Applications of Graph Partitioning Algorithms To Terrain Visibility and Shortest Path Problems

by

Lamis M. Farrag

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

May 1998

Abstract

Geographic Information Systems (GIS) often place great computational demands on sequential computer systems. The need to improve GIS performance lead to research that focuses on applying parallel computing to GIS. In this thesis, graph partitioning theorems are studied in order to investigate the possibility of using graph partitioning algorithms to improve the performance of some substeps in geographic information systems. A vertex-separator algorithm is implemented and used in solving two parallel GIS applications, namely a parallel horizon line computation application and a parallel all-pairs shortest path application. The performance of the vertex-separator algorithm in solving the horizon line computation problem in parallel is compared against the performance of the interval partitioning algorithm in solving the same problem. From the results obtained in this thesis, it is concluded that the vertex-separator algorithm can be used to improve the performance of some GIS applications. In addition, it can be used as a preprocessing phase because it is independent of the algorithm that uses the partitioning results i.e., it can be used to partition the input data and then several algorithms can use the results of the partitioning.

Acknowledgements

I would like to thank both of my supervisors Dr. Jörg-Rüdiger Sack and Dr. Anil Maheshwari. Their knowledge, support, and insight were greatly appreciated throughout the development of this thesis.

I would also like to thank Dr. Aleksandrov for his advice and assistance during the implementation of the vertex-separator algorithm. I would also like to thank all members of the PARADIGM group at Carleton for ideas and discussions.

Finally, I would like to thank my parents and Walied, my husband, for their help and support throughout the development of this thesis.

Contents

List of Tables

List of Figures

List of Symbols

C	a graph separator
A	a graph partition
B	a graph partition
D	a graph partition
$f(n)$	the nonnegative function for a separator theorem
α	constant for the size of the largest partition obtained using a particular graph separator theorem
β	constant for the graph separator size obtained using a particular graph separator theorem
G	a graph
V	vertices of a graph (in Chapter 3 and 5)
E	edges of a graph
n	number of vertices in a graph
m	number of edges in a graph
T	a spanning tree of a graph
BFS	a breadth-first spanning tree of a graph
r	radius of a spanning tree (in Chapter 3)
S	a separation graph
g	the genus of a graph
ε	constant in an ε-separator theorem for the size of the largest partition
$w(G)$	weight of a graph G
ν	number of partitions obtained from the vertex-separator algorithm
Σ	a plane subdivision
σ	a polyhedral terrain
TIN	Triangulated Irregular Network
V	viewpoint (in Chapter 4)

L	list of candidate points
(θ, α)	spherical coordinates
q	number of shortest path queries

Chapter 1

Scope of the Research

Geographic Information Systems (GIS) are computer based tools used for collecting and analyzing geographic data, (Tomlin, 1990). GIS has applications in a large number of public management problems. Examples of GIS applications include traffic management, weather forecasting and mineral exploration. Two important areas in GIS are computation of visibility information on terrains and shortest paths.

Terrain visibility deals with the computation of visible portions of the terrain surface from a selected viewpoint. This visibility information can be used to compute paths with specified visibility characteristics, e.g. hidden paths with respect to a predefined set of observers. It can also be used in navigation and terrain exploration applications as well as other applications.

In shortest path problems, it is required to compute a path from a source location to a target location that has minimum cost. The cost can represent distance, time spent in traveling or energy. Shortest path problems arise in traffic control, search and rescue and navigation.

GIS often require intensive computation operations and the ability to process large volumes of data. On sequential machines, some GIS applications can take enormous computation time. The need to improve GIS performance lead to research that focuses on applying parallel computing to GIS.

In this thesis, graph partitioning theorems are studied in order to investigate the possibility of using graph partitioning algorithms to improve the performance of geographic information systems. The first part of this thesis focuses on graph separator theorems. Using graph separators, graph problems can be solved efficiently by applying a parallel approach. In this parallel approach, the graph is divided into a set of subgraphs that define 'independent' and smaller problems. Every processor is assigned a subgraph and works on its solution. Afterwards, the solutions computed by all processors are combined to provide the solution to the original problem. In this thesis. a vertex-separator algorithm is implemented and used in two parallel GIS applications, namely a parallel horizon line computation application and a parallel all-pairs shortest path application.

The second part of this thesis focuses on solving terrain visibility problems. In this part, computing visibility information is done in object-space format. In other words, the visible portions of the objects are computed. This allows other applications to do further analysis on the computed visibility information. A parallel horizon line computation application is implemented in this thesis. Parallelism is achieved in this application by partitioning the input data among a number of processors and every processor computes the horizon line using the portion of data assigned to it. The parallel horizon line computation application is first implemented using an interval partitioning algorithm. Next, the same application is implemented using the vertex-separator algorithm. The performance of each case is measured and a comparison is performed.

The third part of this thesis focuses on solving all-pairs shortest path problems in parallel by using graph separators. A parallel preprocessing phase for solving all-pairs shortest path is implemented and a detailed description of the shortest path queries applied on the results of this preprocessing phase is given. The parallel preprocessing phase uses the vertex-separator algorithm implemented in the first part of this thesis to partition the input graph among a number of processors.

This thesis is organized as follows. Chapter 2 presents some basic ideas about parallel computing as well as a short discussion about geographic information systems and their importance. Chapter 3 provides an extended discussion about graph

separator theorems and the vertex-separator algorithm implemented in this thesis. Some results of applying the vertex-separator algorithm on TINs are also presented at the end of this chapter. In Chapter 4, various terrain visibility problems are discussed in addition to some related algorithms. A detailed description of the horizon line computation application implemented in this thesis using the interval partitioning algorithm or the vertex-separator algorithm is also given in Chapter 4. Chapter 5 provides an extended discussion on some of the key algorithms that use graph separators in solving shortest path problems. In Chapter 5, the parallel all-pairs shortest path preprocessing phase implemented in this thesis is presented as well as a detailed description of shortest path queries applied to the results of the parallel preprocessing phase. The results and performance analysis of the horizon line computation application and the parallel all-pairs shortest path preprocessing phase are presented in Chapter 6. Finally, Chapter 7 presents some conclusions and directions for future research.

Chapter 2

Introduction

2.1 Parallel Computing

The main theme of this thesis is the application of parallel computing to geographic information systems. In this section, the basic idea for parallel computing is presented as well as a short discussion about parallel hardware, theoretical models for parallel computing and strategies for computational problems' parallel solutions. In the next section, Section 2.2, a basic description of geographic information systems and their importance is presented.

Scientific computing often requires intensive computation operations and a large amount of data storage. The need to solve these complex problems leads to an increasing demand for high performance computers. Problems that fall in this category include environmental modeling, molecular modeling, biomedical imaging, and computational fluid dynamics.

Parallel computing is a branch of computing that offers high performance by making multiple processors work in concert. A basic description for how parallel computing is applied to a problem is the following. The original problem is divided into smaller subproblems that are easier to solve. These subproblems are then solved using a number of processors and their results are combined to form the solution to the original problem. In addition to the need for high performance computing, the falling cost of hardware has made research in parallel computing available to many institutions.

2.1.1 Parallel Computers

A *parallel computer* is one that can execute more than one instruction at the same time. A parallel computer is called *synchronous* if all its processors execute an instruction simultaneously; otherwise, it is called *asynchronous*. A parallel computer is categorized as *SIMD* (single instruction multiple data stream) if all its processors synchronously execute the same instruction on different data. Some old parallel computers classified as SIMD include CM-2, MasPar MP-1 and MP-2. A parallel computer whose processors execute different instructions at the same time is categorized as *MIMD* (multiple instruction multiple data stream). Most modern parallel computers fall in the MIMD category, for example. nCUBE, iPSC, AVX Series, and Cray T3D.

SIMD computers only need one control unit and, therefore, use less memory to store the program. They are suitable for *data parallel programs* where it is required to execute the same set of instructions on a large data set. On the other hand, MIMD computers can be used in a variety of computer applications because every processor can run its own program. However, MIMD computers require more complex techniques for synchronization.

The address-space organization is another factor used in categorizing parallel computers. Parallel computers can have shared memory, distributed memory or distributed-shared memory. A description of shared memory and distributed memory parallel computers is presented next.

Shared Memory

In a shared memory parallel computer, all processors have access to the same address-space. When a processor wants to send data to another processor, it writes the data in the shared memory and the second processor reads it from memory. The shared variables can also be used for synchronization. The logic for shared memory is controlled by the operating system kernel. Shared memory parallel computers include Cray C-90 and KSR-1, (JaJa, 1992), (Fountain, 1994).

Distributed Memory

In a distributed memory parallel computer, every processor has its own local memory. Communications and synchronizations between processors are done using message passing. The advantage of distributed memory parallel computers is that their hardware is simpler to implement; on the other hand, the communication is left to the application developer. Examples of distributed memory parallel computers include the AVX Series II and III, the iPSC and nCUBE, (JaJa, 1992), (Fountain, 1994).

The experimental results presented in this thesis in Chapter 6 were executed on a parallel machine with 16 pentium computers connected with a high speed switch. At the time the experiments were performed, only 15 processors were available.

2.1.2 Theoretical Models for Parallel Computing

The Parallel Random Access Machine (PRAM) is a theoretical model for parallel computing that expresses the power of multiple processors, but uses a rather idealistic communication model. The *Parallel Random Access Machine* model resembles the standard sequential Random Access Machine model. However, ignoring communication in the PRAM model makes algorithms that run quickly on a PRAM, run slowly on a real parallel machine, (JaJa, 1992). Other parallel computing models attempted to deal with this by modeling the connection network. Examples of such models include the mesh and hypercube. In these models, higher lower bounds on the time cost of an algorithm is sometimes imposed by the cost of communication over a network and the specific topologies.

A more realistic model is the *Bulk Synchronous Parallel* (BSP) model. The BSP model concentrates on the parallelism of a program rather than the topology of the connection network. The parallelism of a program is referred to in this model as the *parallel slackness* and it is to a certain extend greater than the actual parallelism available. In the BSP model, a program is divided into a series of phases called *super-steps*. Every super-step consists of a computation stage and a communication stage, (Valiant, 1990).

In the algorithms implemented in this thesis, the communication cost is relatively small compared to the gain in performance due to parallelism. This can be seen from the results presented in Chapter 6.

2.1.3 Strategies for Computational Problems' Parallel Solutions

In designing parallel solutions for computational problems, there are two approaches that can be used, namely a data parallel and a functional parallel approach. The choice of which approach to use is dependent on the nature of the computational problem to be solved.

In the data parallel approach, the input data is partitioned among the processors and every processor executes the same algorithm on its set of data. After the processors complete executing the algorithm, the results are either collected by one processor or remain distributed. This is dependent on what is going to be performed next on the results.

In the functional parallel approach, each processor handles a task of the work different from the rest of the processors. In this approach, the output of one processor may be the input to another processor. An example of using the functional parallel approach is pipelining. In pipelining, each processor is assigned a stage of a multistage computational problem. The first processor reads the input data and executes its task on this data. The second processor takes the output of the first processor, executes its task on it and passes its output to the third processor which passes its output to the next processor in the same way. The output of the whole problem is the output of the last processor. Although the time taken to produce one result using the pipelining method is at least as long as the time to produce this result on a sequential computer, the total computation time for all the input data is less than the corresponding time using a sequential computer.

2.2 Geographic Information Systems (GIS)

> A *Spatial Information System* is a computerized system for gathering, storing, presenting, manipulating and interpreting spatial data for phenomena below, on or above the earth's surface. The system is called *Geographic Information System* (GIS) when the data are geographical data, i.e. the data deals with the surface (or the space close to the surface) of the earth, (Hutchinson et al., 1996).

Geographic information systems generally use two types of data encoding. The first one is the raster format while the second is the vector format. Choosing which format to use is dependent upon which is more suitable for the given problem. A *raster* is a uniform grid imposed on the area of interest. The grid is built by dividing the area into rows and columns at regular intervals. The intersection of a row and column in a raster is called a *cell*. Every cell has data associated with it. The size of a raster is measured by the number of cells it has. For example, if a raster has m cells in every column and n cells in every row then the size of the raster is mn cells.

In the *vector* format, the location of data objects are represented explicitly by their coordinates in some frame reference. Using the vector format is more efficient than using the raster format if the data are distributed sparsely over the area of interest. In such a case, most raster cells will be empty.

Geographic information systems are of increasing economic importance due to their application to a large scale of public management problems. Such problems include:

- Routing Problems: traffic management and snow removal planning
- Weather Forecasting
- Mineral Exploration

Some applications that the PARADIGM group at Carleton worked on include forest fires, air pollution, and ice tracking.

Since GIS often require computation intensive operations and the ability to process large volumes of data, parallel computing seems like a natural choice to improve GIS performance. *In recent years the demand on Spatial Information Systems technology from application areas has sharply increased. In particular, with the EOS (Earth Observation System), terabytes of spatial information will be collected daily. The increase in speed in sequential computing cannot keep up with the demand generated by many systems handling spatial data. Therefore a need for parallel computing in this area is widely recognized,* (Hutchinson et al., 1996). The main goal of this thesis is to show that applying parallel techniques to GIS problems demonstrates a significant improvement in performance.

Chapter 3

Graph Separators

3.1 Introduction to Graph Separators

A graph separator is defined as a small (in size) set C of vertices or edges whose deletion divides the graph of consideration into components that satisfy some conditions. A standard approach for partitioning graphs is to find the set C. Graph separators provide a powerful tool for developing efficient algorithms for a variety of problems. They can be used in important applications such as numerical analysis (Lipton et al., 1979), VLSI design (Bhatt and Leighton, 1984), finding approximate solutions to NP-complete problems (Lipton and Tarjan, 1980), shortest path problems (Frederickson, 1987), the design of parallel algorithms (Goodrich, 1992) and many others. An example of a graph separator is given in Figure 3.1. A graph is represented by a rectangle. A separator, drawn as a dark line, is computed for this graph and it divides the graph into three partitions.

By using graph separators, graph problems can be solved in parallel. In this approach, the graph is divided into a set of subgraphs that define 'independent' and smaller problems. Every processor works on the solution for a subgraph. Finally, the solutions computed by all the processors are combined to provide the solution to the original problem.

There are separator theorems for many classes of graphs, for instance, planar graphs (Lipton and Tarjan, 1979), forests, grids (Lipton et al., 1979), graphs of bounded genus (Gilbert et al., 1984), graphs with an excluded minor (Alon et al.,

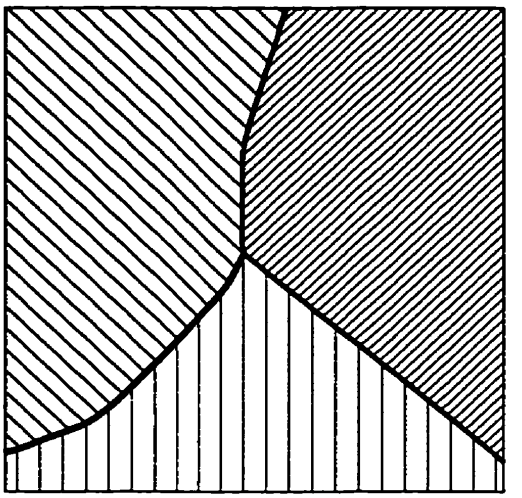


Figure 3.1: An example of a graph separator

1990), geometric graphs (Miller et al., 1991) and others. The planar separator theorem by Lipton and Tarjan (1979) provides a basis for the known graph separator theorems.

Let S be a class of graphs closed under the subgraph relation and $f(n)$ be a nonnegative function. A formal definition for an $f(n)$-separator theorem for S is the following:

> *For any n-vertex graph G in S, there exist constants $\alpha < 1$ and $\beta > 0$ so that G can be partitioned into three sets A, B, and C such that no edge joins a vertex in A with a vertex in B, neither A nor B contains more than αn vertices and C contains no more than $\beta f(n)$ vertices.*

In this theorem, the set C is called the separator and the efficiency of the application algorithms depends on its size. Therefore, once an optimal function $f(n)$ for a class of graphs is found, it is important to make the constant β as small as possible, (Nishizeki and Chiba, 1988).

3.2 Preliminaries

In this section, some notions will be introduced that are going to be used in proceeding sections.

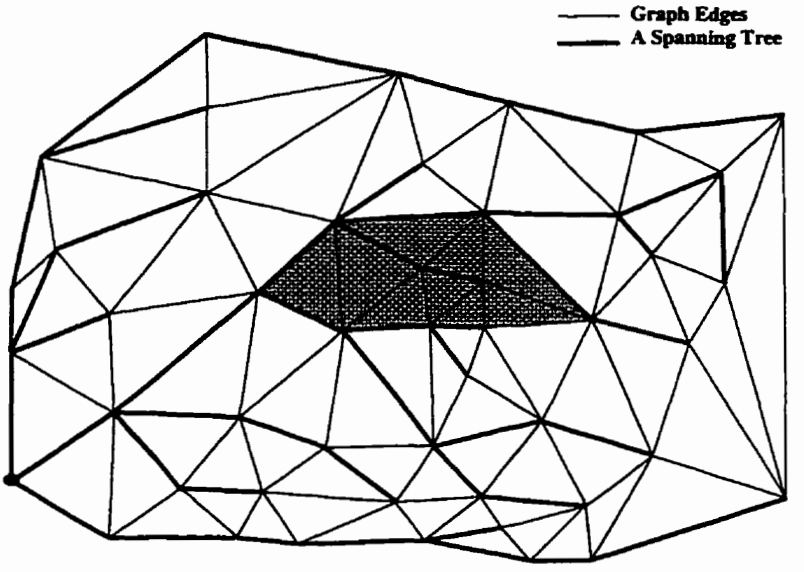


Figure 3.2: An example of a T-cycle. It divides the graph into an inside (gray) and outside part.

3.2.1 Graphs

A *graph* $G=(V,E)$ consists of a finite set of vertices V and a finite set of edges E. The elements of E are pairs of elements of V. Two vertices in a graph are *adjacent* if they are joined by an edge. A graph can be drawn by representing its vertices as points and its edges as segments. A graph is *planar* if it can be drawn on a plane in such a way that no two edges intersect except at an endvertex in common. A planar graph is a *maximal planar graph* if and only if the addition of one edge makes the graph not planar, (Nishizeki and Chiba, 1988).

3.2.2 A Spanning Tree of a Graph

A *spanning subgraph* of a graph $G=(V,E)$ consists of the set of vertices V and a set of edges E', where $E' \subset E$. A *spanning tree* of a graph $G=(V,E)$ is a spanning subgraph of G that is a tree. The *radius* of a rooted spanning tree is the length of the longest path in the tree from the root vertex to a leaf, (Nishizeki and Chiba, 1988).

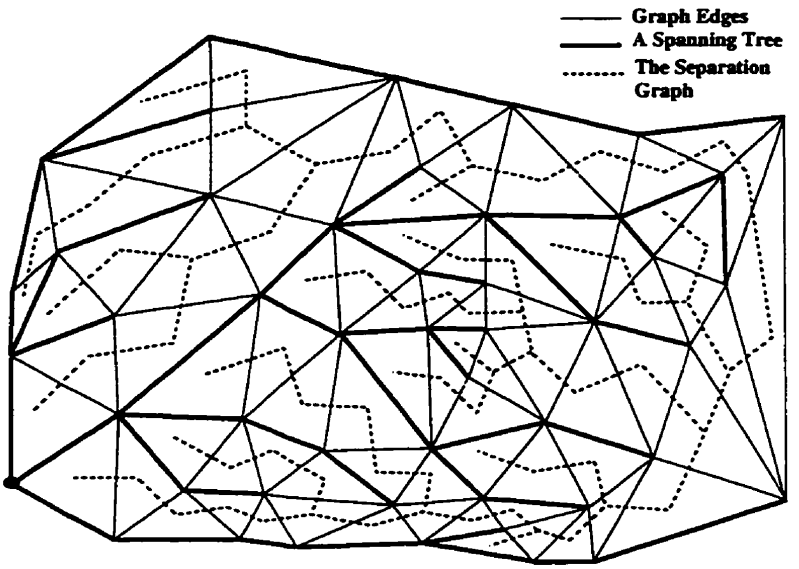


Figure 3.3: Separation Graph

3.2.3 T-cycles

Given a graph G and a rooted spanning tree T with radius r, then each non-tree edge of G forms with the edges of T a unique simple cycle. This cycle is called a *fundamental* or a *T-cycle*. Any T-cycle will contain at most $2r + 1$ vertices of G if it contains the root of T, or $2r - 1$ vertices otherwise. Also, this cycle will divide a graph into two parts, namely, an inside and an outside part. This notion of a T-cycle is related basically to the classic Jordan's theorem proved in the 19th century and it is describes by Djidjev and Venkatesan (1991). Figure 3.2 shows an example of a T-cycle and how it divides a graph into two parts.

3.2.4 Separation Graph

Let G be a graph and T a spanning tree of G. The *separation graph* of G with respect to T, $S(G,T)$, is then constructed as follows. Make every face of G a triangle by adding non-tree edges where necessary. This triangulation is not necessary in constructing separation graphs but it is used here to make the separation graph of degree three. Every face in G will be represented by a vertex in S. If two faces in G share a common non-tree edge, their corresponding vertices in S will be joined by

an edge. Therefore, any edge in S determines a unique T-cycle, namely, the T-cycle formed by the corresponding non-tree edge. If it is required that the separation graph is connected, a vertex is added to the separation graph representing the external face of G. If a face in G shares a non-tree edge with the external face of G, the corresponding vertices in S will be joined by an edge, (Djidjev, 1988).

Accordingly, the separation graph is a subgraph of the dual graph. It is of degree three and it is simpler and sparser than the original graph. Figure 3.3 shows an example of a separation graph. The following equations come from the definition of separation graphs.

$$|V(S)| = |F(G)| \tag{3.1}$$

$$|E(S)| = |E(G)| - |V(G)| + 1 \tag{3.2}$$

Using Euler's formula, Eq(3.3), and the two equations Eq(3.1) and Eq(3.2), a new equation for separation graphs can be derived, Eq(3.4), (Aleksandrov and Djidjev, 1996).

$$|V(G)| - |E(G)| + |F(G)| = 2c - 2g \tag{3.3}$$

where c is the number of connected components of G and g is the genus of G.

$$|E(S)| - |V(S)| = 2g - 1 \tag{3.4}$$

3.2.5 The Genus of a Graph

Closed surfaces are classified into two types, namely *orientable* and *nonorientable surfaces*. An informal definition for an orientable surface is that if a bug starts from a point on a closed curve drawn on the surface and traverses the curve, it will return to the initial point with the same initial orientation. If there is at least one curve where the bug will return to the initial point with the initial orientation reversed, then this surface is a nonorientable surface. Every orientable surface is the sum of a certain number of tori g ,where $g \geq 0$. The number g is called the *genus of the surface*. The *orientable genus surface of a graph* is the orientable surface with minimum genus on which the graph can be embedded. The *genus of the graph* is the genus of its orientable genus surface.

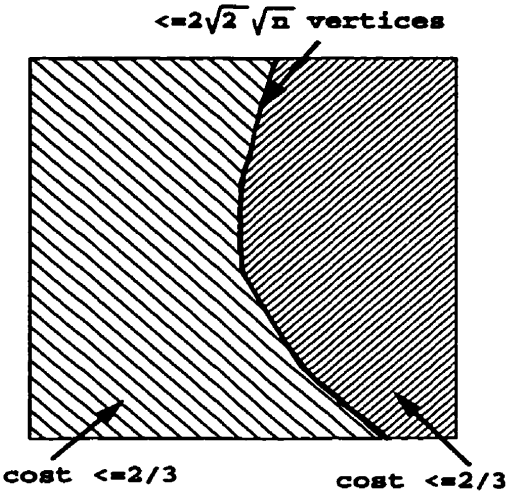


Figure 3.4: An example of Theorem 1

3.3 Separator Theorems for Planar Graphs

Lipton and Tarjan (1979) proved the existence of a $\sqrt{n}$-separator theorem for all planar graphs. This theorem provides the basis for all known separator theorems and it goes as follows:

> **Theorem 1.** (Lipton and Tarjan, 1979) *Let G be any n-vertex planar graph having nonnegative vertex costs summing to no more than one. Then the vertices of G can be partitioned into three sets A, B, C such that no edge joins a vertex in A with a vertex in B, neither A nor B has total cost exceeding $2/3$, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices.*

An example for Theorem 1 is given in Figure 3.4. In a case where there is no cost on the vertices of G, each vertex can be assigned a cost of $1/n$. In this case, the separator splits the vertex set according to cardinality. Another theorem was introduced for the value of $\alpha = 1/2$.

> **Theorem 2.** (Lipton and Tarjan, 1979) *Let G be any n-vertex planar graph having nonnegative vertex costs summing to no more than one. Then the vertices of G can be partitioned into three sets A, B, C such that no edge joins a vertex in A with a vertex in B, neither A nor B has*

total cost exceeding $1/2$, *and* C *contains no more than* $2\sqrt{2}\sqrt{n}/(1-\sqrt{2/3})$ *vertices.* The value of $2\sqrt{2}/(1-\sqrt{2/3})$ is approximately 15.413.

The first theorem was extended to finite element graphs. These are graphs formed from planar embeddings of planar graphs by adding all possible diagonals to each face. An element in the graph will be a region surrounded by a set of vertices and there are no vertices inside this region. The following theorem was proved for such finite element graphs.

Theorem 3. (Lipton and Tarjan, 1979) *Let* G *be any* n*-vertex finite element graph with nonnegative vertex costs summing to no more than one. Suppose no element of* G *has more than* k *boundary vertices. Then the vertices of* G *can be partitioned into three sets* A, B, C *such that no edge joins a vertex in* A *with a vertex in* B, *neither* A *nor* B *has total cost exceeding* $2/3$, *and* C *contains no more than* $4\lfloor k/2 \rfloor \sqrt{n}$ *vertices.*

Djidjev (1982) improved the case of unweighted planar graphs by reducing the constant for the separator size. For $\alpha = 2/3$, the theorem is:

Theorem 4. (Djidjev, 1982) *Let* G *be any* n*-vertex planar graph. The vertices of* G *can be partitioned into three sets* A, B, C *such that no edge joins a vertex in* A *with a vertex in* B, $|A| \leq 2n/3$, $|B| \leq 2n/3$, $|C| \leq \sqrt{6n}$.

It was proved that the smallest constant that can replace $\sqrt{6}$ in Theorem 4 must be at least 1.555, (Djidjev, 1982). Another improvement was also given by Djidjev (1982) for unweighted planar graphs for the case where $\alpha = 1/2$.

Theorem 5. (Djidjev, 1982) *Let* G *be any* n*-vertex planar graph. The vertices of* G *can be partitioned into three sets* A, B, C *such that no edge joins a vertex in* A *with a vertex in* B, $|A| \leq n/2$, $|B| \leq n/2$, $|C| \leq \left(k + \frac{3\sqrt{2}}{\sqrt{3}-1}\right)\sqrt{n}$ *where* $k = \frac{3+\sqrt{21}}{2}$. The value of $\left(k + \frac{3\sqrt{2}}{\sqrt{3}-1}\right)$ is approximately 9.587.

Venkatesan (1987) improved the constant in Theorem 2 which is for partitioning weighted planar graphs into halves. The new theorem was presented as follows:

Theorem 6. (Venkatesan, 1987) *Let G be any n-vertex planar graph having non-negative vertex weights adding to no more than* 1. *Then the vertices of G can be partitioned into three sets A, B, C such that A and B are not connected to each other, neither A nor B has total weight exceeding* $\frac{1}{2}$, *and C contains no more than* $(8 + \frac{16}{81}(\sqrt{6}/(1 - \sqrt{2/3})))\sqrt{n}$ *vertices.* The value of $(8+\frac{16}{81}(\sqrt{6}/(1-\sqrt{2/3})))$ is approximately 10.6367.

For the partitioning of unweighted planar graphs into halves, described in Theorem 5, an improvement was made by Venkatesan (1987) and presented in the following form :

Theorem 7. (Venkatesan, 1987) *Let G be an n-vertex planar graph. Then the vertices of G can be partitioned into three sets A. B, C such that no edge connects A with B, A and B each have* $\leq n/2$ *vertices, and C contains* $(7+1/\sqrt{3})\sqrt{n}$ *vertices.* The value of $(7+1/\sqrt{3})$ is approximately 7.5774.

3.3.1 Separator Theorems For A Regular γ-Partition

A *regular* γ*-partition* is used to partition a graph into three subgraphs. Its formal definition given by Djidjev (1982) is the following:

Let G=(V,E) be an n-vertex graph and $0 \leq \gamma \leq 1$. *The sets A, B, C and D are a regular* γ*-partition of V if A, B. C and D partition V, no edge joins a vertex in A with a vertex in B. a vertex in B with a vertex in C or a vertex in C with a vertex in A,* $|A| \leq (1-\gamma)n$, $|B| \leq (1-\gamma)n$, *and* $|C| \leq \gamma n$.

Djidjev (1982) introduced a theorem for a regular γ-partition in the case of unweighted planar graphs. The theorem states the following.

Theorem 8. (Djidjev, 1982) *Let G be any n-vertex planar graph and* $1/2 \leq \gamma \leq 1$. *Then there exists a regular* γ*-partition of V into four sets A, B. C, D such that* $|D| \leq 3\sqrt{2}\sqrt{n}$.

The value of $3\sqrt{2}$ is approximately 4.243. In the case where $\gamma = 1/2$, this constant can be reduced to $(3+\sqrt{21})/2$ which is approximately equal to 3.791.

Theorem 9. (Djidjev, 1982) *Let G=(V,E) be any n-vertex planar graph. Then there exists a regular 1/2-partition of V into four sets A, B, C, D such* $|D| \leq k\sqrt{n}$, *where* $k = (3+\sqrt{21})/2$.

Venkatesan (1987) improved the constant in Theorem 8 and established the following theorem:

Theorem 10. (Djidjev, 1982) *Let G be an n-vertex planar graph. If* $1/2 \leq \gamma \leq 1$, *then there exists a regular* γ*-partition (A, B, C, D) of G such* $|D| \leq 2\sqrt{3}\sqrt{n}$. $2\sqrt{3}$ is approximately equal to 3.464.

