A FAST HEURISTIC FOR FINDING THE MINIMUM WEIGHT TRIANGULATION

By

Ronald Beirouti

B. Sc. (Computer Science) Université de Montréal

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

....

....

THE UNIVERSITY OF BRITISH COLUMBIA

July 1997

DEPARTMENT OF COMPUTER SCIENCE
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date:

Abstract

No polynomial time algorithm is known to compute the minimum weight triangulation (*MWT*) of a point set. In this thesis we present an efficient implementation of the *LMT-skeleton* heuristic. This heuristic computes a subgraph of the *MWT* of a point set from which the *MWT* can usually be completed. For uniformly distributed sets of tens of thousands of points our algorithm constructs the exact *MWT* in expected linear time and space.

A fast heuristic, other than being usefull in areas such as stock cutting, finite element analysis, and terrain modeling, allows to experiment with different point sets in order to explore the complexity of the *MWT* problem. We present point sets constructed with this implementation such that the *LMT-skeleton* heuristic does not produce a complete graph and can not compute the *MWT* in polynomial time, or that can be used to prove the *NP*-Hardness of the *MWT* problem.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I owe my sincere gratitude to my supervisor Jack Snoeyink for his support, ideas and comments throughout my study at UBC. I would also like to thank David Kirkpatrick for reading and commenting on this thesis.

Many people have contributed to our work on minimum weight triangulations through ideas, discussions and keeping us informed. Mark Keil described the initial idea that initiated our research. Oswin Aichholzer, Siu-Wing Cheng, Matt Dickerson, and Scot Drysdale contributed valuable ideas that were tested in the code and valuable confirmation that their implementations computed the same results as ours. Jack Snoeyink, Patrice Belleville, Jit Bose, Luc Devroye, Matt Dickerson, Will Evans, Mark Keil, and Michael McAllister have all helped to design difficult instances while at or visiting UBC; Scott Drysdale and Jack Snoeyink designed the wire while at DREI workshop in Princeton. Otfried Schwarzkopf's IPE system made the UBC code a valuable tool. Franz Aurenhammer, Naoki Katoh, David Kirkpatrick, and Cao-an also contributed to discussions.

Finally, I would like to thank my wife and daughter, Katherine and Lilianne, for their patience and love throughout my studies.

This thesis was supported by a scholarship from *Le Fonds pour la Formation de Chercheurs et l'Aide la Recherche (FCAR Québec).*

Chapter 1

Introduction

One of the many properties the triangulation of a point set might have, and probably one of the first that comes to mind, is the property of minimum weight. The following definitions are presented in order to state the minimum weight triangulation problem clearly.

Definition 1.0.1 A *triangulation*, $T(S)$, of a 2-dimensional point set S is a maximum set of edges with endpoints in S such that no edges cross. The set of edges in $T(S)$ are said to *triangulate* S.

The reader can check that this concise definition gives what is expected – an embedded graph whose outer face is the convex hull of S and all other faces are triangles.

In the scope of this thesis the *weight of an edge* refers to the Euclidean length of the line segment between its two endpoints. The *weight of a triangulation* will therefore denote the total length of its edges.

Definition 1.0.2 The *weight of a triangulation* $T(S)$ is the sum of the weights of all edges : $w(T(S)) = \sum_{e \in T(S)} w(e)$.

Definition 1.0.3 A *minimum weight triangulation* of a point set S is a triangulation whose weight is minimum: $w(MWT(S)) = \min_{\forall\, T(S)}(w(T(S)))$. The set of edges in $MWT(S)$ is said to *triangulate S minimally.*

This leads to the statement of the problem: find a set of edges T that triangulates S minimally.

No polynomial time algorithm is known for computing the solution of this problem, nor has it been proven that the problem is NP-hard. In fact this problem is one of the few problems stated in Garey and Johnson's book on NP-completeness [GJ79] whose complexity status is still unknown.

This thesis will present an efficient implementation of a heuristic that calculates an exact minimum weight triangulation of most point sets. For all uniformly distributed point sets we tested the heuristic on, we obtained a minimum weight triangulation. However one can construct point sets for which the heuristic does not produce a triangulation. This thesis will also illustrate such examples.

Chapter 2

History

The problem of finding a minimal weight triangulation of a point set has attracted a lot of interest and research. It is a very interesting problem because minimal weight is a natural property of a triangulation and no polynomial time algorithm is known to solve it.

One of three directions is usually taken when addressing this problem. One can look for efficient algorithms that compute the *MWT* for restricted classes of point sets. Otherwise, one can find algorithms that compute triangulations that approximate the weight of the *MWT*. Finally, one can find algorithms that identify edges that must be in a *MWT* and try to construct the optimal triangulation from these edges.

The following sections present the various work done towards computing efficiently the *MWT* of a point set through the different directions.

2.1 *MWT* of Restricted Classes of Point Sets

In considering restricted classes of point sets, Gilbert [Gil79] and Klincsek [Kli80] independently presented a dynamic programming algorithm that computes a minimum weight triangulation of a simple polygon in $O(n^3)$ time.

Recently, Anagnostou and Corneil [AC93] described an $O(n^{3k+1})$ time algorithm that computes the *MWT* of a point set that can be the vertices of k nested convex polygons. Many others have applied dynamic programming with branch and bound techniques to the general problem. Cheng, Golin and Tsang [CGT95] proposed a dynamic programming algorithm that completes a subgraph of a minimum weight triangulation composed of k unconnected components in $O(n^{k+2})$ time.

2.2 Heuristics That Approximate the MWT

It is legitimate to ask if any of triangulations like the Delaunay triangulation or greedy triangulation that have polynomial time algorithms are minimum weight triangulations or are a constant factor approximation of the MWT.

Lloyd [Llo77] showed that in general the Delaunay triangulation is not a minimum weight triangulation. In fact, the Delaunay triangulation does not produce a constant factor approximation of the MWT. Kirkpatrick [Kir80] showed that for each n there exists a set of n points such that the Delaunay triangulation is $\Omega(n)$ times longer than the MWT.

Lloyd [Llo77] also showed that the greedy triangulation is not the minimum weight triangulation. Levcopoulos [Lev87] showed that it does not approximate the MWT better than by a $\Omega(\sqrt{n})$ factor.

Since known triangulations do not provide good approximations of the MWT, work has been done to find algorithms that compute better approximations. Plaisted and Hong [PH87] proposed a heuristic that approximates the MWT with a factor of at most $\Omega(\log n)$. To compute this triangulation the algorithm took $O(n^2 \log n)$ time in the worst case.

Other heuristics were proposed to approximate the minimum weight triangulation. The minimum spanning tree heuristic constructs a triangulation by including the edges in the minimum spanning tree and the edges of the convex hull of a point set. The result is a connected graph where polygonal holes can be completed with the polygon MWT

dynamic programming algorithm described in Section 4.3. The greedy spanning tree heuristic constructs a triangulation similarly but uses the edges of the greedy spanning tree instead. Levcopoulos and Krznaric [CL96] showed that these heuristics produce triangulations that are respectively $\Omega(\sqrt{n})$ and $\Omega(n)$ longer than the *MWT*.

Lingas [Lin85], Levcopoulos et al. [LLS89] and Levcopoulos and Krznaric [LK97] have an in-depth study of the *MWT* of convex polygons. The last paper [LK97] presents an algorithm computing a $(1+\epsilon)$ approximation of the *MWT* of convex polygons in linear time, for any fixed ϵ.

2.3 Subgraphs of the *MWT*

Another direction of attacking the *MWT* problem is by constructing a subgraph of the *MWT*. If the subgraph is connected the *MWT* can be completed by the dynamic programming algorithm presented in Section 4.3.

The β-skeleton, $\beta \geq 1$ of a point set S is the set of edges with endpoints in S such that for each edge $\overline{ab}$ the two circles of radius $\frac{\beta}{2}|\overline{ab}|$ passing through a and b are empty of all points as illustrated in Figure 2.1. The β-skeleton is the Delaunay triangulation when $\beta = 1$ otherwise it is a subgraph of the Delaunay triangulation.

Keil [Kei94] showed that the β-skeleton is also a subgraph of the *MWT* when $\beta \leq \sqrt{2}$. Unfortunately this subgraph usually contains a lot of disconnected components.

Kyoda [Kyo96] combined branch and cut to the β-skeleton and was able to compute the *MWT* of 100 points.

Work has been done to find a smaller β to allow more edges in the subgraph. Cheng and Xu [CX96] showed that the β-skeleton is still a subgraph of the *MWT* for $\beta \leq 1.17682$. The β-skeleton remains disconnected for this value of β. There is little room for improvement of β since Keil [Kei94] also found a four point example such that the

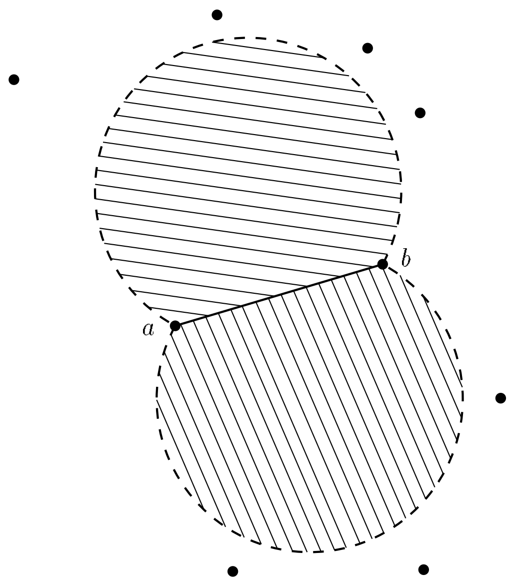

Figure 2.1: An edge of the β-skeleton with the two empty circles of radius $\frac{\beta}{2}|\overline{a}b|$.

the β-skeleton is not a subgraph of the *MWT* for $\beta < 1/\sin(\pi/3) < 1.154701$.

Recently, Keil [Kei94] and Dickerson [DM96] independently described the *LMT-skeleton*, a new subgraph of the *MWT*. Inspired by Keil, Snoeyink implemented the *LMT-skeleton* heuristic as described in Chapter 5. For uniformly distributed sets of up to 1000 points this implementation produced a connected subgraph of the *MWT* in less than half an hour. This implementation which stored all information on edges required at most $O(n^4)$ time and $O(n^2)$ space to compute the *LMT-skeleton*.

Dickerson [DM96] also implemented the heuristic. His implementation, described in Section 3.3, requires $O(n^6)$ time and $O(n^3)$ space to compute the *LMT-skeleton*. Cheng and Katoh [CK96] improved the time and space complexity of Dickerson's implementation by weakening the test.

Independently of our work, Hainz, Aichholzer, and Aurenhammer [HAA97] incorporated local tests and bucketing techniques to compute the *LMT-skeleton* in linear time and space for uniformly distributed points. However, their implementation is based on Dickerson's and still has the same worst case complexity of $O(n^6)$ time and $O(n^3)$ space. For uniformly distributed points, their implementation computes the *LMT-skeleton* of 5000 points in 20 minutes.

This thesis will describe a fast implementation of the heuristic for computing the minimum weight triangulation of a point set. It adds to Snoeyink's implementation of the *LMT-skeleton* heuristic a bucketing technique which allows it to run in expected linear time and space over uniformly distributed point sets. In the worst case this implementation will run in $O(n^4)$ time and $O(n^2)$ space.

Chapter 3

Properties of Minimum Weight Triangulation Edges

This chapter describes the basic underlying principles used by our algorithm to compute the minimum weight triangulation. We first describe the heuristic that constructs a subset of the MWT. We then describe a property shared by all MWT edges that is used before the heuristic to remove edges that cannot be in an MWT, reducing the time that the heuristic requires.

3.1 Local Minimality

Let e be an edge in a triangulation $T(S)$ that is not an edge of the convex hull of S. Then e is the *base edge* of two triangles that form a *quadrilateral* Q. If Q is convex then Q has another diagonal d that crosses e.

Definition 3.1.1 The edge e is *locally minimal* if e is on the convex hull of S, if the adjacent quadrilateral Q is not convex, or if $weight(e) \leq weight(d)$, where d is the crossing diagonal.

Definition 3.1.2 When e is locally minimal, the pair of triangles forming Q is called the *certificate* of e.

Figure 3.2 gives two examples of certificates for an edge e. On the left, e has a certificate because the empty quadrilateral is not convex. On the right, e has a certificate because it is the shortest diagonal of the empty quadrilateral.

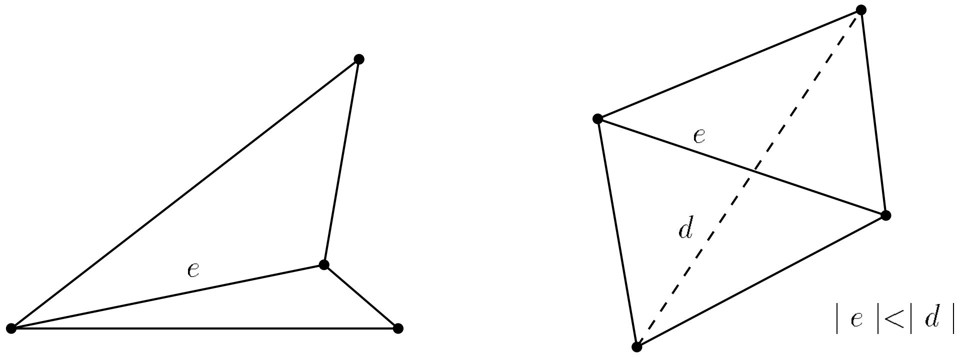

Figure 3.2: Two certificates for the edge e. The empty quadrilateral is not convex, left. The edge e is the shortest diagonal of the convex empty quadrilateral, right.

If the edge e is not locally minimal then we can decrease the weight of the triangulation by *flipping* e: removing e from $T(S)$ and replacing it by d.

Definition 3.1.3 A triangulation $T(S)$ is *a locally minimal triangulation* (LMT) if all edges in $T(S)$ are locally minimal.

The minimum weight triangulation of a point set is a locally minimal triangulation.

3.2 The LMT heuristic

As suggested independently by Keil [Kei94] and Dickerson [DM96] one can use local minimality to generate a subgraph of the MWT. The algorithm they proposed is based on the observation that edges in all locally minimal triangulations are in the minimum weight triangulation and edges that are not in any locally minimal triangulation are not in the minimum weight triangulation.

Definition 3.2.1 The *LMT-skeleton* is the set of edges that are in all locally minimal triangulations.

Given a point set S, the heuristic classifies all edges joining each pair of points in S into three categories. All edges are initially said to be *possible*, which means they are possibly in *MWT(S)*. The heuristic then considers each possible edge e and makes e *impossible* if e has no certificate. Similarly the edge e is made *certain* if e is in all LMT's which can be determined if e is on the convex hull of S or e has a certificate and no possible or certain edge crosses it.

All resulting *certain* edges form the $LMT - skeleton$ of S. If the $LMT - skeleton$ is complete, this subgraph is the *MWT(S)*, except for some non-triangulated empty *holes*. These holes are polygons empty of all points and can be triangulated using the dynamic programing algorithm for the MWT of polygons to obtain the exact *MWT(S)*.

