Elisa Zancolo

Minimum Spanning Trees Using Delaunay Triangulation

DIPLOMA THESIS

to achieve the academic title

Diplom-Ingenieurin

Technische Mathematik

Alpen-Adria-Universität Klagenfurt

Fakultät für Technische Wissenschaften

Supervisor: Univ.-Prof. Dipl.-Ing. Dr. Franz Rendl

Institut für Mathematik

July 2008

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende wissenschaftliche Arbeit selbstständig angefertigt und die mit ihr unmittelbar verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für wissenschaftliche Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die während des Arbeitsvorganges gewährte Unterstützung einschließlich signifikanter Betreuungshinweise ist vollständig angegeben.

Die wissenschaftliche Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

(Unterschrift) (Ort, Datum)

Abstract

This paper compares different methods to construct a Delaunay triangulation and furthermore explains how to get the minimum spanning tree (MST) with the help of the Delaunay triangulation.
First the basic terms and definitions of graphs and triangulations are explained, second related graphs (like the Voronoi diagram) and subgraphs to the Delaunay triangulation are introduced. In the third chapter several algorithms to get the Delaunay triangulation are presented and explained. I also added some more theory like the constrained Delaunay triangulation, several possible data structures for graph triangulations and a runtime comparison of the algorithms in chapter four. The fifth chapter introduces algorithms to get the minimum spanning tree (MST). After constructing the Delaunay triangulation the MST can be performed in $O(n \, log \, n)$ time.
Finally I attached the C++ code of my program which performs the Delaunay triangulation and MST of a random point set in $O(n \, log \, n)$ time. I implemented the incremental algorithm using the directed acyclic graph (DAG) data structure and Kruskal's greedy algorithm. The program returns the needed time and some documentation of the process and prints the triangulation and MST in a FIG–file.

Acknowledgements

I want to thank all my lecturers for teaching me the various fields of mathematics: sparking my interest in a lot of subjects and also showing me those fields I will further avoid. Thank you to all the staff members of Klagenfurt university - I really learned to love this place.
This thesis would not exist without the motivation of Dipl.–Ing. Dr.techn. Univ.–Doz. Oswin Aichholzer and Ao.Univ.–Prof. Mag.rer.nat. Dr.techn. Sophie Frisch from the TU Graz. My supervisor Univ.–Prof. Dipl.–Ing. Dr. Franz Rendl guided me well even from a distance and let me realize my ideas with few constraints or modifications. Thank you!
Furthermore I want to thank all those authors, who publish their papers online for free. This thesis would be less qualitative (or a lot more expensive) without you.

Finally... as I am going to really finish my studies I want to thank those people who believed in me and helped me to get here. Firstly a big thanks to my parents who pretty much put me through this years, not only financially. Also to my sister who really knows how to comfort me when I am feeling blue.
A lot of friends have accompanied me during those last six years. My second family, the mathematicians Michaela Wassner, Olivia Bluder, Sibylle Singer, Veronika Grascher, Anton Ortner, Daniel Finke, Florian Türk, Gerhard Planka and Peter Krierer - I am glad we guys started together, it made those times much easier and even much more fun!
Big thanks to Elisabeth Zienitzer, you will always be an inspiration for me. Furthermore to Markus Kügerl, it sure is not easy to have an excellent mathematician on my side all the time, but I would not want it any other way. Special thanks to my best friend Claudia Zimmermann, I simply cannot imagine life without you.

Thank you to all the people who guided me on my way. You helped me become what I am.

Contents

1 Introduction **7**
1.1 What is a Triangulation? . . . 7
1.2 What is a Delaunay Triangulation? . . . 7
1.3 Assumptions . . . 8
1.4 Properties . . . 8

2 Related Graphs **10**
2.1 Voronoi Diagram . . . 10
2.1.1 Duality . . . 11
2.1.2 Applications . . . 11
2.2 Gabriel Graph . . . 11
2.3 Relative Neighbor Graph . . . 12
2.4 Minimum Spanning Tree . . . 12
2.5 Pairs . . . 13
2.6 Minimum Weight Triangulation . . . 13

3 Algorithms **15**
3.1 Flip . . . 15
3.2 Incremental Construction . . . 16
3.3 Delete and Build . . . 16
3.4 Divide and Conquer . . . 16
3.5 Sweepline . . . 17
3.6 Convex Hull in 3D . . . 17
3.7 Step by Step . . . 18
3.8 Voronoi . . . 19