3.3.2 Separator Theorems For An ε-Separator

An *ε-separator* partitions a graph in such a way that no component has a weight higher than ε. Venkatesan (1987) introduced a theorem for a $\sqrt{n/\varepsilon}$-separator for weighted planar graphs. The theorem states:

Theorem 11. (Venkatesan, 1987) *Let G be an n-vertex planar graph with nonnegative vertex weights adding to* ≤ 1, *and let* $0 < \varepsilon \leq 1$. *Then there exists some set C of* $2\sqrt{6}\sqrt{n/\varepsilon}$ *vertices whose deletion leaves G with no connected component of weight* $> \varepsilon$. *Furthermore the set C can be found in* $O(n \log n)$ *time.* The value of the constant $2\sqrt{6}$ is approximately equal to 4.89898.

Djidjev (1988) improved the size of the separator and the time complexity of the algorithm. The new theorem states the following:

Theorem 12. (Djidjev, 1988) *Let G be an n-vertex weighted planar graph and let* $\varepsilon > 0$. *There exists a set C of* $\leq 4\sqrt{n/\varepsilon}$ *vertices whose removal leads to a graph with no component of weight* $> \varepsilon w(G)$. *Moreover C can be found in* $O(n)$ *time.*

where w(G) is the weight of G. The number of components resulting from applying Theorem 12 is of order $\Omega(1/\varepsilon)$. Theorems 13 and 14 were also introduced by Djidjev (1988) for weighted and unweighted planar graphs, respectively, where $0 < \varepsilon \leq 1/3$.

Theorem 13. (Djidjev, 1988) *Let G be an n-vertex weighted planar graph and let $0 < \varepsilon \leq 1/3$. There exists a set C of $\leq \sqrt{96\varepsilon n}$ vertices whose removal leads to a graph with components of weight $\leq (1-\varepsilon)w(G)$. Moreover C can be found in $O(n)$ time.*

Theorem 14. (Djidjev, 1988) *Let G be an n-vertex planar graph and let $0 < \varepsilon \leq 1/3$. There exists a set C of no more than $8\sqrt{\varepsilon n}$ vertices whose removal leads to a graph with components of $\leq (1-\varepsilon)n$ vertices. Moreover C can be found in $O(n)$ time.*

3.3.3 Simple Cycle Graph Separation

Simple cycle graph separators were first examined by Miller (1984). Later on, Djidjev and Venkatesan (1991) examined simple cycle separators for maximal planar graphs. Theorem 15 and 16 for unweighted and weighted maximal planar graphs, respectively, were established by Djidjev and Venkatesan (1991) and can be implemented with $O(n)$ algorithms.

Theorem 15. (Djidjev and Venkatesan, 1991) *Let G be an n-vertex maximal planar graph, and $\delta \leq 1/3$. Then the vertex set of G can be partitioned into three sets A, B, C such that neither A nor B contains more than $(1-\delta)n$ vertices, no edge from G connects a vertex in A to a vertex in B, and C is a simple cycle with no more than $(\sqrt{2\delta} + \sqrt{2-2\delta})\sqrt{n} + O(1)$ vertices if $0.1 \leq \delta \leq 1/3$ and no more than $\sqrt{32\delta n}$ vertices if $0 \leq \delta \leq 0.1$.* For the case when $\delta = 1/3$, $(\sqrt{2\delta} + \sqrt{2-2\delta})$ will be approximately 1.971.

Theorem 16. (Djidjev and Venkatesan, 1991) *Let G be an n-vertex maximal planar graph with nonnegative vertex weights adding to one.*

Then there exists a simple cycle (2/3)*-separator of G of no more than* $2\sqrt{n} + O(1)$ *vertices.*

3.3.4 Partitioning Planar Graphs With Costs And Weights On Vertices

Many applications require the association of costs with the vertices or the edges of the graph. For instance, costs could be used to represent the amount of data needed in the communication between processes in a distributed system, or the sizes of circuit elements needed. Therefore, such applications' type need another measurement for the separator size besides its cardinality. They need a representation for the cost of the separator. Djidjev (1997) introduced separator Theorems 17 and 18 for graphs that have costs and weights on their vertices. Costs are used to represent the size of the separator where as the weights are used to represent the sizes of the graph components obtained after the partitioning.

Theorem 17. (Djidjev, 1997) *Let G be a planar graph with nonnegative vertex weights that sum up to one and nonnegative vertex costs whose squares sum up to N(G). There exists a* 2/3 *vertex separator of G with total vertex cost no more than* $\sqrt{8N(G)}$. *Such a separator can be found in* $O(|G|)$ *time.* This theorem is optimal within a constant factor.

Theorem 18. (Djidjev, 1997) *Let G be a planar graph with nonnegative vertex weights that sum up to one and nonnegative vertex costs whose squares sum up to N(G). There exists a* 1/2 *vertex separator of G of cost* $O(\sqrt{N(G)})$, *which can be found in* $O(|G|)$ *time.*

3.3.5 A Partitioning Algorithm For Planar Graphs

The partitioning described in Theorem 1 can be computed using an $O(n)$ algorithm, (Lipton and Tarjan, 1979). This algorithm can be modified to compute the partitions described in Theorem 4, 5, 8 and 9. The following is a description of the main steps of the algorithm.

Step 1: Find a planar embedding of G.

Step 2: Find the connected components of G and compute the cost of each one.

- If no component has cost exceeding 1/3:
  - Let the components of G be $G_1, G_2, \cdots, G_k$ with vertex sets $V_1, V_2, \cdots, V_k$, respectively.
  - Find the minimum index i such that the total cost of $V_1 \cup V_2 \cup \cdots \cup V_i$ exceeds 1/3.
  - Let $A = V_1 \cup V_2 \cup \cdots \cup V_i$, $B = V_{i+1} \cup V_{i+2} \cup \cdots \cup V_k$ and $C = \emptyset$.
- Else if no component has cost exceeding 2/3:
  - Let the components of G be $G_1, G_2, \cdots, G_k$ with vertex sets $V_1, V_2, \cdots, V_k$, respectively.
  - Let G_i be the component with the cost between 1/3 and 2/3.
  - Let $A = V_i$, $B = V_1 \cup \cdots \cup V_{i-1} \cup V_{i+1} \cup \cdots \cup V_k$ and $C = \emptyset$.
- Else go to step 3.

Step 3: Construct a breadth-first spanning tree for the most costly component. Compute the level of each vertex and the number of vertices $L(l)$ in each level l.

Step 4: Find a level index i such that the total cost for the vertices from level 0 to level $l_i - 1$ does not exceed 1/2, but the total cost for the vertices from level 0 to level l_i does exceed 1/2. Let k be the number of vertices from level 0 to level l_i.

Step 5: Find the highest level $l_o \leq l_i$ such that $L(l_o) + 2(l_i - l_o) \leq 2\sqrt{k}$. Find the lowest level $l_2 \geq l_i + 1$ such that $L(l_2) + 2(l_2 - l_i - 1) \leq 2\sqrt{n-k}$.

Step 6: Delete the vertices from level l_2 and all the levels above. Shrink the rest of the graph using the following method.

- Delete all vertices from level 0 to level l_o and replace them by one vertex call it x.

- Construct a boolean table with one entry for every vertex.
  - The entries for the vertices from level 0 to level l_o will have the value TRUE.
  - The entries for the vertices from level $l_o + 1$ to level $l_2 - 1$ will have the value FALSE.
  - Call the subtree from level 0 to l_o of the original breadth-first spanning tree T.
  - Scan clockwise the edges incident to T. For an edge (v,w) with v in T
    * If the table entry for w is TRUE, then delete (v,w).
    * If the table entry for w is FALSE, change it to TRUE, delete (v,w) and add an edge (x,w).
- Call the new graph G'

Step 7: Construct a breadth-first spanning tree rooted at x for G'. Make all the faces of G' triangles by adding non-tree edges where necessary. Let every vertex in the tree point to its parent vertex and know the the cost of all its descendants including itself.

Step 8: Select any non-tree edge (v_l, w_l) and compute its corresponding T-cycle. Compute the cost of each side of the T-cycle and call the side with greater cost the inside of the T-cycle.

Step 9: If the cost of the inside of the current T-cycle exceeds 2/3 find a better cycle using the following method.

- Call the current non-tree edge forming the current T-cycle (v_i, w_i).
- Find the triangle (v_i, y, w_i) which has one of its edges the non-tree edge (v_i, w_i) and lies inside the T-cycle.
- If one of the other two edges is a tree edge and the second is a non-tree edge, make the non-tree edge the current (v_{i+1}, w_{i+1}) and compute the cost inside its T-cycle.

- If non of the other two edges are tree-edges, determine the tree path from y to (v_i, w_i) by following parent pointers from y.
- Let z be a vertex on the (v_i, w_i) T-cycle and on the tree path from y to (v_i, w_i).
- Compute the costs of the inside of the two new T-cycles and choose the one with more inside cost inside it.
- Repeat step 9 until a T-cycle with inside cost not exceeding 2/3 has been found.

Step 10: Let A be the inside of the T-cycle, C will contain the vertices on levels l_o,l_2, and the T-cycle (excluding the vertex x). B will contain the remaining vertices.

3.4 Separator Theorems for Embedded Graphs of Bounded Genus

Another class of graphs for which separator theorems were introduced is the bounded genus class of graphs. Gilbert et al. (1984) proved three separator theorems for graphs of bounded genus. The first theorem is for partitioning an unweighted graph of bounded genus into components with no more than $2/3n$ vertices. The second theorem is for weighted graphs of bounded genus and finally, the third theorem is for the existence of an ε-separator for weighted graphs of bounded genus. Those theorems are stated as follows:

Theorem 18. (Gilbert et al., 1984) *A graph of genus g with n vertices has a set of at most $6\sqrt{gn} + 2\sqrt{2n} + 1$ vertices whose removal leaves no component with more than $2n/3$ vertices.* Such a separator can be found in $O(n + g)$ time.

Theorem 19. (Gilbert et al., 1984) *Let G be an n-vertex graph of genus g whose vertices have nonnegative weights. The vertices of G can*

be partitioned into three sets A, B, and C such that no edge joins a vertex in A with a vertex in B, C contains $O(\sqrt{gn})$ vertices, and neither A nor B contains more than half the total weight. The set C can be found in $O(n+g)$ time.

Theorem 20. (Gilbert et al., 1984) *Let G be an n-vertex graph of genus g whose vertices have nonnegative weights summing to no more than one. Let $0 < \varepsilon \leq 1$. Then there is a set C of $O(\sqrt{gn/\varepsilon})$ vertices whose removal leaves no connected component with total weight exceeding ε. The set C can be found in $O((n+g)\log n)$ time.*

Djidjev (1988) improved the time for the algorithm for an ε-separator for graphs of bounded genus for the case of unweighted graphs.

Theorem 21. (Djidjev, 1988) *Let G be an n-vertex graph of genus g and let $0 < \varepsilon \leq 1/3$. There exist a set C of $\leq 8\sqrt{(4g\varepsilon+1)\varepsilon n}$ vertices whose removal leads to a graph with components of $\leq (1-\varepsilon)n$ vertices each. Moreover C can be found in $O(n)$ time.*

Aleksandrov and Djidjev (1990) improved the constant in Theorem 18 by a factor of $\sqrt{2}$. This improvement was realized by using the notion of separation graphs in order to obtain better results in partitioning graphs of bounded genus. The theorem that was established is for unweighted graphs of genus g; however, according to Aleksandrov and Djidjev (1990), another version of the theorem can be proved for the case of weighted graphs. Aleksandrov and Djidjev (1990) also presented a theorem for graphs of non-orientable genus.

Theorem 22. (Aleksandrov and Djidjev, 1990) *Let G be an n-vertex graph of genus g. The vertices of G can be divided into three sets A, B, C such that no edge joins A and B, $max(|A|,|B|) \leq (2/3)n$ and $|C| \leq \sqrt{6(g+1)n}$.* Given the embedding of G, a linear time algorithm can be implemented to find such a separator.

Theorem 23. (Aleksandrov and Djidjev, 1990) *Let G be an n-vertex graph of non-orientable genus q. The vertices of G can be divided into three sets A, B, C such that no edge joins A and B, $max(|A|, |B|) \leq (2/3)n$ and $|C| \leq \sqrt{(3q+9)n}$.*

According to Theorem 22, for the case where G is a toroidal ($g = 1$), C will contain $\leq \sqrt{12n}$ vertices. The value of $\sqrt{12}$ is approximately 3.4641. The lowest constant that can replace this value is $\sqrt{2\pi/\sqrt{3}}$ which is approximately 1.9046, (Aleksandrov and Djidjev, 1990). For the case where the graph has costs and weights on its vertices, there is a 2/3-separator theorem by Djidjev (1997).

Theorem 24. (Djidjev, 1997) *Let G be an n-vertex graph embedded on a surface of genus g with non-negative vertex weights that sum up to one and non-negative vertex costs whose squares sum up to $N(G)$. There exists a 2/3-separator of G with total vertex cost no more than $\sqrt{(16g+8)N(G)}$. Such a separator can be found in $O(|G|)$ time, given the embedding of G.*

Aleksandrov and Djidjev (1996) proved an ε-separator theorem for weighted graphs embedded on a surface of orientable genus g. In that theorem, the separator has no more than $4\sqrt{(g+1/\varepsilon)n}$ vertices; moreover, given the embedding of the graph, such a separator can be found in $O(n+g)$ time. Theorem 25 presents the case where $\varepsilon \in (0,1)$, while Theorem 26 presents a special case for the value of $\varepsilon \in (0, 1/15)$.

Theorem 25. (Aleksandrov and Djidjev, 1996) *Let G be an n-vertex graph with non-negative vertex weights and $I(G)$ be a 2-cell embedding of G on an orientable surface of genus g. For any $\varepsilon \in (0,1)$ there exists an ε-separator C of G of no more than $4\sqrt{(g+1/\varepsilon)n}$ vertices. The separator C can be found in $O(n+g)$ time, given $I(G)$.*

Theorem 26. (Aleksandrov and Djidjev, 1996) *Let G be an n-vertex graph with non-negative vertex weights and $I(G)$ be a 2-cell embedding*

of G on an orientable surface Z of genus g. For any $\varepsilon \in (0, 1/15)$ there exists an $(1-\varepsilon)$-separator C_I of G of no more than $2+30\sqrt{2}\sqrt{n\varepsilon(2g\varepsilon+1)}$ vertices. The separator C_I can be found in $O(n+g)$ time, given $I(G)$.

3.5 The Vertex-Separator Algorithm

Previously, computing an ε-separator of size $O(\sqrt{ng/\varepsilon})$ for graphs of genus g was done by recursively computing a 2/3-separator. This approach required an algorithm with time complexity $O(n \log n)$. Therefore, the results of Aleksandrov and Djidjev (1996) are an improvement to the previous known results not only in the size of the separator but also in the time complexity of the separator construction algorithm. The separator size presented in the theorem by Aleksandrov and Djidjev (1996) is smaller than the best previous separators for the planar case and asymptotically optimal for arbitrary genus.

The algorithm to find such a separator uses the notion of separation graphs because of their simple structure and sparsity. More precisely, a theorem is used that states that by constructing an edge-separator for a separation graph S, a separator for the original graph can be constructed. This theorem is used because Aleksandrov and Djidjev (1996) were able to construct edge-separators for sparse graphs in optimal linear time.

Using the separation graph and T-cycles only to construct a separator will give a limit on how small the size of the separator can be because of the fact that the length of a T-cycle has $\Omega(|V|)$. Therefore, in order to achieve an optimal separator size besides an optimal running time for the algorithm, a radius-reducing technique was combined with the use of separation graphs.

The following describes the main steps of the vertex separator algorithm. The algorithm described here is restricted to the case $g = 0$ because the cases for $g >= 0$ are not covered within the scope of this thesis. The vertex-separator algorithm constructs an ε-separator of size no more than $4\sqrt{(1/\varepsilon)n}$ for a given graph G.

Input:

- An embedding of a graph G.
- The non-negative weight function for the vertices of G. The sum of the weights in G does not exceed 1.
- A number $\varepsilon \in (0, 1)$.

Output:

- An ε-separator C of G.
- The set of partitions (subgraphs) for G.

Part I:

Step 1.1: For every vertex v in $V(G)$, compute a list of its adjacent vertices in G, call it *Adj(v)*.

Step 1.2: Choose a vertex v_o and construct a breadth-first spanning tree T of G with v_o as the root vertex.

Step 1.3: Compute the radius of T and call it r.

Step 1.4: For every level j from 0 to $r - 1$

- Construct a list $L(j)$ for the vertices on level j.
- Compute the number of vertices on level j, call it $l(j)$.

Step 1.5: Find a set of indices $j_1 < j_2 < \cdots < j_\mu$ such that

- $j_o = 0$
- $j_{\mu+1} = r$
- $h(\varepsilon, n)$ is a tuning function. $h(\varepsilon, n) = 1/2\sqrt{\varepsilon n}$
- The total number of the vertices $\cup_{i=1}^{\mu} l(j_i)$ is minimum.
- $0 < j_1 < h$
- $|l(j_i)| = min(|l(j_{i-1} + h + \omega)|)$ where $\omega = -1, 0, 1$ for $i = 2, 3, \cdots, \mu$.

Step 1.6: Compute the cycles formed by the chosen indices.

Step 1.7: Construct the subgraphs G_i for $i = 1, 2, \cdots, \mu$. G_i consists of the faces that have at least one vertex of level less than j_i, and that do not belong to G_{i-1}.

Step 1.8: Find the connected components of each G_i, for $i = 1, 2, \cdots, \mu$. Call the total number of connected components μ' where $\mu' \geq \mu$. These connected components, $\Pi = \{G_1, G_2, \cdots, G_{\mu'}\}$ form an initial partitioning of G.

Part II:

Step 2.1: For each component, G_i, in Π that has more than εn vertices do the following

- Construct the separation graph $S = S(G_i, T)$ for G_i with respect to the breath-first spanning tree T.
- Assign weights to the edges of S that reflect the total weight inside the T-cycles in G_i defined by T.
- Find a small set of T-cycles in G_i that partition it in such a way that no connected component will have more than ε total weight.
- Compute the set of components resulting from partitioning using the T-cycles and replace G_i with the new set of components.

Step 2.2: Call the total number of connected components that partition the graph until now ν'. ν' will be $\geq \mu'$ and it will be also $\geq 1/\varepsilon$. Call the total set of connected components $\Pi = \{G_1, G_2, \cdots, G_{\nu'}\}$.

Part III:

Step 3.1: Construct the graph $G(\Pi)$

- $G(\Pi)$ will have ν' vertices representing the connected components in Π.
- Every vertex will be associated with a weight that has the value of the weight of the connected component the vertex represents.

- If two connected components are adjacent in G, there will be an edge connecting their corresponding two vertices in $G(\Pi)$.
- The edges will be assigned weights equal to the size of the boundary in G they represent.

Step 3.2: Shrink edges in $G(\Pi)$ to vertices in such a way that

- The total number of vertices in $G(\Pi)$ is $1/\varepsilon$.
- The total weights on the edges is minimized.
- No vertex has weight more than εn.

Step 3.3: Construct the final partitions of G corresponding to the vertices of $G(\Pi)$.

- A partition is constructed from the subgraph represented by a vertex in $G(\Pi)$.
- The separator consists of all the boundary vertices of the partitions.

3.6 Implementation of the Vertex-Separator Algorithm

In this thesis, the algorithm described in Section 3.5 was implemented. It is a difficult algorithm that poses problems both in comprehension and implementation. In addition, the implementation of graph separators is discussed only briefly in the literature. Most of the publications about graph separators discuss its theoretical aspects. One of the contributions of this thesis is the implementation of this vertex-separator algorithm because this is the first implementation that the author of this thesis is aware of. The design and implementation of the algorithm were done in a three and a half months period (full time) and the implementation consists of approximately 1800 lines of code.

Some basic goals were set before the implementation of the vertex-separator algorithm. These goals include the following.

- The first goal was to make it easy to use the algorithm in many different applications.
- The second goal was to make the implementation easy to understand and modify.
- The third goal was not to put any requirements on the application that will use the vertex-separator algorithm.

The following decisions were made in order to achieve the goals specified.

- In order to achieve the first goal, the interface of the algorithm had to be as simple as possible. In other words, the algorithm does not require many interactions or setups from the application side; on the contrary, it needs only a simple input from the application.
- The second goal was accomplished by dividing the algorithm into a set of steps. Those steps were implemented in modular functions, i.e. the interaction between the functions is only through their interface and there are no global flags or variables that affect the behavior of a function.
- In order to accomplish the third goal, no special libraries were used in the implementation of the algorithm. For instance, LEDA was not used in the algorithm implementation because if it were, any application that uses the vertex-separator algorithm has to have access to LEDA and use it.

The input to the algorithm is a graph and the number of required partitions. It was decided not to take ε as input and take the number of required partitions instead in order to make it easy to use the algorithm in parallel applications. If ε was used as input, the user of the algorithm will not have total control on how many graph partitions will the algorithm's output have. On the other hand, by letting the algorithm's user define the number of graph partitions that the algorithm will compute, the output partitions can be directly used in parallel applications where every processor will be assigned a graph partition and no further load balancing will be needed.

Because the choice of the root of the breadth-first spanning tree built by the algorithm affects the final output of the algorithm, the algorithm's user was given an option to decide on the root. The root vertex can be supplied as input to the algorithm if the user wishes to control it; otherwise, a default root vertex will be chosen by the algorithm.

The input graph consists of three sets: vertices, edges, and graph faces. Every element in these sets has an identification number. It was decided that the output partitions must keep these identification numbers the same. Accordingly, any information that an application has that relates to the graph's vertices, edges or faces will remain valid for the graph partitions produced by the vertex-separator algorithm.

3.6.1 Part I of the Vertex-Separator Algorithm

The first function in the vertex-separator algorithm is *ConstructAdjacencyList*. This function computes for every vertex in the input graph a list of its adjacent vertices. After this adjacency information is computed, the second function *ConstructBFS* is used to build a breadth-first spanning tree of the input graph rooted at either the vertex specified by the algorithm's user or the default root vertex. The breadth-first spanning tree built by this function must have enough information that if some consecutive tree levels were chosen the subgraph represented by these levels can be constructed. In other words, the information from the chosen breadth-first spanning tree levels must be enough to relate it to the input graph information and produce a subgraph that consists of a set of vertices, tree-edges, non-tree edges, and graph faces, all having the same identification number as the corresponding ones in the input graph.

After the breadth-first spanning tree is built, a set of levels is chosen from this tree by the function *ChooseLevels*. The function $h(\varepsilon, n)$ used to select levels in the spanning tree was slightly modified from its description in Section 3.5 to $h(\nu, f) = 1/2\sqrt{f/\nu}$ where f is the total number of faces in the input graph and ν is the required number of partitions from the algorithm's input. This change was made

because the number of faces is a more representative parameter for the size of the partitions. However, this did not have any significant effect on the output. This was experimentally observed by comparing the graph partitions obtained using the number of vertices with those obtained using the number of faces.

The cycles formed by the selected breadth-first spanning tree levels is then computed. This was one of the most difficult parts in the implementation of the vertex-separator algorithm. Some of the reasons for this difficulty include: one tree level can form more than one cycle, not all vertices on one tree level participate in the cycles formed by this tree level, some vertices on the tree level are included in more than one cycle formed by that level, not all edges of the tree level participate in the level's cycles. every cycle has to be computed in order, i.e the order of visiting the vertices and edges of the cycle either in a clockwise or counterclockwise direction. In order to implement this step in a modular approach, it was divided into substeps and one function was implemented for each substep. Only one function interacts with the rest of the algorithm and internally it calls the functions for the substeps.

After the cycles were computed, the subgraphs described in *Step 1.7* and *1.8* of the vertex-separator algorithm in Section 3.5 are computed. This task was also subdivided into smaller steps and implemented in a modular approach. A set of functions were implemented for the different substeps and one function interacts between these functions and the rest of the algorithm. The implementation of this task was difficult because of the definition of the subgraphs that are to be computed. For instance, the inclusion of three vertices in a subgraph does not necessarily mean that the face they construct will be included in the subgraph. The output of this step is a set of subgraphs consisting of a set of vertices, edges and faces, all having the same identification number as the corresponding ones in the input graph. Moreover, every subgraph is connected.

3.6.2 Part II of the Vertex-Separator Algorithm

In Part II, the separation graph was built for components that had a total number of faces greater than f/ν, where f is the total number of faces in the input graph

and ν the required number of graph partitions. The function *BuildSeparationGraph* receives as input one component (obtained from Part I) and constructs its separation graph. If a component has holes inside, a star-shaped triangulation is computed for the vertices of the border of each hole before the separation graph is computed. This triangulation is performed by computing the angular order of the vertices around a hole, choosing one vertex from these vertices and connecting it to the rest of the vertices in such a way that every face is a triangle. This triangulation is computed in order to get connected subgraphs while traversing the graph using the method described next.

The function *TraverseSeparationGraph* traverses the constructed separation graph starting from the vertices of degree one upwards. In other words, all vertices of degree one are considered as 'leaves' and the graph is traversed upwards from all leaves one level at a time. While traversing the graph, the weights inside the T-cycles formed by the various parts of the separation graph traversed so far are computed. An inside part of a T-cycle is cut from the separation graph exactly before its weight exceeds f/ν. The output of this phase is a set of subgraphs where every subgraph consists of exactly one connected component. Moreover, the subgraphs' vertices, edges and faces have the same identification number as their corresponding ones in the input graph.

3.6.3 Part III of the Vertex-Separator Algorithm

Part III consists of merging the graph partitions obtained so far to get exactly the number of required partitions. Two conditions have to be satisfied by the merging. The first one is that every partition must have at most $f/\nu+c$ number of faces where c is a small constant and the second condition is that the size of the separator is minimized. Since trying all possible combinations for merging the graph partitions leads to an exponential algorithm, the merging was implemented using a method to get a good merging but not necessary the best merging.

First the function *GetAdjacentPartitions* computes for every graph partition a list of graph partitions that share a boundary with it, i.e. are adjacent to it. After

that the function *MergePartitions* use these lists to merge adjacent partitions. The function starts with the smallest partition and merges it with one of its adjacent partitions in such a way as to satisfy the two required conditions. This step is repeated until only the required number of partitions is left. Using this merging strategy, the ratio between the size of the biggest and smallest output partition was suggested by Aleksandrov (1998) to be $\nu' - 1/\varepsilon$ in the worst case, where ν' is the number of partitions obtained from Part II.

The output of this phase is the graph partitions where every partition is connected. The vertices, edges and faces of every partition have the same identification number as the input graph and are associated with the same information of their corresponding ones in the input graph. The partitions border edges are marked because their vertices represent the separator and they represent the part where an application will merge its computation results obtained from each graph partition if necessary. In addition, some adjacency information is also included in the output, for instance, every edge knows its left and right face. Every output partition has the same representation as the input graph. Accordingly, any algorithm that was developed for the original graph can also be used for the output partitions.

List of Difficulties

The following are some of the difficulties faced during the implementation of the vertex-separator algorithm.

- Constructing a subgraph from a set of consecutive levels of the breadth-first spanning tree. The elements of the subgraph must have the same identification numbers as in the input graph.
- Computing cycles formed by the breadth-first spanning tree levels.
- Computing the connected components obtained after computing the cycles.
- Computing weights inside T-cycles.
- Constructing a subgraph from the separation graph.

3.7 Applying the Vertex-Separator Algorithm to a TIN

In this thesis, the algorithm described in Section 3.6 was tested on triangulated irregular networks (TINs). Every vertex in a TIN was assigned a weight value equal to $1/n$ where n is the total number of vertices. Different roots were chosen for the breadth-first spanning tree.

Since the choice of the root of the breadth-first spanning tree affects the structure of the tree itself, the partitions obtained from Part I were dependent on the choice of the breadth-first spanning tree root. The output of that phase was a number of subgraphs slightly greater or equal the required number of partitions. Figure 3.5 shows an example of the output of Phase I. The rectangle represents the entire TIN and the root of the breadth-first spanning tree is marked with a black circle. In this example, the required number of partitions entered was four. Four levels in the breadth-first spanning tree were selected and their corresponding cycles computed. Those cycles provide an initial partitioning for the TIN by dividing the TIN into a set of connected components. Eight connected components were computed after computing the selected breath-first spanning tree levels' cycles.

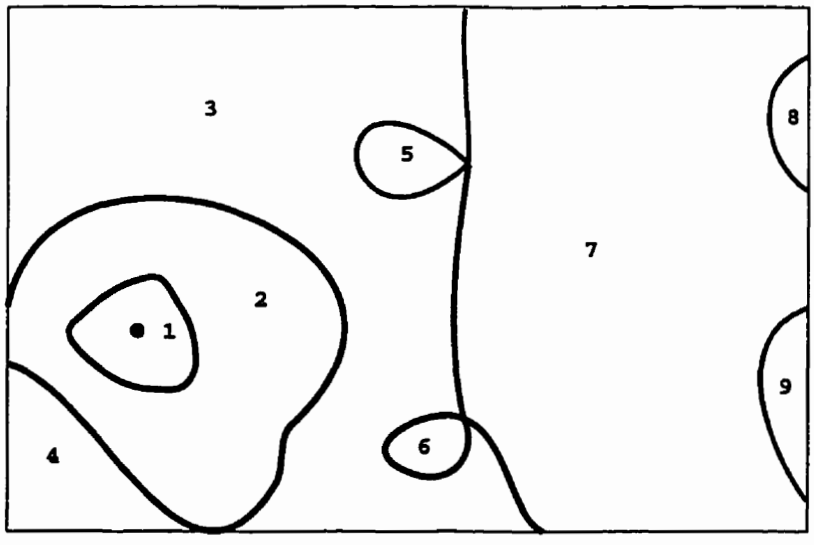


Figure 3.5: Output of Phase I of the vertex-separator algorithm

In Part II, the separation graph was built for components that had total number of faces greater than f/ν where f is the total number of faces in the TIN and ν the required number of partitions. The main purpose for applying Part II is to remove large partitions obtained from Part I by repartitioning them. After applying Part I and Part II, all connected components will have number of faces $\leq f/\nu$. Moreover, the approach of using the cycles formed by breadth-first spanning tree levels in combination with separation graphs reduces the size of the separator. Figure 3.6 shows an example of the output of Part II. The two large connected components were further partitioned into two and three connected components, respectively.

Finally, the purpose of applying Part III is to: (a) combine partitions in order to get exactly the required number of partitions ν and (b) minimize the separator size. After applying Part III, the algorithm will output exactly ν partitions where every partition is connected and has number of faces $\leq f/\nu + c$, where c is a small constant.