3.3 Dickerson's View

The algorithm described by Dickerson [DM96] differs from the one described here and in [BKMS96] mainly in two points. Dickerson not only stores all edges in P but also the set of all *possible* triangles, $candTris$, which initially contains all empty triangles in P.

When checking for *certificates* for every *possible* edge e, the algorithm looks for a *certificate* of e in all the pairs of triangles in $candTris$ that border e. If no *certificate* of e is found, e is made *impossible* and all triangles containing e are removed from $candTris$ (see algorithm in 3.3).

For a general set of n points, P, there are $O(n^3)$ empty triangles and $O(n^2)$ empty edges. The time taken to list all empty triangles in P (Step 1) is $O(n^3)$. Listing all empty edges (Step 2) takes $O(n^2)$ time. Step 4 can be done in $O(n \log n)$ time. There are $O(n^2)$ pairs of triangles to consider around each edge e which, for all $O(n^2)$ possible edges, would take $O(n^4)$ time. As for the space required, we need to store all $O(n^3)$ empty triangles. The complexity of the algorithm is therefore $O(n^3)$ in space and $O(n^4)$

1. *candTris* := A list of all empty triangles in P.

2. *possibleEdges* := A list of all empty edges in P.

3. *certainEdges* := A new empty list.

4. Remove convex hull edges from the *possibleEdges* list and add them to the *certainEdges* list.

5. For each edge $e \in$ *possibleEdges*

   (a) If there does not exist a pair of triangles left and right of e, t_i and t_j, such that e is locally minimal with respect to t_i and t_j, then remove e from *possibleEdges* and remove all triangles containing e from *candTris*.

   (b) If e intersects no other edge in *possibleEdges* or *certainEdges*, then add e to the *certainEdges*

Figure 3.3: Dickerson's partial LMT-skeleton algorithm

in time.

Furthermore, since edges that are eliminated can be part of the certificate of previously checked edge, more edges might become *impossible* if Step 5 is run over again. In that sense the algorithm finds a *partial LMT-skeleton*. Dickerson suggests to find the *LMT-skeleton*, by repeating this step until no edges change status. This could be repeated at most $O(n^2)$ times which brings to $O(n^6)$ the time complexity of computing the *LMT-skeleton*.

Dickerson's implementation triangulated a uniform random distribution of 250 points in about 12 hours on a Power Macintosh 8500/120, where the space for storing the possible triangles is the bottleneck.

3.4 The Diamond Property

In order to minimize the time and space the heuristic requires to construct the *LMT-skeleton* of a point set, we use the *diamond property* to eliminate edges before applying the heuristic.

Das and Joseph [DJ89] argued that all edges in an *MWT* have the following *diamond property*.

Theorem 3.4.1 [DJ89] If an edge $\overline{ab}$ is in *MWT(S)* then at least one of the two isosceles triangles with base $\overline{ab}$ and base angles $\pi/8$ contains no points of P (see figure 3.4).

The *diamond test* eliminates from the initial possible edge set those edges with least one point in each of the two triangles of the diamond. As shown in Lemma 3.4.1, for uniformly distributed point sets, only an expected linear number of edges pass the diamond test. On average, less than $50n$ directed edges pass the diamond test for such point sets.

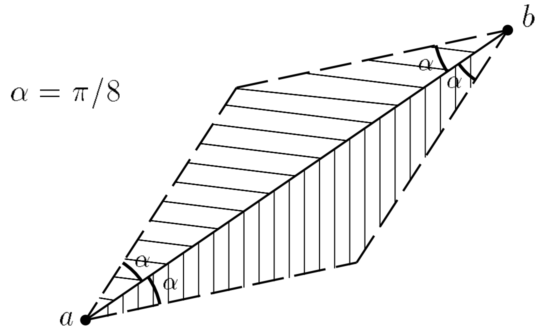

Figure 3.4: The diamond region of the line segment $\overline{ab}$. Edge $\overline{ab}$ is not in *MWT(S)* if there is a point of P in each of the isoceles triangles t_1 and t_2.

Suppose that we have a point set uniformly distributed in the unit square. If we ignore the effect of the boundary, it is not hard to calculate the average number of edges that pass the diamond test.

Lemma 3.4.1 Around one fixed point, the expected number of edges that pass the complete diamond test in a uniformly distributed point set is less than 50.

Proof Fix one point p_0 as the origin and number the remaining points by increasing distance from p_0. Let d be the distance from p_0 to p_{i+1}.
Consider the probability that point p_{i+1} passes the diamond test. The i previous points, $p_1, \ldots, p_i$, are distributed uniformly at random in the circle of radius d that is centered at p_0. Thus, they fall into a triangle of the diamond with probability $q = \sin(\pi/8)/4\pi < 0.03$. The probability that a given triangles is empty is $(1-q)^i$; that one of the two is empty is $2(1-q)^i - (1-2q)^i$, where subtracting the second term avoids double-counting diamonds that are empty.
The expected number of points that pass the diamond test with a given origin, is thus

$$\sum_{0 \le i \le n} 2(1-q)^i - (1-2q)^i = 2\frac{1-(1-q)^i}{q} - \frac{1-(1-2q)^i}{2q} < \frac{3}{2q} < 50$$

□

By first eliminating points that fail the diamond test the *LMT-skeleton* heuristic would only need to consider the expected $O(n)$ edges that passed instead of all $O(n^2)$ edges in the complete graph of S. This would dramatically improve the time and space required by the heuristic.

A straightforward implementation of the diamond test could consider each edge individually and simply check all points against the two triangles. This would take $\Theta(n^3)$ time. In chapters 6 and 7 we give methods that are faster—in chapter 6 by weakening the property to allow a more efficient test, and in chapter 7 by bucketing under the assumption that few edges pass the diamond test, as occurs in practice.

Chapter 4

Basic Data Structure and Algorithms

The following sections present the data structure and basic procedures used by the *LMT-skeleton* heuristic. The implementation of the main algorithm will be described in terms of these procedures.

4.1 The Edge Data Structure

The bottleneck of the algorithm described by Dickerson (Section 3.3) was the $O(n^3)$ space required to store all empty triangles. Our *LMT-skeleton* algorithm tries to minimize memory use by not keeping a list of empty triangles but by scanning for the empty triangles when they are needed.

Our data structure is edge-based, storing origin and destination points of the edge, two radially-sorted lists of edges around each endpoint, and two edge pointers i and j used to scan for empty triangles. We actually use the directed edge structure of Figure 4.5, which we now describe more precisely.

Between every two points a and b there are two directed edges, $\overline{ab}$ and $\overline{ba}$, where $\overline{ba} = \overline{ab} \rightarrow rev$ and $\overline{ab} = \overline{ba} \rightarrow rev$. The edge $\overline{ab}$ would hold a pointer to its destination endpoint, $\overline{ab} \rightarrow dest = b$. The point a is then $\overline{ab} \rightarrow rev \rightarrow dest$. Variable s stores the edge's status: whether the edge is *possible*, *certain* or *impossible*.

The pointer $\overline{ab} \rightarrow next$ points to the next counter-clockwise edge in the radially sorted list of edges around a; pointer $\overline{ba} \rightarrow next$ points to the next counter-clockwise edge around b. The radially sorted lists $\overline{ab} \rightarrow next$ only contains *possible* or *certain* edges; they allow

$$
\begin{array}{lll}
Edge := \{ & \text{Point ptr} & dest \\
 & \text{Edge ptr} & rev \\
 & \text{Status} & s \\
 & \text{Edge ptr} & next \\
 & \text{Edge ptr} & i \\
 & \text{Edge ptr} & j \\
 & \text{Edge ptr} & dstart \\
 & \text{Edge ptr} & rightPoly \\
 & \text{Edge ptr} & leftPoly \\
 & \text{Edge ptr} & polyWeight \\
 \} & &
\end{array}
$$

Figure 4.5: Data structure elements for one directed edge; values described in the text

us to scan edges to find empty triangles that can participate in a certificate for $\overline{ab}$.

When a certificate is found for $\overline{ab}$, the pointers $\overline{ab} \rightarrow i$ and $\overline{ab} \rightarrow j$ point to the edges of the empty triangle to the left of $\overline{ab}$ and the pointers $\overline{ab} \rightarrow rev \rightarrow i$ and $\overline{ab} \rightarrow rev \rightarrow j$ point to the two edges of the other empty triangle as seen in figure 4.6.

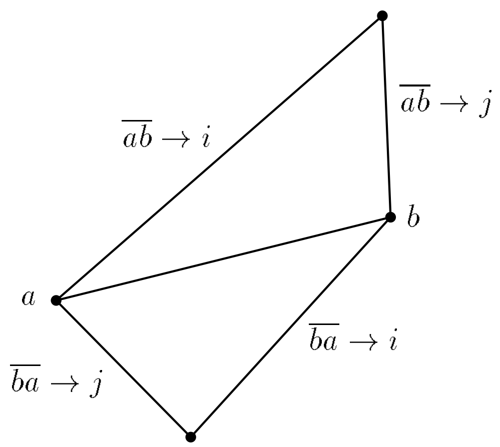

Figure 4.6: Pointers i and j identifying the certificate of $\overline{ab}$.

The edge $\overline{ab} \rightarrow dstart$, used in scanning for empty triangles, is the first edge $\overline{bc}$ clockwise around b such that c is not left of $\overline{ab}$ and $\overline{bc} \rightarrow next$ is left of $\overline{ab}$ or is $\overline{ab}$.

Finally, *rightPoly*, *leftPoly* and *polyWeight* are used to triangulate the polygonal holes remaining in the *LMT-skeleton* and will be described in Section 4.3.

4.2 Scanning for Empty Triangles

The procedure *advance* in figure 4.7 is the elementary operation used in scanning for empty triangles. Given two edges $\overline{ab} \to i$ and $\overline{ab} \to j$, *advance* finds the next pair of edges i and j forming an empty triangle with $\overline{ab}$ such that $i \to dest = j \to dest$ is left of $\overline{ab}$. The algorithm of *advance,* which scans through the radially-sorted lists to find the next empty triangle, is described in Figure 4.7. Before the first time *advance* is called on the edge $\overline{ab}$, $\overline{ab}$ is *reset*: $\overline{ab} \to i$ and $\overline{ab} \to j$ are initialized to $\overline{ab} \to next$ and $\overline{ab} \to dstart$. Advance has found the next empty triangle $\triangle abc$ when $c = ab \to i \to dest = ba \to j \to dest$.

procedure *advance(segment* $\overline{ab}$*)*
  repeat
    while $\overline{ab} \to i \to dest$ is not left of $\overline{ab} \to j$
      $\overline{ab} \to i := \overline{ab} \to i \to next$
    while $\overline{ab} \to j \to dest$ is right of $\overline{ab} \to i$
      $\overline{ab} \to j := \overline{ab} \to j \to next$
  until $\overline{ab} \to i \to dest = \overline{ab} \to j \to dest$

Figure 4.7: The *advance* procedure

To *scan* through the list of empty triangles left of $\overline{ab}$, call *advance*($\overline{ab}$) until the destination vertex of $\overline{ab} \to j$ is a. Both the *LMT-skeleton* heuristic and the polygon *MWT* algorithm of the next section use this scanning.

4.3 Minimum Weight Triangulation of Polygons: a Dynamic Programming Algorithm

Let P be a simple polygon with n vertices labelled v_0, v_1, $\ldots$, v_{n-1}. All further use of vertex indices will be implicitly modulo n. The segment $\overline{v_i v_j}$ is an *edge* of P whenever $|i - j| = 1$, otherwise the segment $\overline{v_i v_j}$ is a *diagonal* of P.

A simplified variant of the minimum weight triangulation problem is to find a *minimum weight triangulation of a polygon P, MWT(P)*. That is, find a set of diagonals and edges of P such that the sum of the weights of the diagonals and edges is minimum. As shown in [Kli80] and described in this section, there exists a dynamic programming algorithm to solve this problem.

The algorithm presented in this section will solve the following problem. Given a polygon P and a subset of *possible* diagonals D such that $MWT(P) \subseteq D \cup P$ find a set of diagonals in D that triangulate P minimally. Figure 4.8 shows an example of a polygon with diagonals and its minimum weight triangulation.

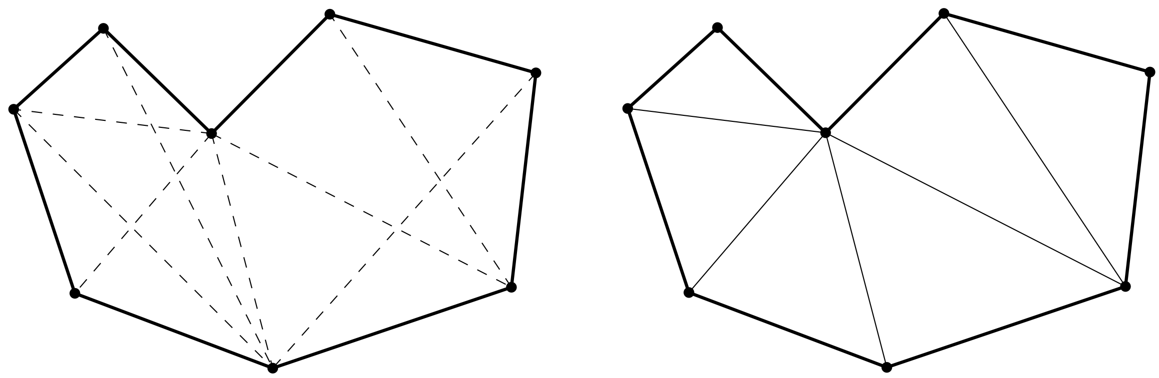

Figure 4.8: Example of a polygon and its minimum weight triangulation.

The principle of dynamic programming is to solve a problem bottom up, solving all the smaller sub-problems first and then building larger sub-problems with the solutions of smaller ones until the solution of the whole problem is found. The following definition describes the sub-problems used in the algorithm.

Definition 4.3.1 If the line segment $\overline{v_i v_j}$ is a diagonal of P then there are two subsets of $P \cup \{\overline{v_i v_j}\}$ that form polygons. The *sub-polygon* π_{ij} is the polygon such that v_j and v_i are in counter-clockwise order of vertices. The diagonals of π_{ij} is the set of diagonals in D that have both endpoints on π_{ij}.

The dynamic programming algorithm constructs the MWT of a sub-polygon using the MWT of smaller sub-polygons. The following two lemmas present its basic principle.

Lemma 4.3.1 Let $\overline{v_i v_j}$ be a diagonal or edge of a polygon P, if $j-i=1$ then $MWT(\pi_{ij}) = \{\overline{v_i v_k}\}$. Otherwise if $j-i>1$ then $MWT(\pi_{ij}) = \{\overline{v_i v_j}\} \cup MWT(\pi_{ik}) \cup MWT(\pi_{kj})$ for k satisfying $\min_{i<k<j}(w(MWT(\pi_{ik})) + w(MWT(\pi_{kj})))$ and such that $\overline{v_i v_k}$ and $\overline{v_k v_j}$ are edges or diagonals of π_{ij}.