4 Additional Theory about Delaunay Triangulations **20**
4.1 Constrained Delaunay Triangulation . . . 20
4.2 Data Structure . . . 21
4.2.1 Doubly Connected Edge List . . . 21
4.2.2 Quad Edge Structure . . . 22
4.2.3 Directed Acyclic Graph . . . 22

4.2.4 Tree Structure . . . 25
4.3 Applications . . . 25
4.4 Runtime Comparison . . . 25

5 Minimum Spanning Tree Algorithms **27**
5.1 Greedy Algorithm . . . 28
5.2 Kruskal's Algorithm . . . 29
5.3 Reverse–Delete Algorithm . . . 30
5.4 Prim's Algorithm . . . 30
5.5 Boruvka's Algorithm . . . 31

A Programming Example **34**
A.1 point.h . . . 37
A.2 point.cpp . . . 37
A.3 edge.h . . . 38
A.4 edge.cpp . . . 39
A.5 dag.h . . . 42
A.6 dag.cpp . . . 43
A.7 functions.h . . . 49
A.8 functions.cpp . . . 51
A.9 main.cpp . . . 59

1 Introduction

Most people have no idea what it is like to study technical mathematics and even less of writing a diploma thesis in this subject. So I developed an easy understanding way to explain the basic principle of my thesis to my relatives and friends.
Delaunay triangulations and their duals, Voronoi diagrams, are very famous and well studied in computational geometry. About one out of 16 papers in computational geometry discusses those topics.
The applications, theorems and fundamental terms in this chapter can be found in almost every basic work about combinatorics and graph theory, for example Jungnickel [Jun90] or de Berg et al [dBvKOS97].

1.1 What is a Triangulation?

Triangulations concern triangles. If we have several points on a plane and need to connect them (without crossing edges) we get triangles. And with this so–called triangulation we may achieve to find the shortest path including all points etc.
The triangulation problem can (in a more mathematical way of terms) be applied by guarding an art gallery. A polygonal gallery is triangulated by diagonals between pairs of vertices.

Theorem 1.1. *A triangulation of a simple polygon with n vertices consists of exactly $n-2$ triangles.*

By three–coloring the points (setting each node of a triangle to a different color) we see that $\lfloor n/3 \rfloor$ cameras or watchmen are sufficient to guide the gallery.
When triangulating a simple polygon we start with the given edges and only add edges inside the polygon. So it is a little different from triangulating a normal point set, where points define the convex hull without given edges.

1.2 What is a Delaunay Triangulation?

Boris Delaunay was a Russian mathematician (1890-1980) and a quite famous mountain climber. His name is the french version of his Russian name 'Delone', french being the language of science those days. Delaunay studied various fields of mathematics like abstract algebra, number theory and crystallography. He found the Delaunay triangulation in 1934.

The Delaunay triangulation is a triangulation with some special qualities. It maximizes the minimal angle of all triangles and therefore avoids skinny triangles. In other words no point is inside the circumcircle of any triangle. Delaunay triangulations can also be used in higher dimensional space but this paper will only refer to triangulations in the plane.

1.3 Assumptions

For simplification we suppose that any point set mentioned in this paper is in 'general position'. This means that no three points are collinear and no more than three points lie on any circle. Small point sets which are generated at random almost ever happen to be in general position. In higher dimensions no $n+1$ points are allowed to be on the same hyperplane and no $n+2$ points on the same hypersphere, n being the dimension.
If the points are not in general position the Delaunay triangulation is not unique. Anyway all the possible triangulations fulfill the properties of a Delaunay triangulation.
We further let the distance between two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ be the Euclidean distance. In the plane this is

$$dist(p, q) := \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

It is also possible to use a different metric, but in that case the Delaunay triangulation may not be unique or even existing.
The Manhattan or L_1–metric for example defines the distance as

$$dist_1(p, q) := |p_x - q_x| + |p_y - q_y|$$

Generally said the L_n–metric defines a distance of two points p, q as

$$dist_n(p, q) := \sqrt[n]{|p_x - q_x|^n + |p_y - q_y|^n}$$

1.4 Properties

Theorem 1.2. *Let P be a planar point set with n points and k points lying on the convex hull of P. Then any triangulation of P has $2n - 2 - k$ triangles and $3n - 3 - k$ edges.*

Definition 1.3. *(Delaunay criterion) A triangulation of a point set P is called 'Delaunay triangulation' if and only if the circumcircle of any triangle contains no other point of P in its interior.*

The Delaunay triangulation of a point set in the plane is a planar graph.

The Delaunay triangulation is angle–optimal which means the smallest angle is greater or equal than the smallest angle in any other triangulation of this point set. On the other hand the Delaunay triangulation does not minimize the maximum angle (or the longest edge).