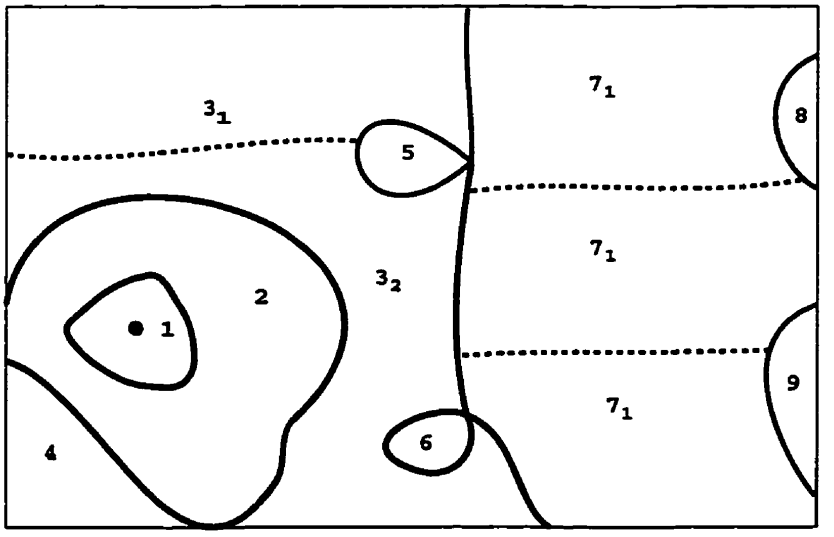


Figure 3.6: Output of Phase II of the vertex-separator algorithm

3.8 Results

In this section, some examples of applying the vertex-separator algorithm to TINs are presented. Figures 3.7-3.10 show how the output partitions are balanced in the

number of faces in every partition.

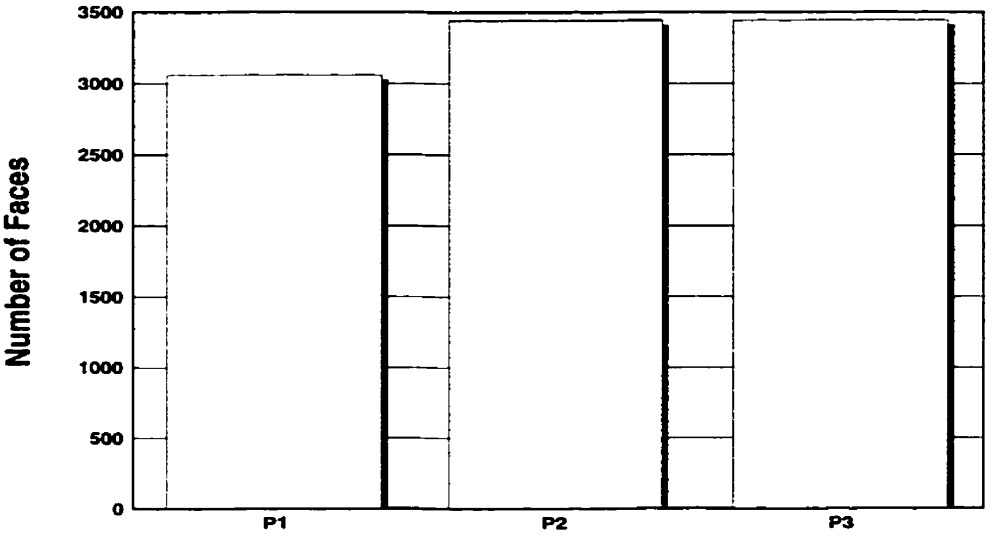


Figure 3.7: Partitioning a TIN with 9944 faces and 5000 vertices into three partitions. The separator has 75 vertices.

Figure 3.11 presents some examples of partitioning a TIN with 50244 faces and 25440 vertices into four partitions. In every example, a different root vertex for the breadth-first spanning tree was selected in order to show how the choice of the breadth-first spanning tree root affects the final output of the algorithm. The partitioning obtained is balanced in the number of faces in every partition. In the example that used vertex R 5120 as a root for the breadth-first spanning tree, the smallest partition obtained is 24.93%, the largest is 25.08%, and the median is 25.0% of the total number of faces in the TIN. In the example that used vertex R 54640 as a root for the breadth-first spanning tree which is the largest interval in Figure 3.11, the smallest partition obtained is 23.54%, the largest is 26.588%, and the median is 24.93% of the total number of faces in the TIN.

Figure 3.12 presents some examples of partitioning a TIN with 50244 faces and 25440 vertices into five partitions. Like the previous set of examples, a different root vertex for the breadth-first spanning tree was selected in every example in order to show how the choice of the breadth-first spanning tree root affects the final output of the algorithm. The partitioning obtained was also balanced in the number of

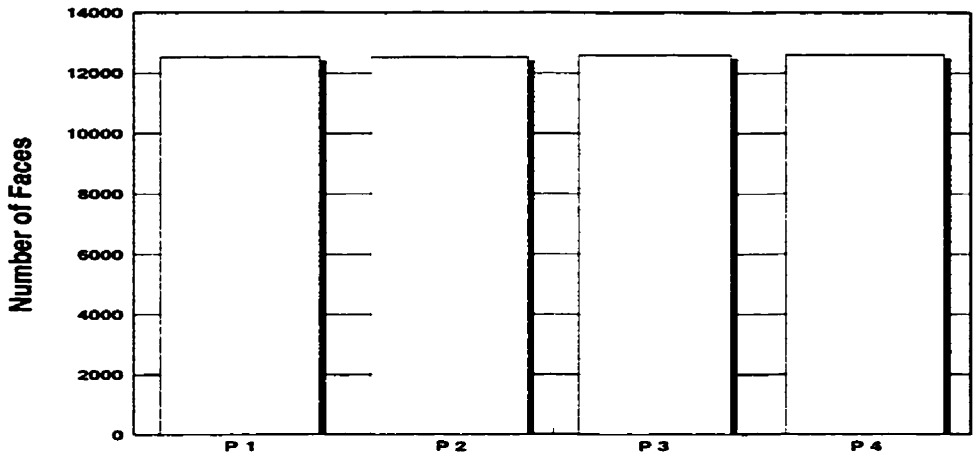


Figure 3.8: Partitioning a TIN with 50244 faces and 25440 vertices into four partitions. The separator has 433 vertices.

faces in every partition. In the example that used vertex R 7343 as a root for the breadth-first spanning tree, the smallest partition obtained is 18%, the largest is 21.67%, and the median is 20.6% of the total number of faces in the TIN. In the example that used vertex R 5900 as a root for the breadth-first spanning tree which is the largest interval in Figure 3.12, the smallest partition obtained is 14.6%, the largest is 24.08%, and the median is 20.27% of the total number of faces in the TIN. The median value is very close to 20.0% which is the optimal value.

Figure 3.13 presents some examples of partitioning a TIN with 50244 faces and 25440 vertices into six partitions. A different root vertex for the breadth-first spanning tree was also selected in every example. The partitioning obtained was also balanced in the number of faces in every partition. In the example that used vertex R 438 as a root for the breadth-first spanning tree, the smallest partition obtained is 14.21%, the largest is 19.50%, and the median is 16.9% of the total number of faces in the TIN. In the example that used vertex R 400 as a root for the breadth-first spanning tree which is the largest interval in Figure 3.13, the smallest partition obtained is 10.76%, the largest is 20%, and the median is 17.8% of the total number of faces in the TIN. The median value is very close to 16.67% which is the optimal value. Figure 3.14 and 3.15 presents sample outputs of the vertex-separator algorithm, where the separator is the darker line.

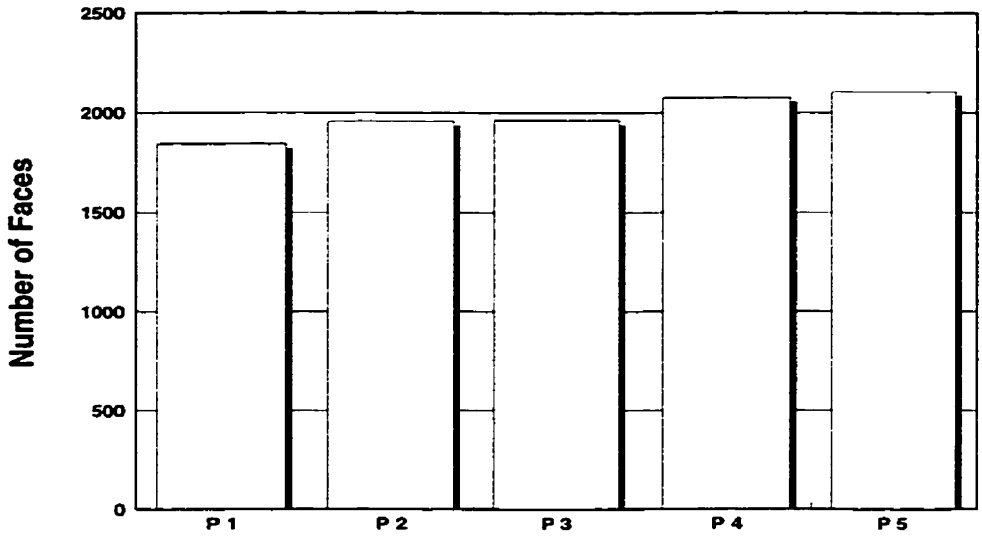


Figure 3.9: Partitioning a TIN with 9944 faces and 5000 vertices into five partitions. The separator has 149 vertices.

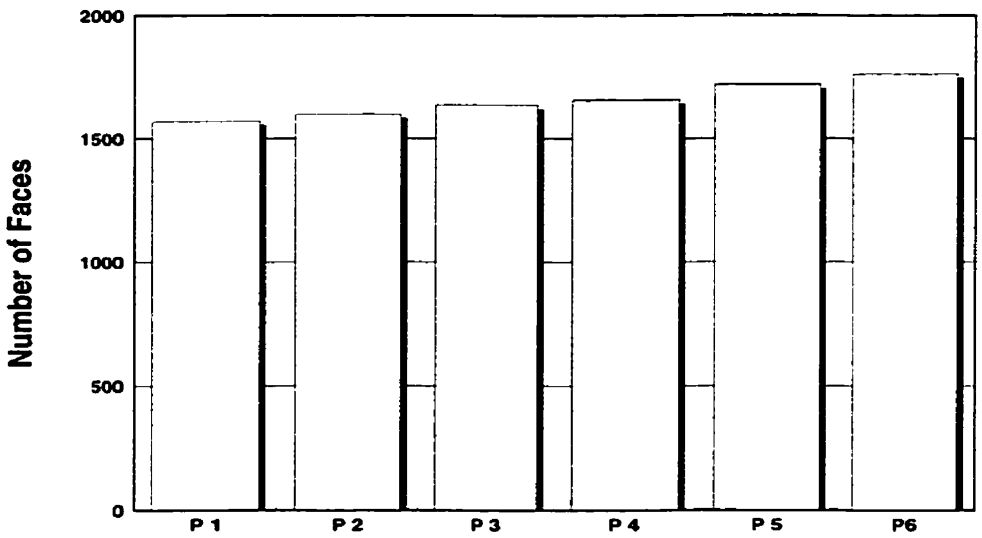


Figure 3.10: Partitioning a TIN with 9944 faces and 5000 vertices into six partitions. The separator has 153 vertices.

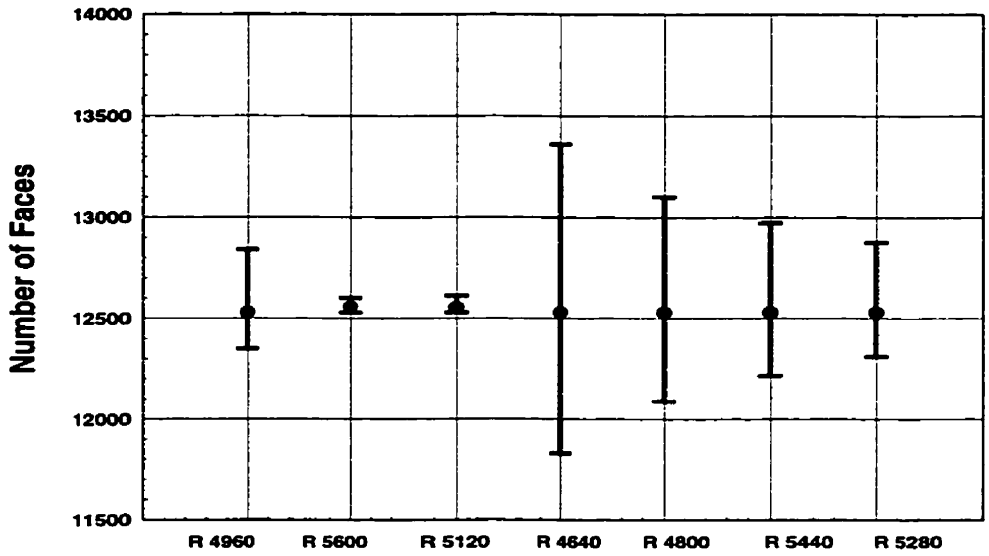


Figure 3.11: Partitioning a TIN with 50244 faces into four partitions. Showing the minimum, median, and maximum number of faces in the output partitions

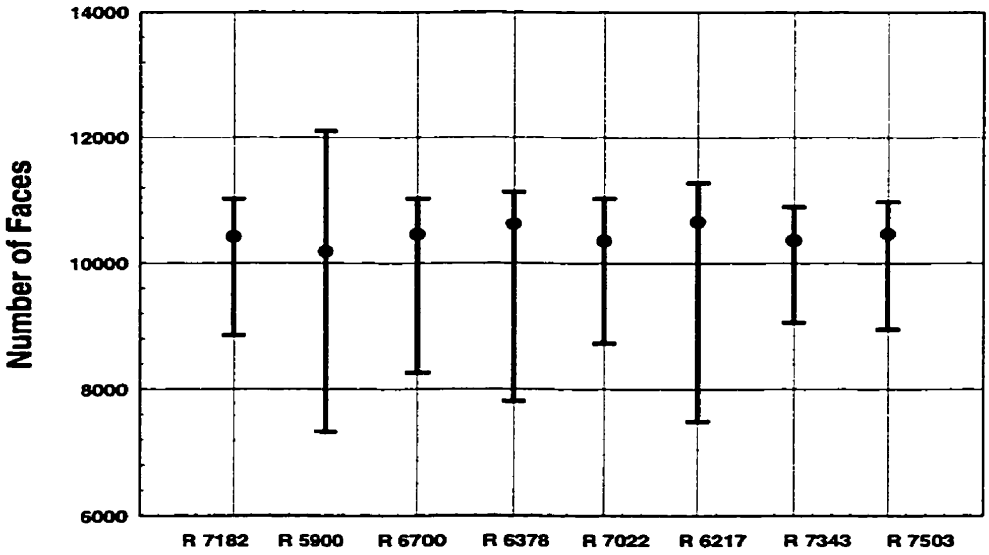


Figure 3.12: Partitioning a TIN with 50244 faces into five partitions. Showing the minimum, median, and maximum number of faces in the output partitions

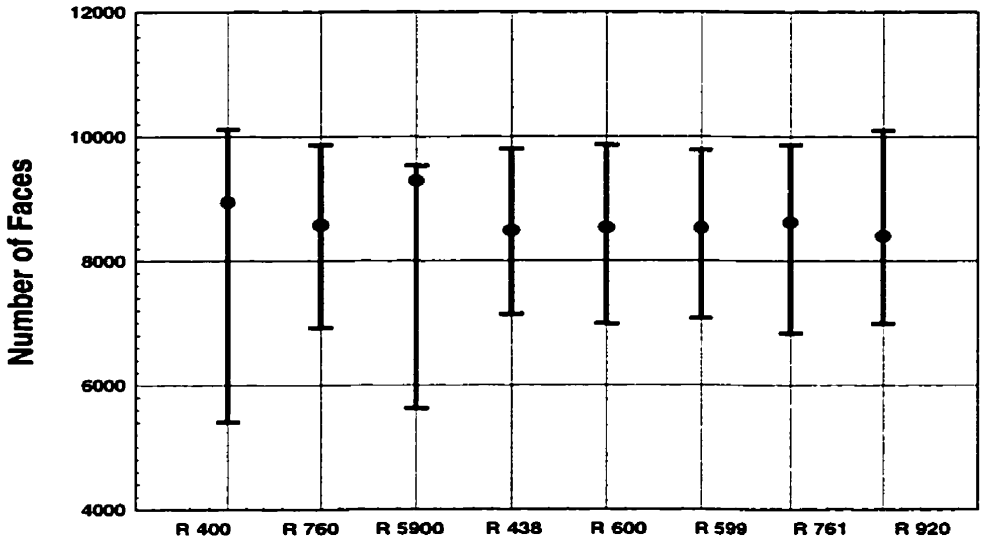


Figure 3.13: Partitioning a TIN with 50244 faces into six partitions. Showing the minimum, median, and maximum number of faces in the output partitions

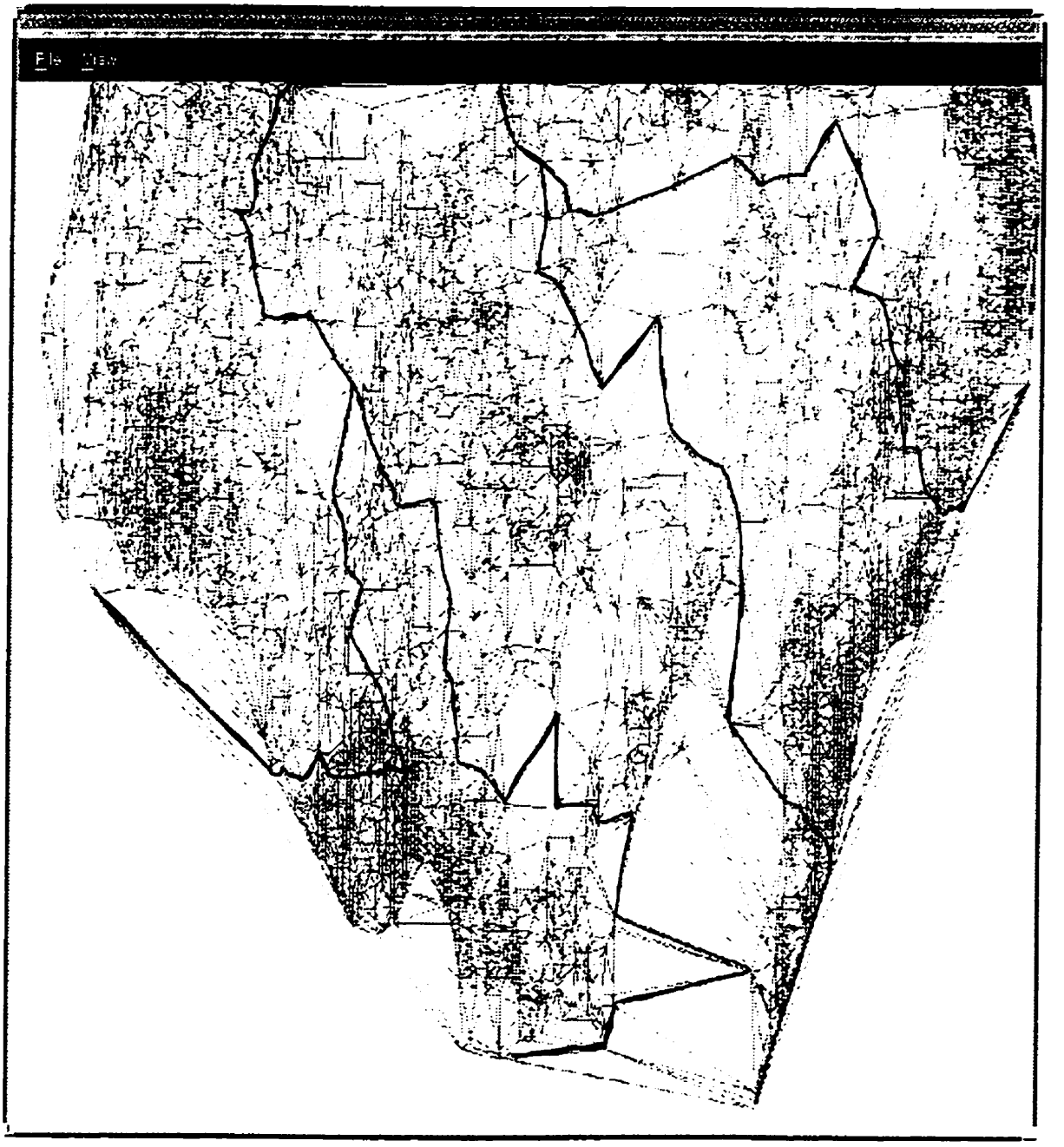


Figure 3.14: A sample output for partitioning a TIN into five partitions

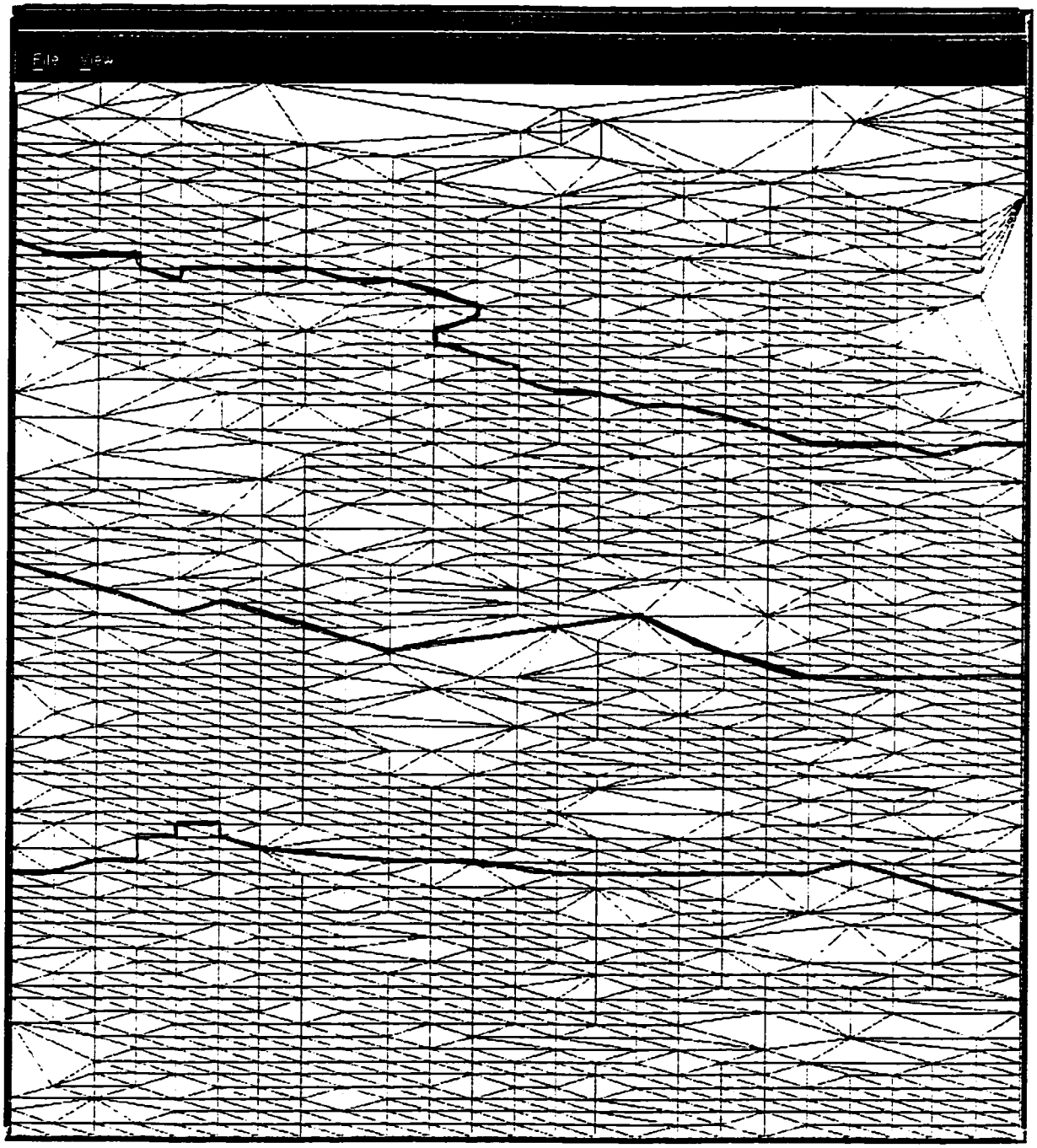


Figure 3.15: A sample output for partitioning a TIN into four partitions

Chapter 4

Terrain Visibility

4.1 Introduction to Terrain Visibility

The computation of visibility information on terrains is needed in many GIS applications. Some of these applications are related to the computation of a set of optimal observation points, terrain navigation, terrain exploration, determining the location of forest fire observation towers to provide maximum coverage of the terrain, determining optimal location of transmitters and receivers for line-of-sight communication, extraction of significant topographic features, search and rescue applications, and many others (Cazzanti et al., 1991), (Cole and Sharir, 1989), (De Floriani et al., 1994a), (Goodchild and Lee, 1989), (Lee, 1991), (Lee, 1992), (Teng et al., 1993). Visibility information also has a variety of other non-GIS related applications. For example, visibility information is needed in hidden surface removal in 3D computer graphics applications.

There are several versions of visibility problems. For instance, there are different 'viewpoint' types and these include points (fixed or moving), line segments, and regions. The objects for which the visibility information is computed include points, line segments, regions, polygons, and polyhedra. In addition, some restrictions can be added to visibility problems, for instance angular restrictions and distance restrictions.

In this thesis, visibility information is not computed in the display aspect which can be done in linear time using a numbering scheme, (De Berg, 1993). The focus in

this thesis is on computing visibility information in object-space format, i.e. computing what portions of the objects are visible. This allows other applications to do further analysis on the computed visibility information.

In this thesis, the problem of computing visibility information on models of natural terrains is investigated. More specifically, the problem of computing *visibility on a terrain* is addressed. In this case, the viewpoint or the observation point is located on or above the terrain surface. In the literature, terrain visibility algorithms usually operate on digital terrain models (DTMs) or on polyhedral models.

4.2 Preliminaries

In this section, some definitions will be introduced that are going to be used in proceeding sections.

4.2.1 Candidate and Observation Points and Visual Rays

A *polyhedral terrain* σ is the graph of a polyhedral function $z = F(x, y)$ defined over the entire xy-plane. A point $a = (x_o, y_o, z_o)$ is above σ if $z_o > F(x_o, y_o)$. A *candidate point* is any point located on or above a polyhedral terrain σ. An *observation point* or *viewpoint* is also located on or above σ. Two candidate points are considered mutually visible if the straight line segment joining them lies above the terrain. In other words, the straight line segment does not intersect the terrain (except where it touches the two candidate points), see Figure 4.1 for an illustration. A *visual ray* is any ray originating from a viewpoint, (Cole and Sharir, 1989), (De Floriani and Magillo, 1994).

4.2.2 The Infront/Behind Relationship

The *infront/behind* relationship is used often in visibility computation, especially in determining the depth order of objects with respect to a viewpoint. It is determined in the following way: Given a plane subdivision Σ, an observation point V, an edge e_1, and an edge e_2, e_1 of Σ is infront of e_2 (and e_2 behind e_1) with respect to V

(a) P is visible from V (b) P is invisible from V

Figure 4.1: Visibility between two candidate points

if and only if there exists a ray r originating from V and intersecting both e_1 and e_2 and the intersection of r and e_1 lies nearer to V than the intersection of r and e_2, see Figure 4.2 for an illustration. The infront/behind relationship is defined in the same way for regions of Σ. A subdivision plane Σ is called *acyclic* if it has the infront/behind relationship as a partial order relationship. In other words, it is acyclic if a depth order can be computed for all its faces with respect to a viewpoint. In this depth order the faces are listed according to their distance from the viewpoint using the infront/behind relationship between every pair of faces, (Foley et al., 1990) (De Floriani and Magillo, 1994).

4.2.3 The Face Up/Down Relationship And Blocking Edges

If a face f_i of a TIN has its external side oriented towards a viewpoint V, it is considered *face up* with respect to V; otherwise, it is considered *face down*. If f_1 is the closest face incident to an edge e and f_2 the furthest, then e is called a blocking edge, (De Floriani et al., 1994b). These relationships are illustrated in Figure 4.3.

4.3 Classification of Terrain Visibility

Visibility computations on terrains are classified into point, line and region visibility. Throughout the literature, several versions of visibility problems are addressed, for

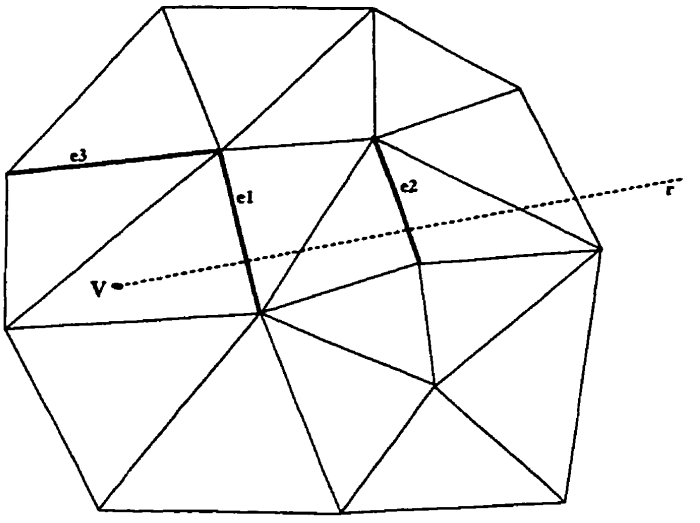


Figure 4.2: e1 is infront of e2 with respect to V and e3 is not related to e1 nor e2

example, in static and dynamic settings, with fixed and moving observation points, on hierarchical terrain models or 3-d scenes. The following subsections give some explanation for these classifications.

4.3.1 Point Visibility

Point visibility problems consist of computing the intervisibility between pairs of points. In general, it is required to compute a subset of a given candidate point set that is visible from one observation point. Given a terrain M, an observation point V, and a set of candidate points L, it is required to compute a subset, L', that is visible from V. The subset L' is the *discrete visible region of V in M*. A more generalized form of the problem arises when instead of having only one observation point, a set of observation points L_1 is given. In this case, for every point in L_1, a subset of L is computed that is visible from that point. This type of problem is called computing the *discrete visibility model* of a terrain, (De Floriani and Magillo, 1994).