Proof For all k such that $\overline{v_i v_k}$ and $\overline{v_k v_j}$ are in $D \cup P$ the weight of the triangulation of π_{ij} is $w(T(\pi_{ik})) + w(T(\pi_{kj})) + w(\overline{v_i v_j})$. For this sum to be minimal $w(T(\pi_{ik}))$ and $w(T(\pi_{kj}))$ must be minimal. $MWT(\pi_{ij})$ is therefore the triangulation such that $w(MWT(\pi_{ik})) + w(MWT(\pi_{kj})) + w(\overline{v_i v_j})$ is minimal for all k such that $\overline{v_i v_k}$ and $\overline{v_k v_j}$ are in $P \cup D$. $\Box$

Lemma 4.3.2 The minimum weight triangulation of the polygon P, $MWT(P) = \pi_{ij}$ for all $i = j+1$.

Proof The line segment $\overline{v_i v_j}$ is an edge of P when $i = j+1$ such that $\pi_{ij} = P$. Therefore π_{ij} is the minimum weight triangulation of P. $\Box$

4.3.1 Algorithmic Details for Dynamic Programming

The algorithm uses the data structure described in Section 4.1 to store polygon diagonals and edges. If $\overline{v_i v_j}$ is a diagonal, then the field $\overline{v_i v_j} \rightarrow$*polyWeight* stores the weight of minimum weight triangulation of π_{ij}. The two pointers $\overline{v_i v_j} \rightarrow$ *leftPoly* and $\overline{v_i v_j} \rightarrow$ *rightPoly* hold the edges $\overline{v_i v_k}$ and $\overline{v_k v_j}$ in $MWT(\pi_{ij})$.

As mentioned in Section 4.2, the procedure *scan* uses *advance* to visit all empty triangles to the left of $\overline{v_i v_j}$. The code in Figure 4.9 also checks empty triangles to find the one that can be in $MWT(\pi_{ij})$. *Scan* is applied on every edge and diagonal, starting with those with endpoint indices $j - i = 2$ and continuing incrementally with edges that satisfy $j - i = 3, 4, \ldots, n - 2$. *Scan* is finally applied on one edge such that $j - i = n - 1$. The pointers *rightSubTri* and *leftSubTri* of this last edge then form a binary tree that holds all the edges of an $MWT(P)$.

procedure $scan(\overline{v_i v_j})$

$a := \overline{v_i v_j} \rightarrow next$
$b := \overline{v_i v_j} \rightarrow dstart$

$weight := \infty$

while $a \rightarrow dest$ is left of $\overline{v_i v_j}$
  $advance(\overline{v_i v_j})$
  if $w(MWT(\pi_a)) + w(MWT(\pi_b)) + w(\overline{v_i v_j}) < weight$ **then**
    $weight := w(MWT(\pi_a)) + w(MWT(\pi_b)) + w(\overline{v_i v_j})$
    $rightSubTri := b$
    $leftSubTri := a$

Figure 4.9: The *scan* procedure for the *MWT* of polygons

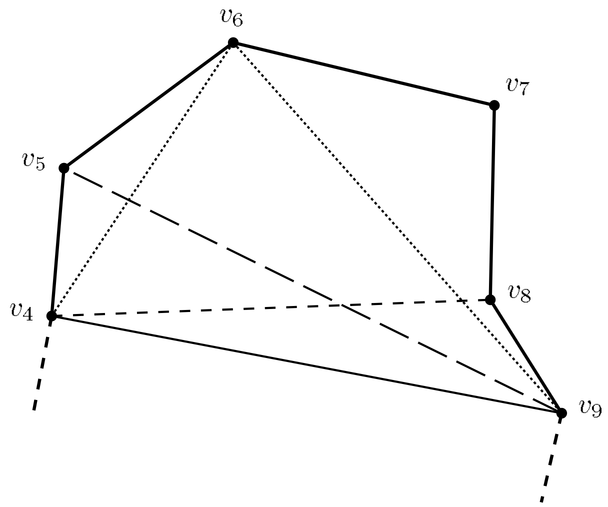

Figure 4.10: The minimum weight triangulation of a sub-polygon.

4.3.2 Complexity Analysis

Lemma 4.3.3 The time required by the *scan* procedure applied on an edge $\overline{v_i v_j}$ is $O(d)$ where d is the maximum degree of the vertices v_i and v_j.

Proof *Scan* visits each edge of the two radially sorted lists at most once. Sideness tests are done in constant time. With at most d segments in each list, scanning edge $\overline{v_i v_j}$ takes at most $O(d)$ time. □

Lemma 4.3.4 The time required to compute the minimum weight triangulation of a polygon P is $O(dm)$ where $m = |D \cup P|$ is the number of diagonals in D and edges in P, and d is the maximum degree of vertices of P.

Proof *Scan* is applied once for each edge or diagonal. From Lemma 4.3.3 each scan takes at most $O(d)$ time. For m edges and diagonals, the $MWT(P)$ algorithm therefore completes in $O(md)$ time. □

4.3.3 The Sum of Square Roots Problem

In presenting this dynamic programming algorithm, we have implicitly assumed that one can evaluate sums of radicals. Specifically, consider the sum

$$S = \sum_{i=1}^{k} c_i \sqrt{q_i}$$

where c_i and $q_i \in \mathcal{R}$. Determining if S is less, equal or greater than zero is commonly known as the *sum of square roots problem.* No polynomial time algorithm is known to solve this problem [Blö91] for machine models that are not given square roots as a primitive operation.

When the dynamic programming algorithm selects which triangulation of sub-polygon π_{ij} has minimum weight, it needs to decide if

$$w(T(\pi_{ik})) + w(T(\pi_{kj})) > w(T(\pi_{il})) + w(T(\pi_{lj}))$$

to know that it should use vertex p_k rather than p_l. The weight of each triangulation is a sum of square roots since the weight of each edge is the Euclidean distance between its endpoints. Thus, evaluating this inequality is equivalent to determining if a sum of square roots is positive.

The sum of square roots problem is common in geometric optimization problems that involve computing Euclidean lengths. In the the rest of this thesis we assume that the dynamic programming algorithm can use square roots as a primitive and, thus, compute the exact MWT. It is notable that the heuristic does not involve comparing sums of square roots, but works with at most two radicals at a time. This allows it to be implemented exactly, even in fixed precision models of computation.

Chapter 5

The *LMT-skeleton* Algorithm

Our implementation of the *LMT-skeleton* heuristic presented in Section 3.2 does not store the $O(n^3)$ empty triangles, as does the algorithm described by Dickerson [DM96]. Instead, all information is stored around the edge data structure presented in the previous chapter. Furthermore, the algorithm does not need to be repeated until no further edges are eliminated. For sets of tens of thousands of uniformly distributed points, the algorithm computes a connected graph that can be completed with the polygon MWT dynamic programming algorithm.

This chapter will describe the different parts of the *LMT-skeleton* algorithm before presenting the main algorithm.

5.1 Initializing the Data Structure

The algorithm must first create edges between each pair of points. The three main components to initialize in the edge data structure are the radially-sorted lists of possible edges around each point, the pointers to the reverse edges $\overline{ab} \rightarrow rev$ and the $dstart$ pointers used for checking certificates.

Given a point set S the list around a point o contains initially the edges between all points in $S \backslash \{o\}$ and o. These edges are inserted in the list and then sorted radially in counter-clockwise order.

Lemma 5.1.1 Given the set of n points S and the set of m edges E, all radially-sorted lists can be initialized in $O(nd \log d)$ time where d is the maximum degree of the graph.

Proof Since the maximum degree of the resulting graph is d there are at most d edges in each list. Around each point, the time taken for insertion is therefore $O(d)$. Each list of at most d edges can be sorted in $O(d \log d)$ time. To sort all n lists therefore takes $O(nd \log d)$ time. □

To initialize the reverse pointer of an edge $\overline{ab}$, $\overline{ab} \rightarrow rev$, the algorithm checks if the edge list around the point b has been created. If so, it looks through this list for the reverse edge $\overline{ba}$ and sets both pointers $\overline{ab} \rightarrow rev$ and $\overline{ba} \rightarrow rev$. If the list around b has not yet been initialized nothing is done: $\overline{ab} \rightarrow rev$ will be initialized later when edge $\overline{ba}$ is initialized.

Lemma 5.1.2 Given the set of n points S and the set of m edges E, intitializing all reverse pointers takes at most $O(m \log d)$ time where d is the maximum degree of the resulting graph.

Proof Given edge $\overline{ab}$, to find edge $\overline{ba}$ the program needs to search through the radially-sorted list of edges around b. Each of these searches takes $O(\log d)$ using binary search. This search is done for half of all the m edges therefore requiring at most $O(m \log d)$ time. □

The $\overline{ab} \rightarrow dstart$ pointer is initialized to the last edge around b such that $\overline{ab} \rightarrow dstart \rightarrow dest$ is not left of $\overline{ab}$. To find the edge $\overline{ab} \rightarrow dstart$ the algorithm scans through the list of edges around b starting with $\overline{ba}$ until it finds an edge with endpoint left of $\overline{ab}$.

Lemma 5.1.3 Given the graph G that contains m edges, initializing all *dstart* pointers takes at most $O(m)$ time.

Proof Once the *rev* pointers are established, one can assign *dstart* pointers by scanning the list of edges around every vertex with two pointers that are maintained at 180°. These scans look at every edge at most twice on each end. Thus, to intialize the *dstart* pointers for all m edges therefore takes at most $O(m)$ time. □

Lemma 5.1.4 Given a set S of n points, initializing all edges between each pair of points takes at most $O(n^3)$ time.

Proof There are $O(n^2)$ edges between pairs of points and there are $O(n)$ edges in the list around each point therefore the maximum degree d is $O(n)$. From Lemma 5.1.1 creating the sorted lists takes $O(n^2 \log n)$ time. Initializing the reverse pointers can be done in $O(n^2 \log n)$ time from Lemma 5.1.2 while Lemma 5.1.3 shows that initializing the *dstart* pointers takes at most $O(n^2)$ since the number of edges $m = n^2$. Therefore the whole initialization process takes at most $O(n^2 \log n)$ time. □

5.2 Checking If an Edge Has a Certificate

The procedure *check certificate* uses *advance* to find the next certificate of an edge $\overline{ab}$. This certificate is a pair of empty triangles $\triangle abc$ and $\triangle bad$ where c is left of $\overline{ab}$ and d is right of $\overline{ab}$ and such that $\overline{ab}$ is the shortest edge of the quadrilateral Q formed by the two triangles or the quadrilateral Q is not convex.

The first time *check certificate* is executed on edge $\overline{ab}$, the pointers $\overline{ab} \to i$, $\overline{ab} \to j$, $\overline{ba} \to i$ and $\overline{ba} \to j$ need to be initialized. $\overline{ab} \to i$ and $\overline{ab} \to j$ initially point respectively to $\overline{ab} \to next$and $\overline{ab} \to dstart$ which are the first edges in the lists around a and b such that their endpoint is left of $\overline{ab}$. This operation will be called to *reset* $\overline{ab}$. Similarly, $\overline{ba}$ is *reset*.

As described in Figure 5.11, *check certificate* advances to the next triangle that is left of $\overline{ab}$ until a certificate is found. If all the empty triangles left of $\overline{ab}$ were traversed without finding a certificate, then *check certificate resets* $\overline{ab}$ and *advances* $\overline{ba}$. This is repeated until a certificate is found or until all empty triangles left $\overline{ba}$ were traversed. In this later case, all pairs of empty triangles have been tested and $\overline{ab}$ has no certificate.

One of the edges of the certificate of $\overline{ab}$ can become impossible further in the heuristic.

procedure *check certificate*($\overline{ab}$)

if *check certificate* was never applied on $\overline{ab}$
  reset($\overline{ab}$); *reset*($\overline{ba}$); *advance*($\overline{ba}$)

while no certificate is found
  advance($\overline{ab}$)
  if all triangles left of $\overline{ba}$ were visited
    $\overline{ab}$ has no certificate; **break**
  if all triangles left of $\overline{ab}$ were visited
    advance($\overline{ba}$); *reset*($\overline{ab}$)
  else
    if the quadrilateral Q formed by the two triangles is not convex
        or $\overline{ab}$ is the shorter diagonal of Q
      $\overline{ab}$ has a certificate; **break**

Figure 5.11: The *check certificate* procedure

If so, *check certificate* is called again and tries to find the next certificate of $\overline{ab}$ without reseting the pointers i and j. This avoids visiting a pair of empty triangles more than once.

Lemma 5.2.1 The time taken by *check certificate* on the edge $\overline{ab}$ is at most $O(d^2)$, where d is the maximum degree of points a and b.

Proof If the number of edges in the lists around a and b is smaller than or equal to d then there are at most d triangles with edge $\overline{ab}$. Therefore there are less than d^2 different pairs of triangles. In the worst case where $\overline{ab}$ has no certificate all pairs of triangles are considered thus requiring $O(d^2)$ time. □

5.3 Checking for Crossing Edges

An edge $\overline{ab}$ is certain if it is possible and no other *possible* or *certain* edges cross it. The procedure *check crossing edges* verifies if there exists such an edge that crosses $\overline{ab}$.

In a general set of line segments, this is an operation that must either test every segment against every other, or use large or complicated data structures for an efficient test. We can make use of the fact that the possible edges always contain a triangulation—if $\overline{ab}$ can be omitted while still completing the triangulation, then there is a triangle incident on a that is crossed by $\overline{ab}$. Thus, the algorithm looks at all edges in the list around a. Edge $\overline{ab}$ has a crossing edge if, and only if, for some edge $\overline{ac}$ in this list there is an edge $\overline{cd}$ around c such that $\overline{cd}$ crosses $\overline{ab}$. The procedure *check crossing edges* is further described in Figure 5.12.

procedure *check crossing edges*($\overline{ab}$)

$u := \overline{ab} \rightarrow next$
while $u \rightarrow dest$ is not left of $\overline{ab}$
    $v := u \rightarrow rev$
    while b is left of v
        if v crosses $\overline{ab}$
            return $\overline{ab}$ has crossing edge
        $v := v \rightarrow next$
    $u := u \rightarrow next$
return $\overline{ab}$ has no crossing edge

Figure 5.12: The *check crossing edges* procedure on edge $\overline{ab}$

Lemma 5.3.1 Given a graph G with maximum degree d, the *check crossing edge* procedure takes at most $O(d^2)$ time for any edge in G.

Proof Since there are at most d edges around any point in the graph G each of the two nested loops are repeated at most d times. The *check crossing edges* procedure therefore takes $O(d^2)$ time. □

5.4 Lazy Deletion of Edges

An edge that is found to be *impossible* could be removed from the radially-sorted lists in order to reduce their size and reduce the runtime of the procedures that use those lists. However, several pointers from other edges can be directed to this edge. Therefore, when an edge is found to be *impossible*, only its status s is changed. It is removed from the list later through lazy deletion.

Whenever a procedure scans through the radially-sorted lists of edges it applies the *lazy delete* to each edge. This procedure checks if the next edge $\overline{ab} \rightarrow next$ is already marked *impossible*. If so, then it sets $\overline{ab} \rightarrow next$ to $\overline{ab} \rightarrow next \rightarrow next$.