2 Related Graphs

There are many graphs related to the Delaunay triangulation. Most of them are a part of the triangulation and therefore easier to obtain with the help of Delaunay.

2.1 Voronoi Diagram

The Voronoi diagram was officially found by Georgy Voronoi (1868-1908), another Russian mathematician and teacher of Delaunay, although it is known to be used already in some form in 1644 by the french scientist René Descartes. The Voronoi diagram is also referred to as Dirichlet tesselation, Thiessen polygon or 'Post Office Problem'.
The Voronoi diagram starts with several points in the plane and calculates the area around each point, so that any spot lies in the area of the nearest point (comparable with the next available post office near someone's location). Mathematically said we denote n points $P := \{p_1, p_2, ...p_n\}$ in the plane as 'sites'. We define the Voronoi diagram of P as the subdivision of the plane into n cells, one for each site in P, with the property that a point q lies in the cell corresponding to a site p_i if and only if $dist(q, p_i) < dist(q, p_j)$ for all $p_j \in P$ with $j \neq i$ (see for example [dBvKOS97, p. 147f]).

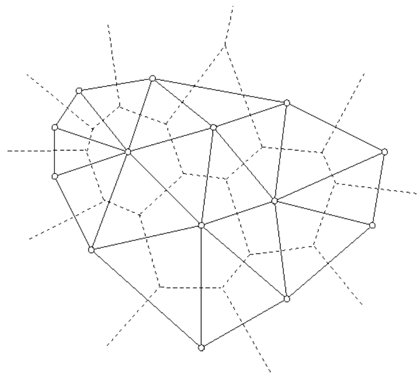

Figure 2.1: Delaunay Triangulation (solid) and Voronoi Diagram (dashed)

Theorem 2.1. *If all sites are collinear then the Voronoi diagram consists of $n-1$ parallel lines and n cells. Otherwise the Voronoi diagram is connected and its edges are either segments or half–lines.*

Theorem 2.2. *The number of vertices in the Voronoi diagram of a set of n sites in the plane is at most $2n - 5$ and the number of edges is at most $3n - 6$ (compare Theorem 1.2).*

Theorem 2.3. *If in general position every vertex of the Voronoi diagram in the plane has degree* 3.

2.1.1 Duality

Duality means a transformation of a mathematical object into another mathematical object. In geometry a graph G has a dual graph D if each vertex of G is a region in D and vice versa. Concerning triangulations the dual graph is unique (except for collinearity).
The important thing about the Voronoi diagram for this paper is its dual relationship to the Delaunay triangulation. If we connect all the pairs of areas from the Voronoi diagram that share a vertex (have adjacent cells) we exactly get the Delaunay triangulation (see Figure 2.1).

2.1.2 Applications

The Voronoi diagram is very useful for computational geometry problems. The properties of the diagram help to find the nearest neighbor for any given point, the edges show the most distant places from the given points. This may be helpful for robot motion planning (avoiding given obstacles), facility location (building a new store as far as possible from already existing stores), pattern recognition (optical character recognition) and computational morphology (like fire spreading out from given points will meet at the Voronoi edges). [Mou00, Lecture 16]

2.2 Gabriel Graph

A Gabriel graph is a plane graph where two points p and q are connected by an edge if and only if the disc having diameter $\overline{pq}$ contains no other point in its interior. As shown in Figure 2.2 the points a and b are connected but b and c are not. The Gabriel graph is a subgraph of the Delaunay triangulation and can be calculated in $O(n)$ time given the Delaunay triangulation.

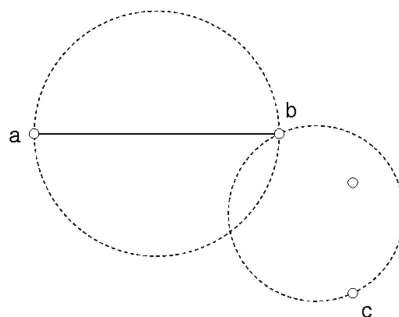

Figure 2.2: Connections in a Gabriel Graph

2.3 Relative Neighbor Graph

The relative neighbor graph is a plane graph where two points p and q are connected by an edge if and only if for all other points s the following is true:

$$dist(p, q) < max\{dist(p, s), dist(q, s)\}$$

As shown in Figure 2.3 the points b and c are connected but a and b are not. The relative neighbor graph is a subgraph of the Gabriel graph and therefore a subgraph of the Delaunay triangulation.