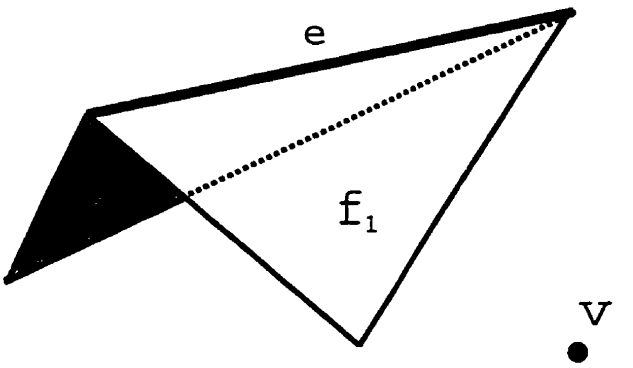


Figure 4.3: e is a blocking edge with respect to V, triangle f_1 is face-up, and triangle f_2 is face-down, (De Floriani et al., 1994b)

4.3.2 Algorithms For Point Visibility

Computing point visibility can be reduced to computing mutual visibility between pairs of points. The rest of this section will present some algorithms for computing point visibility.

A Brute-Force Algorithm For Point Visibility

The following is a representation for an algorithm presented by De Floriani and Magillo (1994) that computes the discrete visibility region of a viewpoint V with respect to a set of candidate points L.

Input:

- A terrain
- An observation point V
- A set of candidate points L

Step 1. Compute the projection of the terrain on the x-y plane.

Step 2. For every point P in L do the following

- Compute the straight line segment joining V and P. Call it s.
- Compute the projection of s on the x-y plane. Call it s'.

- Compute the intersections between s' and the projected terrain edges.
- At each intersection point, compute whether s lies above or below the terrain edge.
- If all intersection points lie above the terrain, then P is visible from V; otherwise, P is invisible from V.

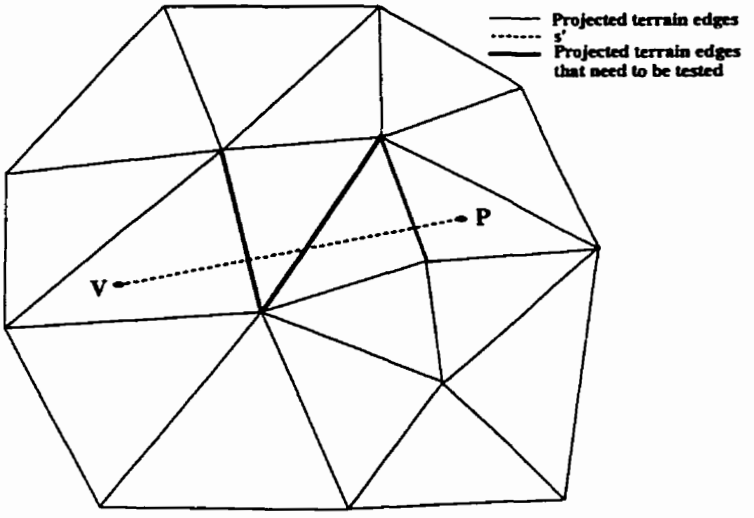


Figure 4.4: Determining the intervisibility between V and P

This process of determining whether a candidate point is visible from the viewpoint has a linear time complexity in the number of terrain edges, $O(n)$. In the case where the given set of candidate points L has k points, applying the brute-force algorithm will require $O(nk)$ time. If the terrain was a regular square grid, the time complexity of the algorithm reduces to $O(\sqrt{n})$, (De Floriani and Magillo, 1994).

A Polylogarithmic Time Algorithm For Point Visibility

A polylogarithmic time algorithm for computing point visibility is based on the solution for solving ray shooting problems. In ray shooting problems, a polyhedral terrain, an observation point V, and a view direction (θ, α) (see Figure 4.7 for an illustration on spherical coordinates) are given and it is required to compute the

first face of the terrain hit by the visual ray r origination from V in the direction (θ, α).

Cole and Sharir (1986) presented a data structure to be used in ray shooting problems, namely the horizon tree (see Section 4.3.3 for a definition of a horizon line). The horizon tree with respect to a viewpoint V can be built for any acyclic polyhedral terrain. It is a balanced binary tree and its depth is logarithmic in the number of terrain edges. The root of the horizon tree represents the entire horizon with respect to V. The left subtree represents a partial horizon constructed from half the terrain edges that are closer to V, while the right subtree represents the other partial horizon constructed from the second half of the terrain edges. The size of the horizon tree is $O(n\alpha(n)\log n)$, where $\alpha(n)$ is the inverse of Ackermann's function, and it can be constructed in optimal $O(n\alpha(n)\log n)$ time. Ray shooting queries can be answered in $O(\log^2 n)$ using the horizon tree, (Cole and Sharir, 1986).

In order to use the above data structure to solve point visibility, the terrain has to be represented as a polyhedral terrain model. In this case, mutual visibility between the observation point V and a candidate point P will be computed in the following way, (De Floriani and Magillo, 1994).

- Construct the segment $\overline{VP}$.
- Determine the direction of $\overline{VP}$ with respect to V, call it β.
- Answer the ray shooting query for the direction β. Call the first face hit by the ray f.
- If V and P lie on the same side of the plane with respect to f, then P is visible from V; otherwise, P is not visible from V.

4.3.3 Line Visibility

A line visibility problem of practical importance in GIS is computing the *horizon line* of a viewpoint on a terrain. Horizon line computation provides information about the maximum terrain height that can be seen from a given viewpoint on a terrain.

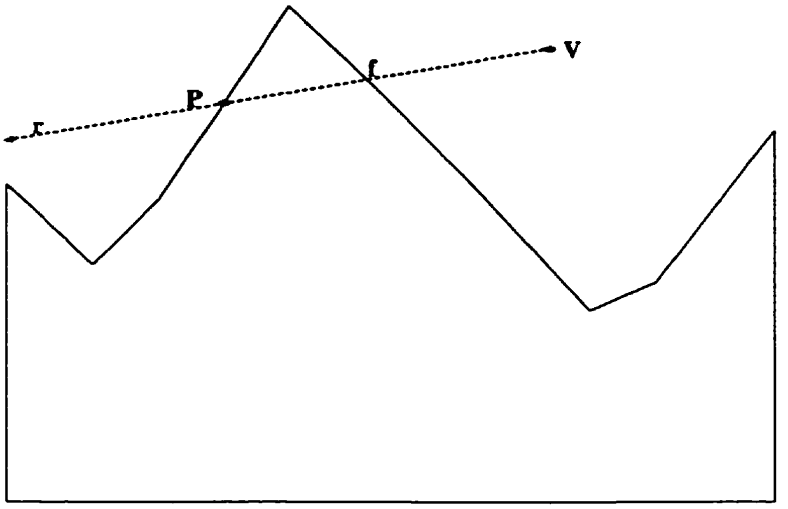


Figure 4.5: Determining the intervisibility between V and P. f is the first face hit by r. Since V and P lie on opposite sides with respect to f, then P is invisible from V, (De Floriani and Magillo, 1994)

A formal definition for the horizon line was given by De Floriani and Magillo (1994) as:

> *Given a terrain and a point of view V, the horizon of the terrain with respect to V is a function $\alpha = h(\theta)$, defined for $\theta \in [0, 2\pi]$, such that, for every radial direction θ, $h(\theta)$ is the maximum value α such that each ray emanating from V with direction (θ, β), with $\beta < \alpha$, intersects the terrain.*

The horizon line of the terrain provides information that can be used in ray shooting problems. It provides, for every radial direction, the minimum elevation that a visual ray must have to pass above the terrain.

4.3.4 Algorithms For Line Visibility

The problem of computing the horizon can be reduced to computing the *upper envelope* of a set of possibly intersecting segments in the plane. A formal definition

for the upper envelope for a set of intersecting segments was given by De Floriani and Magillo (1994).

> Given p segments in the plane, i.e. p linear functions $y = f_i(x), i = 1 \ldots p$, each defined on an interval $[a_i, b_i]$, the upper envelope of such segments is a function $y = F(x)$, defined over the union of the intervals $[a_i, b_i]$, and such that $F(x) = max_{i|x \in [a_i, b_i]}(f_i(x))$.

In other words, the upper envelope determines for every x value, the maximum y value in the given set of segments. The complexity of an upper envelope is the number of distinct segment pieces that form the upper envelope. The upper envelope of a set of nonintersecting segments can be constructed in optimal $O(n \log n)$ time. If the segments' endpoints are sorted from left to right, the upper envelope can be computed in $O(n)$ time, (Asano et al., 1986). On the other hand, if the set of segments intersect, the worst-case complexity of the upper envelope will be $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function. (Hart and Sharir, 1986), (Wiernik and Sharir, 1988). Several divide-and-conquer algorithm were established that compute the upper envelope in $O(n\alpha(n) \log n)$ time, for instance the algorithms by Atallah (1983) and Hart and Sharir (1986). Figure 4.6 shows an example of an upper envelope.

Before computing the upper envelope for the terrain edges, they will be represented using the spherical coordinate system centered at the viewpoint. Figure (4.7) shows a ray r and how its spherical coordinates, (θ, α), are computed.

4.3.5 Divide-And-Conquer Algorithms For Upper Envelope Computation

Atallah (1983) used a divide-and-conquer based approach to solve the problem of computing the upper envelope of a set of possibly intersecting segments in a plane. In his algorithm, the set of segments is recursively split into halves, and then the results are merged pairwise. The merging phase is done using a sweep-line technique for intersecting two monotonic chains of segments; it has a time complexity linear

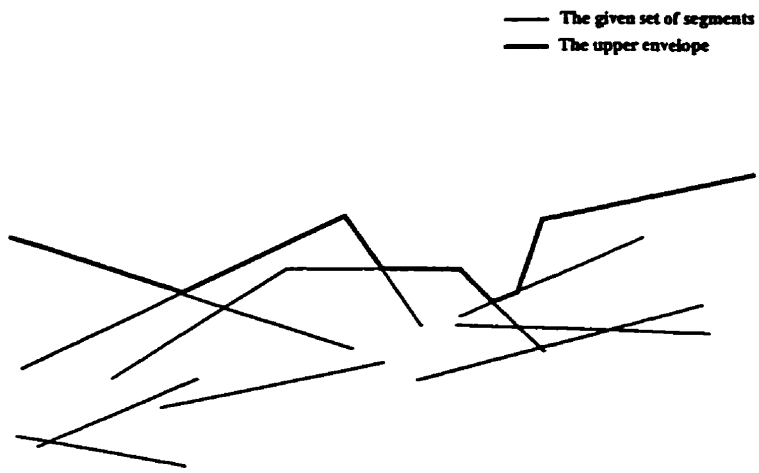


Figure 4.6: The upper envelope of a set of intersecting segments

in the size of the two upper envelopes being merged. Therefore, the worst case time complexity for using this approach is $O(p\alpha(p)\log p)$ where p is the number of segments and $\alpha(p)$ is the inverse of Ackermann's function.

The sweep-line algorithm for the merging phase takes as input two upper envelopes. The sweep line, r, is a vertical line that moves from left to right having the merged part of the envelope on its left and the part not yet merged on its right. The event points are the endpoints of the upper envelopes' segments and the intersection points between the two upper envelopes. The two segments, one from each upper envelope, currently intersecting the sweep line represent the current status of the sweep line. These two segments are ordered according to their heights. At the beginning of the algorithm, all left and right endpoint events are queued, while the intersection event points are computed during the merging. In a left endpoint event, the event's segment is inserted to the current sweep-line status in the correct order and it is tested whether it intersects any other segment. In a right endpoint event, the segment is tested whether it is the upper segment in the current sweep line status. If it is the upper segment, the upper envelope between the last event point and the current event point is computed and inserted in the merged upper envelope. The segment is then deleted from the current sweep-line status. Finally,

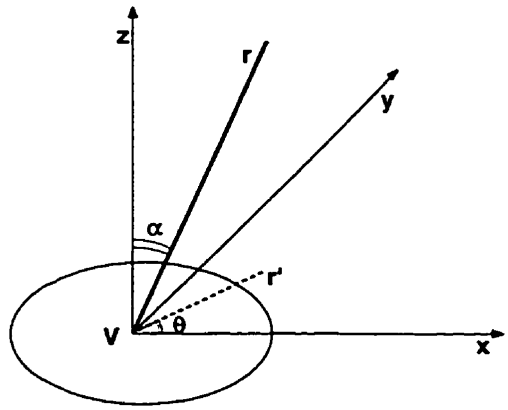
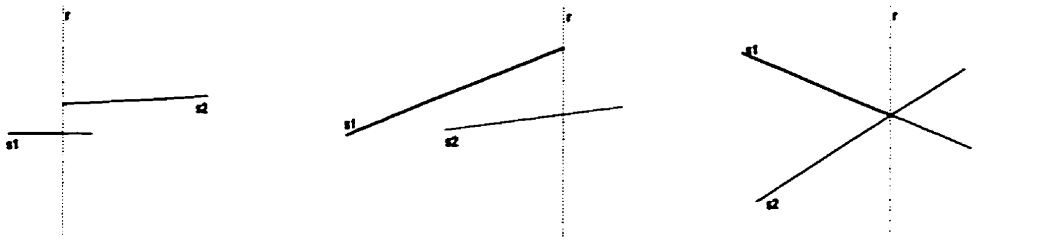


Figure 4.7: Spherical coordinate system, (De Floriani and Magillo, 1994)

in an intersection point event, the order of the two intersecting segments is swapped and the upper envelope between the last event and the current one is inserted in the constructed upper envelope.

Figure 4.8 shows examples of event points. In Figure 4.8(a), at the left endpoint event of *s2*, the upper envelope between this point and the last point in the upper envelope computed so far is computed and *s2* is added in the correct order in the sweep line status. Figure 4.8(b) shows an example of a right endpoint event, namely the right endpoint event of *s1*. At this event, the upper envelope between this point and the last point in the upper envelope computed so far is computed. The segment *s1* is then deleted from the sweep line status and *s2* is replaced in the correct order in the sweep line status. Figure 4.8(c) shows an example of an intersection event point. At this event, the upper envelope between the point where *s1* and *s2* intersect and the last point in the upper envelope computed so far is computed. The segments order in the sweep line is then swapped.

Hershberger (1989) improved the time complexity of the above algorithm to $O(n \log n)$, which is optimal. The basic idea that was used to improve the time complexity is to divide the given set of segments into subsets such that the upper envelope of every subset is linear in the size of the subset. The algorithm consists



(a) Left endpoint event (b) Right endpoint event (c) Intersection event point

Figure 4.8: Examples of event points

of three phases, namely a partitioning phase and two merging phases.

In the first phase, the partitioning phase, the segments' endpoints are sorted in increasing order by their x-coordinate. This partitions the x-axis into a set of intervals. The endpoints sorting can be done in $O(n \log n)$ time, then a balanced binary tree with $O(\log n)$ levels is built. The nodes of the tree represent the $2n$ endpoints of the segments. This tree structure can also be explained as: every node represent a vertical line with an x value equal to the value of the node's endpoint. The segments are assigned to the nodes creating subsets of segments. A segment is assigned to the lowest common ancestor of the two nodes representing the endpoints of the segment. In this way, the upper envelope of every subset of size m has complexity $O(m)$ and subsets on the same tree level lie in disjoined vertical slabs. All the segments subsets on one tree level are then combined into one subset producing a total of $O(\log n)$ subsets also having the property that the upper envelope of an m-size subset has complexity $O(m)$.

In the second phase, the upper envelope of every subset is computed in $O(n)$ time where n is the size of a subset. This will take a total of $O(n \log n)$ time. Finally, the third phase consists of merging the $O(\log n)$ upper envelopes to get the final upper envelope. The merging can be done using a divide-and-conquer approach where the merge tree will have depth $O(\log \log n)$. Therefore, phase three can be computed in

$O(n\alpha(n)\log\log n)$ time.

Another algorithm for computing the upper envelope of a set of segments is the following. It starts with an empty structure and adds one segment at a time in any order. When adding a new segment, its intersections with the constructed envelope are tested and the envelope is updated. The worst case time complexity of this algorithm is $O(p^2\alpha(p))$ where p is the number of segments and $\alpha(p)$ is the inverse of Ackermann's function.

4.3.6 Region Visibility

Region visibility problems consist of computing the subset of the terrain surface which is visible (or invisible) from a predefined observation point. Given a terrain σ and an observation point V, it is required to compute the subset σ' of σ that is visible from V. The subset σ' is called the *visible region of* V and its complement in σ is the invisible region of V. If instead of only one observation point a set of observation points is given, the collection of visible regions with respect to the set of observation points is called the *continuous visibility model* of the terrain with respect to the set of observation points, (De Floriani and Magillo, 1994).

In a digital terrain model, the region visibility information with respect to a viewpoint can be represented as a map, called a *continues visibility map*. In this representation type, the terrain is partitioned into maximal connected regions, where each region is labeled either visible or invisible. The rest of this section describes some algorithms for computing region visibility on terrains.

4.3.7 Algorithms For Region Visibility

Two Sequential Algorithms

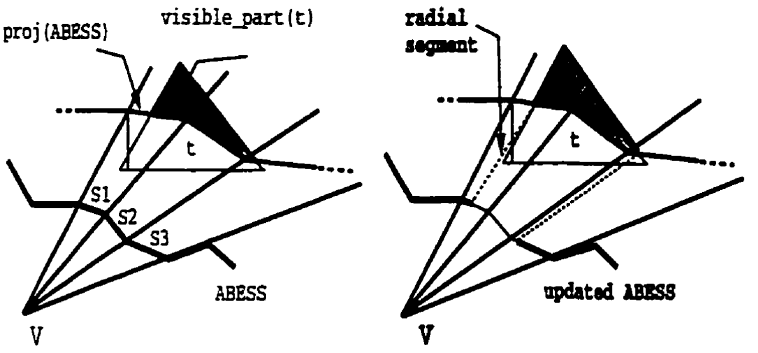
De Floriani et al. (1989) introduced an algorithm to compute region visibility on acyclic TINs. Given an acyclic TIN σ and a viewpoint V, the algorithm computes the invisible portions of the faces of σ. The algorithm consists of two phases, namely a radial sorting phase and a visibility computation phase.

In the radial sorting phase, the triangles of σ are sorted by increasing distance with respect to the projection $\bar{V}$ of the viewpoint V on the $x-y$ plane. The radial sorting is performed by building a star-shaped polygon Π around $\bar{V}$. An initial polygon is first formed by the triangles of σ incident on $\bar{V}$ and then triangles are added to the polygon one at a time. The step of adding triangles is done as follows. The algorithm chooses an edge, l, on the boundary of Π to examine the triangle t which is adjacent outwards to l. If the triangle t is adjacent to Π along two edges, called an intruding triangle, it is added to Π. If t is adjacent to Π along a single edge ,l, called a protruding triangle, it will be added to Π only if the opposite vertex of t lies in the radial sector defined by $\bar{V}$ and l. The boundaries of the polygon Π are stored in C_{Π}. The radial sorting phase takes linear time with respect to the number of triangles in σ.

In the visibility computation phase, an active circular list of blocking edges is constructed. This list is called the Active Blocking Edge Segment Sequence, ABESS, which contains the portions of blocking edges from the triangles already visited. The ABESS is initially empty. The triangles of σ are examined in the order defined by the radial sorting phase and edges are added to ABESS.

The following conditions are used to test the triangles. A triangle t is totally invisible from a viewpoint V if it is face down with respect to V. Otherwise, the segments in ABESS that project onto t given the observation point V are used to compute the visible portions of t. The visible portions of t are the parts of t that are above ABESS. After that the ABESS segments that project into t are replaced by the corresponding upper edges (or part of edges) of t. In this test, only the edges of t that are not part of ABESS are examined, called the active edges. Figure (4.9) shows an example for the visibility computation phase. This phase has worst-case time complexity $O(n^2\alpha(n))$ where $\alpha(n)$ is the inverse of Ackermann's function. In the worst-case, the size of ABESS is $O(n\alpha(n))$.

Later on, Boissant and Dobrindt (1992) presented an algorithm for computing the lower envelope of a set of disjoint triangles that can be applied on TINs to compute region visibility. The algorithm is based on an incremental randomized approach. It inserts one triangle at a time in the lower envelope constructed so



(a) Clipping the visible part of the current triangle t

(b) Updating ABESS

Figure 4.9: Visibility computation phase, (De Floriani et al., 1994b)

far. The algorithm maintains the trapezoidal decomposition of the lower envelope constructed so far as well as the history of its construction. The randomized expected time of inserting triangle i is $O(i \log i)$ and the entire algorithm has $O(n^2 \log n)$ time complexity.

A Parallel Algorithm

De Floriani et al. (1994b) introduced a parallel version of the sequential algorithm to compute region visibility presented by De Floriani et al. (1989). In the parallel algorithm, the sorting phase is unchanged, only the visibility computation phase is parallelized. In the visibility computation phase, parallelism is achieved by partitioning the input dataset in such a way that independent visibility computations can be performed. The input data set σ is partitioned into sectors with their common vertex at the viewpoint.

At initialization time, the input data sets are loaded on each processing element *pe*. Then the algorithm computes the local sectors and it marks the triangles on the border of the sector with a border triangle mark. ABESS and C_Π only connect triangles which are partially or completely in the local sector. There is only one significant modification to the sequential algorithm. The visibility of a border triangle

t_b depends on the triangles contained in both adjacent sectors. So the triangle t_b is split into two regions by the sector border plane (which is orthogonal to the x-y plane and intersects with it at the border line between the two sectors. So visibility is computed separately in the two regions.

In order to achieve load balancing, a dynamic scheduling policy has been used. This dynamic scheduling policy is implemented as follows. A number of partitions greater than the number of processing elements is statically selected and a master process allocates tasks to worker nodes. When a worker completes its visibility computation task, it sends to the master and the master assigns a new sector to the worker.

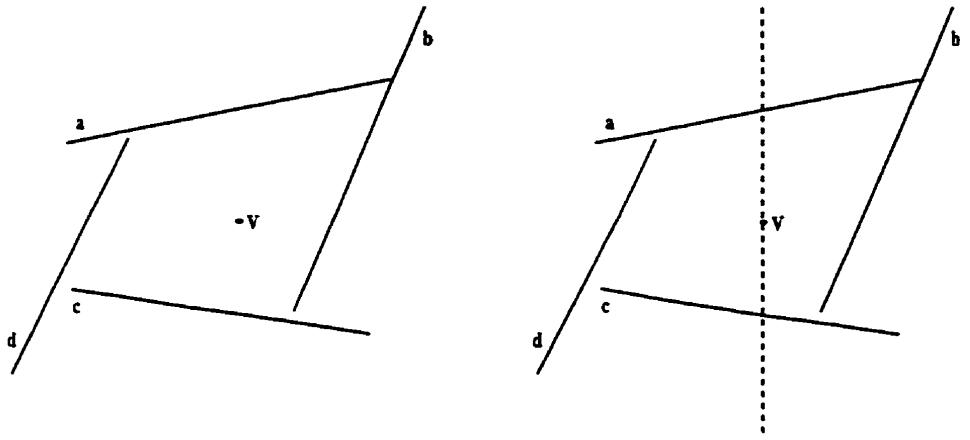
The shortcomings of this parallel algorithm include the following. Although every processor works on only one partitions of the input TIN, the data of the entire TIN must be present on all processors. The triangles on the borders are cut. The load balancing used introduces an overhead. In this thesis, some of these problems were addressed for a particular problem, called the horizon line computation.

4.4 Horizon Line Computation

In this thesis, the horizon line algorithm by Atallah (1983) described in Section 4.3.5 was implemented. The algorithm implemented here consists of three phases: constructing a left and right subproblem, computing the projection of the terrain edges on the (θ, α)-plane, and computing the upper envelope of the projected edges for each subproblem.

The first phase takes as input the terrain and the viewpoint. It then divides the terrain into two halves along the vertical line that passes through the given viewpoint. The output of this phase is two lists of terrain edges, one for the left half of the terrain and the second for the right half. The reason behind introducing this phase at the beginning of the horizon line algorithm is to allow the algorithm later to be used in other GIS applications, in particular applications that are related to computing terrain visibility with a moving viewpoint or ray shooting applications.

In computing terrain visibility with a moving viewpoint, the terrain edges are



(a) A cycle formed by the terrain edges with respect to the viewpoint V

(b) Breaking the edges' cycle by dividing the edges into two sets.

Figure 4.10: Showing a cycle formed by the terrain edges with respect to a viewpoint and the method to break it.

first sorted producing polygonal chains that are monotone with respect to lines perpendicular to f^* (where f^* is the projection on the xy-plane of the viewpoint path). After that, the horizon line will be computed at specific points along the viewpoint path called *critical points* using a different subset of polygonal chains for every critical point. The subset of polygonal chains will be chosen according to their total order obtained at the beginning of the algorithm, (Bern et al., 1994).

In ray shooting applications, a tree of horizon lines is computed where every node represent the horizon line formed by using a subset of the terrain edges. Those subsets are chosen after a total order of the terrain edges with respect to the viewpoint is computed, (Cole and Sharir, 1989).

In order to compute the total order of the terrain edges with respect to a viewpoint, there has to be no cycles formed by the terrain edges. Figure 4.10(a) shows an example of such a cycle. The edge a is infront of b, b is infront of c, c is infront of d, and d is infront of a. Therefore, in order to break any cycles in the terrain, the terrain is divided into two halves along the vertical line that passes through the

viewpoint. As a result, the viewpoint will be outside each terrain half avoiding the possibility of having cycles , (De Berg, 1993). Figure 4.10(b) shows an example of dividing the terrain and breaking a cycle formed by the terrain edges.

In the second phase of the implemented horizon line algorithm, the projection of the terrain edges on the (θ, α)-plane are computed. The input to this phase is the two lists obtained from Phase 1 and the viewpoint and the output is two lists of projected edges, one for the left half of the terrain and the second for the right half. The projection is computed by computing the spherical coordinates of the edges' endpoints where the viewpoint is the origin of the spherical coordinate system. Figure 4.7 shows how the θ and α values are computed.

The third phase, namely computing the upper envelope of the projected terrain edges, is the most complicated phase among the three phases. The difficulty in implementing this phase was to cover all conditions needed for merging two upper envelopes; in other words, covering all the needed relationships between the two upper envelopes being merged. The input to this phase in the two lists of projected edges produced by Phase 2 and the output is the horizon line for the left half of the terrain and the horizon line for the right half with respect to the given viewpoint. In this phase, LEDA was used for two main reasons. The first reason is that in this phase many functions that deal with segments' computation were needed and some of these functions were already implemented in LEDA. Therefore, it was practical to use already completed and tested functions. The second reason is that many float computations are needed in this phase and some errors arose from the accumulated approximation from float computation. Therefore, the *real* data type in LEDA was used to prevent such errors to occur.

In the following two subsections, 4.4.1-4.4.2, a description of how the horizon line algorithm was parallelized is presented. Parallelism was achieved here by partitioning the terrain data among the processors, executing the horizon line algorithm on every processor separately, and finally merging the results to get a final left and right horizon lines.

4.4.1 Parallel Horizon Line Computation Using The Interval Partitioning Scheme

In the interval partitioning scheme, all segments are on one plane and a set of intervals on the horizontal axis is computed. All segments that fall in one interval construct one partition. The intervals are computed by first selecting a sample from the segments and then splitters are chosen from the samples in such a way to balance the partitions. The splitters define the start of each interval. Segments that fall in more than one interval are cut into subsegments in such a way that every subsegment will fall in only one interval. Figure 4.11 shows an example of partitioning a set of segments using the interval partitioning scheme.

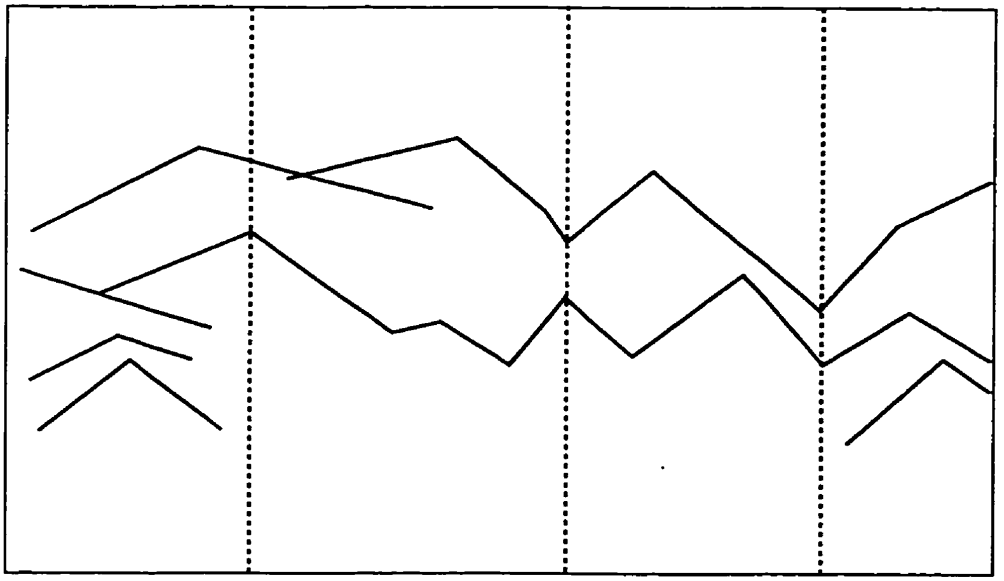


Figure 4.11: Example of the interval partitioning scheme

A parallel version of the horizon line algorithm using the interval partitioning scheme was implemented in this thesis. In this parallel version, Phase 1 (constructing a left and right subproblem) and Phase 2 (computing the projection of the terrain edges on the (θ, α)-plane) of the horizon line algorithm are first executed on one processor. After that, the interval partitioning scheme is applied to the set of left projected terrain edges and the set of right projected terrain edges separately. Finally, every processor will compute the upper envelope for the left and

right projected terrain edges assigned to it.