5.5 Restacking Edges Whose Certificates Became Invalid

When an edge $\overline{ab}$ is found to be *impossible* and is part of the certificate of another edge $\overline{cd}$, this certificate is no longer valid. Once a certificate becomes invalid, it will never become valid again.

The procedure *restack edges* looks for all the edges having $\overline{ab}$ as part of their certificate and pushes them onto the stack of edges that need their certificates checked.

Restack edges scans through all empty triangles with edge $\overline{ab}$ and stacks the edges of triangles if their i or j pointers do point to $\overline{ab}$ or $\overline{ba}$. The same procedure is also applied to edge $\overline{ba}$. Figure 5.13 describes the details of *restack edges*.

This procedure allows the algorithm to compute what Dickerson calls the *extended LMT-skeleton* without repeating the whole algorithm $O(n^2)$ times. Pointers i and j not only facilitate finding the edges whose certificates are no longer valid, but also let *check certificate* resume from the last certificate.

Lemma 5.5.1 Given an edge $\overline{ab}$, the procedure restack edges takes at most $O(d)$ time where d is the maximum degree of the points a and b.

procedure *restack edges*($\overline{ab}$)

$reset(\overline{ab})$
for all empty triangles left of $\overline{ab}$
  $advance(\overline{ab})$
  if $\overline{ab} \to i \to rev \to j = \overline{ab}$
    push $\overline{ab} \to i$
  if $\overline{ab} \to j \to i = \overline{ab} \to rev$
    push $\overline{ab} \to j$

Figure 5.13: The *restack edges* procedure on edge $\overline{ab}$

Proof As in Lemma 5.2.1 the number of empty triangles with edges $\overline{ab}$ is at most $O(d)$. Restack edges scans only once through these triangles, thus taking at most $O(d)$ time. □

5.6 The *LMT-skeleton* Algorithm

The tools needed to construct the *LMT-skeleton* were described in the previous sections. This section describes the main algorithm in terms of the procedures defined previously while Figure 5.14 sums up the main steps.

The algorithm first initializes the edge data structures as described in Section 5.1. Then all edges are pushed on the stack of edges that need to be checked for certificates. The stack is sorted so that *check certificate* is applied to the longest edges first since they are more likely to have no certificates.

Each edge $\overline{ab}$ of the stack is then popped and the procedure *check certificate* attempts to find a certificate for $\overline{ab}$. If no certificate is found $\overline{ab}$ is either a convex hull edge or is impossible. So *check crossing edges* is applied to find a possible edge that crosses $\overline{ab}$. If no crossing edge is found then $\overline{ab}$ is a convex hull edge and its status is changed to *certain*. Otherwise the status is set to *impossible*. In this case $\overline{ab}$ can be part of the certificate of another edge. The procedure *restack edges* is therefore applied to find all the edges whose certificates have become invalid. Those edges are pushed back on the stack.

Initialize data structures
Push all edges onto the stack ST
Sort ST so that longest edges are on top
while ST is not empty
   e := pop ST
   check certificate of e
   if e has no certificate
      check crossing edges
      if e has no crossing edge
         $e \rightarrow s := certain$
      else
         $e \rightarrow s := impossible$
         restack edges

Figure 5.14: The *LMT-skeleton* algorithm

When the stack is empty the algorithm then considers all remaining *possible* edges. These edges have certificates and *check crossing segments* is applied on each. If an edge is *possible* and has no crossing edges then it is marked *certain*. Otherwise the heuristic can not determine if it is in the minimum weight triangulation. The *LMT-skeleton* is then the set of all certain edges.

The sets of edges that remain *possible* can be isolated in polygonal regions. If these regions are simply connected, then the minimum weight triangulation can be completed by the algorithm described in Section 4.3.

5.7 Complexity Analysis

The *LMT-skeleton* heuristic starts with the $O(n^2)$ edges joining all pairs of points in the set S of size n. The degree of each point in the initial graph is $n-1$.

Lemma 5.7.1 The algorithm described in this section takes at most $O(n^4)$ time to calculate the *LMT-skeleton* of n points.

Proof From Lemma 5.1.4 initializing the data structures takes $O(n^3)$ time. The procedure *check certificate* is the most expensive to compute. For each edge this operation takes at most $O(n^2)$ time; therefore $O(n^4)$ time is sufficient for the $O(n^2)$ edges. $\square$

Lemma 5.7.2 The algorithm described in this section takes $O(n^2)$ space to compute the *LMT-skeleton* of n points.

Proof The algorithm can easily store the $O(n^2)$ edges and $O(n)$ points in $O(n^2)$ space. $\square$

5.8 Experimental Results

The algorithm was implemented in *cweb* and run on an *SGI Indy* [BKMS96]. To correctly perform all arithmetic operations in 53-bit double precision, the input points were scaled to 20 bit positive integers. Degeneracies, as colinearities and equal lengths were handled.

Trials showed that for sets of uniformly distributed random points, the algorithm computes the complete *MWT*. For 250 and 1000 points, the algorithm completed in, respectively 25 seconds and half an hour. The observed time was proportional to $n^3 \log n$. The bottleneck is definitely the $\Theta(n^2)$ space required.

Chapter 6

The Diamond Test Algorithm

In Section 3.4 we presented the diamond property, a local property that can identify edges that are not in an *MWT*. In this and the next chapter we present two algorithms that eliminate the edges that don't satisfy the diamond property. This pretest is run before the *LMT-skeleton* heuristic is applied to reduce the running time and space required by the heuristic.

The *diamond test* verifies if an edge possesses the diamond property. This chapter describes an efficient way of eliminating edges that fail the diamond test. The diamond test is applied to the edges while initializing edges, before applying the *LMT-skeleton* heuristic.

6.1 Applying the Diamond Test

When initializing the edge data structures, *quick sort* is used to sort edges in counter-clockwise order in the circular edge lists. The sorting algorithm also applies the diamond test on the edges being sorted. At each partitioning step *quick sort* chooses a pivot p. It then scans through the list of edges to find the first one that must be clockwise of p. While scanning edges, the diamond test is applied to each edge traversed with respect to p. In other words, for each edge e traversed the algorithm checks if the endpoint of p lies within one of the two isosceles triangles of e. If so, e is marked accordingly. If at another partitioning step the endpoint of the pivot lies in the other isosceles triangle, the edge is removed from the list. Symmetrically, *quick sort* scans clockwise for the first edge that must be counter-clockwise of p and marks edges when the endpoint of p lies in one of their diamond triangles.

The pivot is always chosen as the shortest edge in the partition, which enables the pivot to kill the most edges. In random point sets it also reduces the chance that the worst case behavior of *quick sort* occurs since the position of the pivot in the radially sorted list is independent of the length of the edge.

The diamond test as applied by *quick sort* is weaker than the test proposed by Das and Joseph [DJ89]. Because an edge in a partition is tested only against the pivot endpoint, edges are not tested for the diamond property with regard to all points. In particular, endpoints of edges that are killed at one partitioning step are never used to kill other edges.

One can apply a *complete* diamond test by removing killed edges only after the list is sorted and by testing all edges against each pivot. With this approach, however, sorting takes longer and the diamond test takes cubic time in the average case; we found it better to speed up the test at the expense of letting a few more edges pass.

6.2 Complexity analysis

Performing the diamond test doesn't change the $O(n^4)$ worst-case complexity of the algorithm. In a typical case, however, it eliminates a significant number of edges. In fact, for uniformly distributed random point sets, we observed that only $O(n)$ edges passed the diamond test. After the test, the average degree of each point was indeed bounded by 50, even with the weaker test.

Lemma 6.2.1 Given the set of n points S, let the graph G over S contain all edges that pass the diamond test. Initializing data structures takes at most $O(n^2 \log d)$ time, or $O(nd \log d)$ time after sorting edges radially, where d is the maximum degree of G.

Proof Around each vertex, the n edges can be sorted by a combination of quicksort and the diamond test in $O(n \log d)$ time. (To make this true in the worst case, one should use medians to choose pivots in every other step.)
The algorithm initializes the *rev* and *dstart* pointers only after the edge lists are sorted and the diamond test applied. Initializing *rev* pointers takes $O(nd \log d)$ total time by Lemma 5.1.2; initializing *dstart* pointers takes $O(nd)$ time by Lemma 5.1.3.
The overall time taken for initialization is therefore $O(n^2 \log d + nd \log d)$. □

Lemma 6.2.2 Given the initialized data structure, the *LMT-skeleton* algorithm takes at most $O(d^3 n)$ time where d is the maximum degree of the initial graph and n is the number of points.

Proof For each edge the procedures *check certificate* and *check crossing edges* are the most expensive, requiring at most $O(d^2)$ time. For all $O(nd)$ edges the *LMT-skeleton* heuristic therefore takes at most $O(d^3 n)$ time to complete. □

Note that to achieve the $O(d^3 n)$ time bound the algorithm has to omit sorting the edges in order of length before checking for certificates. However in practice it is more efficient to sort edges.

Lemma 6.2.3 If the maximum degree d of the graph obtained after the diamond test is applied is constant then computing the *LMT-skeleton* heuristic takes at most $O(n^2)$.

Proof From Lemma 6.2.1, if d is constant then the time required for initialization is $O(n^2)$. From Lemma 6.2.2 the rest of the heuristic can be computed in worst case linear time. The overall heuristic therefore takes $O(n^2)$ time. □

If the graph containing all edges that passed the diamond test has constant degree and therefore the number of edges that pass the diamond test is linear then the rest of the algorithm can be computed in linear time. The bottleneck becomes the time to sort edges around points and apply the diamond test, which takes $O(n^2 \log d)$ time.

6.3 Experimental Results

The algorithm was implemented in *cweb* and run on an *SGI XZ* with a 200 *MHz IP22* processor. The algorithm was run on several sets of uniformly distributed random points ranging from 100 to 40000 points.

For such point sets the degree of the points of the graph obtained after applying the diamond test is constant. For n points, about $45n$ directed edges do pass the diamond test.

For every point set a complete graph was obtained, that means the algorithm was able to compute *MWT(S)* by triangulating the unresolved polygonal holes. Table 6.1 and Figures 6.15 and 6.3 give a summary of the times observed as well as the number of edges that passed the diamond test, while Figure 6.17 illustrates the exact *MWT* of a random uniformly distributed point set.

n	# edges pass diamond test	avg degree after test	time in secs
100	1511	30.21	0.18
200	3711	37.11	0.60
400	8088	40.44	2.58
800	12227	42.43	4.14
1000	21386	42.77	7.32
2000	44846	44.85	11.46
4000	92974	46.49	44.64
6000	141290	47.10	390.48
8000	189438	47.36	721.38
10000	238079	47.62	1283.94
12000	285956	47.66	2210.94
16000	383538	47.94	4818.90
20000	483174	48.32	8242.50
30000	729612	48.64	20018.76
40000	973547	48.68	36998.64

Table 6.1: Statistics observed while running the *MWT* algorithm with diamond test on uniformly distributed point sets.