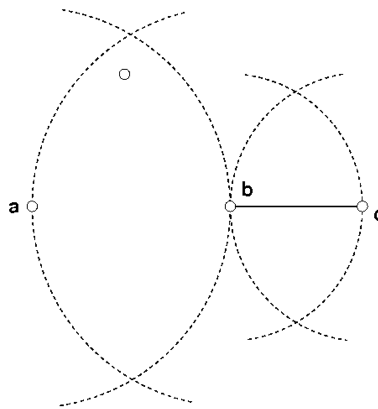

Figure 2.3: Connections in a Relative Neighbor Graph

2.4 Minimum Spanning Tree

A tree is a connected simple graph with no cycles. It usually has got a special vertex called root and several branchings. Each vertex can have several edges, a vertex having no continuing edge (but the one toward the root) is called leaf. Every tree with more than one vertex must have at least two leaves (one probably being the root).
A minimum spanning tree (abbr.: MST) or Euclidean minimum spanning tree is a connected point set with a minimal total edge length. Any vertex can be reached from any other vertex by exactly one path with smallest possible distance (see Figure 2.4). There may be more than one minimum spanning tree but all having the same total edge length.

Theorem 2.4. *A minimum spanning tree of a point set* $P := \{p_1, p_2, ...p_n\}$ *has* $n - 1$ *edges.*

It is very useful to find the minimum spanning tree in a planar graph for getting the shortest connection between any two points. More information about the MST and its connection to the Delaunay triangulation can be found in chapter 5.
The MST is a subgraph of the relative neighbor graph and therefore a subgraph of the Delaunay triangulation.

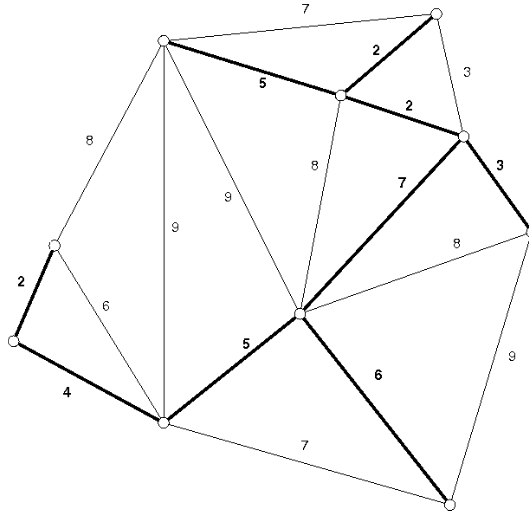

Figure 2.4: Minimum Spanning Tree

2.5 Pairs

There are even structures that are subgraphs of the MST (and therefore of the Delaunay triangulation as well).

A **nearest neighbor pair** connects two points p and q by an edge if and only if they are nearest neighbors. That means the distance from p to q is smaller or equal than the distance from p or q to any other point of the point set.

A **min pair** is an edge, identifying the smallest distance between points in the point set. For k min pairs the following is true

$$1 \leq k \leq \binom{n}{2}$$

as there are $\binom{n}{2}$ possibilities to choose 2 points out of n. The k closest pairs of a point set of n points can be computed in $O((n+k)\ log\ n)$ time and $O(n+k)$ space.

2.6 Minimum Weight Triangulation

The minimum weight (or greedy) triangulation is as the name suggests the triangulation with the minimal total edge length. We may be tempted to think that the Delaunay triangulation minimizes the total edge length, but this assumption is wrong. The Delaunay property (that the circumcircle of a triangle does not contain any other point) allows us to find a counterexample.

As shown in Figure 2.5 the circumcircle of the three points nearly on a line include the other point so Delaunay has to use the longer horizontal edge than the minimum weight triangulation.

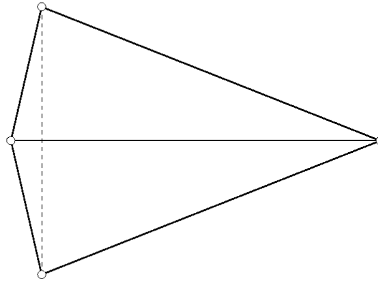

Figure 2.5: Difference between DT (solid) and MWT (dotted)

Recently Wolfgang Mulzer and Günter Rote showed that finding the minimum weight triangulation is NP–hard (see [MR06]).

3 Algorithms

There are several methods to get the Delaunay triangulation that are presented and compared in this chapter.