The reason behind partitioning the terrain edges after they were projected on the (θ, α)-plane is to omit the need for a merging phase. i.e., the horizon lines computed by every processor are part of the real horizon line and they can be just gathered and concatenated to get the entire horizon line. Another advantage of this method of parallelizing the horizon line algorithm is that the communication amount needed is very small. In fact, the only communication needed is to assign a subset of the projected terrain edges to every processor. In this parallel version of the horizon line algorithm, the partitioning of the data among the processors is dependent on the viewpoint because it is computed after the terrain edges are projected on the (θ, α)-plane. Therefore, for every viewpoint, the result of the partitioning is different. Accordingly, the partitioning cannot be considered as a preprocessing phase.

4.4.2 Parallel Horizon Line Computation Using The Vertex-Separator Algorithm

A second parallel version of the horizon line algorithm was implemented in this thesis. In this parallel version, the vertex-separator algorithm was used to partition the terrain data among the processors. It was used as a preprocessing phase because its result is independent of the choice of the viewpoint. In other words, the same partitioning result can be used in computing the horizon line for many viewpoints.

After the vertex-separator algorithm computes the terrain partitions, every processor computes Phase 1, Phase 2, and Phase 3 of the horizon line algorithm on its partition. Because the partitioning is independent of the viewpoint, the horizon lines computed by every processor are local to the processor's partition. In other words, the horizon lines on every processor are not parts from the final horizon line and a final merging phase is needed to get the horizon line of the entire terrain with respect to the viewpoint.

Several merging strategies were studied in order to select the most efficient one in this particular implementation. One of the merging strategies studied was to build a binary tree where the processors will represent the leafs of the tree. Every two

processors will do the following: One of them will send its partial horizon line to the second processor that will merge the two horizon lines. The processors that have the results of the mergings represent the next upper level in the tree. This process is repeated until the horizon line is merged completely and resides on one processor that will be represented by the root of the tree.

This merging strategy contains many communication steps. Therefore, the amount of communications required was compared to the total amount of computation done by all processors in merging the partial horizons. The result of this comparison was that the communication overhead is large compared to time saved by computing the mergings in parallel. Therefore, another merging strategy was chosen based on the fact that the size of the partial horizons is small and therefore the amount of computation needed to merge them is small. In the selected merging strategy, one processor collects the partial horizons from the rest of the processors and computes the merging recursively.

Chapter 5

Shortest Path

5.1 Introduction to Shortest Path

Shortest path computation is a fundamental and well studied problem in graph theory (Aho et al., 1974; Deo and Pang, 1984). It is also an important area that GIS need in many applications including traffic control, search and rescue, water flow analysis, navigation, routing, robotics.

The shortest path computation problem is defined for undirected and directed graphs with real-valued weights on the edges. Weights on the edges can represent distance, time, energy , etc., depending on the application that uses the shortest path problem. The weight of an edge $[v, w]$ will be denoted in this thesis by $weight(v, w)$. The definition of the shortest path from a source vertex s to a target vertex t is the path from s to t where the summation of all edges weights in this path is minimum. There is a shortest path from s to t if and only if no path from s to t contains a cycle with negative weight (a negative cycle). There are four versions of the shortest path problem presented by Tarjan (1983):

The single pair problem Compute the shortest path from a given source to a given destination.

The single source problem Compute the shortest path from a given source to v for every vertex v.

The single destination problem Compute the shortest path from v to a given destination for every vertex v.

The all pairs problem Compute the shortest path from s to t for every pair of vertices s and t.

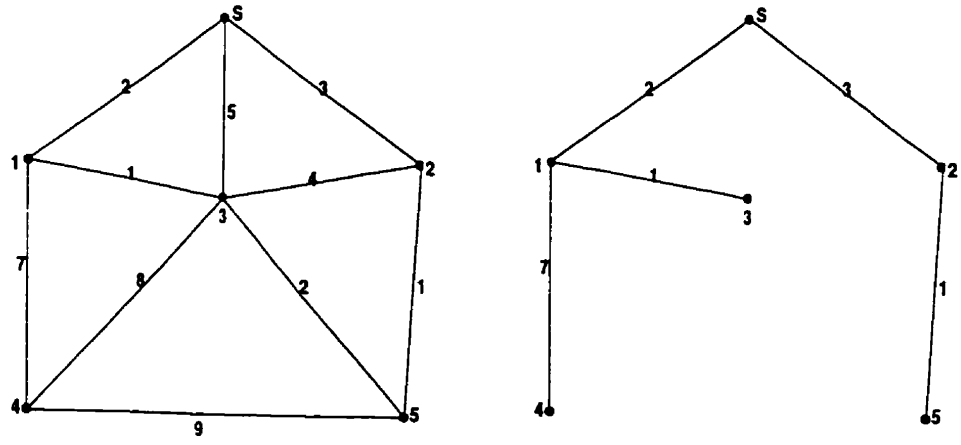
In case of directed graphs, the single source and single destination problems are directional duals of each other. In other words, given a directed graph G, reversing the edges of G converts the single source problem to the single destination problems, and vice versa.

In this thesis, an algorithm for computing all-pairs shortest path was developed. The algorithm has a preprocessing phase where the input graph is partitioned using the graph partitioning algorithm also implemented in this thesis. This chapter contains a review of the literature deemed most pertinent to the problem of computing shortest paths using graph separators. The first section deals with the single source shortest path problem while the second section deals with the all-pairs shortest path problem. Some solutions to the dynamic version of the shortest path problem use graph separators, and therefore are also discussed in this thesis.

5.2 The Single Source Shortest Path Problem

Computing the shortest path from a single source s to all vertices in a given graph G can be done by constructing a *shortest-path tree*. This tree is a spanning tree T of G rooted at s where every path is a shortest path. The weight of the path from s to v in a spanning tree T will be called $distance(v)$. According to Tarjan (1983), T will be a shortest-path tree if and only if $distance(v) + weight(v, w) \geq distance(w)$ for every edge $[v, w]$ in G. In Figure 5.1(a), an example is shown for a given weighted undirected graph. It is required to find the shortest path from the vertex s to all other vertices in this graph. The solution was found by computing the shortest-path tree shown in Figure 5.1(b).

Dijkstra (1959) proposed an iterative single source shortest path algorithm for directed or undirected graphs with nonnegative edge costs. The following is a brief



(a) A given weighted undirected graph

(b) Its shortest-path tree with the root at vertex s

Figure 5.1: An example of a single source shortest Path problem

description of its main steps. While running the algorithm, the graph vertices can have one of two states, either open or closed. A closed state means that the shortest path to that vertex has been already computed, while an open state means that the shortest path to the vertex is still not known. At the beginning, all the vertices are initialized to open and the $path(v)$ for all v in the graph is set to infinity, except $path(s)$ is set to 0.

Step 1. The vertex v with the minimum $path(v)$ is chosen and its state is changed to closed.

Step 2. For every edge (v, w), the value of $path(v) + weight(v, w)$ is compared to the current value of $path(w)$ and the minimum among them is chosen to be the new $path(w)$.

These two steps are repeated until all vertices are in a closed state, (Dijkstra, 1959).

Solving the single source shortest path problem on either directed or undirected graphs with nonnegative edge weights has time complexity $O(n^2)$ using Dijkstra's algorithm, (Dijkstra, 1959). This time complexity can be reduced to $O(n \log n)$ for

any graph with $O(n)$ edges by using a heap in the implementation of Dijkstra's algorithm. (Johnson, 1977).

5.2.1 Single Source Shortest Path Computation Using Graph Separators

Frederickson (1987) improved on Dijkstra's algorithm by adding a preprocessing phase that divides the graph into subsets of vertices in such a way that allows conducting the iterative search on each subset separately. In that phase, the separator theorem for planar graphs introduced by Lipton and Tarjan (1979) is used to partition the graph into regions. After the partitioning, the graph vertices are classified into one of two types, namely interior vertices and boundary vertices. Interior vertices are contained in exactly one region and are adjacent to vertices contained in the same region, while boundary vertices are shared among at least two regions. The shortest path between all pairs of boundary vertices in each region is then computed.

In order to make the algorithm more efficient, the regions obtained from partitioning the graph should have specific size and adjacency properties. These properties will not be guaranteed from just directly applying the separator theorem. Therefore, some modifications in partitioning the graph were introduced in order to obtain the required properties in the resulting regions. Frederickson (1987) presented some divisions types with the required size and adjacency properties and a data structure used in solving the shortest path problem.

An r-Division

Given an n-vertex graph and a parameter r, an *r-division* of the graph is a partitioning of the graph into $\Theta(n/r)$ regions each having $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices, which makes $\Theta(n/\sqrt{r})$ boundary vertices in total. An r-division can be obtained in the following way. First, the graph vertices are assigned weights $1/n$ each and then the separator algorithm presented by Lipton and Tarjan (1979) is applied to the graph with $1/3 \leq \alpha \leq 2/3$ yielding three subgraphs A, B and C. From this partitioning, two regions can be obtained. These two regions are A_1 and

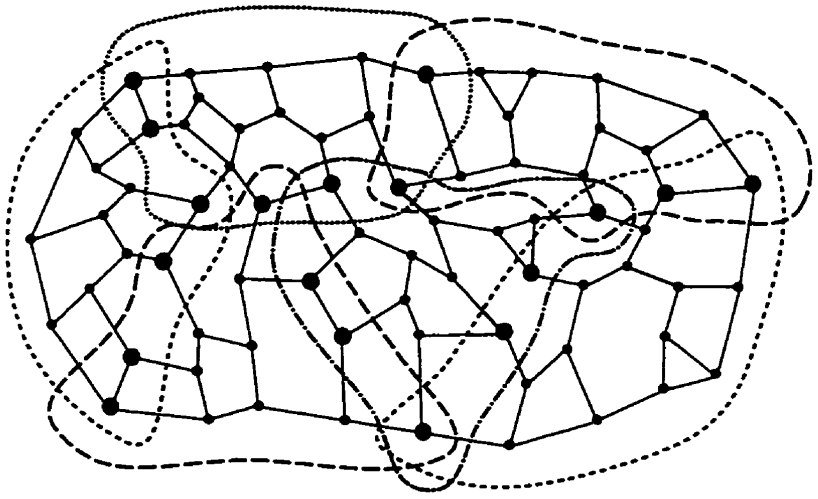


Figure 5.2: An example of dividing a planar graph into regions, (Frederickson, 1987)

A_2 where $A_1 \subseteq A \cup C$ and $A_2 \subseteq B \cup C$ of sizes $\alpha n + O(\sqrt{n})$ and $(1-\alpha)n + O(\sqrt{n})$, respectively. The above procedure is applied to the subgraphs A_1 and A_2 recursively until no subgraph has more than r vertices. This will be done in $O(n \log(n/r))$ total time.

In order to guarantee the condition for $O(\sqrt{r})$ boundary vertices in each region, the planar separator theorem is then applied to any region with more than $c\sqrt{r}$ boundary vertices, for some constant c. For the region's n' boundary vertices, every boundary vertex will be assigned a weight equal to $1/n'$ and the interior vertices will be assigned weight zero. Using the procedure described above, constructing an r-division for an n-vertex planar graph will take $O(n \log n)$ total time, (Frederickson, 1987).

Not only reducing the number of boundary vertices is important, but also reducing the number of regions that share a boundary vertex; therefore, the initial planar graph is transformed to get a planar graph with no vertex having degree greater than three. This transformation can be done using the transformation presented by Harary (1969). Figure 5.2 shows an example of an r-division. The graph is partitioned into six regions, each marked with a circle. The boundary vertices are drawn bold.

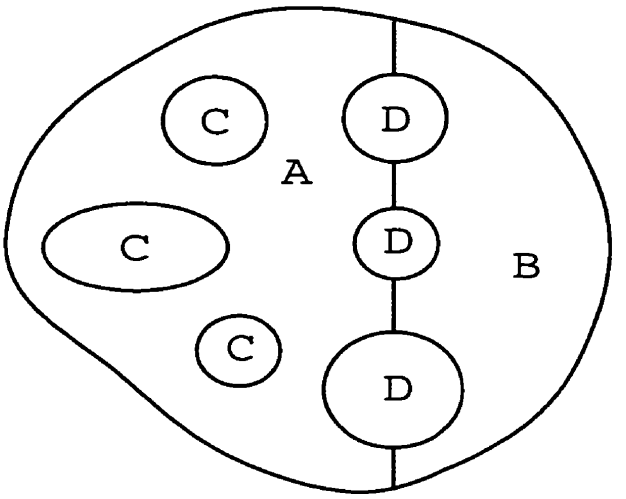


Figure 5.3: An example of regions that are unions of connected components, (Frederickson, 1987)

A Suitable r-Division

A *suitable r-division* of a planar graph is an r-division with the following two extra conditions. The first condition is that each boundary vertex is contained in at most three regions. The second condition is for disconnected regions. It states that any region that is disconnected, its connected components must share boundary vertices with exactly the same set of either one or two connected regions.

The following procedure is used to guarantee that the first condition holds. The planar separator algorithm is applied to a planar graph producing three subgraphs, A, B and C. Calculate the set of vertices in C not adjacent to any vertex in $A \cup B$. This set will be called C' and the set C'' will be $C'' = C - C'$. Compute the connected components $A_1, A_2, \cdots\cdots, A_q$ in $A \cup B \cup C'$. Any vertex v in C'' that is adjacent to a vertex in A_i but not adjacent to any vertex in A_j where $i \neq j$ will be deleted from C'' and inserted into A_i. Therefore, by making a vertex v a boundary vertex in a connected subgraph A_i (if v is adjacent to any vertex in A_i), any boundary vertex will be in at most three connected subgraphs because all the vertices are of degree three. Afterwards, if there is a connected subgraph with more than $c\sqrt{r}$ boundary vertices for some constant c, the planar separator algorithm will be applied to that connected subgraph.

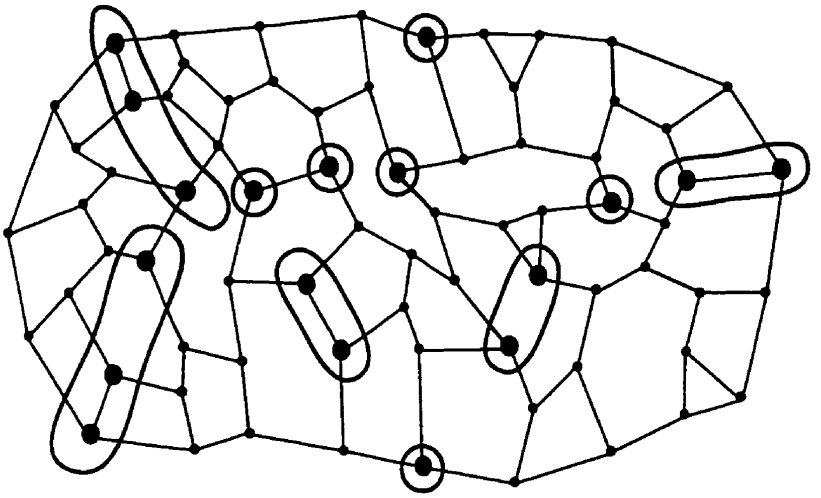


Figure 5.4: Boundary sets for the regions in Figure 5.2, (Frederickson, 1987)

By recursively applying the above procedure to any connected subgraph with more than r vertices, the resulting division can consist of more than $\Theta(n/r)$ connected subgraphs. Therefore, after applying the above steps, a greedy approach will be used to generate $\Theta(n/r)$ regions from the connected subgraph set obtained so far. Every connected subgraph will be considered a region and assigned an index. If there are two regions sharing a common boundary vertex and each has at most $r/2$ vertices and $c\sqrt{r}/2$ boundary vertices, then these two regions will be merged together. In case there are still more than $\Theta(n/r)$ regions, if there are two regions each having at most $r/2$ vertices and $c\sqrt{r}/2$ boundary vertices and are also adjacent to the same set of either one or two regions, these two regions will be merged. The resulting division will contain regions consisting of either one connected subgraph or a union of connected subgraphs that share boundary vertices with the same set of regions. A suitable r-division of a planar graph can be computed in $O(n \log n)$ time. Figure 5.3 shows an example of merging regions. The region C consists of three connected components each of which share a boundary with region A. Also region D consists of connected components that are adjacent to regions A and B.

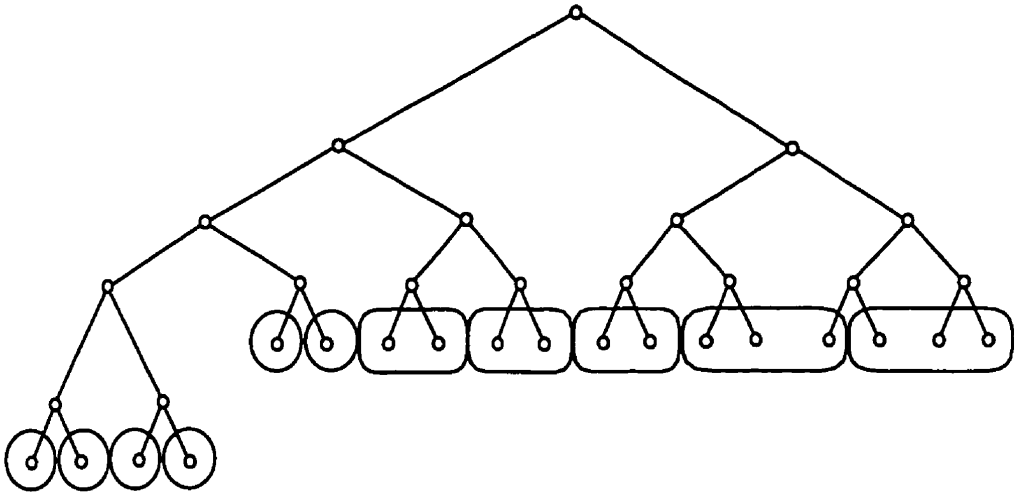


Figure 5.5: The topology-based heap for the boundary vertices sets in Figure 5.4, (Frederickson, 1987)

A Topology-Based Heap And Batched Update Operations

A *topology-based heap* is organized based on a suitable r-division by applying the following steps. The boundary vertices of a suitable r-division are partitioned into maximal subsets such that every subset contains boundary vertices that share exactly the same set of regions. This partitioning will result in $\Theta(n/r)$ boundary sets. The topology-based heap on boundary vertices is represented by a balanced tree. The leaves of the tree represent the boundary vertices and boundary vertices in one set are being represented in consecutive leaves. The topology-based heap can be created in $O(n/\sqrt{r})$ time. A *batched update* is an update operation performed on the topology-based heap where the values associated with all the vertices in one boundary set are updated. Figure 5.4 shows how the boundary vertices from Figure 5.2 are partitioned into maximal subsets and Figure 5.5 shows the corresponding topology-based heap.

A Single Source Shortest Path Algorithm

The following describes an algorithm to find a single source shortest path tree in a planar graph in $O(n\sqrt{\log n})$ time. Here, the suitable r-division can be computed in $o(n \log n)$ time, (Frederickson, 1987).

Input:

- A weighted planar directed graph.

Output:

- A Single Source Shortest Path Tree.

Step 1. Compute a suitable r_1-division of G where $r_1 = \log n$. Call the regions obtained from this division level 1 regions.

Step 2. For each level 1 region, find a suitable r_2-division where $r_2 = (\log \log n)^2$. Call the resulting regions level 2 regions. In order to minimize the number of boundary vertices, start with the boundary vertices of level 1 regions being the boundary vertices of level 2 regions. This way, no more than $\Theta(n/\sqrt{r_2})$ boundary vertices of level 2 regions will be created.

Step 3. For each level 2 region, find the shortest path between every pair of boundary vertices using Dijkstra's algorithm.

Step 4. Using the shortest path information computed so far, find the shortest path between every pair of boundary vertices in each level 1 region.

Step 5. Using Dijkstra's algorithm, compute the shortest path tree representing the shortest path to each level 1 boundary vertex in the graph. The topology-based heap is used in this step in order to improve its time complexity.

Step 6. Within each region, the shortest paths are then extended to the interior vertices using Dijkstra's algorithm. The topology-based heap is also used in this step.

Frederickson (1987) also proposed a slightly more complicated algorithm that is based on the above algorithm and the same graph partitioning techniques. This algorithm performs more recursive partitioning in the preprocessing phase in order to answer shortest path queries in $O(n)$ time.

5.2.2 Dynamic Single Source Shortest Path Computation Using Graph Separators

Dynamic algorithms for graph problems is considered a significant research area. The aim of these algorithms is to design a data structure that allows modifications to be done on the graph as well as answering queries. The difference here is that after modifications are done on the graph, it is required to obtain the new solution without having to recompute everything from scratch, (Feuerstein and Marchetti-Spaccamela, 1993).

The dynamic version of the shortest path problem has many important applications. Some of these applications include: dynamic maintenance of a maximum s-t flow in a planar graph (Hassin, 1981), computing a feasible flow between multiple sources and sinks and finding a perfect matching in bipartite planar graphs (Miller and Naor, 1991). The dynamic shortest path problem is also used in applications that include incremental computations, for instance, data flow analysis and interactive systems design, (Ramalingan and Reps, 1991), (Yellin and R., 1991).

Feuerstein and Marchetti-Spaccamela (1993) introduced a data structure and an algorithm for maintaining shortest distances in planar graphs where the edge weights are modified dynamically. More precisely, the data structure maintains a suitable r-division of a planar graph as well as the all-pairs shortest distances between boundary vertices. A preprocessing phase is needed that takes $O(n \log n)$ time. Using this data structure, the modifications of the shortest distances after an edge weight update are computed in $O(n\sqrt{\log \log n})$ time and the update operation is performed in $O((\log n)^3)$ time. The space required for the data structure is $O(n)$ because there are $O(n/\sqrt{r})$ boundary vertices. In case it is required to store not only the distances but also the shortest-path trees, the time complexity of the preprocessing phase, the modification computation and the update operation are the same. Only the space requirement will change to $O(n \log n)$.

Two operations were defined on weighted planar graphs. The first operation is for modifying an edge's weight. It is called $update(i, j, w)$ where it is required to modify the weight of the edge $[i, j]$ to the value w, where $w \geq 0$. The second

operation is for computing the shortest distances from a source vertex s to all the rest of the vertices in the graph. This operation is called *shortest-path*(s). The main steps for *shortest-path*(s) are following.

Step 1. If s is an interior vertex, the shortest distances from s to all the vertices within its region are computed.

Step 2. The shortest distances from s to all boundary vertices are computed.

Step 3. The shortest distances from boundary vertices to interior vertices in every region is then computed.

Step 4. If s is an interior vertex, the shortest distances from the vertex s to interior vertices in its region obtained from *step 1* are compared to those obtained from *step 3* and the shorter distances are chosen.

The algorithm presented by Frederickson (1987) is used to perform *Step 1* and, therefore, the time complexity for this step is $O(r\sqrt{\log r})$. In *Step 2*, the topology-based heap also presented by Frederickson (1987) is used leading to a time complexity for this step $O(n + (n/\sqrt{r})\log n)$. In *Step 3*, Frederickson (1987) algorithm is used in all regions. The topology-based heap is initialized with the distances from the vertex s to each boundary vertex of the region containing s. The time required for each region is $O(r\sqrt{\log r})$ and applying this to $\Theta(n/r)$ regions requires a total of $O(n\sqrt{\log r})$ time. Finally, *step 4* takes $O(r)$ time. By having $r = (\log n)^2$, the total algorithm *shortest-path*(s) will take $O(n\sqrt{\log\log n})$ time.

The other operation, namely the *update* operation, is used to modify the values of the all-pairs shortest distances for boundary vertices in regions that contain the modified edge. These will be at most three regions. A preprocessing phase has to be done before the *update* operation. In the preprocessing phase, the regions of the graph obtained from applying a suitable r-division are further divided. Each region is divided into a suitable r'-division where the boundary vertices from the suitable r-division (level 1 boundary vertices) are made as boundary vertices for the suitable r'-division (level 2 boundary vertices). This will produce a total of $O(\sqrt{r} + r/\sqrt{r'})$ boundary vertices in each region. After the partitioning, all-pairs shortest

distances between the boundary vertices of every subregion and every region are then computed. The division part takes $O(n\sqrt{\log n})$ time while the shortest distances computation part takes $O((n/\sqrt{r})(r + (r/\sqrt{r'})\log r))$ time. Setting the value of r' to be equal to $(\log r)^2$, the preprocessing phase will take a total of $O(n \log n)$ time. The update operation itself takes $O((\log n)^3)$ time. The following describes the main steps of the *update* operation.

Step 1. All-pairs shortest distances between level 2 boundary vertices of the subregion containing the modified edge are computed.

Step 2. All-pairs shortest distances between all level 1 boundary vertices of the regions containing the modified edge are computed.

The same algorithms presented by Feuerstein and Marchetti-Spaccamela (1993) can also be used to maintain one-pair shortest distance for an arbitrary pair of vertices in the graph. The only modification required for the case of one-pair shortest distance is in the *shortest-path*(s) algorithm. The third step in that algorithm is performed only in the region that contains the target vertex. Therefore, the time complexity of the *shortest-path* operation will be linear. The space requirement will remain the same. $O(n)$. The same applies to the case where it is required to maintain one-pair shortest path for an arbitrary pair of vertices in the graph. The time complexity of the *shortest-path*(s) and *update*(s) algorithm is also the same. Only the space requirement, which is $O(n \log n)$, is different than the shortest distance case.

5.3 All Pairs Shortest Path

In the all-pairs shortest path problem, a weighted directed graph with no negative cycles is given and it is required to find the shortest path between every pair of vertices in the graph. The classic algorithms for all-pairs shortest path computation are those by Dijkstra (1959) and Floyd (1962). Dijkstra's single source shortest path algorithm can be executed once for every vertex in the graph being the source vertex. If Fibonacci heaps are used in the implementation of priority queues, this algorithm

will take a total of $O(mn + n^2 \log n)$ time, where m is the number of graph edges and n is the number of vertices in the graph, (Johnson, 1977). Unlike Dijkstra's algorithm, the all-pairs shortest path algorithm introduced by Floyd (1962) can handle graphs with negative weights associated with the edges. The graph must not have negative cycles. The algorithm works by dynamic programming and has time complexity $\Theta(n^3)$.

Spira (1973) introduced another all-pairs shortest path algorithm that is based on Dijkstra's algorithm. In this algorithm, if the weights associated with the graph edges are independently and identically distributed, the expected run time will be $O(n^2(\log n)^2)$. Bloniarz (1980) introduced another algorithm for all-pairs shortest path computation that has expected run time $O(n^2 \log n(\log n)^*)$. Frieze and Grimmet (1985) developed another algorithm with expected run time $O(n^2 \log n)$, but this algorithm is suitable only for random graphs. Frederickson (1987) proposed an algorithm that uses the separator theorem by Lipton and Tarjan (1979) for solving the all-pairs shortest path problem on n-vertex planar graphs with either undirected edges and no negative costs or directed edges and negative costs but no negative cycles.

5.3.1 All Pairs Shortest Path Computation Using Graph Separators

Feuerstein and Marchetti-Spaccamela (1993) proposed an algorithm for all-pairs shortest-path computation in planar graphs. The proposed algorithm uses the separator theorem established by Lipton and Tarjan (1979) and it distinguishes between three cases. The first case is when the number of shortest path queries required is small compared to the number of vertices in the graph while the second and third case are when the number of shortest path queries is large with respect to the number of graph vertices. The reason behind this distinction is that when the number of queries is small compared to the number of graph vertices, the preprocessing and the shortest path query algorithm presented by Frederickson (1987) can be used. This will take $O(n \log n)$ preprocessing time, which will be done only once, and an $O(n)$

time per shortest path query. In the second and third case, all-pairs shortest path will be computed. The three cases are described in more details in the following. q represents the number of shortest path queries while n represents the number of graph vertices.

Case 1: $q \leq n^{1/2}$. When the value of q is small compared to n, the algorithm proposed by Frederickson (1987) is used. The preprocessing phase will take $O(n \log n)$ time and the single-source shortest path tree is computed in $O(n)$ time. Accordingly, for q queries, the total time required will be $O(qn + n \log n)$.

Case 2: $n^{1/2} < q \leq n$. In this case, the preprocessing phase consists of computing a suitable r-division recursively and then a single-source shortest-path tree once for each boundary vertex being the root. The recursive partitioning will take $O(n \log n)$ time and will result in $O(n/\sqrt{r})$ boundary vertices. Each single-source shortest-path tree computation takes $O(n)$ time and applying this once for every boundary vertex requires a total of $O(n^2/\sqrt{r})$ time. Therefore, the preprocessing time will take a total of $O(n \log n + n^2/\sqrt{r})$ time. The following steps were described by Feuerstein and Marchetti-Spaccamela (1993) in order to compute the shortest distance between two vertices, namely i and j. Computing the shortest path will follow the same approach.

dis(i,j)

Step 1. If either i or j is a boundary vertex, then the distance between them is already known from the preprocessing phase and no computation is needed.

Step 2. If neither i nor j is a boundary vertex and they lie in different regions, then the shortest distance between i and j will be the one satisfying the following condition:

$$dis(i,j) = min\{d(i,b) + d(b,j)\}$$

where b is a boundary vertex of i's region.

Step 3. If neither i nor j is a boundary vertex and they lie in the same region, then the shortest distance between i and j is computed limited

to the region. The shortest distance will satisfy the following condition:

$$dis(i,j) = min(d(i,j), min\{d(i,b) + d(b,j)\})$$

where b is a boundary vertex of i's region.

The algorithm $dist(i,j)$ takes $O(r)$ time and , therefore, the total time required for case 2 is $O(n \log n + n^2/\sqrt{r} + qr)$.

Case 3: $q > n$. In this case, the preprocessing cost is increased in order to reduce the shortest path query cost. A step is added in the preprocessing phase that computes all-pairs shortest-path distances within each suitable r-division region. This added step requires $O(r^2)$ time and, therefore, the total time required for the preprocessing phase will be $O(n \log n + nr + n^2/\sqrt{r})$. For the $dist(i,j)$ algorithm, *step 3* is modified to the following:

Step 3. If neither i nor j is a boundary vertex and they lie in the same region, the shortest distance between i and j will be:

$$dis(i,j) = min(d(i,j), min\{d(i,b) + d(b,j)\})$$

The time complexity of the modified *step 3* will be $O(\sqrt{r})$ and, therefore, the total time required for case 3 will be $O(n \log n + nr + n^2/\sqrt{r} + q\sqrt{r})$.