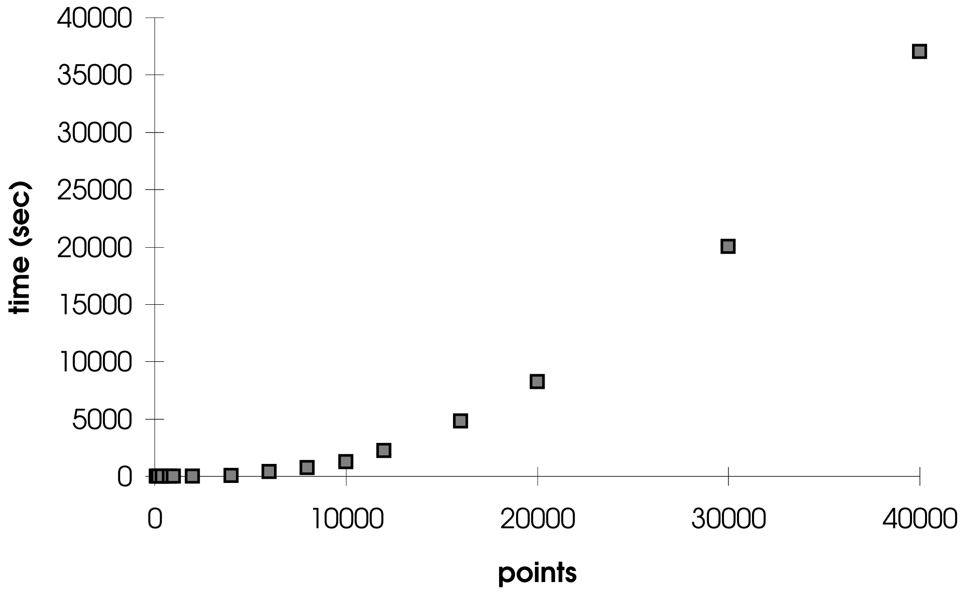

Figure 6.15: Time required to compute the MWT with the diamond test

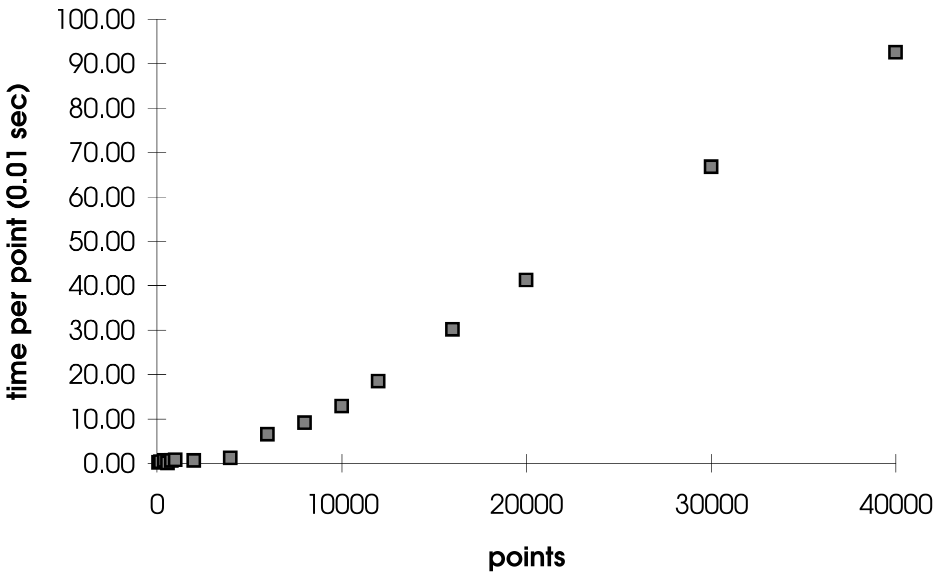

Figure 6.16: Time per point required to compute the MWT with the diamond test

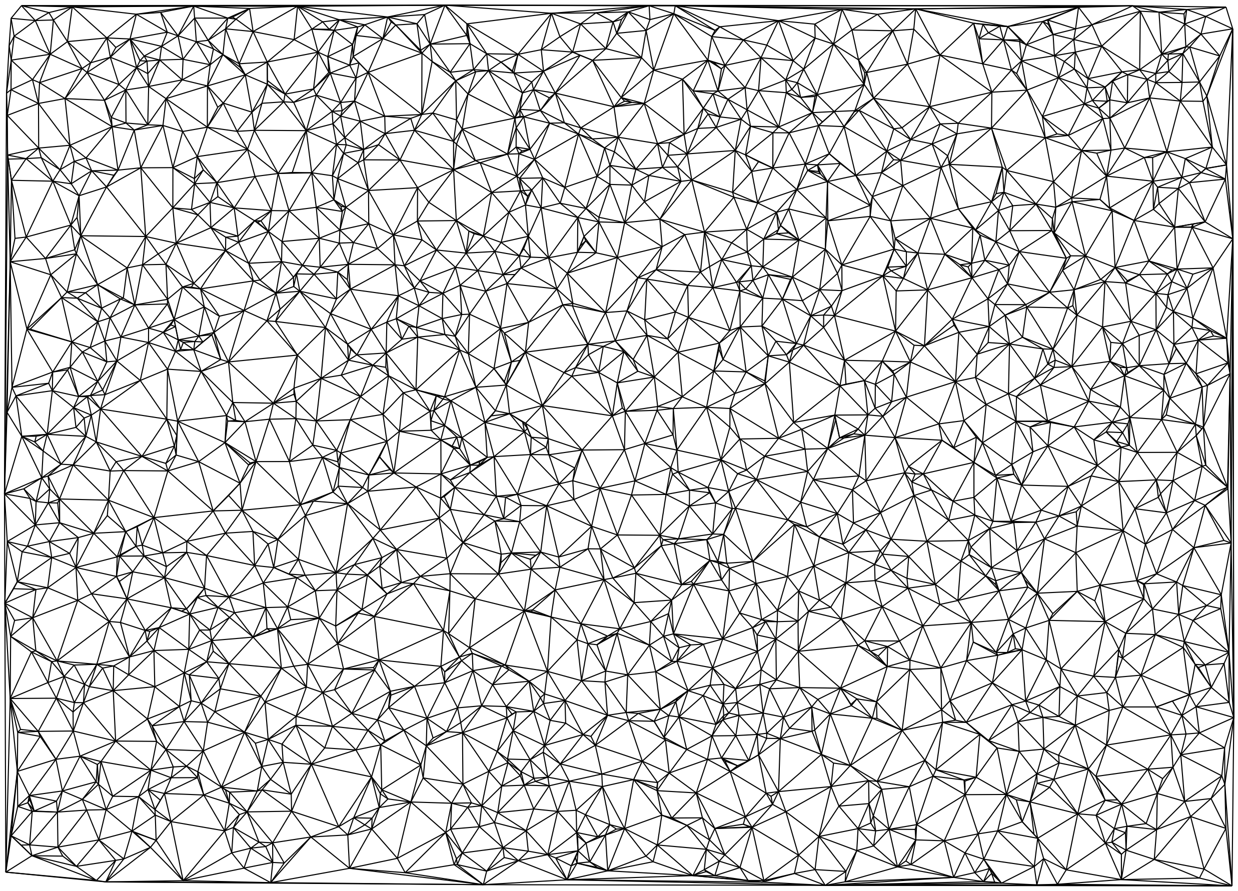

Figure 6.17: The exact minimum weight triangulation of 2000 uniformly distributed random points.

Chapter 7

Eliminating Edges by Buckets

The diamond test defined in previous chapter is faster than the straightforward algorithm that tests all diamond, but it still must look at every edge. Bucketing techniques, which have been successfully applied in greedy triangulations [DDMW94] [DRA95] can be used to throw out clusters of edges at a time. In this chapter we report on our application of bucketing to radially sort only those edges that pass the diamond test around each vertex.

Using this method, our experiments showed that for uniformly-distributed points the edge initialization step is computed in linear time and therefore the overall observed time and space required to compute the *LMT-skeleton* is linear.

7.1 Using the Diamond Property to Discard Regions

To eliminate sets of edges the algorithm uses the diamond property described in Section 3.4.

Given three points o, a and b with angle $\angle aob$ less than $\pi/4$, there is a *dead sector* S consisting of all the points p such that $\overline{op}$ fails the diamond test by having a and b in the isosceles triangles to the left and right of $\overline{op}$.

Define *left sector* LS to be the region consisting of all points p such that a is in the isosceles triangle left of $\overline{op}$ as illustrated in Figure 7.18. LS is bounded by the ray $\overrightarrow{oa}$, the same ray rotated $\pi/8$ clockwise, and is outside the circle centered at c such that $\triangle odc$ is isosceles with base $\overline{oa}$ and c, to the right of $\overline{oa}$, forms $\angle oca$ of $\pi/4$. For all points t

outside this circle, angle $\angle ota$ is less than $\pi/8$.

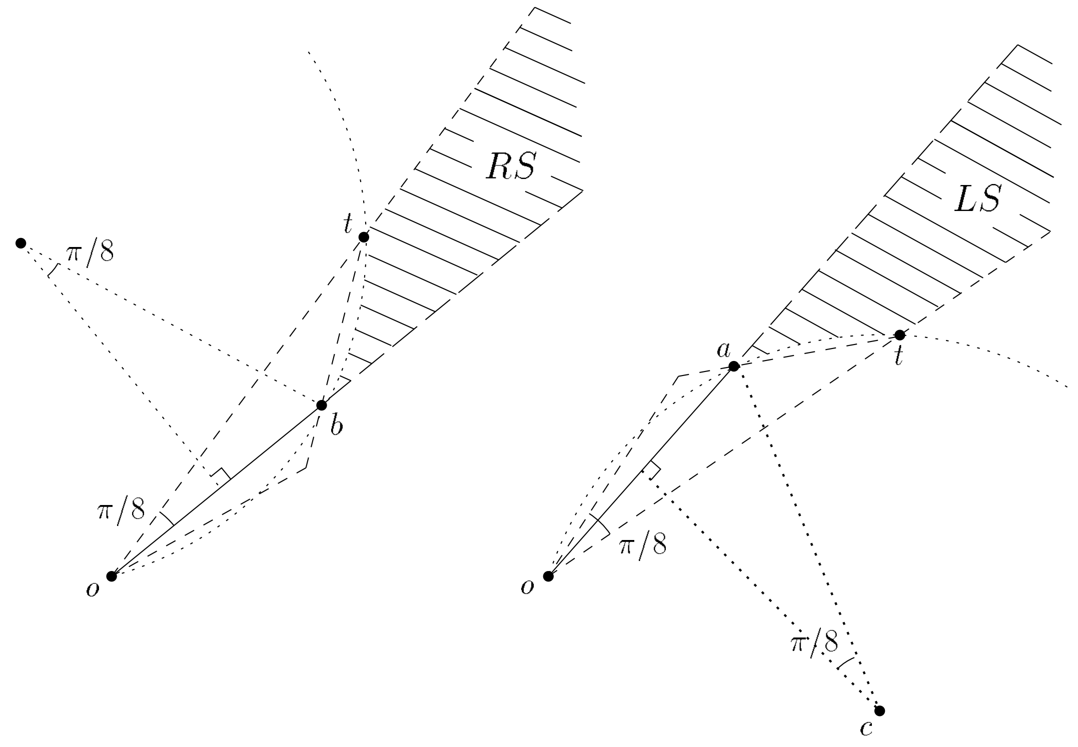

Figure 7.18: The sectors LS and RS

Define *right sector* RS symmetrically to be the region for which b is in the right triangle. Then the dead sector is the intersection $S = LS \cap RS$. Figure 7.19 illustrates two examples of dead sectors.

7.2 Definitions

The goal of the algorithm is to eliminate sets of edges at a time. In order to do so, points are stored into *buckets*: cells of a homogeneous square grid covering the plane. The size of the buckets is set to satisfy the specified average number of points in each bucket. In a grid B of size $m \times n$, individual buckets are denoted $b_{i,j}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.

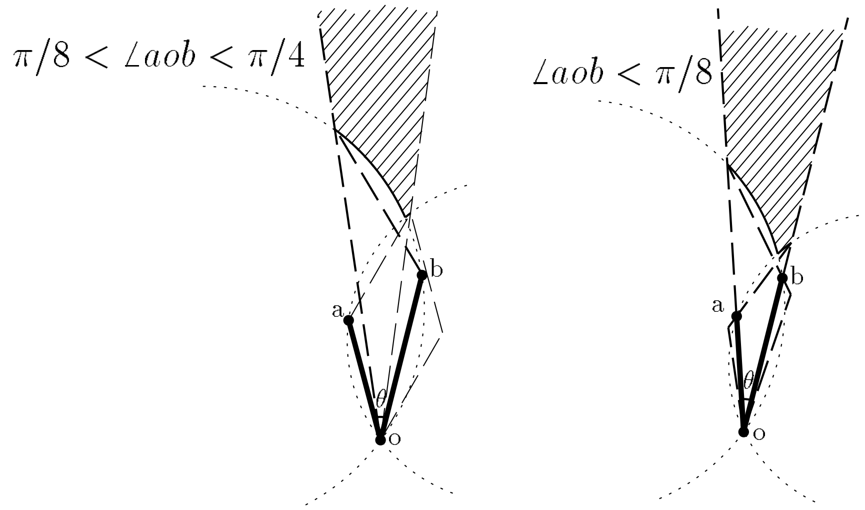

Figure 7.19: Sectors defined by two line segments

A *layer* of buckets of level l around a bucket $b_{i,j}$ is the set of buckets that can be reached by crossing exactly l horizontal grid lines or exactly l vertical grid lines. In other words it is the set

$$\{b_{i+l,k} : j-l \leq k \leq j+l\} \cup \{b_{i-l,k} : j-l \leq k \leq j+l\} \cup$$
$$\{b_{k,j+l} : i-l \leq k \leq i+l\} \cup \{b_{k,j-l} : i-l \leq k \leq i+l\}.$$

A *layer row* is the set of buckets defined by one of the terms of the previous union. Buckets in a layer form a square configuration. A layer row is the set of buckets along one side of the square configuration.

A *layer line* is the inside edge of a layer row.

The *origin point* refers to the point around which the algorithm is constructing a radially-sorted edge list. The *origin bucket* contains the origin point.

A point, bucket, layer row, or layer is *dead* if it lies completely inside dead sectors, otherwise it is *alive*.

7.3 The Bucketing Algorithm

The main idea of the algorithm is to construct the radially-sorted list of edges around an origin point o by considering edges from shortest to longest. A list of dead sectors is maintained until the dead sectors cover the whole 2π range as illustrated in Figure 7.20. Only edges $\overline{op}$ such that p lies in the central *alive* region need to be considered.

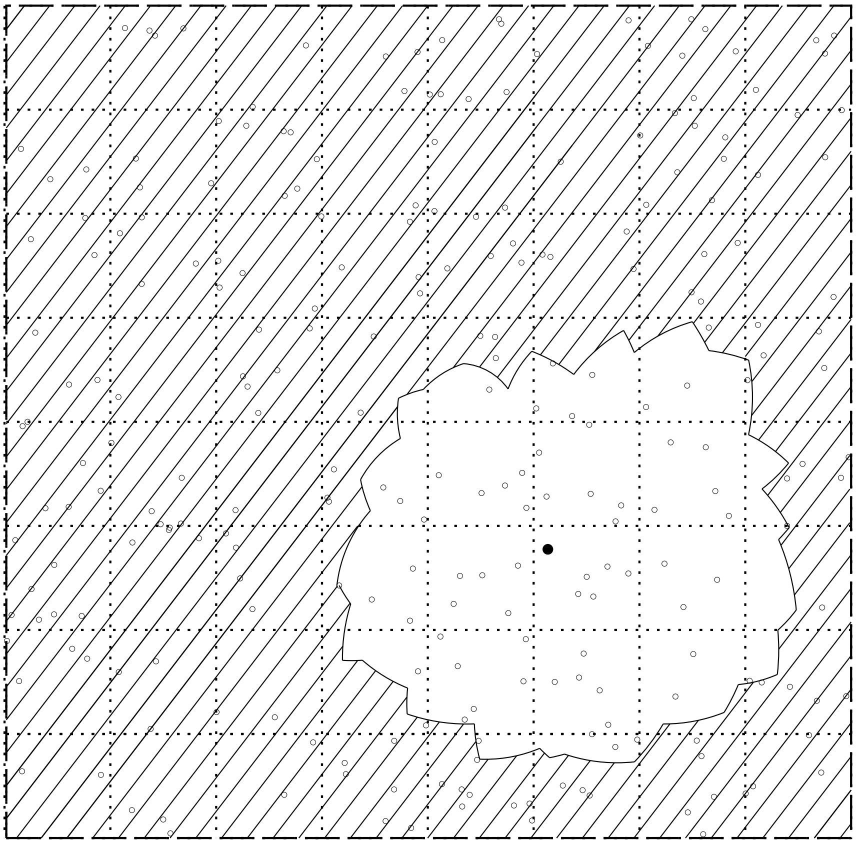

Figure 7.20: Dead sectors covering the 2π range

To construct the list of edges around an origin point o the algorithm starts by considering all points p in the origin bucket. Each segment $\overline{op}$ is inserted in the list if it passes the diamond test with regards to the endpoints of the two neighbouring edges.

Dead sectors are then generated with each pair of edges in the list. Points contained in the buckets of the next layer are similarly inserted in the list. The two previous steps are repeated until the next layer is dead.

There are several variations on how to consider points on the next layer. Considering all points in a layer can provide an easy implementation but is expensive when only a few buckets in that layer are alive. The algorithm would apply the diamond test to all edges with endpoints in that layer although some of these edges are known to fail the test because their endpoints are contained in dead buckets. Our implementation considers only edges with endpoints in alive buckets.

7.4 Implementation

Computing the actual dead sectors would be inefficient because it would require computing expensive trigonometric functions. Furthermore the data structure that would hold the exact dead sectors would be quite intricate because of the complex shape of these sectors. Instead the algorithm keeps a radially sorted list of the endpoints of edges that passed the diamond test. At each new layer considered, the algorithm computes the intersection of the dead sectors these points generate with the layer lines. The intersections are stored as intervals in four ordered lists, one for each layer line. Note that no information is lost by storing intersections instead of sectors.

If there are k endpoints in the list then there are $\binom{k}{2}$ pairs of endpoints. As described in Figure 7.21, the program builds the lists of dead sector intersections without requiring $\binom{k}{2}$ operations. Instead, to build each of the lists, it construct two independent ordered lists: *RSlist* contains the intersection of the right sectors RS with the layer line and *LSlist* contains the intersection of the left sectors LS with the layer line. In Figure 7.21 $RS(\overline{ab})$ and $LS(\overline{ab})$ refer to the right sector and left sector generated by the edge $\overline{ab}$ while l

procedure *makeDSlist*

$DSlist := empty$
$RSlist := empty$
$LSlist := empty$

For each edge $\overline{ab}$ in the radially-sorted edge list
  $RSlist := (RS(\overline{ab}) \cap l) \cup\ RSlist$
  $LSlist := (LS(\overline{ab}) \cap l) \cup\ LSlist$
  $\overline{ab} := \overline{ab} \rightarrow next$

$DSlist := RSlist \cap\ LSlist$

Figure 7.21: Creating a list of dead sectors

represents the layer line. The dead sector intersection list *DSlist* is constructed by taking the intersection of the right and left sector lists.