3.1 Flip

The flipping algorithm was proposed by Charles Lawson in 1972, who also proved that any triangulation of a planar point set can be transformed in any other triangulation by flipping operations. The algorithm starts with an arbitrary triangulation and turns it into the Delaunay triangulation by flipping edges. It is a statical algorithm which means the Delaunay criterion is not fulfilled until the algorithm has ended. This method is quite slow and can reach a worst case duration of $O(n^2)$ but the technique is needed in many further algorithms.
If two adjacent triangles do not fulfill the Delaunay criterion we can switch the common edge to get triangles that do fulfill them (see Figure 3.1). We may get the illegal triangles by checking the circumcircle, as point $d = (d_x, d_y)$ lies inside the circle through $a = (a_x, a_y), b = (b_x, b_y), c = (c_x, c_y)$ (in mathematical positive order) if and only if

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} > 0$$

Assuming there are more than 3 points on a circle then there is more than one way to get a legal triangulation. For the uniqueness of the Delaunay triangulation all points have to be in general position.
Another way to find incorrect Delaunay triangles is to sum up the angles opposite the common edge. If this sum is greater than 180° the edge needs to be flipped. So small angles are avoided as settled in the Delaunay triangulation properties.
After flipping an edge we need to check the surrounding edges recursively as they may have become illegal now and need to be flipped as well.

Theorem 3.1. *Each triangulation of a point set P can be turned in a Delaunay triangulation with $O(n^2)$ flips.*

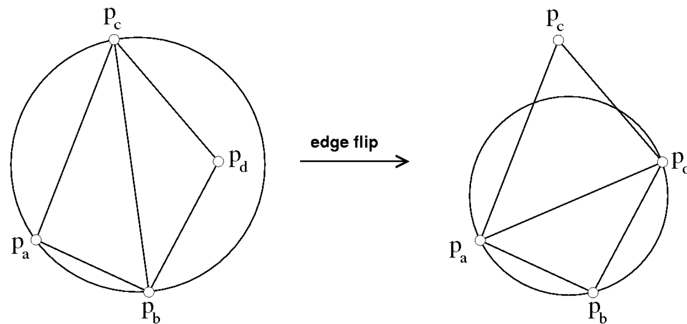

Figure 3.1: Edge Flip

3.2 Incremental Construction

A simple and straightforward algorithm is the incremental construction of the Delaunay triangulation originally found by Lawson. It is simply done by adding one point after another. If the new triangle is illegal one or more edges have to be flipped. Lawson showed that this procedure converges after a finite number of steps. The incremental construction is a dynamical algorithm which means the Delaunay criterion is fulfilled each time before a new point is added.
A similar version of this algorithm is the incremental search method. This way we just join correct points and triangles to the existing triangulation so that no flips are necessary. The Delaunay triangulation is finished when we got the convex hull of the point set triangulated. This may be done like the step by step algorithm (see section 3.7).

3.3 Delete and Build

The delete and build method is also an incremental and dynamical algorithm. By inserting a new point we delete all affected edges. A triangle is affected if its circumcircle includes the new point. So all inner edges are deleted and new edges to every corner are built (see Figure 3.2).
This algorithm is also known as the Bowyer–Watson algorithm because it was found independently by both men at the same time.

3.4 Divide and Conquer

The divide and conquer algorithm was originally found by Lee and Schachter but further on improved by Guibas and Stolfi to being worst case optimal. Dwyer found a modification of the algorithm so that it may run in $O(n\ log\ log\ n)$ and in practice even in linear time. Another optimization of Dwyer is to use horizontal and vertical cuts alternatively to avoid long thin strips and therefore to speed up the algorithm.

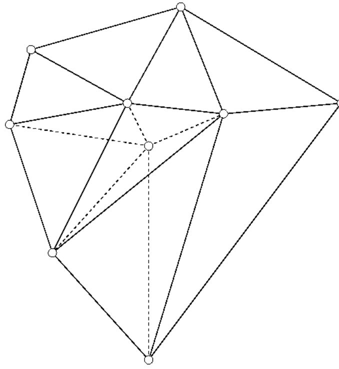

Figure 3.2: Delete old and Build new Edges

Like every divide and conquer algorithm the problem is recursively split into subproblems (handling only half of the original points) that are easier to solve and finally merged together. Small point sets are trivial to triangulate so the most difficult part is the merging which once more includes flipping edges. This algorithm is also used for superlarge point sets that cannot be put in the RAM.