According to Feuerstein and Marchetti-Spaccamela (1993), by choosing a value of r that minimizes the total running time, the time complexity for the described three cases can be the following.

Case 1. $q \leq n^{1/2}$, the running time is $O(nq + n \log n)$.

Case 2. $n^{1/2} \leq q \leq n$, setting $r = n^{4/3}/q^{2/3}$ the running time is $O(n^{4/3}q^{1/3})$.

Case 3.

- $n < q \leq n^{4/3}$, setting $r = n^{2/3}$ the running time is $O(n^{5/3})$.
- $n^{4/3} < q \leq n^2$, setting $r = n^2/q$ the running time is $O(n\sqrt{q})$.

The algorithm proposed by Feuerstein and Marchetti-Spaccamela (1993) depends on knowing the value of q in advance. However, in case where the value of q cannot

be known in advance, the queries can be processed in phases. During one such phase, the algorithm neither changes the preprocessing nor the query strategy. Each phase starts with a preprocessing based on the optimal value r and the query strategy for the number of queries processed so far. The total number of phases is minimized in order to minimize the total cost of preprocessing. A new phase begins either when q reaches a threshold value at which the preprocessing and the query strategy must change or when the value of q doubles. Accordingly, the total number of phases is $O(\log q)$. When using phases, the time to answer a query may differ from the time defined by the above algorithm; however, the difference will only be a constant factor. Also, while using phases, the total processing time might differ by a constant factor, (Feuerstein and Marchetti-Spaccamela, 1993).

5.3.2 Dynamic All Pairs Shortest Path Computation Using Graph Separators

Feuerstein and Marchetti-Spaccamela (1993) proposed a dynamic version of the all-pairs shortest path algorithm where the weights of the edges can be modified by an update operation, namely, *update*(i, j, w). This update operation sets the weight of the edge(i, j) to the new value w. The proposed all-pairs shortest path algorithm distinguishes between three cases.

Case 1. $q \leq n^{1/2}$. When the number of queries is small compared to the number of graph vertices, the preprocessing phase and the single-source shortest path algorithm proposed by Frederickson (1987) is used. The preprocessing phase will take $O(n \log n)$ time while one query will take $O(n)$ time, and, therefore, the total time for one update and q queries will be $O(n \log n + qn)$.

Case 2. $n^{1/2} < q \leq n^{4/5}$. In this case, the cost of the *update* operation is increased in order to reduce the cost of a query. The cost of a query will be $O(r)$. Accordingly, the total time for one update and q queries is $O(n \log n + n^2/\sqrt{r} + qr)$.

Case 3. $n^{4/5} < q \leq n^2$. In this case, the cost of the *update* operation is also increased in order to reduce the cost of a query making it $O(\sqrt{r})$. An *update*(i, j, w) computes all-pairs shortest path within the suitable r-division regions containing ei-

ther i or j. After that, the single-source shortest-path trees rooted at each boundary vertex is computed. Therefore, one update operation takes $O(n \log n + r^2 + n^2/\sqrt{r})$ time and the time required for one update and q queries is $O(n \log n + r^2 + n^2/\sqrt{r} + q\sqrt{r})$.

According to Feuerstein and Marchetti-Spaccamela (1993), by choosing a value of r that minimizes the total running time, the time complexity for the described three cases can be the following.

Case 1. $q \leq n^{1/2}$, the running time is $O(n \log n + qn)$ and the preprocessing cost is $O(n \log n)$.

Case 2. $n^{1/2} \leq q \leq n^{4/5}$, setting $r = n^{4/3}/q^{2/3}$ the running time is $O(n^{4/3}q^{1/3})$ and the preprocessing cost is $O(n^{4/3}q^{1/3})$.

Case 3.

- $n^{4/5} < q \leq n^{6/5}$, setting $r = n^{4/5}$ the running time is $O(n^{8/5})$ and the preprocessing cost is $O(n^9/5)$.
- $n^{6/5} < q \leq n^2$, setting $r = n^2/q$ the running time is $O(n\sqrt{qn})$ and the preprocessing cost is $O(n^3/q + n\sqrt{q})$.

5.4 All Pairs Shortest Path Computation

In this thesis, a parallel preprocessing phase for computing all-pairs shortest path in planar graphs was implemented. The preprocessing implemented here consists of four phases: partitioning the input graph using the vertex-separator algorithm implemented in this thesis, computing all-pairs shortest path for every graph partition, collecting the all-pairs shortest distances between boundary vertices of every partition, computing all-pairs shortest path between all boundary vertices.

The first phase takes as input a graph G and the number of partitions required, ν. It uses the vertex-separator algorithm implemented in this thesis to partition the input graph. The output of this phase is ν partitions, $\{G_0, G_1, \cdots\cdot, G_{\nu-1}\}$. In the second phase, every processor P_i is assigned a graph partition G_i obtained from

Phase 1, where $0 \leq i < \nu$. It computes all-pairs shortest path for the vertices belonging to G_i. It does not use any edges or vertices of other partitions in its computation. Dijkstra's single source shortest path algorithm is used in this phase. It is used once for every vertex in G_i being the source vertex.

In the third phase, one processor P_0 collects a subset of the results computed by every processor. It collects from every processor P_i the shortest distances between the boundary vertices of G_i. The boundary vertices of G_i are the vertices that belong to the vertex-separator computed in Phase 1 and also belong to G_i. The fourth phase is executed only by processor P_0, the processor that collected the intermediate results in Phase 3. This processor creates a new graph G'. The vertices of G' are the set of vertices of the vertex-separator, i.e., all boundary vertices. The edges of G' are constructed from the shortest distances collected in Phase 3. Every edge represents the shortest distance between two boundary vertices of G_i restricted only to G_i. After that, all-pairs shortest path is computed for G'. Dijkstra's algorithm is also used here once for every vertex in G' being the source vertex.

After the preprocessing phase, queries for a given pairs of vertices (i, j) will be processed in the following way. The query is sent to processor P_0. Processor P_0 checks three possible conditions for the pair (i, j).

Vertices i and j are boundary vertices: In this case, the shortest distance between (i, j) is known by processor P_0.

Only vertex i is a boundary vertex: Assume that the graph partition containing vertex j, G_j, was assigned to processor P_j. Processor P_j sends to P_0 the shortest distances from vertex j to all boundary vertices in G_j. After that, processor P_0 constructs a set S of shortest distances. One shortest distance in S consists of the following. The shortest distance from vertex i to a boundary vertex u in G_j plus the shortest distance from u to j. This is computed for every boundary vertex u in G_j and stored in S. The shortest distance between i and j is the minimum shortest distance in S.

Both vertices i and j are not boundary vertices: Assume that the graph partition containing vertex i, G_i, was assigned to Processor P_i and the graph partition

containing vertex j, G_j, was assigned to processor P_j. Processor P_i sends to processor P_0 the shortest distances from i to all boundary vertices in G_i and processor P_j sends to processor P_0 the shortest distance from j to all boundary vertices in G_j. After that, processor P_0 constructs a set of shortest distances S. Every shortest distance in S consists of the following. The shortest distance from vertex i to a boundary vertex u in G_i plus the shortest distance from vertex u to a boundary vertex v in G_j plus the shortest distance from v to j. This is repeated for all possible combinations of u and v and stored in the set S. If vertices i and j lie in the same graph partition, the shortest distance between them restricted to that partition is also sent from P_i to P_0 and added to S. Finally, the shortest distance between i and j is the minimum shortest distance in S.

If not only the shortest distance is required but the shortest path, this can be retrieved from the processors that contain the graph partitions that contain subsets of the shortest path. These graph partitions include:

- The graph partition that contains vertex i, G_i.
- The graph partition that contains vertex j, G_j.
- Any graph partition $G_{(u,v)}$ that contains two boundary vertices, u and v participating in the shortest path.

The running time analysis for the preprocessing phase is as follows. For an input graph G with n vertices, Phase 1 takes linear time using the vertex-separator algorithm, $O(n)$. Every output partition has $O(\frac{n}{\nu})$ vertices and the total number of boundary vertices in all partitions is $O(\sqrt{n\nu})$. Accordingly, Phase 2 can be computed in $O((\frac{n}{\nu})^2 \log \frac{n}{\nu} + (\frac{n}{\nu})^2)$ time. This phase is executed in parallel, i.e., every processor is assigned a graph partition and computes all-pairs shortest path for that graph partition. In Phase 3, the total amount of information sent to P_0 is $O(n\nu^2)$ in the worst case. Finally, Phase 4 can be computed in $O(n\nu \log \sqrt{n\nu} + n\nu\sqrt{n\nu})$ time.

Chapter 6

Results and Performance Analysis

The key measurement in describing the benefits of parallel computing is the *speedup*. Speedup is defined as the improvement in performance by using p processors instead of one processor for a given problem. Let T_s be the time for a sequential solution to the given problem and T_p be the time for the parallel solution on p processors to the given problem. The speedup S_p on p processors is defined in Equation 6.1 as

$$S_p = \frac{T_s}{T_p} \tag{6.1}$$

Ideally, T_s should be the time for the optimal sequential algorithm for the current problem on the best sequential hardware. If no optimal solution is available, the fastest sequential algorithm known is used. However, this does not provide a true picture of the gain obtained from parallelizing the problem. The sequential hardware used must be comparable in performance to a single node in the parallel hardware, especially because state of the art parallel machines do not always use the most recent sequential processors, (JaJa, 1992).

An ideal parallel application should have a speedup equal to or greater than the number of processors used. However in real parallel applications, there is usually a limit on the speedup that is obtained by adding more processors. This limit is due to the amount of time spent in communication, load balancing, or waiting for data to be received from other processors. In other words, there is a point after which there is no increase in the performance no matter how many more processors are added. On the contrary, a drop off in performance may occur after that point due

Table 6.1: The selected TINs attributes

Number of Faces	Exaggerated Heights	Data Source
2854	NO	real DEM
2854	YES	real DEM
9944	NO	real DEM
9944	YES	real DEM
50244	NO	RANDOM

to load balancing and communication overhead.

In the rest of this chapter, the performance results of the horizon line and all pairs shortest path algorithms implemented in this thesis are presented and analyzed. All the experiments were executed on a cluster of workstations consisting of 15 166MHz Pentium processors interconnected by a 100MHz Ethernet switch, running Linux and using the LAM MPI implementation.

6.1 Horizon Line Computation

A natural subclass of polyhedral terrain is a *Triangulated Irregular Network* or TIN. A TIN is constructed from a triangulated point set in the plane where every point is associated with a height value. In geographic information systems, terrain visibility algorithms often use TINs to model the terrain. Accordingly, the experimental results of the horizon line algorithm presented in this thesis are for TINs although the algorithm can be applied on any polyhedral terrain.

It is conceivable that the performance of an algorithm can be affected by different TIN characteristics. While choosing the TINs to be used in the experiments, the following different TIN characteristics were taken into consideration: TIN's size (i.e. number of faces), height characteristics (i.e. smooth or spiky), and data sources (i.e. random or sampled from Digital Elevation Models (DEM)). The TINs chosen to perform the experiments upon were created from real DEM and one was randomly

created. Three TINs of different sizes were chosen to perform the experiments upon. Two of these three TINs were stretched by multiplying their heights by five in order to study the effect of the terrain height on the algorithm's performance. Experiments were performed on those two TINs one time before stretching and a second time after stretching. Table 6.1 shows a summary of the TINs used in the experiments.

For the TINs with small number of faces and medium number of faces, five viewpoints were computed in order to study the effect of the viewpoint position on the performance of the algorithm. The viewpoints were chosen along the diagonal and then shifted a small amount either to the left or to the right of the diagonal. The first viewpoint is in the center of the diagonal (where $x_1 = 50$), the second is in the corner (where $x_1 = 10$), the third is at $x_1 = 25$, the fourth is at $x_1 = 66$ and the fifth is at $x_1 = 83$ along the diagonal. Figure 6.1 explains how the viewpoints were distributed along the diagonal. The height of a viewpoint was the height of the terrain at the viewpoint's position plus a small constant. The experiments were performed on those two TINs one time before stretching and a second time after stretching.

In the following two subsections, 6.1.1 and 6.1.2, the results of the horizon line algorithm using the interval partitioning scheme and the vertex-separator algorithm are presented. In both cases, the algorithm starts after the TIN data are loaded on all processors and when the algorithm finishes execution the output is present on one processor. Regarding where the output should reside after the algorithm finishes, this is considered application dependent. In other words, it depends on whether the application that uses the horizon line algorithm requires the output to reside on all processors for further processing or not.

6.1.1 Using Interval Partitioning

In Figures 6.2-6.7, the computation time consists of the time for:

1. Dividing the TIN into a left and a right part.

2. Computing the projection of the TIN's edges on the θ-α-plane.

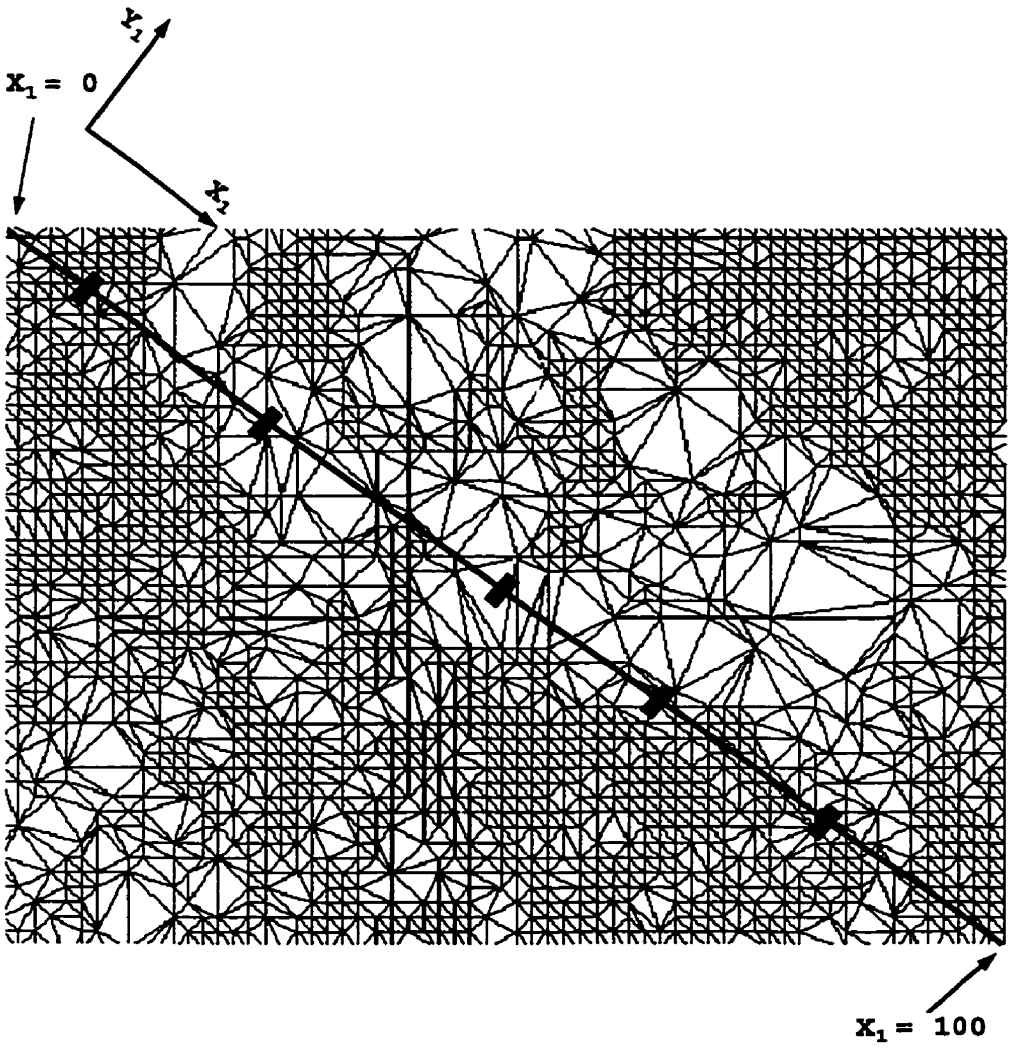


Figure 6.1: Showing the different viewpoint positions used in the experiments

3. Computing the pivots for partitioning.

4. Partitioning the projected edges (without the time spent in communication).

5. Computing the left and right upper envelope of the projected edges.

6. Gathering the result on one processor.

In every experiment, the time of the slowest processor was considered the computation time of the experiment. Some of the experiments were executed two or three times, and it was found that the difference in the running time was insignificant. Therefore, the rest of the experiments were executed only once.

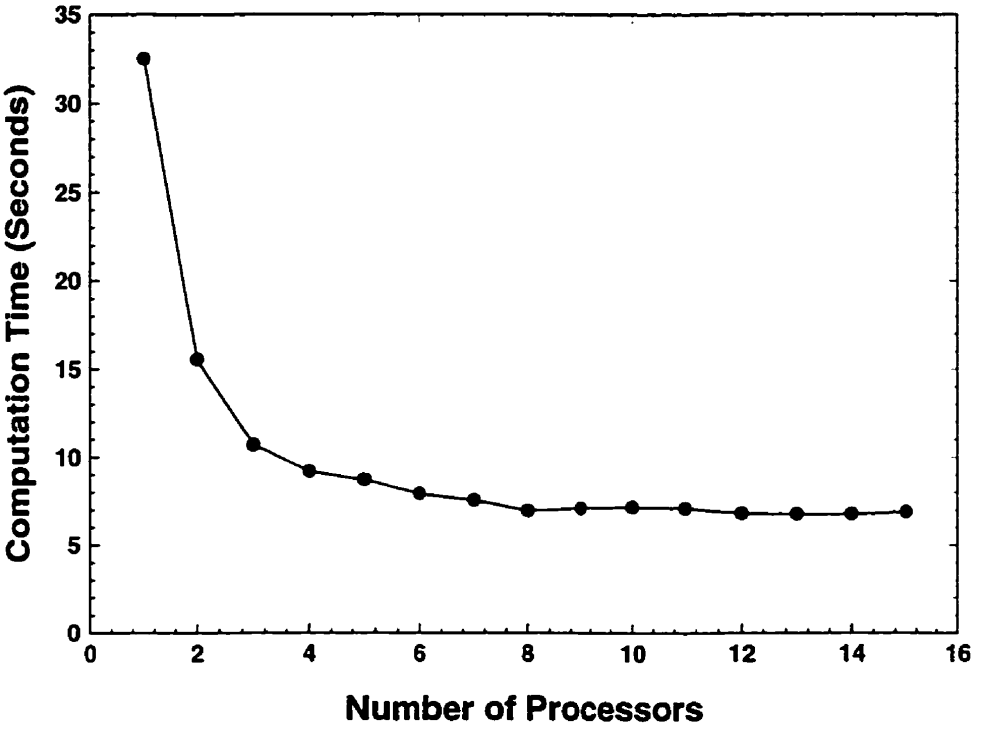


Figure 6.2: Performance of the horizon line algorithm using a TIN with 50244 faces.

The results of applying the horizon line algorithm on a TIN with 50244 faces are presented in Figure 6.2. This figure shows that as the number of processors working on the problem increases, the amount of time taken to compute the final result dramatically decreases at the beginning and then levels off. There are several factors that contribute to this behavior. As the number of processors used increases, the time taken to partition the data increases. In addition, the intervals assigned to the processors will decrease in width causing an increase in the number of segments that are cut, and therefore, an increase in the total number of segments that have to be processed. For instance, Figure 6.2 shows that when ten processors are used a large number of segments are cut, and, therefore, there is a slight increase in the computation time.

In Figure 6.3, the performance of applying the algorithm on a TIN with 9944 faces is presented. Similar to the results in Figure 6.2, Figure 6.3 shows that as the number of processors working on the problem increases, the amount of time taken

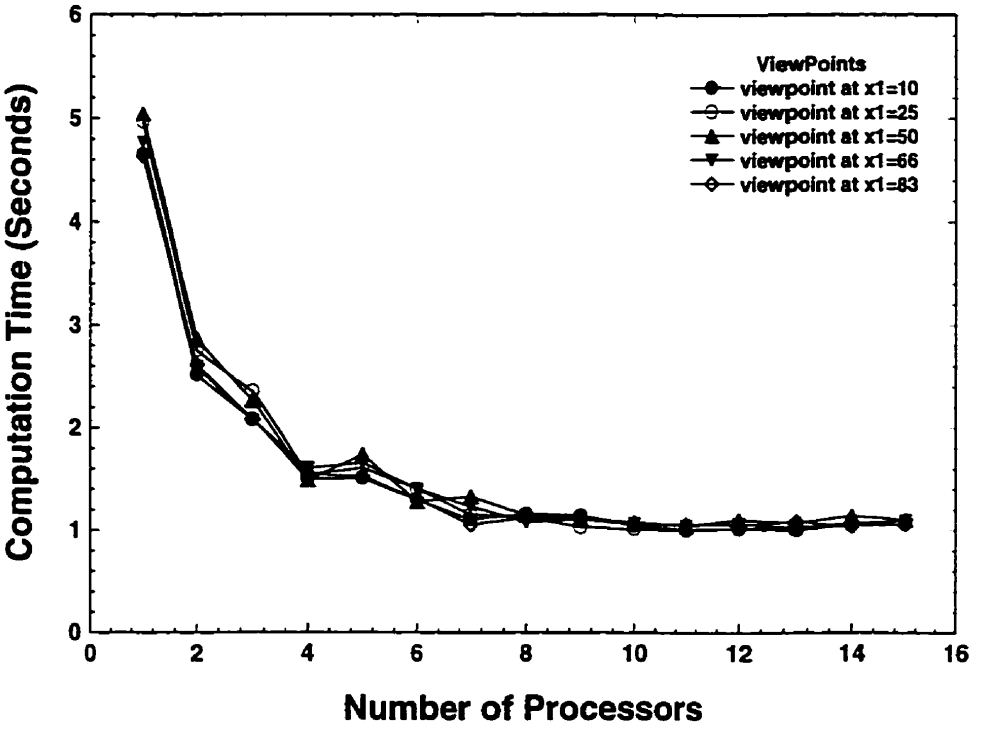


Figure 6.3: Performance of the horizon line algorithm using a TIN with 9944 faces. Showing the effect of the viewpoint position on the algorithm's performance.

to compute the final result dramatically decreases at the beginning and then levels off. This figure also shows that the viewpoint position has no major effect on the algorithm's performance in the studied cases.

Figure 6.4 shows the effect of stretching the TIN on the algorithm's performance. In this figure, the vertices' heights of the TIN used were multiplied by five and the same viewpoints positions were used. It can be seen from Figure 6.4 that stretching the TIN has no major effect on the algorithm's performance in the studied cases. The only difference is when the algorithm was run on one processor. The computation time for the stretched TIN on one processor is less than the corresponding time for the same TIN before stretching. It can be concluded from this, that the effect of stretching the TIN was negligible in the studied cases that it appeared only when running the algorithm on one processor and diminished when running the algorithm on multiple processors. The same experiments were applied on a TIN

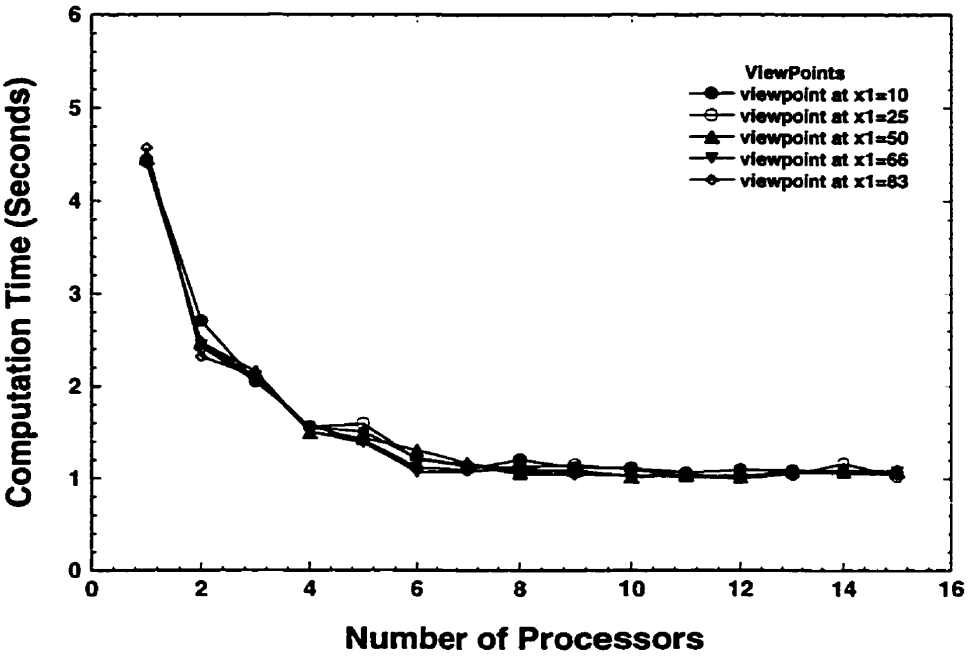


Figure 6.4: Performance of the horizon line algorithm using a TIN with 9944 faces. Showing the effect of the viewpoint position and the effect of stretching the TIN on the algorithm's performance.

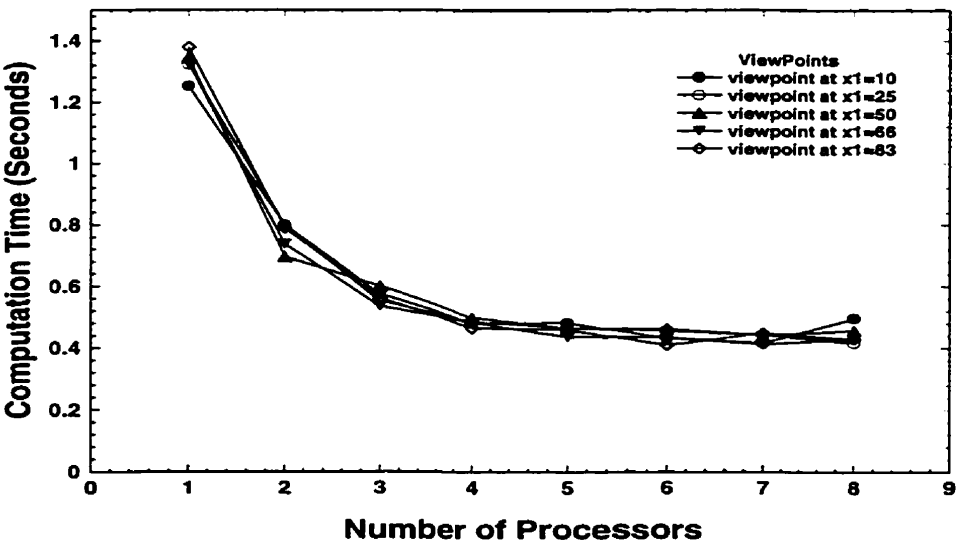


Figure 6.5: Performance of the horizon line algorithm using a TIN with 2854 faces. Showing the effect of the viewpoint position on the algorithm's performance.

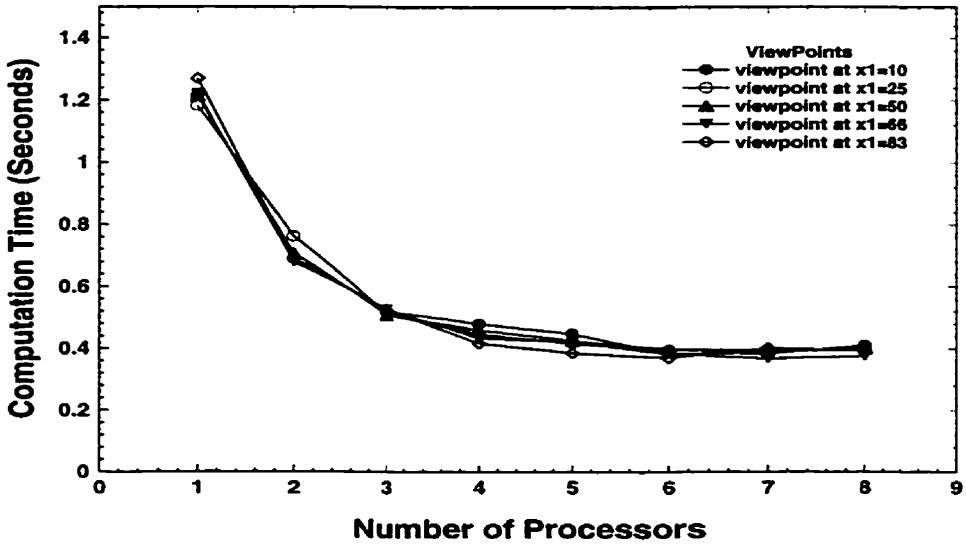


Figure 6.6: Performance of the horizon line algorithm using a TIN with 2854 faces. Showing the effect of the viewpoint position and the effect of stretching the TIN on the algorithm's performance.

with 2854 faces, one time before stretching and a second time after stretching. The results of these experiments are shown in Figure 6.5 and Figure 6.6, respectively. The behavior in these figures is similar to the the corresponding one using the TIN with 9944 faces.

Figure 6.7 shows the performance of the horizon line algorithm applied to different TIN sizes. For the small and medium size TINs, the computation time is the average of the computation time for the five different viewpoints.