Once *DSlist* is constructed, the algorithm runs through the buckets in a layer row. Each bucket is tested against the dead sector list corresponding to the layer row. If a bucket is alive, then the edges with endpoints in that bucket that pass the diamond test are added to the sorted edge list. If all buckets are dead, the layer row is dead and the algorithm does not need to consider layer rows in that direction. If all four layer rows are dead, then all remaining buckets in the grid must also lie in dead sectors and the list of edges around the origin point is complete. Figure 7.22 describes the algorithm to construct the radially-sorted list of edges around a point o.

Note that this pretest may allow a few edges through that do not satisfy the diamond property, because edges that are never inserted in the list don't participate in eliminating other edges.

procedure *makeEdgeList(o)*

For each layer $(0 \ldots l)$
  For each alive layer rows
    For each alive bucket in the layer row
      For each point p in the bucket
        If the edge op passes the diamond test with
        regards to the two neighbouring edges.
          Insert the edge at the edge at the appropriate
          position in the radially-sorted list.
  If no buckets were alive then stop
  Construct *RSlist*
  Construct *LSlist*
  makeDSlist

Figure 7.22: The procedure *makeEdgeList(o)* that constructs the radially-sorted edge list around point o

7.5 Calculating the Dead Sectors

Consider the intersection of the sector LS defined earlier, generated by the left diamond triangle and the line l tangent to the inside edge of the right layer row (Figure 7.23). The intersection of the sector LS with the line is the line segment $\overline{pq}$ where the point p is the intersection of the line m tangent to $\overline{oa}$. For the sake of simplicity, in the following equations the point o is also the origin of the Cartesian plane. The coordinate p_x is known since l is vertical and p_y is

$$p_y = \frac{a_y}{a_x}\ p_x$$

The point q is the intersection of the lines n and l if t is left of l, otherwise it is the intersection of the arc at with l.

In the first case the intersection of n and l is

$$q_y = \frac{s_y}{s_x}\, p_x,$$

where s is the apex of the right diamond triangle. In the second case q is the intersection of the circle centered at c and l, where

$$c = \frac{a}{2} + \mathrm{perp}(\frac{a}{2})\ \cot(\pi/8).$$

Since both q and r lie on the circle

$$(q-c)\cdot(q-c) = r \cdot r,$$

so

$$q \cdot q = 2\ \ r \cdot q.$$

In this case the larger solution of the quadratic is required

$$q_y = \frac{1}{2}\left(a_x + a_y \cot(\pi/8) + \sqrt{(-a_y + a_x \cot(\pi/8))^2 - 4q_x\,(-a_x + q_x - a_y \cot(\pi/8))}\right).$$

All other intersections can be similarly obtained without using time consuming trigonometric functions.

7.6 Complexity Analysis

The following analysis is based on the number of edges k and the number of buckets b that are considered to construct a radially sorted edge list. We show that if both values are constant the algorithm presented here constructs the *LMT-skeleton* in linear time.

Lemma 7.6.1 Constructing a dead sector list generated by the list of k radially sorted edges takes $O(k)$ time.

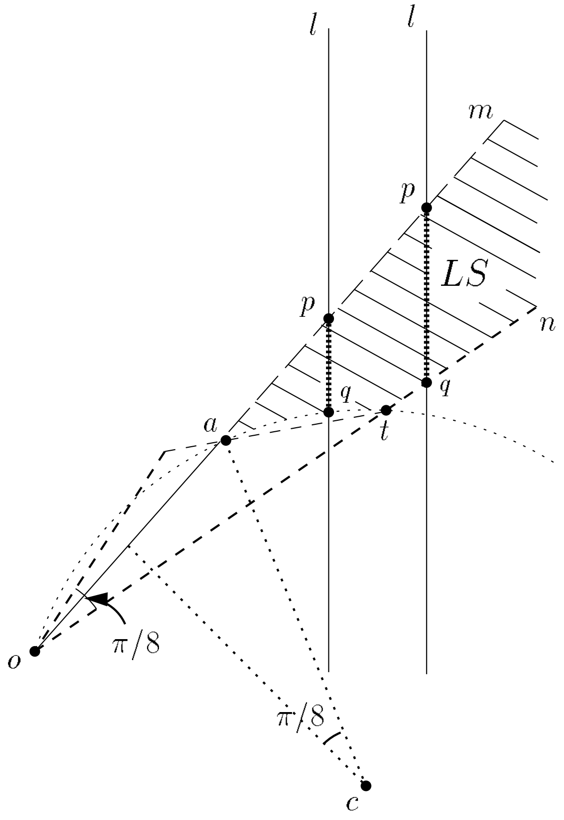

Figure 7.23: Two cases for the intersection between the sector LS and the line l

Proof The ordered list of intersections *RSlist* and *LSlist* can be constructed $O(k)$ time by a simple scan of the radially-sorted edge list. The intersection of the two lists can be done in $O(k)$ time since there are most k intervals in each list. □

Lemma 7.6.2 The time taken to construct the radially-sorted list of edges L around a point o is $O(b\ k + k^2)$ where b and k are the number buckets and edges considered to build the list.

Proof If there are l layers considered the algorithm needs to reconstruct the lists of dead sectors at most l times. From Lemma 7.6.1 the time taken to compute each list is $O(k)$. Constructing the l lists can therefore be done within $O(b\ k)$ time. Furthermore, at most k edges are added into the sorted edge list requiring at most $O(k^2)$ time. □

Lemma 7.6.3 If the maximum number of edges k and the maximum number of buckets b considered to construct a radially-sorted edge list are constant then the *LMT-skeleton* heuristic can be computed in linear time.

Proof If b and k are constant, Lemma 7.6.2 implies that each edge list can be constructed in constant time. Constructing the n lists can then be done in linear time. By Lemma 6.2.2, the rest of the heuristic can also be computed in linear time. □

Note that for uniformly distributed point sets there is a constant number of points in each bucket. For such point sets, values k and b are therefore interchangeable in the previous lemmas.

7.7 Optimizations and Experimental Results

The buckets on the first few layers are usually alive. The algorithm considers all buckets on the first two layers saving the time of constructing the dead sector lists and checking which buckets are alive. Furthermore, the algorithm sets the size of the buckets so that an average of five points lie in each bucket.

We ran our *C++* implementation on a SGI with 200MHz IP22 processor on uniformly distributed point sets of 500 to 40 000 points. As we had hoped, only a constant number of layers and buckets are visited around each point. The time saved by not sorting all $n-1$ edges for each point was reflected in the observed linear behavior of the overall algorithm.

A significant improvement in the run time was observed in our implementation of the algorithm. For uniformly distributed point sets, the exact minimum weight triangulation of 40 000 points was found in less than 5 minutes. Statistics of our implementation can be found in Table 7.2 and Figures 7.24 and 7.25. Figure 7.26 compares the performances of the heuristic using the diamond test and the one using bucketing.

n	total buckets	edges per point passing	average degree	avg bkts visited	avg layers visited	time (10^{-2} secs)
100	16	17.05	34.10	13.34	6.00	0.20
200	36	18.80	37.59	18.81	8.96	0.30
400	64	21.19	42.37	21.10	10.30	0.31
600	100	22.24	44.47	23.19	11.29	0.36
800	144	22.63	45.28	25.33	12.23	0.39
1 000	196	22.78	45.55	27.31	13.02	0.41
2 000	400	23.77	47.53	30.11	14.17	0.53
4 000	784	24.62	49.24	31.31	14.73	0.53
6 000	1 156	24.99	50.00	31.63	14.93	0.56
8 000	1 600	25.20	50.39	32.90	15.38	0.57
10 000	1 936	25.41	50.83	32.62	15.32	0.60
12 000	2 304	25.53	51.07	32.72	15.34	0.63
16 000	3 136	25.73	51.47	33.41	15.62	0.66
20 000	3 969	25.69	51.39	33.74	15.72	0.69
25 000	4 900	25.70	51.40	33.69	15.76	0.71
30 000	5 929	25.70	51.39	34.00	15.88	0.74
35 000	6 889	25.76	51.54	34.10	15.92	0.73
40 000	7 921	25.78	51.56	34.25	15.97	0.73

Table 7.2: Statistics observed while running the *MWT* algorithm with bucketing on uniformly distributed point sets.

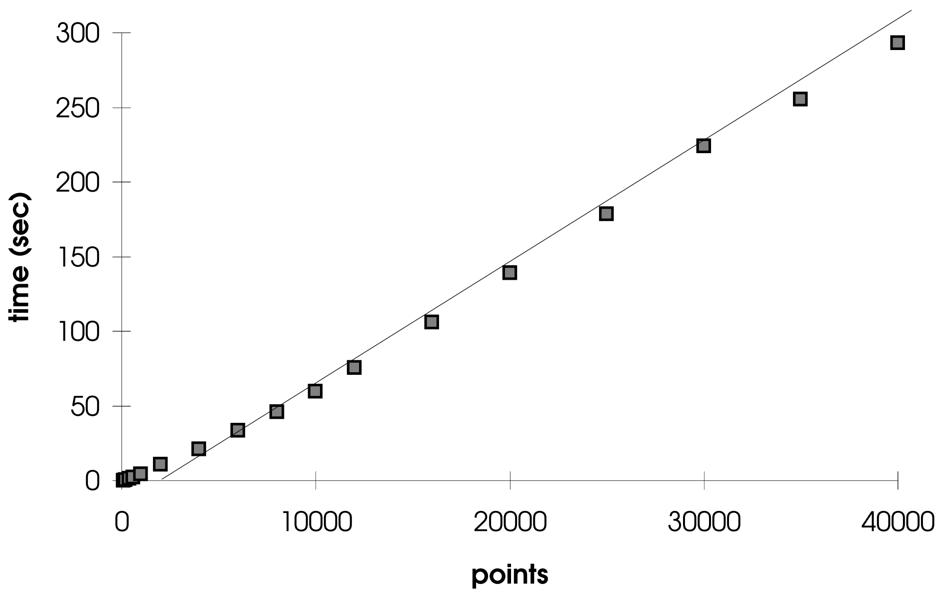

Figure 7.24: Time required to compute the *MWT* with bucketing

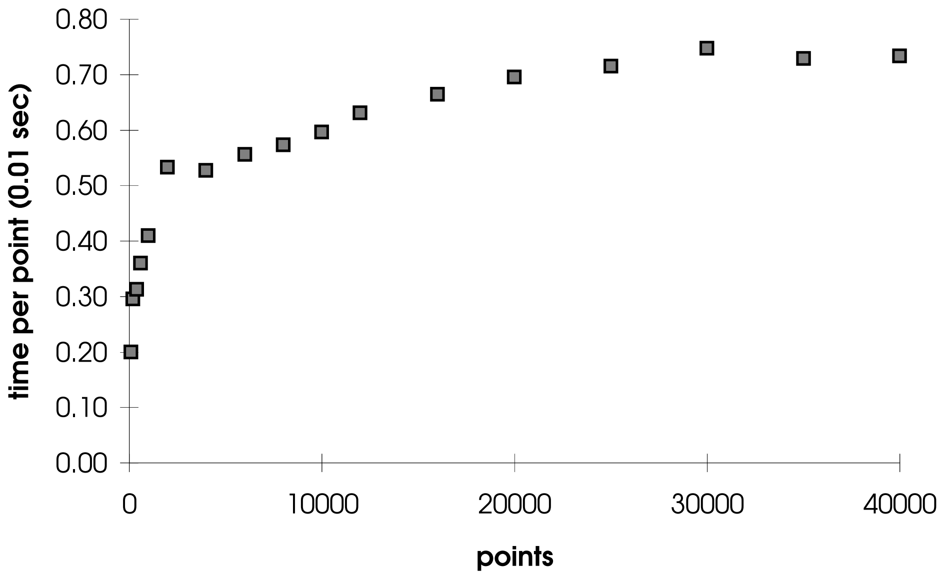

Figure 7.25: Time per point required to compute the MWT with bucketing

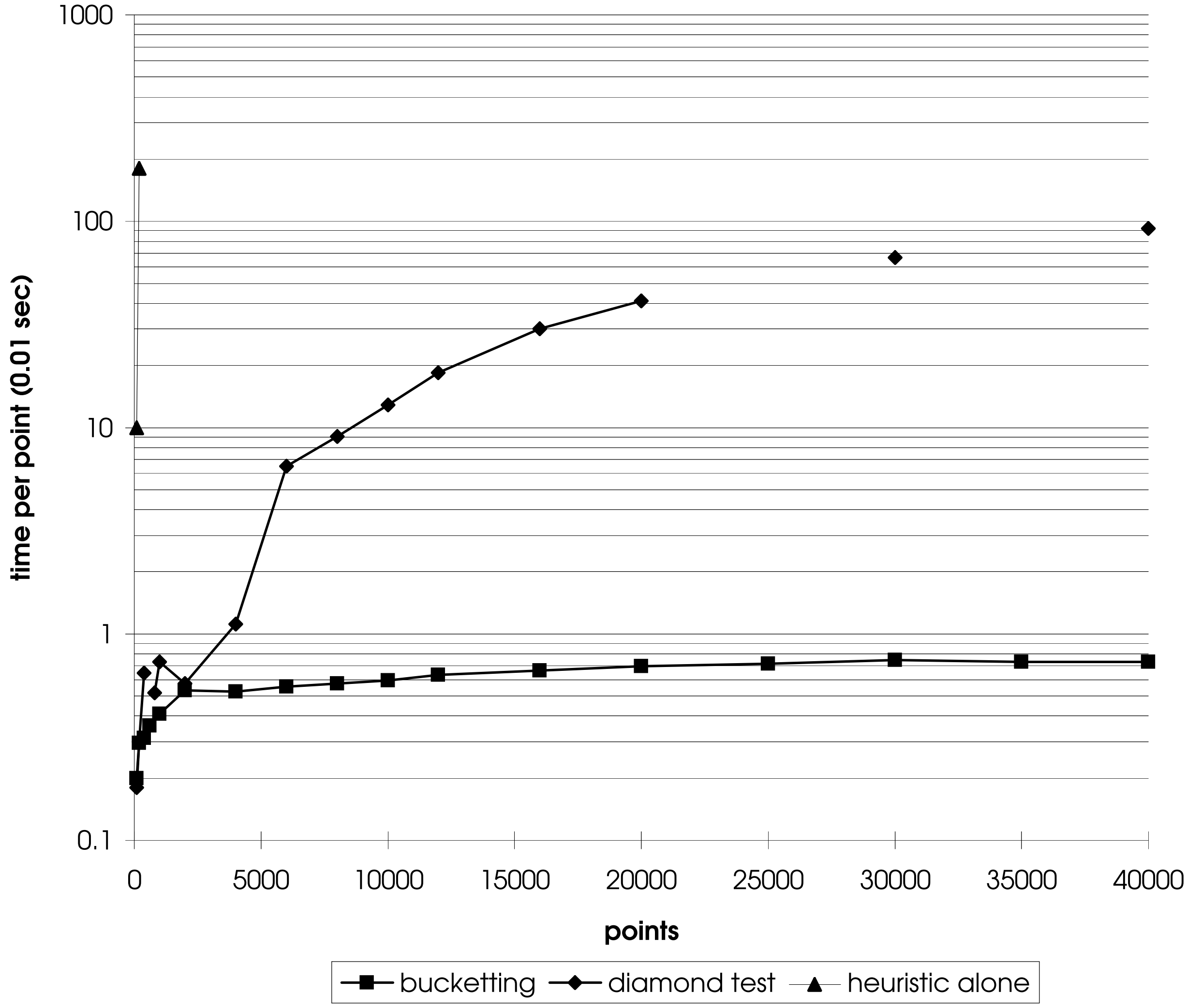

Figure 7.26: Time required by *LMT-skeleton* heuristic: diamond test versus bucketing.