3.5 Sweepline

The sweepline is an imaginary line that sweeps over the point set from left to right (or sometimes from top to bottom). For this purpose the points are ordered by their increasing x–value. If an event occurs like reaching a new point (site event) or passing a circle formed by three adjacent points (circle event) the algorithm updates the triangulation. When the line reaches a point, a new parabola through the point is created and inserted in the sweepline. A circle event leads to a vertex of the Delaunay triangulation by intersecting two of the parabolae (see Figure 3.3). So the area behind the line gets to form a Delaunay triangulation. The algorithm ends when the sweepline has passed all sites and events.
The sweepline algorithm is worst case optimal and was first found by Steven Fortune in 1985 for constructing Voronoi diagrams. In Fortune's version of the algorithm the sweepline is a list of half–edges, also called the beachline because it looks live moving waves on a beach. Jonathan Richard Shewchuck found a sweepline algorithm for constructing higher–dimensional Delaunay triangulations.

3.6 Convex Hull in 3D

A plane Delaunay triangulation can be computed as the three–dimensional convex hull of a paraboloid, the missing third coordinate being calculated as $z = x^2 + y^2$. The proof by Seidel and Edelsbrunner is available for example at Dave Mount's homepage [Mou00, Lecture 17].

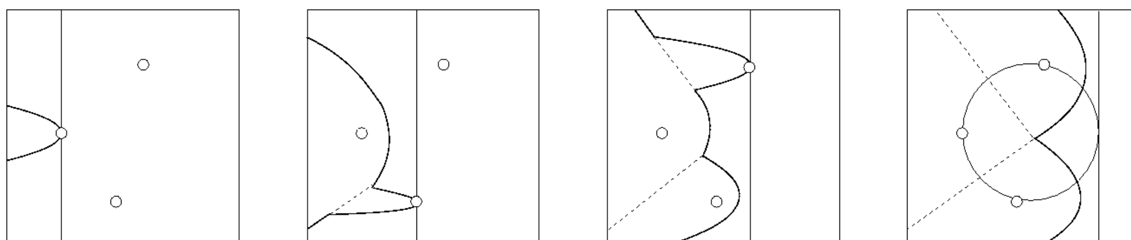

Figure 3.3: Sweepline Algorithm

This method is even applicable in higher dimensions, a n–dimensional triangulation always being a convex hull in dimension $n+1$.

3.7 Step by Step

The step by step algorithm is another statical algorithm which uses the Delaunay criterion (see Definition 1.3) to add a new point. Starting from a baseline between two points (usually the smallest edge) all points that may possibly form a Delaunay triangle are compared. The point with the maximum angle to the vertices of the baseline is the next legal connection for the Delaunay triangulation. The two new edges of the triangle form the new baselines (see Figure 3.4).

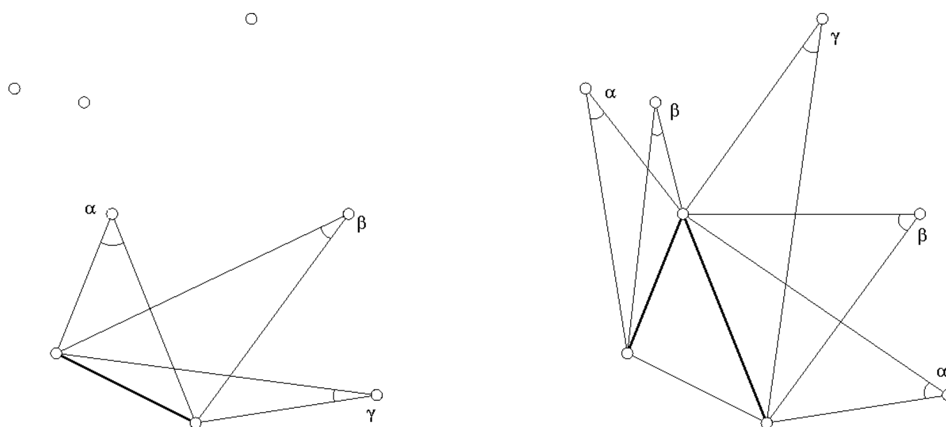

Figure 3.4: Step by Step Algorithm (comparing angles)

Another algorithm called 'Gift Wrapping' is quite similar to this one, introduced by Chand and Kapur. Starting from a point on the convex hull we try to find a vertex with the smallest angle to the first point. Repeating this procedure clockwise or counterclockwise until we get to the start point again, we get the convex hull. In three dimensions the edges are faces and the algorithm works like wrapping a gift into paper.