Figure 6.8 presents the speedup of applying the horizon line algorithm to a TIN with 9944 faces at five different viewpoints. This figure shows that there is a gain in performance until processor number 12. When using more than 12 processors, there is no gain in performance; on the contrary, the performance starts to drop off. The figure also shows that the speedup for using different viewpoints is approximately the same for the studied cases. Figure 6.9 presents the speedup of applying the horizon line algorithm to the same TIN after stretching it. The speedup for five different viewpoints are also presented. For two of these viewpoints, the maximum speedup is obtained when using eight processors. For the other three viewpoints,

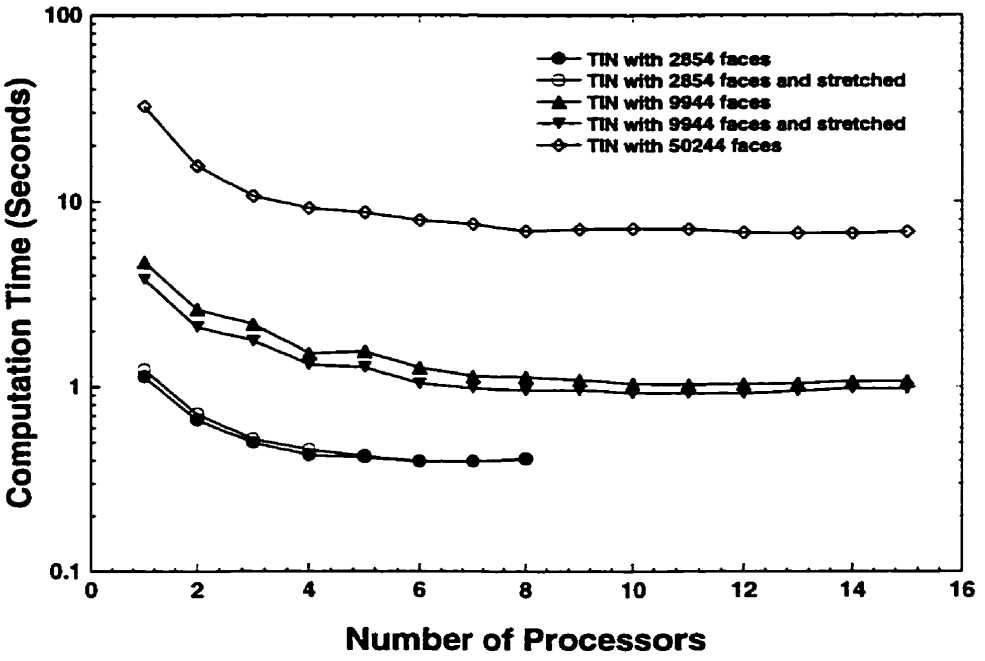


Figure 6.7: Performance of the horizon line algorithm using different TIN sizes.

the maximum speedup is obtained when using 12 processors.

Figure 6.10 presents the speedup of applying the horizon line algorithm to a TIN with 2854 faces using five different viewpoints. The results in this figure shows that there is an increase in the speedup obtained until the use of six processors for four of the viewpoints. As for the fifth viewpoint, the maximum speedup is obtained when using seven processors. Still, the performance for all the viewpoints is relatively close within the studied cases. In Figure 6.11, the same TIN was stretched and used with the same five viewpoints. The performance in this case has the same tendency as the case when the TIN was not stretched. Four viewpoints have maximum speedup when using six processors and the fifth viewpoint has maximum speedup when using seven processors. In addition, the performance for the different viewpoints is close.

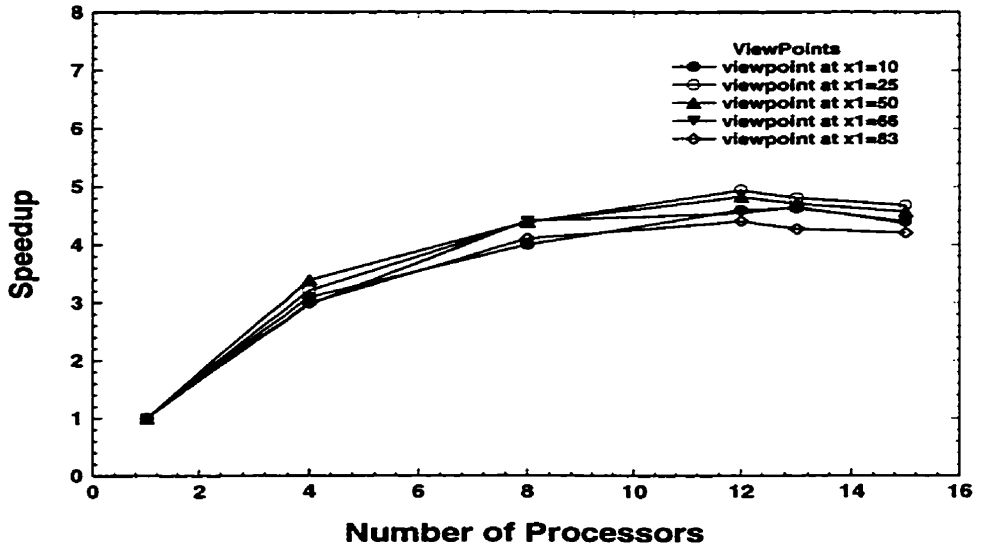


Figure 6.8: Speedup of the horizon line algorithm using a TIN with 9944 faces.

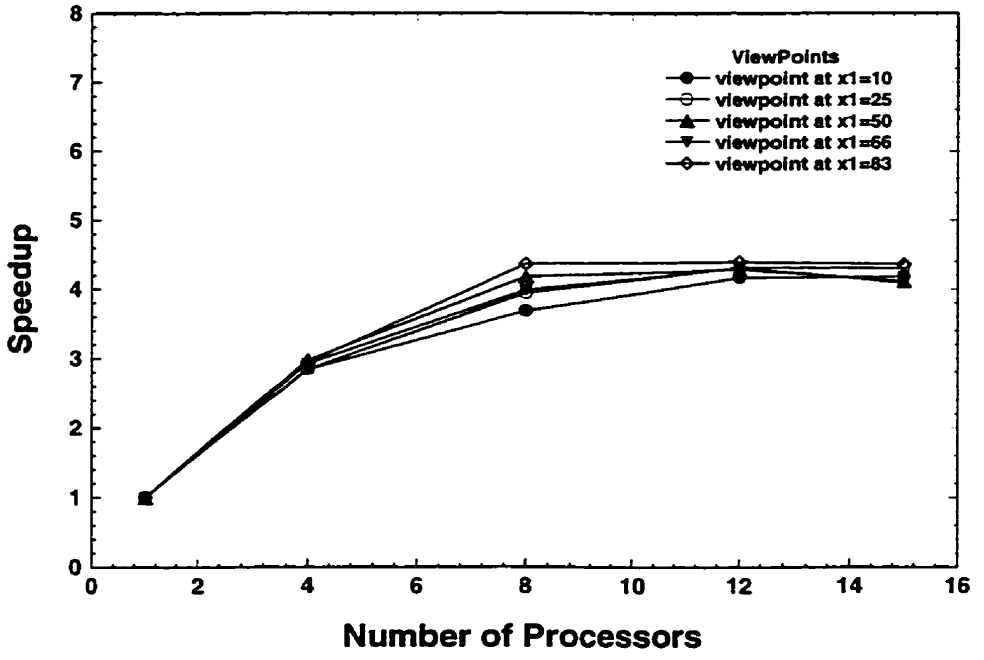


Figure 6.9: Speedup of the horizon line algorithm using a TIN with 9944 faces and stretched.

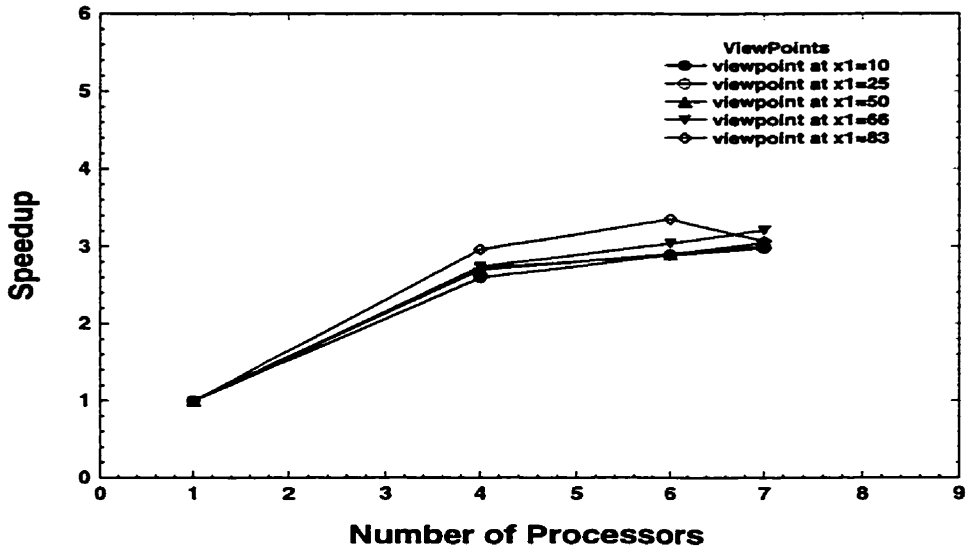


Figure 6.10: Speedup of the horizon line algorithm using a TIN with 2854 faces.

Figure 6.12 presents the speedup of the horizon line algorithm on TINs with different characteristics. For the small and medium size TINs, the speedup is computed from the average computation time for the five different viewpoints. Figure 6.13 presents the relationship between the TIN size and the speedup obtained. This figure shows that as the TIN size increases, the speedup increases. Based on these results, the rate of speedup increase is found to be higher for TINs with small sizes. This effect is due to the fact that using the interval partitioning scheme, the number of segments cut increases as the TIN size increases. Therefore, the total number of segments that need to be processed increases. Another reason for this behavior is that as the TIN size increases the time spent in partitioning the input data increases. Also, Figure 6.13 shows that the maximum speedup for applying the horizon line algorithm on a small size TIN is gained when six processors are used. However, as eight processors are used, there is a drop off in performance due to parallelization overhead. Finally, for the medium size TIN, the maximum gain in speedup is obtained when using 12 processors. This is also the case for the large size TIN, as seen in Figure 6.13.

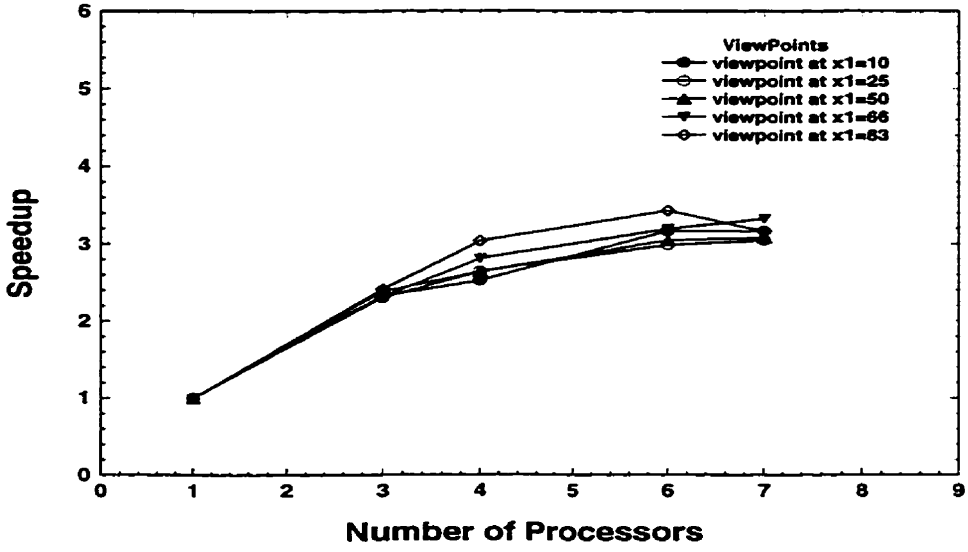


Figure 6.11: Speedup of the horizon line algorithm using a TIN with 2854 faces and stretched.

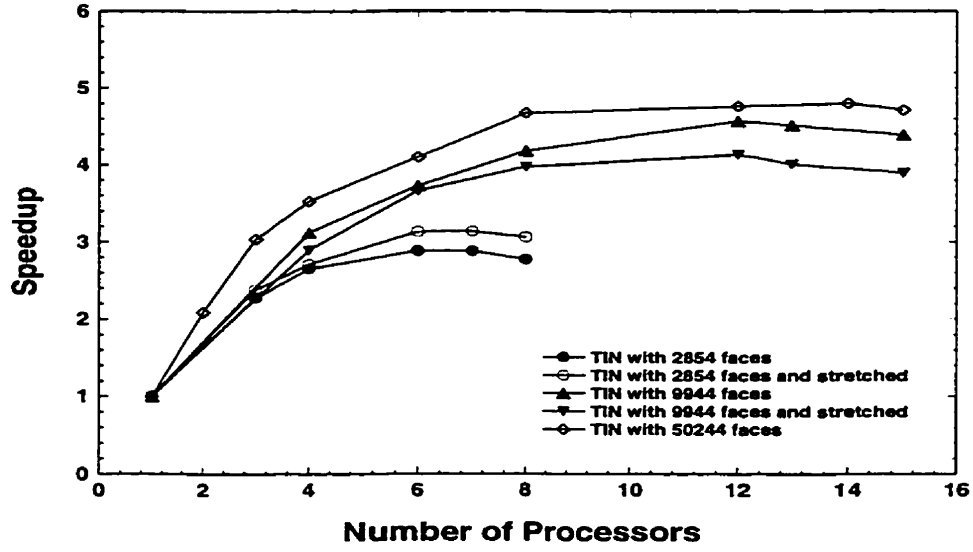


Figure 6.12: Speedup of the horizon line algorithm using different TINs

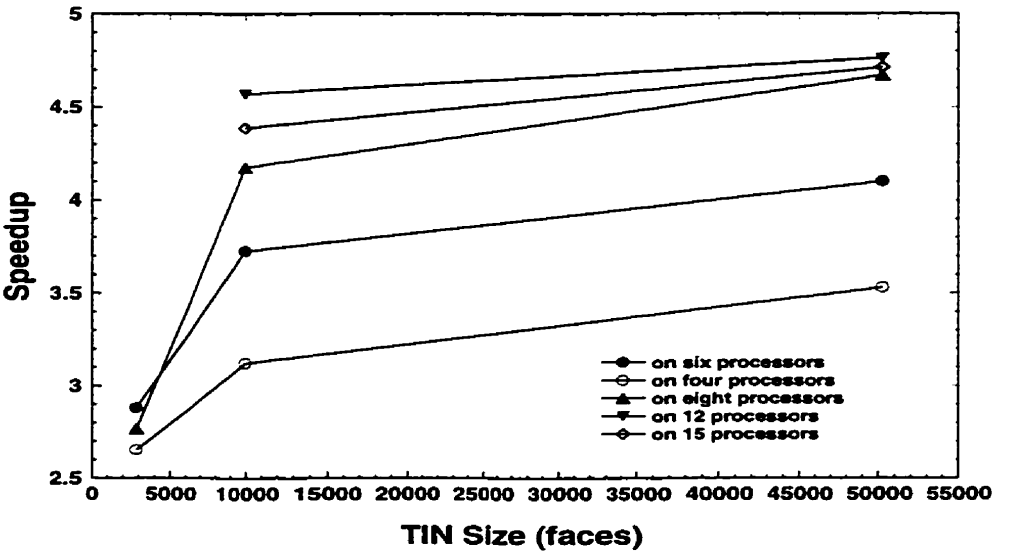


Figure 6.13: The relation between TIN sizes and speedup in the horizon line algorithm

6.1.2 Using Vertex-Separator

In Figures 6.14-6.17, the computation time consists of the time spent by one processor for:

1. Dividing the TIN into a left and a right part.
2. Computing the projection of the TIN's edges on the θ-α-plane.
3. Computing the left and right upper envelope of the projected edges.
4. Collecting and merging the horizon lines obtained from all processors.

In every experiment, the time of the slowest processor was considered the computation time of the experiment. The running time for the vertex-separator algorithm for the largest TIN used was approximately three seconds. Some of the experiments were executed two or three times, and it was found that the difference in the running time is insignificant. Therefore, the rest of the experiments were executed only once.

The results of applying the horizon line algorithm on a TIN with 50244 faces are presented in Figure 6.14. This figure shows that as the number of processors

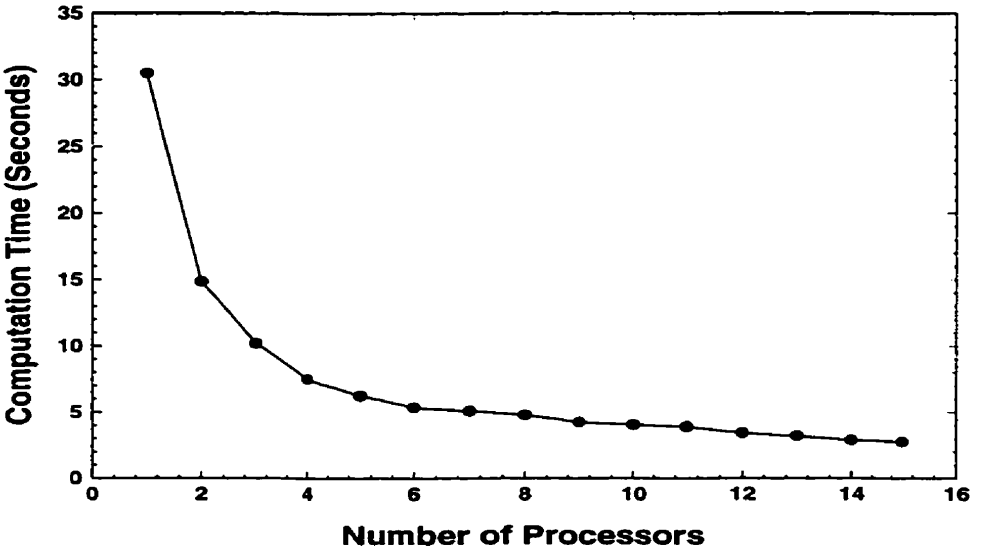


Figure 6.14: Performance of the horizon line algorithm using a TIN with 50244 faces.

working on the problem increases, the amount of time taken to compute the final result decreases. The computation time decreases with a higher rate for a small number of processors than for a large number of processors. This is due to the fact that as the number of processors increases, the time taken to collect and merge the partial horizon lines obtained from all processors increases.

In Figure 6.15, the performance of applying the algorithm on a TIN with 9944 faces is presented. Similar to the results in Figure 6.14, Figure 6.15 shows that as the number of processors working on the problem increases, the amount of time taken to compute the final result decreases. The computation time decreases with a higher rate for the use of six or less processors. As a larger number of processors is used, change in computation time is found to be less significant. This figure also shows that the viewpoint position has no major effect on the algorithm's performance in the studied cases. Figure 6.16 shows a similar performance with the application of the horizon line algorithm on a TIN with 2854 faces.

Figure 6.17 shows the performance of the horizon line algorithm applied to different TIN sizes. For the medium size TIN, the computation time is the average of the computation time for the two viewpoints. Figure 6.18 presents the performance

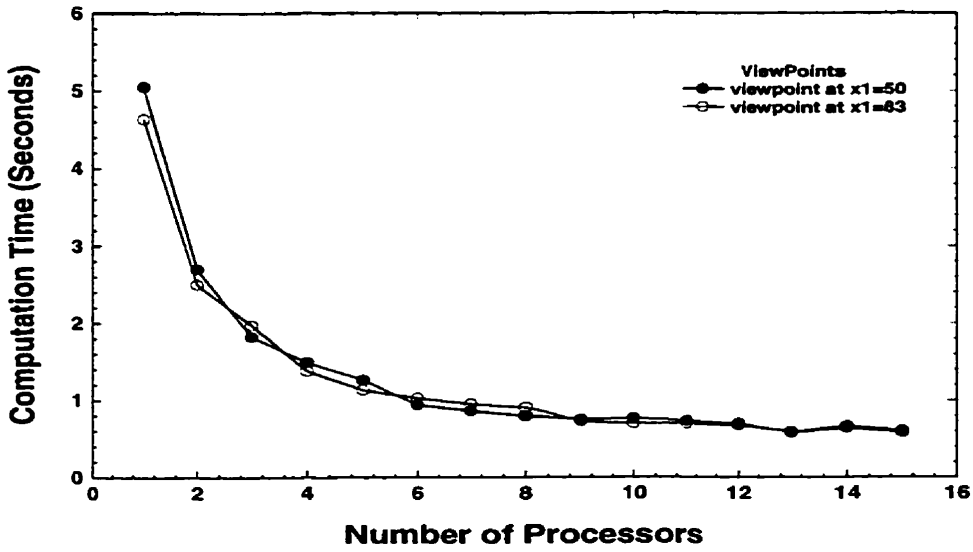


Figure 6.15: Performance of the horizon line algorithm using a TIN with 9944 faces. Showing the effect of the viewpoint position on the algorithm's performance.

of the horizon line algorithm on TINs with different sizes. For the medium size TIN. the speedup is computed from the average computation time for the two viewpoints. This figure shows that for the small and medium size TINs maximum speedup is obtained when using 13 processors. For the large size TIN, a continuous increase in its speedup is noticed using the maximum number of processors available. The rate of speedup increase using a larger number of processors might be the subject of future work. Figure 6.19 presents the relationship between the TIN size and the speedup obtained. This figure shows that as the TIN size increases, the speedup increases. The rate of speedup increase is found to be higher for TINs with small sizes. This is due to the fact that as the TIN size increases, the partial horizon lines computed by the processors increase in size and therefore the time needed to collect and merge these partial horizon lines increases.

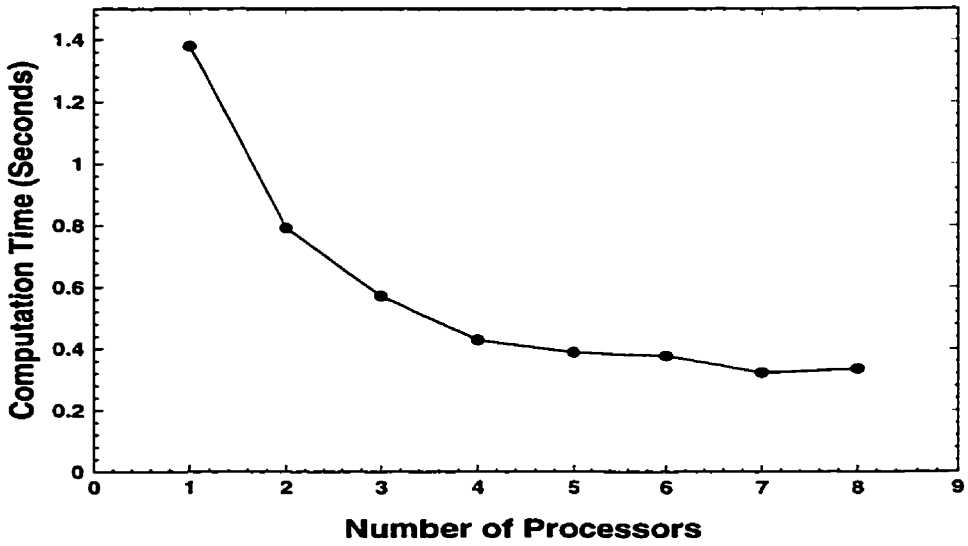


Figure 6.16: Performance of the horizon line algorithm using a TIN with 2854 faces.

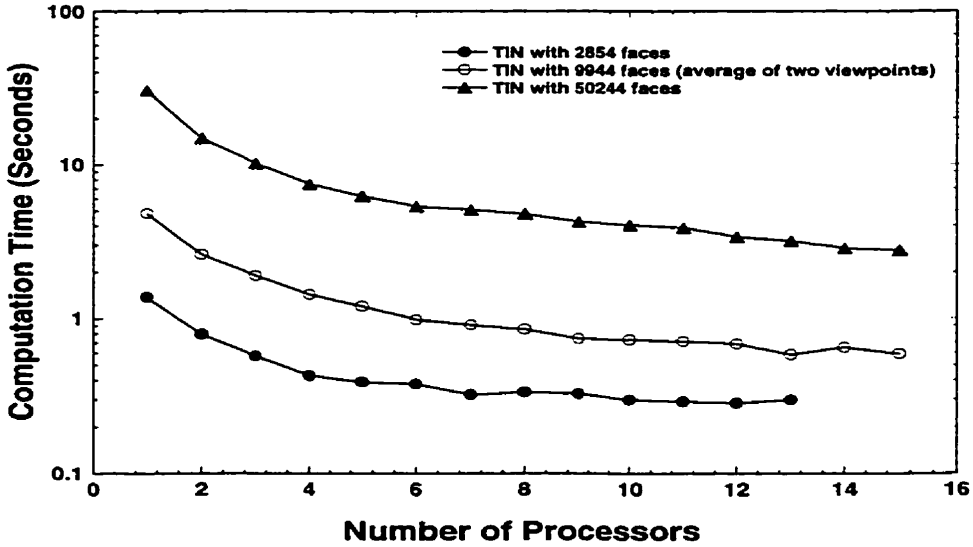


Figure 6.17: Performance of the horizon line algorithm using different TIN sizes.

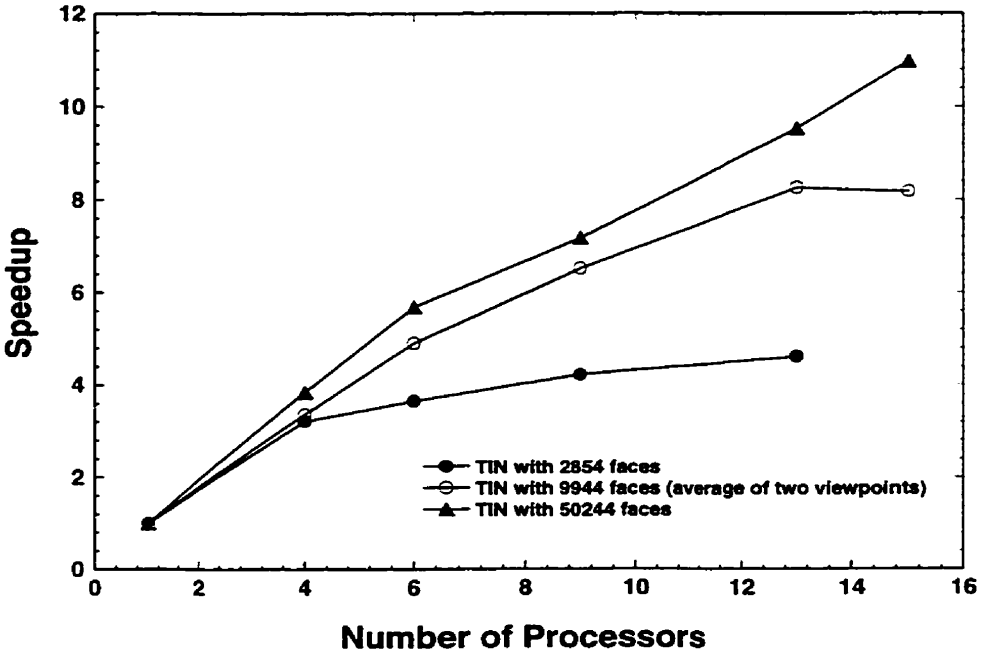


Figure 6.18: Speedup of the horizon line algorithm using different TINs

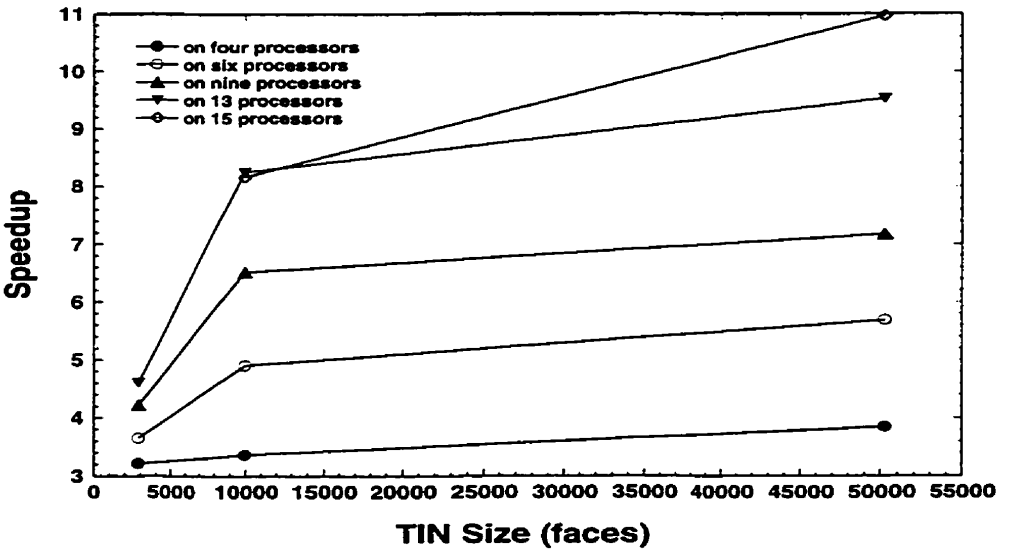


Figure 6.19: The relation between TIN sizes and speedup in the horizon line algorithm

6.1.3 Discussion

In this subsection, the factors that affect the performance of the horizon line algorithm are discussed. Also, a comparison is made between the performance of this algorithm using the interval partitioning scheme and its performance using the vertex-separator algorithm implemented in this thesis.

Based on the results presented in Sections 6.1.1 and 6.1.2, the factors that affect the performance of the horizon line algorithm are:

1. **TIN Size:** As the TIN's size increases, the time taken to compute the horizon line increases.

2. **Viewpoint Position:** The performance of the algorithm is affected by the position of the viewpoint; however, this effect is not major in the studied cases. This effect is explained in point Number 4.

3. **Stretching a TIN:** Stretching a TIN has no significant effect on the algorithm's performance in the studied cases. This effect is also explained in point Number 4.

4. **Complexity of Horizon Line in Intermediate Steps:** Not only the number of segments inputted to the algorithm and the size of the final horizon line affect the computation time, but also the size of intermediate horizon lines obtained in the recursion. This was measured by counting the number of event points during all recursive steps of the algorithm and comparing these with the computation time.

Figure 6.20 shows the computation time for TINs with different characteristics for both partitioning schemes. This figure shows that there is a decrease in computation time obtained from parallelism and this decrease becomes more significant when the TIN size increases. Figure 6.20 also shows that the computation time for the experiments done using the vertex-separator algorithm is less than the corresponding experiments done using the interval partitioning scheme.