Chapter 8

Effectiveness and Future Directions

The *LMT-skeleton* is the set of edges that are in all locally minimal triangulations. If a point set has more than one locally minimal triangulation then there are some edges that will remain *possible* after the *LMT-skeleton* heuristic is applied. In most cases the *LMT-skeleton* is a connected graph. In this case, edges that remain possible are isolated in polygons and the *MWT* can be completed with the dynamic programming algorithm described in Section 4.3.

However some structures that admit more than one locally minimal triangulation do generate a disconnected subgraph of the *MWT*. This chapter will present the different structures that *block* the *LMT-skeleton* heuristic.

8.1 The Wheel Configuration

This section will first present a structure and show that it blocks the *LMT-skeleton* heuristic. It then presents the work of Bose, Devroye and Evans [BDE96], which shows that this structure can be expected to occur linearly-many times in uniformly distributed point sets.

8.1.1 The Structure of the Wheel Configuration

One can construct a *wheel* by placing a point o at the center of a circle and all other points on the circle such that any $\pi/3$ sector contains at least three points.

Let o be the point located at the center of the circle and $p_0, p_1, \ldots, p_{n-1}$ be the n

points labelled clockwise on the circle such that each $\pi/3$ sector contains at least three points. All further point indices will be modulo .

Lemma 8.1.1 All edges $\overline{p_i p_{i+2}}$ and $\overline{op_i}$ in a wheel have certificates. Therefore the *LMT-skeleton* of a wheel is a graph where o is disconnected from all other points.

Proof

Suppose all edges op_i and $p_i p_{i+2}$ are possible, then all these edges have certificates.
Consider any three consecutive points p_i, p_{i+1}, p_{i+2} and the point o as shown in Figure 8.27. Since any $\pi/3$ sector contains at least three points the angle $\angle p_i o p_{i+2}$ is at most $\pi/3$. Therefore $|\overline{p_i p_{i+2}}| \leq |\overline{p_{i+1} o}|$. This means that the edge $\overline{p_i p_{i+2}}$ has a certificate consisting of the two triangles $\triangle p_i p_{i+1} p_{i+2}$ and $\triangle p_i o p_{i+2}$. If the triangle edges are possible then all edges $\overline{p_i p_{i+2}}$ have certificates.
Consider the points p_i, p_{i+2}, o and the point p_j left of $\overline{op_i}$ such that p_j can not lie in the same $\pi/3$ sector than p_i and p_{i+2}. Since $|\overline{p_j p_{i+2}}| \geq |\overline{op_i}|$, the edge op_i has a certificate consisting of the two empty triangles triangles $\triangle op_i p_j$ and $\triangle op_i p_{i+2}$ if the edges of the two triangles are possible.
Therefore all edges op_i and $p_i p_{i+2}$ have certificates. Since each of these edges is possible and has at least one crossing edge, none can become certain. The point o therefore remains disconnected from all other points p_i. $\square$

Figure 8.28 is an example of a wheel configuration to which the *LMT-skeleton* heuristic was applied. Dotted edges are the ones that remained possible. Points do not have to be co-circular to block the heuristic and such a structure can be expected in a uniformly distributed point set as shown in the next section.

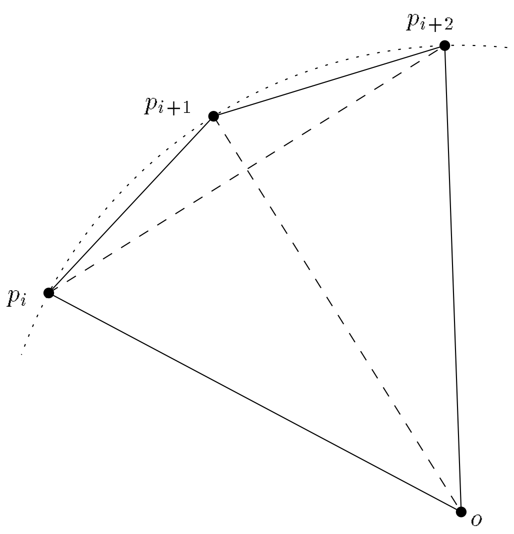

Figure 8.27: Certificate of edge $\overline{p}_i p_{i+2}$.

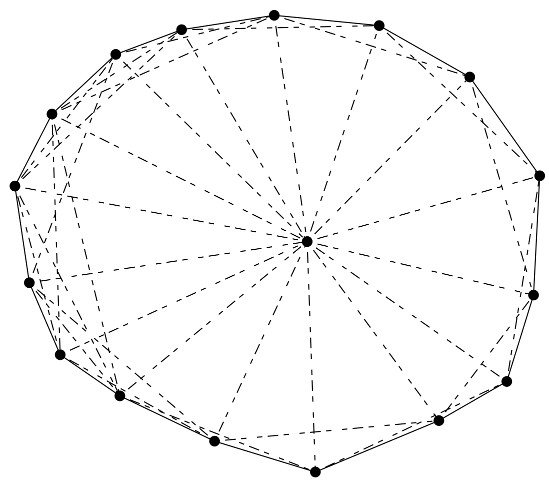

Figure 8.28: Example of a point set forming a wheel.

8.1.2 The Diamond Configuration

Given a point set S of n points, a point o is a *diamond* if o is the center of a circle c of radius $1/\sqrt{n}$ that only contains o. *Facets* are the 18 regions located between the regular 18-gon in which c is inscribed (see Figure 8.29). In each facet lies exactly one point of S. Note that the diamond as described in this section does not relate to the one defined in Section 3.4 and used in the previous chapters.

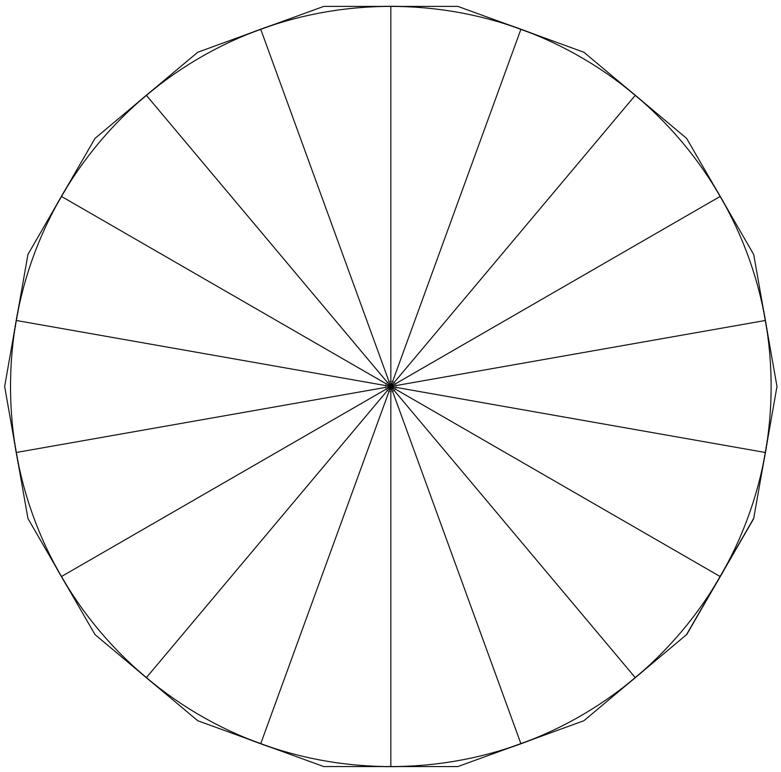

Figure 8.29: The diamond configuration.

Let $p_0, p_1, \ldots, p_{n-1}$ be the 18 points lying in the facets in clockwise order. For the same reasons as the wheel structure presented in the previous section all edges $\overline{p_i p_{i+2}}$ remain possible after the *LMT-skeleton* heuristic is applied. Point o is therefore disconnected from the other points.

Bose, Devroye and Evans [BDE96] show that the probability that a diamond occurs in a uniformly distributed point set is constant. The expected number of diamonds in a point set exceeds one only when the set contains more than 10^{51} points. However this number is high because of the constraining structure of the diamond configuration. Points can be placed in a looser pattern and still generate an isolated point in the center. One can even isolate a set of edges with endpoints near the center as seen in Figure 8.30.

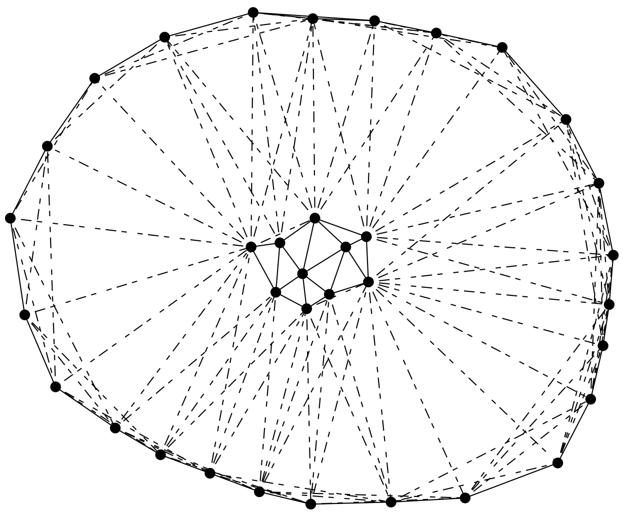

Figure 8.30: Example of a point set forming a wheel with several disconnected edges.

8.2 Tiling Wheels

Even if the wheel configuration occurs in a point set it only contains a constant number of points. Using a brute force approach finds the MWT of each wheel in constant time.

Belleville, Keil, McAllister and Snoeyink [BKMS96] tiled the wheel configuration to obtain the structure in Figure 8.31 and Figure 8.32.

The tiling structure is constructed by placing the wheels at the vertices of a hexagonal grid in such a way that the heuristic produces a graph with disconnected 18-gons as shown

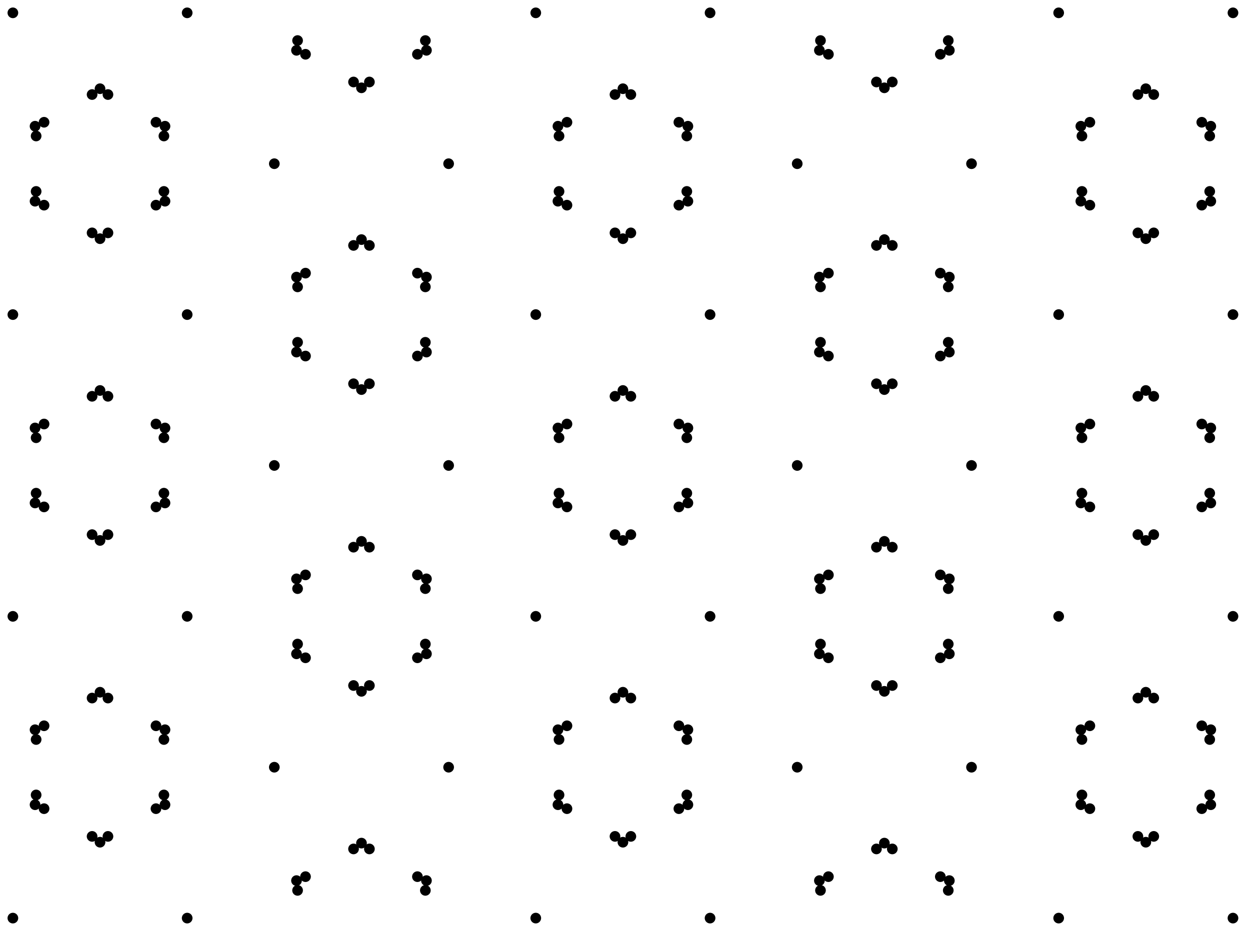

Figure 8.31: Tiling wheels in the plane along a hexagonal lattice.