3.8 Voronoi

Another option to get the Delaunay triangulation is to first find the Voronoi diagram and calculate its dual which can be done in linear time.
The Voronoi diagram can be defined as the intersection of half planes. Given two sites p and q the point set that is closer to site p than to site q is exactly the half plane with a bounding line to the half plane of site q, denoted by $h(p,q)$. So a point s lies in the Voronoi cell of p if and only if s lies within the intersection of $h(p,q)$ for all $p \neq q$ [Mou00, Lecture 16].
This approach gives a construction algorithm by merging all half planes by the bisecting line between the site and all other sites in the plane. For each site in the plane the bisecting lines to any other sites are merged together to create the Voronoi cell (see Figure 3.5).

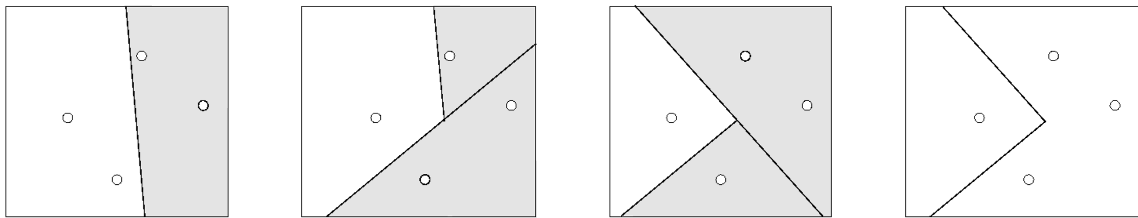

Figure 3.5: Voronoi Algorithm (intersection of half planes)

This method is rather complex with $O(n^2)$ needed time. The already discussed plane sweep or divide and conquer algorithm can compute the Voronoi diagram much faster.

4 Additional Theory about Delaunay Triangulations

The following chapter contains information about the constrained Delaunay triangulation, a special form of the Delaunay triangulation. Furthermore I added some useful data structures for an implementation and finally the applications and runtime comparison of the algorithms presented in the previous chapter.

4.1 Constrained Delaunay Triangulation

A constrained Delaunay triangulation (CDT) is a triangulation due to visibility constraints. A point p is visible to point q if and only if the direct connection between p and q is not intersected by an edge.
In a CDT there are some predetermined edges and the Delaunay triangulation is built around them (see Figure 4.1). So it does not have to be a correct Delaunay triangulation but comes as close as possible to be one. The Delaunay criterion (see Definition 1.3) only has to be valid for points that are visible from the interior of the triangle whereas the predetermined edges are handled as visible and all other edges as invisible.

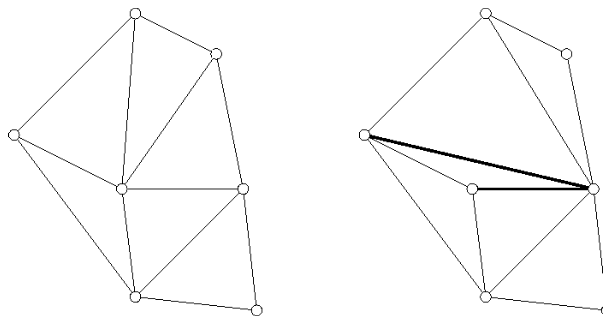

Figure 4.1: DT versus CDT

The predetermined edges are not allowed to be flipped or changed in any way. So the CDT has to be build around them.
A CDT can be computed in $O(n\ log\ n)$ time using a divide and conquer or a sweepline algorithm. If the points and constraint edges form a simple polygon Klein and Lingas found out it can be computed in linear time [KL93, p. 124ff].

4.2 Data Structure

There are different ways to store the details of a triangulation. The primitive use of a list or an array proves to be bad if we need to find neighboring edges as for flip operations (see section 3.1). But there are also better data structures that need $O(n)$ memory and are able to find neighboring edges more quickly.

4.2.1 Doubly Connected Edge List

The doubly connected edge list (DCEL) is a widely used data structure in computational geometry. Muller and Preparata first found it for efficiently saving three–dimensional convex polyhedra information. The DCEL structure is very useful for finding neighbors of vertices, faces or edges.
The system is based on splitting every edge in two opposite directed half–edges. Each half–edge stores a pointer to its twin (on the other side of the edge) and to the point it originates. Furthermore it stores a pointer to the face that touches it. Finally a pointer stores the next edge originating in the current endpoint and bordering the same face (see Figure 4.2). This way we get a whole polygon by following the next pointer till getting to the start half–edge again.