Figure 6.21 shows the speedup obtained for the experiments done using the interval partitioning scheme and the vertex-separator algorithm. For the experiments

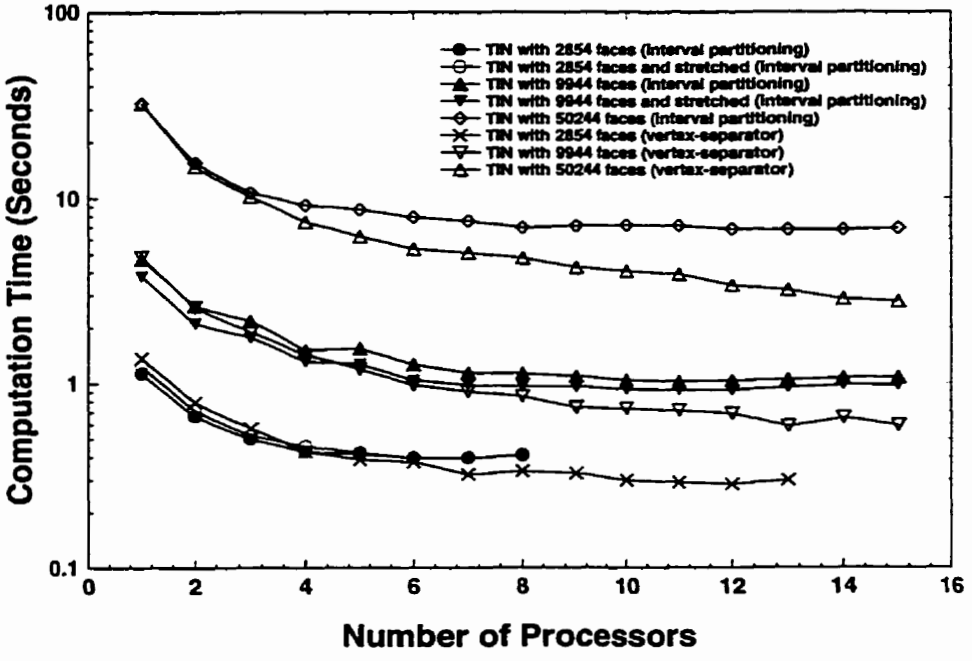


Figure 6.20: Performance of the horizon line algorithm.

done using the interval partitioning scheme and the small size TIN. the maximum gain in speedup is obtained when six processors are used. Using more than six processors, the obtained speedup starts to drop off. This behavior is similar for the case when the same TIN was stretched. For the experiments done using the interval partitioning scheme and the medium size TIN, the maximum gain in speedup is obtained when 12 processors are used and after that the speedup start to drop off. Similar behavior is also found when using the same TIN but stretched. For the experiments done the large TIN size and the interval partitioning scheme. the maximum speedup gain is also obtained when using 12 processors. Figure 6.21 shows that as the TIN size increases, the speedup obtained from parallelism also increases. Limit on the speedup that can be obtained when using the interval partitioning scheme is due to the following facts. As the number of processors used increases, the time taken to partition the data increases. In addition, the intervals assigned to the processors decrease in width causing an increase in the number of segments that are cut and ,therefore, an increase in the total number of segments that have to be processed.

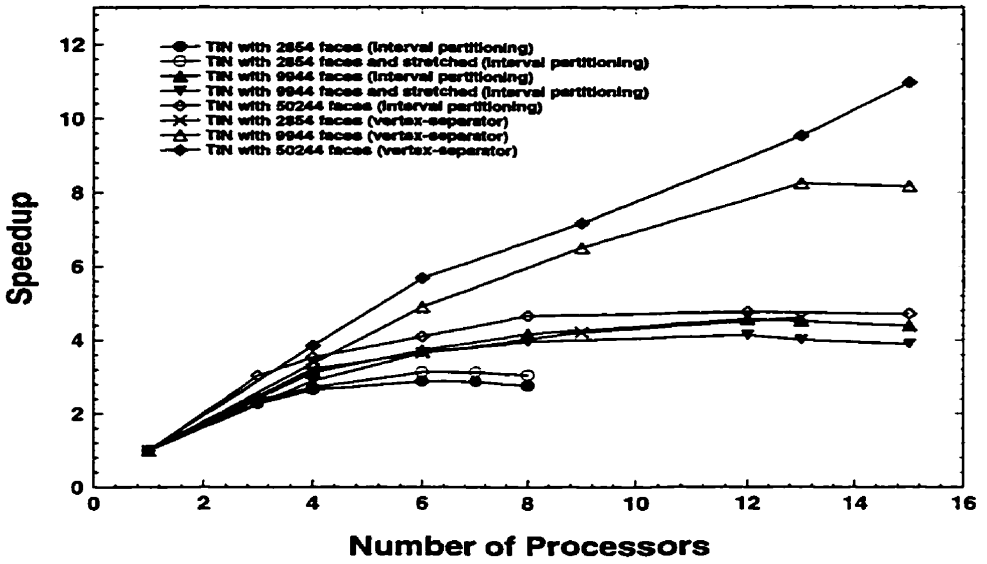


Figure 6.21: Speedup of the horizon line algorithm.

For the experiments done using the vertex-separator algorithm presented in Figure 6.21, the speedup gain obtained from parallelism increases as the TIN size increases. Figure 6.21 also shows that for the small and medium size TINs maximum speedup is obtained when using 13 processors. For the large size TIN, a continuous increase in its speedup is noticed using the maximum number of processors available. The rate of speedup increase is found to be higher when using small number of processors. This is due to the fact that as the number of TIN partitions increases, the number of partial horizon lines computed by the processors increases and ,therefore, the time needed to collect and merge these partial horizon lines increases.

It can be concluded from Figure 6.21 that the speedup obtained when using the vertex-separator algorithm was greater than the corresponding one obtained when using the interval partitioning scheme in the studied cases. Also, as the TIN size increases, the speedup obtained from parallelism increases more when using the vertex-separator algorithm than when using the interval partitioning scheme in the studied cases. Finally, it can be concluded that the performance of the horizon line algorithm in the studied cases was better when using the vertex-separator algorithm.

Table 6.2: The selected TINs attributes

Number of Vertices	Number of Edges	Data Source
1500	4353	DEM
2601	7600	Random
5000	14943	DEM

6.2 All Pairs Shortest Path

In geographic information systems, shortest path algorithms for terrains often use TINs to model the terrain. Accordingly, the experimental results of the all-pairs shortest path algorithm presented in this thesis are for TINs.

It is conceivable that the performance of the all-pairs shortest path algorithm can be affect by different TINs' characteristics. While choosing the TINs to be used in the experiments, the following different TINs characteristics were taken into consideration: number of vertices and number of edges. Three TINs of different sizes were chosen to perform the experiments upon. Table 6.2 shows a summary of the TINs used in the experiments. In the following subsection, the performance results for the preprocessing phase of the all-pairs shortest path algorithm implemented in this thesis are presented and analyzed.

6.2.1 Using Vertex-Separator

In Figure 6.22, the computation time consists of the time for:

1. Computing all-pairs shortest path for one TIN partition.

2. One processor collecting from every other processor the shortest distances between all vertices on the boundary of the partition assigned to that processor.

3. Computing the all-pairs shortest path for a graph that consists of:

   - The vertices are the boundary vertices of all TIN's partitions.

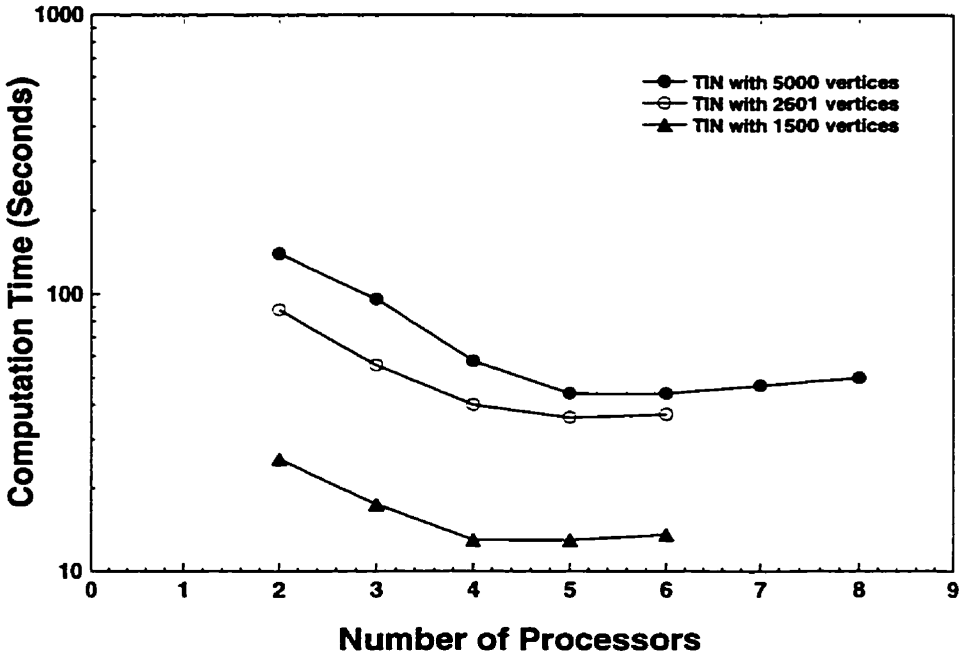


Figure 6.22: Performance of the preprocessing phase for different TIN sizes

- The edges are the shortest distances between all boundary vertices of every TIN partition.

In every experiment, the time of the slowest processor is considered the computation time of the experiment. The time for partitioning the largest TIN used in these experiments is approximately one second. The first set of experiments were repeated one or two times and the difference in computation time was insignificant; therefore, the rest of the experiments were done only once.

The results for applying the preprocessing phase of the all-pairs shortest path algorithm on TINs with different sizes are presented in Figure 6.22. This figure shows that as the TIN size increases, the computation time increases. Also, this figure shows that for every TIN as the number of processors used increases, the computation time continues to decrease until a certain point after which the computation time increases slowly. For the two TINs with 1500 vertices and 2601 vertices, the computation time continues to decrease as the number of processors used is less than four. When using five processors, the computation time remains the same.

Using more than five processors, the computation time increases slowly. For the third TIN, the TIN with 5000 vertices, the computation time continues to decrease as the number of processors used is less than five. When six processors are used, the computation time remains the same and when more than six processors are used the computation time increases slowly.

The reason for this behavior was found to be the following: As the number of processors used increases, the partitions assigned to every processor will decrease in size leading to a decrease in the time taken to compute all pairs shortest path for one partition. However, as the number of partitions increases the separator size increases and ,accordingly, the number of boundary vertices increases. As the number of boundary vertices increases, the time taken to collect the results from the processors increases. In addition, the graph formed from the boundary vertices will have a larger number of vertices and a greater number of edges. This leads to more time to compute the shortest path between all boundary vertices. At some point, this increase in time exceeds the time saved from computing all-pairs shortest path for a partition with smaller size. This causes the slight increase in the total computation time at the end of the graphs in Figure 6.22.

Finally, the following can be concluded from the results obtained above:

- The computation time of the preprocessing phase depends on the size of the TIN. As the Tin's size increases, the computation time increases.

- There is gain in performance obtained from parallelizing the preprocessing phase of the all-pairs shortest path algorithm: however, there is a limit on the maximum gain that can be obtained. This limit is related to the TIN's size and the number of processors used.

Chapter 7

Conclusion and Future Work

The main theme of this thesis is the application of parallel computing to geographic information systems. Graph partitioning theorems are studied in order to investigate the possibility of using graph partitioning algorithms to develop high performance parallel geographic information systems. The first part of this thesis focuses on graph separator theorems and a vertex-separator algorithm is implemented.

Several tests are performed to evaluate the performance of the vertex-separator algorithm. The two outputs of the vertex-separator algorithm that measure its performance are the size of the separator and the size of the output partitions. In the performed tests, the size of the separator is proved to be within the size stated in the theorem of this algorithm. In addition, the output partitions in the tests' result are balanced. The effect of changing the root of the BFS spanning tree is also investigated and it is concluded that the choice of the BFS spanning tree root vertex affects the size of the output partitions in the tests performed.

In the second part of this thesis, the performance of the vertex-separator algorithm in solving the horizon line computation problem in parallel is compared against the performance of the interval partitioning algorithm in solving the same problem. In addition, the effect of the TIN's size and structure and the viewpoint position on the performance of horizon line computation is investigated. The tests' results show that as the TIN's size increases, the gain obtained from parallelizing the problem increases. They also show that the effect of the viewpoint position and the structure of the TIN on the performance of horizon line computation is insignificant

in the tests performed.

When using the interval partitioning algorithm in solving the horizon line computation problem in parallel, a limit on the maximum speedup that can be obtained is found. This limit is due to the following facts. As the number of processors used increases, the time taken to partition the data increases. In addition, the intervals assigned to the processors decrease in width causing an increase in the number of segments that were cut and, therefore, an increase in the total number of segments that must be processed.

On the other hand, from the results obtained on the performance of the vertex-separator algorithm for solving the horizon line computation problem in parallel, it is concluded that the speedup gain obtained from parallelism increases as the TIN's size increases. The rate of speedup increase is found to be higher when using small number of processors. This is due to the fact that as the number of TIN partitions increases, the number of partial horizon lines computed by the processors increases and, therefore, the time needed to collect and merge these partial horizon lines increases.

The tests' results shows that the speedup obtained when using the vertex-separator algorithm is greater than the corresponding one obtained when using the interval partitioning algorithm in the studied cases. Also, as the TIN's size increases, the speedup obtained from parallelism increases with higher rate when using the vertex-separator algorithm than when using the interval partitioning algorithm in the studied cases. Finally, it can be concluded that the performance of the horizon line algorithm in the studied cases is better when using the vertex-separator algorithm.

In the third part of this thesis, a parallel preprocessing phase for solving the all-pairs shortest path problem is implemented and a detailed description of the shortest path queries applied on the results of this preprocessing phase is given. The vertex-separator algorithm is used in the preprocessing phase to partition the input graph among a number of processors. Several tests are performed to measure the performance of the preprocessing phase and from the tests' results it is concluded that as the TIN's size increases, the computation time increases. It is also concluded

that there is a gain in the performance obtained from parallelizing the preprocessing phase of the all-pairs shortest path algorithm; however, there is a limit on the maximum gain that can be obtained. This limit is due to the fact that as the number of processors used increase, the number of boundary vertices increases and, therefore, the graph formed from the boundary vertices will have a greater number of edges. This leads to more time to compute the shortest path between all boundary vertices. At some point, this increase in time exceeds the time saved from computing all-pairs shortest path within smaller size partitions in parallel.

From the results obtained in this thesis, it is concluded that the vertex-separator algorithm can be used to improve the performance of some GIS applications. In addition, it can be used as a preprocessing phase because it is independent of the algorithm that uses the partitioning results, i.e., it can be used to partition the input data and then several algorithms can use the results obtained from the partitioning.

7.1 List of Contributions

- Implementation of a vertex-separator algorithm which is the first implementation that the author is aware of.
- Implementation of a parallel horizon line computation application.
- Analysis of the factors that affect the performance of horizon line computation.
- A comparison of the performance of the vertex-separator algorithm in solving the horizon line computation problem in parallel against the performance of the interval partitioning algorithm in solving the same problem.
- Implementation of a parallel preprocessing phase for solving the all-pairs shortest path problem that uses the vertex-separator algorithm.

7.2 Future Work

Based on the above, the following points represent some of the future work that could be related to this thesis.

- The performance of the merging step in Phase III of the vertex-separator algorithm can be improved by using a different heuristic.
- Some modifications can be done in the vertex-separator algorithm to make it valid for graphs with genus $g \geq 0$.
- The vertex-separator approach is likely a good method for solving the region visibility problem which will do substantial improvement on the available solutions described in this thesis. The implementation of the solution for this problem is not covered within the scope of this thesis, but, similar techniques can be applied to solve the region visibility problem.
- Applying shortest path queries on the preprocessing phase results presented in this thesis.
- Using the vertex-separator algorithm as a preprocessing phase in solving other GIS applications.

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D., 1974, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

Aleksandrov, L., 1998, *Based on Close Communication.*

Aleksandrov, L. G. and Djidjev, H. N., 1990, "Separation of Graphs of Bounded Genus," Tech. Rep. 175, School Of Computer Science, Carleton University, Ottawa, Canada.

Aleksandrov, L. G. and Djidjev, H. N., 1996, "Linear Algorithms for Partitioning Embedded Graphs of Bounded Genus," *SIAM Journal on Discrete Mathematics*, Vol. 9, No. 1, pp. 129–150.

Alon, N., Seymour, P., and Thomas, R., 1990, "A Separator Theorem for Graphs With an Excluded Minor and Its applications," in: *Proceedings of the 22nd Annual A.C.M. Symposium on Theory of Computing*, Baltimore, MD.

Asano, T., Asano, T., Guibas, L., Hershberger, J., and Imai, H., 1986, "Visibility of Disjoint Polygons," *Algorithmica*, Vol. 1, No. 1, pp. 49–63.

Atallah, M., 1983, "Dynamic Computational Geometry," in: *Proceedings of the 24th Symposium on Foundations of Computer Science*, Tuscon, Ariz.

Bern, M., Dobkin, D., Eppstein, D., and Grossman, R., 1994, "Visibility With a Moving Point of View," *Algorithmica*, Vol. 11, pp. 360–378.

Bhatt, S. N. and Leighton, F. T., 1984, "A Framework for Solving VLSI Graph Layout Problems," *J. Comput System Sci*, Vol. 28, pp. 300–343.

Bloniarz, P. A., 1980, "A Shortest-Path Algorithm With Expected Time $O(n^2 \log n(\log n)^*)$," Tech. Rep. 3, Department of Computer Science, State University of New York.

Boissant. J. D. and Dobrindt, K., 1992, "On-Line Construction of the Upper Envelope of Triangles in R^3," in: *Acts of the 4th Canadian Conference on Computational Geometry*, Newfoundland, pp. 311–315.

Cazzanti, M., De Floriani, L., Puppo, E., and Nagy, G., 1991, "Visibility Computation on a Triangulated Terrain," in: *Proceedings of the 8th International Conference on Image Analysis and Processing*, Singapore.

Cole, R. and Sharir, M., 1986, "Visibiliy Problems for Polyhedral Terrains," Tech. Rep. 32, Courant Institute, New York University, New York.

Cole, R. and Sharir, M., 1989, "Visibility Problems for Polyhedral Terrains," *Journal of Symbolic Computation*, Vol. 7, pp. 11–30.

De Berg, M., 1993, *Ray Shooting, Depth Orders and Hidden Surface Removal*, Vol. 703 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany.

De Berg, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O., 1998, *Computational Geometry: Algorithms and Applications*, Springer, Berlin, Germany.

De Floriani, L., Falcidieno, B., Nagy, G., and Pienovi, C., 1989, "Polyhedral Terrain Description Using Visibility Criteria," Tech. Rep. 17, Institute For Applied Mathematics, National Research Council.

De Floriani, L. and Magillo, P., 1994, "Visibility Algorithms on Triangulated Digital Terrain Models," *International Journal of Geographical Information Systems*, Vol. 8, No. 1, pp. 13–41.

De Floriani, L., Marzano, P., and Puppo, E., 1994a, "Line-of-Sight Communication on Terrain Models," *International Journal of Geographical Information Systems*, Vol. 8, No. 4, pp. 329–342.

De Floriani, L., Montai, C., and Scopigno, R., 1994b, "Parallelizing Visibility

Computations on Triangulated Terrains," *International Journal of Geographical Information Systems*, Vol. 8, No. 6, pp. 515–531.

Deo, N. and Pang, C., 1984, "Shortest-Path Algorithms: Taxonomy and Annotation," *Networks*, Vol. 14, pp. 275–323.

Dijkstra. E. W., 1959, "A Note on Two Problems in Connexion With Graphs," *Numer. Math.*, Vol. 1, pp. 269–271.

Djidjev, H. N., 1982, "On the Problem of Partitioning Planar Graphs," *SIAM Journal on Algebraic and Discrete Methods*, Vol. 3, No. 2, pp. 229–240.

Djidjev, H. N., 1988, "Linear Algorithms for Graph Separation Problems," in: *Proceedings of 1st Scandinavian Workshop on Algorithm Theory*, Halmstad, Sweden.

Djidjev, H. N., 1997, "Partitioning Graphs With Costs And Weights On Vertices: Algorithms And Applications," in: *Lecture Notes in Computer Science*, Vol. 1284.

Djidjev, H. N. and Venkatesan, S. M., 1991, "Reduced Constants for Simple Cycle Graph Separation," Tech. Rep. 186, School Of Computer Science, Carleton University, Ottawa, Canada.

Feuerstein, E. and Marchetti-Spaccamela, A., 1993, "Dynamic Algorithms for Shortest Paths in Planar Graphs," *Theoretical Computer Science*, Vol. 116, pp. 359–371.

Floyd, R. W., 1962, "Algorithm 97: Shortest Path," *Communications of the A.C.M.*, Vol. 5.

Foley, D., Van Dam, A., Feiner, S., and Hughes, J., 1990, *Computer Graphics: Principles and Practice*, The Systems Programming Series, Addison-Wesley, Reading, Massachusetts.

Fountain, T. J., 1994, *Parallel Computing: Principles and Practice*, Cambridge University Press, University College, London.

Frederickson, G. N., 1987, "Fast Algorithms for Shortest Paths in Planar Graphs, With Applications," *SIAM Journal on Computing*, Vol. 16, No. 6, pp. 1004–1021.

Frieze, A. M. and Grimmet, G. R., 1985, "The Shortest-Path Problem for Graphs With Random Arc-Lengths," *Discrete Applied Mathematics*, Vol. 10.

Gilbert, J. R., Hutchinson, J. P., and Tarjan, R. E., 1984, "A Separator Theorem for Graphs of Bounded Genus," *Journal Of Algorithms*, Vol. 5. pp. 391–407.

Goodchild, M. F. and Lee, J., 1989, "Coverage Problems and Visibility Regions on Topographic Surfaces," *Annals of Operations Research*, Vol. 18, pp. 175–186.

Goodrich, M. T., 1992, "Planar Separators and Parallel Polygon Triangulation," in: *Proceedings of the 24th Symposium on Theory of Computing*, Victoria, BC.

Harary, F., 1969, *Graph Theory*, Addison-Wesley, Reading, MA.

Hart, S. and Sharir, M., 1986, "Nonlinearity of Davenport-Shizel Sequences and of Generalized Path Compression Schemes," *Combinatorica*, Vol. 6, pp. 151–177.

Hassin, R., 1981, "Maximum Flow in (s,t)-Planar Networks," *Information Processing Letters*, Vol. 13, pp. 107–??

Hershberger, J., 1989, "Finding the Upper Envelope of n Line Segments in $O(n \log n)$ Time," *Information Processing Letters*, Vol. 33, pp. 169–174.

Hutchinson, D., Küttner, L., Lanthier, M., Nussbaum, D., Maheshwari, A., Roytenberg, D., and Sack, J.-R., 1996, "Parallel Neighbourhood Modeling," in: *Proceedings of the Symposium on Parallel Algorithms and Applications*, pp. 204–207.

JaJa, J., 1992, *An Introduction to Parallel Algorithms*, Addison Wesley.

Johnson, D. B., 1977, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the A.C.M.*, Vol. 24, pp. 1–13.

Lee, J., 1991, "Analysis of Visibility on Topographic Surfaces," *International Journal of Geographical Information Systems*, Vol. 5, pp. 413–429.

Lee, J., 1992, "Visibility Dominance and Topographic Features on Digital Elevation Models," in: *Proceedings of the Fifth International Symposium on Spatial Data*

Handling.

Lipton, R. J., Rose, R. E., and Tarjan, R., 1979, "Generalized Nested Dissection," *SIAM Journal on Numerical Analysis*, Vol. 16, pp. 346–358.

Lipton, R. J. and Tarjan, R., 1979, "A Separator Theorem for Planar Graphs," *SIAM Journal on Applied Mathematics*, Vol. 36, pp. 177–189.

Lipton, R. J. and Tarjan, R., 1980, "Applications of a Planar Separator Theorem," *SIAM Journal on Computing*, Vol. 9, pp. 615–627.

Miller, G., 1984, "Finding Small Simple Cycle Separators for 2-Connected Planar Graphs," in: *Proceedings of The 16th Annual ACM Symposium On Theory of Computing*, pp. 376–382.

Miller, G. and Naor, J., 1991, "Flows in Planar Graphs With Multiple Sources and Sinks," in: *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pp. 112–117.

Miller, G. L., Teng, S., and Vavasis, S. A., 1991, "A Unified Geometric Approach to Graph Separators," in: *Proceedings of the 23rd Annual A.C.M. Symposium on Theory of Computing*, New Orleans, LA.

Nishizeki, T. and Chiba, N., 1988, *Planar Graphs: Theory and Algorithms*, Vol. 32 of *Annals Of Discrete Mathematics*, North-Holland Mathematics Studies, Amsterdam.

Ramalingan, G. and Reps, T., 1991, "On the Computational Complexity of Incremental Algorithms," Tech. rep., University of Wisconsin-Madison.

Spira, P. M., 1973, "A New Algorithm for Finding All Shortest Paths in A Graph of Positive Arcs in Average Time $O(n^2(\log n)^2)$," *SIAM Journal of Computing*, Vol. 2, No. 1, pp. 28–32.

Tarjan, R. E., 1983, *Data Structures and Network Algorithms*, Society For Industrial And Applied Mathematics, Philadelphia, Pennsylvania.

Teng, Y. A., Dementhon, D., and Davis, L. S., 1993, "Region-to-Region Visibil-

ity Analysis Using Data Parallel Machines," *Concurrency Practice and Experience*, Vol. 5, pp. 379–448.

Tomlin, C. D., 1990, *Geographic Information Systems and Cartographic Modeling*, Prentice Hall, Englewood Cliffs, New Jersey.

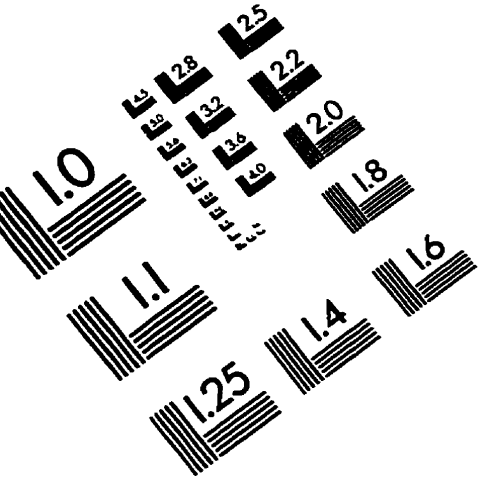
Valiant, L. G., 1990, "A Bridging Model for Parallel Computation," *Communications of the A.C.M.*, Vol. 33, No. 8.

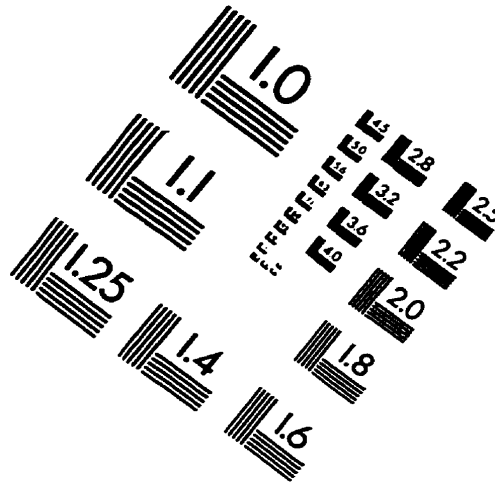
Venkatesan, S. M., 1987, "Improved Constants for Some Separator Theorems," *Journal Of Algorithms*, Vol. 8, pp. 572–578.

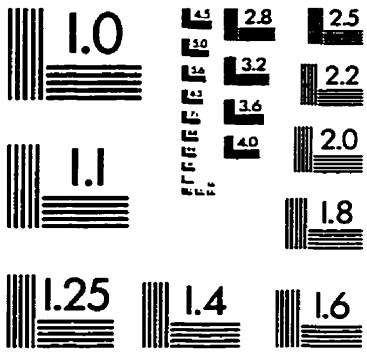
Wiernik, A. and Sharir, M., 1988, "Planar Realization of Nonlinear Davenport-Shizel Sequences by Segments," *Discrete Computational Geometry*, Vol. 3, pp. 15–47.

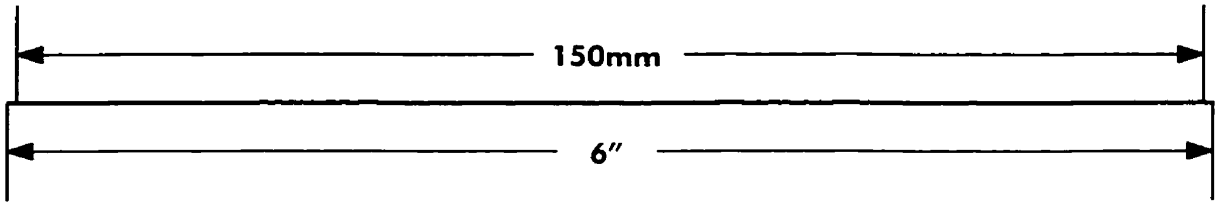
Yellin, D. and R., S., 1991, "INC: A Language for Incremental Computations," *ACM Trans. Prog. Lang. Systems*, Vol. 13, No. 2, pp. 211–236.

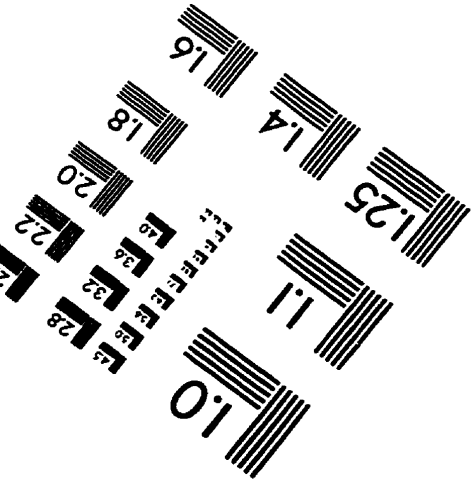
IMAGE EVALUATION
TEST TARGET (QA-3)

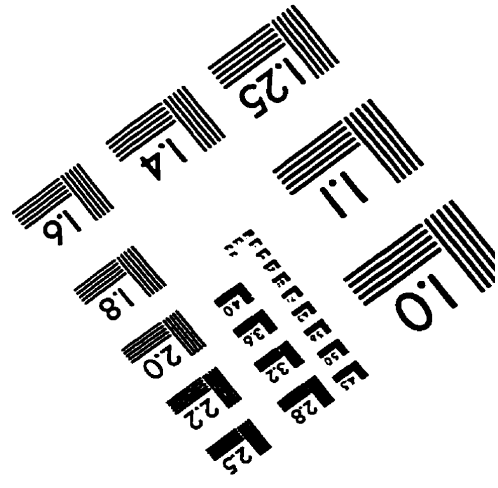












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989