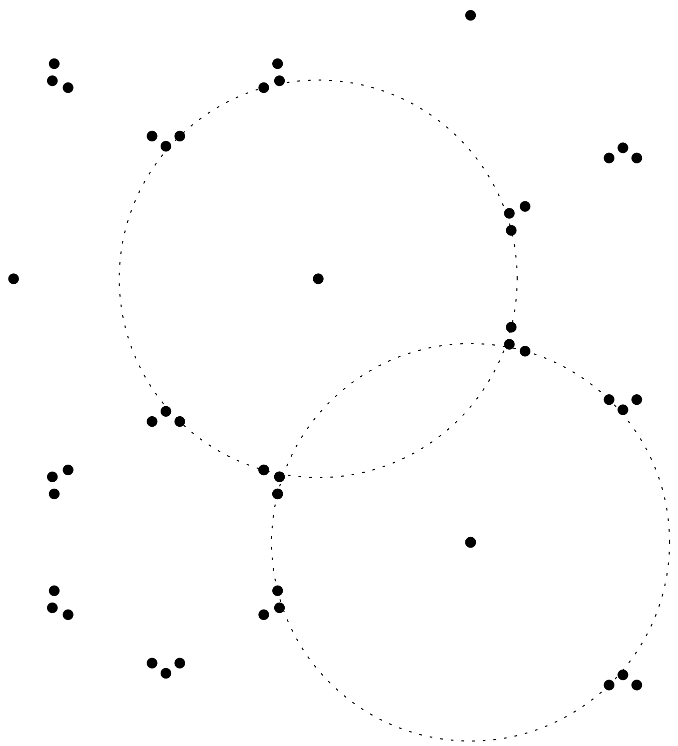

Figure 8.32: A close-up at the tiled wheels.

if Figure 8.33. With n points one can generate as many as $2n/19 - o(n)$ disconnected regions.

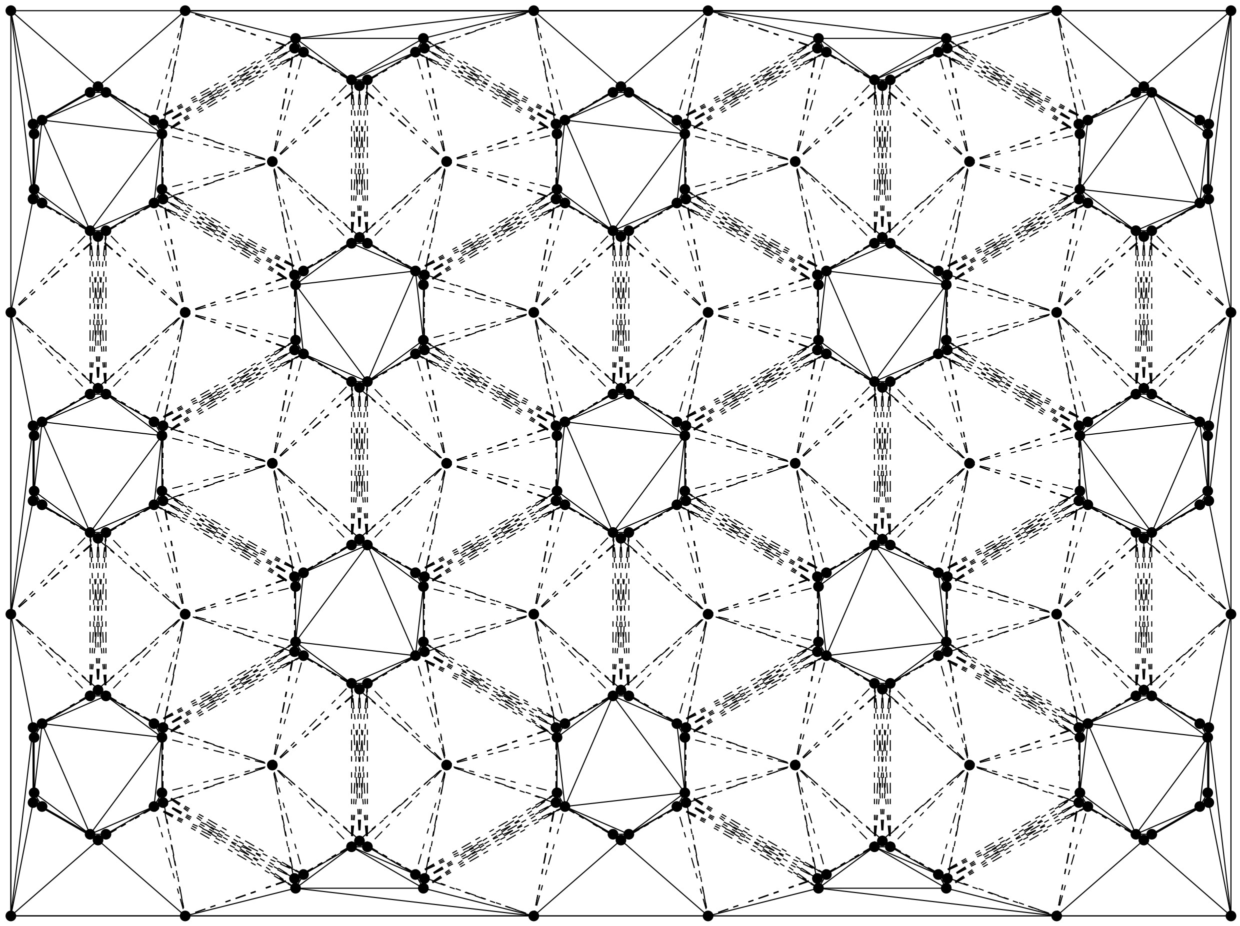

Figure 8.33: Tiled wheels, after applying the *LMT-skeleton* heuristic.

Such a structure shows that the *LMT-skeleton* heuristic does not provide a polynomial time algorithm to solve the minimum weight triangulation problem.

8.3 The wire

Since the minimum weight is a global property of a triangulation there is good reason to believe that the *MWT* problem is **NP**-complete or even **NP**-hard. One of the ways to prove **NP**-completeness of a problem is to reduce a known **NP**-complete problem to it. This section presents a structure that can serve in reducing the problem of satisfiability of a boolean expression (SAT).

To accomplish the reduction, we need point structures that act as variables, gates and wires. Snoeyink and Drysdale used the program described in the previous section to experiment with different point sets to find such structures.

A *wire* is a set of points that admits two different minimum weight triangulations. This structure can allow a boolean value to be transmitted along its length. Requiring that an edge e on one of the extremities of the wire be in the *MWT* entails that some edge f at the other extremity of the wire be in the *MWT*. However, requiring that an other edge g on the first extremity be in the *MWT* entails that some other edge $h \neq f$ is in the *MWT*. Figure 8.34 shows how a wire admits two minimum weight triangulations. This wire can be as long as desired and can also turn.

A *variable* can be any structure that admits two triangulations. One can build a variable by constructing a wire such that the two extremities connect.

No structures have been found to represent gates. Furthermore, although we do have structures for variables and wires we do not know any way to link them together.

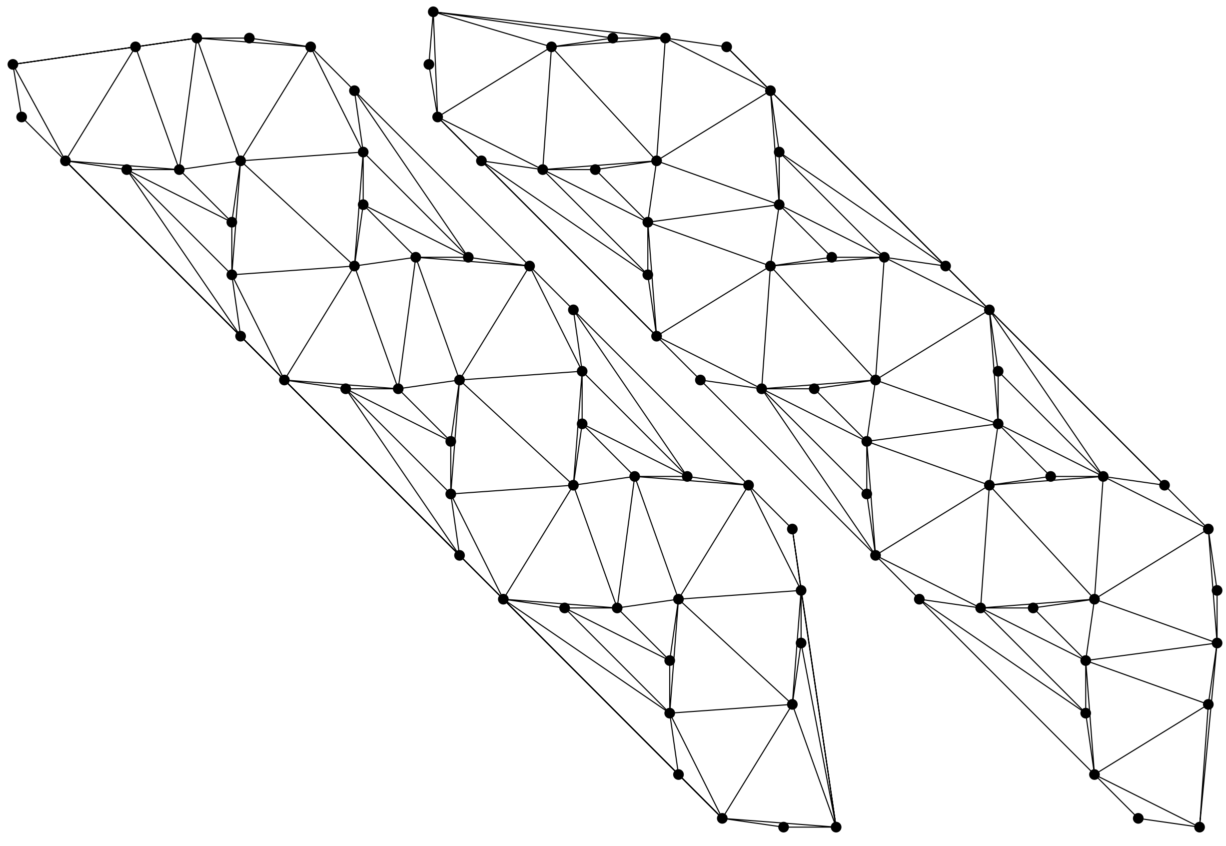

Figure 8.34: Minimum weight triangulations of two closely-related wires

Chapter 9

Conclusion

We presented an efficient implementation of the *LMT-skeleton* heuristic and seen that it usually produces a complete subgraph of the minimum weight triangulation for uniformly distributed point sets. If the graph is complete we compute the exact *MWT* by applying the polygon *MWT* dynamic programming algorithm to the remaining non-triangulated polygonal holes.

The algorithm was implemented in *C++* and run on an SGI with 200MHz IP22 processor. For uniformly distributed point sets of tens of thousands of points our experiments show that the algorithm computes the exact minimum weight triangulation in linear time and space. The *MWT* for uniformly distributed sets of 40,000 points are computed in less than 5 minutes.

Using this fast implementation allows us to experiment with finding point sets such as *wheels*, for which the heuristic does not compute a connected graph. Furthermore, by tiling wheels, we can construct point sets whose *LMT-skeleton* contains a linear number of disconnected components. This shows that the *LMT-skeleton* does not provide a polynomial-time algorithm for solving the *MWT* problem.

The complexity status of the *MWT* problem is still open. We do not know whether it is **NP**-hard or whether it can be solved by a polynomial time algorithm. One can use this implementation to experiment with different point sets in order to find structures to prove the **NP**-hardness of the problem. In the last chapter the wire was presented as one component that can be used to reduce a satisfiability problem to the *MWT* problem.

Structures for the remaining components and how the components can be connected together are still unknown.

Bibliography

[AC93] E. Anagnostou and D. Corneil. Polynomial time instances of the minimum weight triangulation problem. *Comput. Geom. Theory Appl.*, 3:247–259, 1993.

[BDE96] P. Bose, L. Devroye, and W. Evans. Diamonds are not a minimum weight triangulation's best friend. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 68–73, 1996.

[BKMS96] Patrice Belleville, Mark Keil, Michael McAllister, and Jack Snoeyink. On computing edges that are in all minimum-weight triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages V7–V8, 1996.

[Blö91] J. Blömer. Computing sums of radicals in polynomial time. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 670–677, 1991.

[CGT95] Siu Wing Cheng, Mordecai J. Golin, and Jeffrey C. F. Tsang. Expected case analysis of β-skeletons with applications to the construction of minimum-weight triangulations. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 279–284, 1995.

[CK96] Siu-Wing Cheng and Naoki Katoh. A note on the lmt-skeleton. *unpublished*, 1996.

[CL96] Drago Krznaric C. Levcopoulos. Tight lower bounds for the minimum weight triangulation heuristics. *Information Processing Letters*, 57(3):129–135, 1996.

[CX96] Siu-Wing Cheng and Yin-Feng Xu. Approaching the largest β-skeleton within a minimum weight triangulation. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 196–203, 1996.

[DDMW94] M. T. Dickerson, R. L. S. Drysdale, S. A. McElfresh, and E. Welzl. Fast greedy triangulation algorithms. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 211–220, 1994.

[DJ89] G. Das and D. Joseph. Which triangulations approximate the complete graph. In *Proc. International Symposium on Optimal Algorithms*, volume 401 of *Lecture Notes Comput. Sci.*, pages 168–192. Springer-Verlag, 1989.

[DM96] Matthew T. Dickerson and Mark H. Montague. A (usually?) connected subgraph of the minimum weight triangulation. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 204–213, 1996.

[DRA95] Robert L. Scot Drysdale, Günter Rote, and Oswin Aichholzer. A simple linear time greedy triangulation algorithm for uniformly distributed points. Technical Report IIG-408, Institutes for Information Processing, Technische Universität Graz, February 1995.

[Gil79] P. D. Gilbert. New results in planar triangulations. Report R-850, Coordinated Sci. Lab., Univ. Illinois, Urbana, IL, 1979.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, NY, 1979.

[HAA97] R. Hainz, O. Aichholzer, and F. Aurenhammer. New results on minimum weight triangulations and the lmt-skeleton. In *CG'97 13th European Workshop on Computational Geometry (Extended Abstract)*, Würzburg Germany., 1997.

[Kei94] M. Keil. Computing a subgraph of the minimum weight triangulation. *Comput. Geom. Theory Appl.*, 4:13–26, 1994.

[Kir80] D. G. Kirkpatrick. A note on Delaunay and optimal triangulations. *Inform. Process. Lett.*, 10:127–128, 1980.

[Kli80] G. T. Klincsek. Minimal triangulations of polygonal domains. *Discrete Math.*, 9:121–123, 1980.

[Kyo96] Yoshiaki Kyoda. A study of generating minimum weight triangulation within practical time. *Master's Thesis. University of Tokyo.*, 1996.

[Lev87] C. Levcopoulos. An $\Omega(\sqrt{n})$ lower bound for the nonoptimality of the greedy triangulation. *Inform. Process. Lett.*, 25:247–251, 1987.

[Lin85] A. Lingas. A linear-time heuristic for minimum weight triangulation of convex polygons. In *Proc. 23rd Allerton Conf. Commun. Control Comput.*, 1985.

[LK97] C. Levcopoulos and D. Krznaric. A near-optimal heuristic for the minimum weight triangulation of convex polygons. *unpublished*, 1997.

[Llo77] E. L. Lloyd. On triangulations of a set of points in the plane. In *Proc. 18th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 228–240, 1977.

[LLS89] Christos Levcopoulos, Andrzej Lingas, and Jorg-R. Sack. Heuristics for optimum binary search trees and minimum weight triangulation problems. *Theoret. Comput. Sci.*, 66(2):181–203, August 1989.

[PH87] D. A. Plaisted and J. Hong. A heuristic triangulation algorithm. *J. Algorithms*, 8:405–437, 1987.