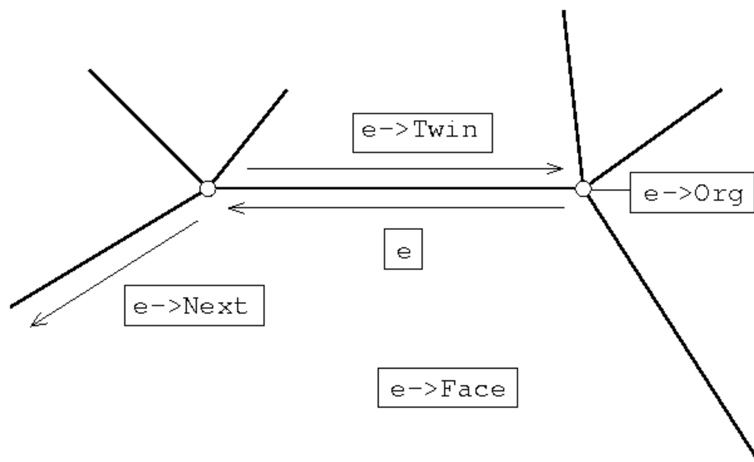

Figure 4.2: Doubly Connected Edge List

There are two more object types besides the half–edges: the vertex, which stores a pointer to an arbitrary edge originating from that point, and the face, which stores a pointer to an arbitrary edge at its border. It is helpful to define an infinite face outside the convex hull, which the outer half–edge pointers may point at.
Additionally we may add a previous pointer to the half–edge ending in the current origin point. Sometimes an additional incremental numeration of the edges may be useful, some other time an attribute information may be needed.

4.2.2 Quad Edge Structure

The quad edge data structure has the big advantage of representing the topology of the dual graph as well. Every edge is defined as a line between two vertices and also between two faces. So the dual graph can be build by just replacing vertices by faces and vice versa. This means the quad edge structure is very useful if we want to get the Delaunay triangulation from the Voronoi diagram or the other way round.
Quad edge is an advancement of the winged edge structure (from Bruce Baumgart 1975) and was invented by Guibas and Stolfi in 1985. Every edge object stores the origin and destination vertex, the left and the right face of the edge and four more edges. One edge starting in origin and having the left face as its right face, one edge starting in origin and having the right face as its left face and two more edges starting in the destination vertex bordering the two faces as well (see Figure 4.3).

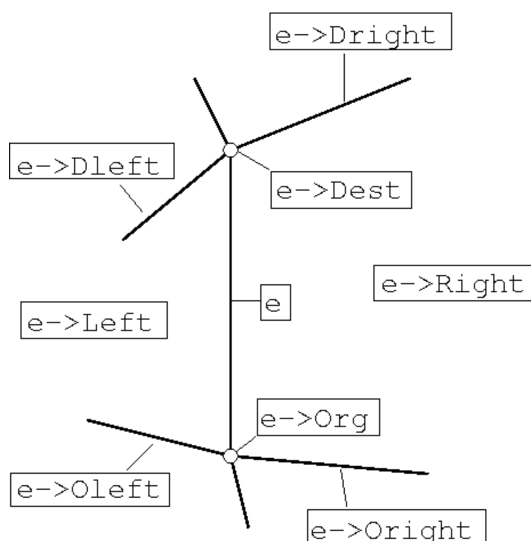

Figure 4.3: Quad–Edge Structure

Again we have two more object types: each vertex stores a pointer to an outgoing edge and each face stores a pointer to an edge on its border. We may also store additional attribute information if necessary.
Dobkin and Laszlo transformed the quad edge idea to three–dimensional space in 1989, but their structure has been found difficult to be implemented.

4.2.3 Directed Acyclic Graph

A directed acyclic graph (DAG) is a directed graph containing no cycles. Each element of the graph stores the edges and vertices of a triangle in the triangulation process. The leaves of the DAG correspond to the triangles of the final Delaunay triangulation. This structure makes point location fast and easy.

By inserting a new point into a triangle we create three new triangles and three new leaves in the DAG. We may have to do some flip operations that may change existing triangles and therefore the DAG.
The DAG may be used for implementing the incremental algorithm. To start with, it is useful to create a big triangle containing all points of a given point set P (like suggested in [dBvKOS97]). After triangulating the point set including the new points, these points p_{-1}, p_{-2}, p_{-3} and their edges are deleted. Therefore they have to be chosen far enough, so they do not interfere with any other triangle of the original point set P. To assure this but not to get too huge coordinates we take

$$p_{-1} := (3M, 0)$$

$$p_{-2} := (0, 3M)$$

$$p_{-3} := (-3M, -3M)$$

where M is the maximum absolute coordinate of any point of P.
Adding a new point p_r to the triangulation is done by locating the triangle including the point and splitting it into new triangles. If p_r lies on an already existing edge, there are only two new triangles to be build (see Figure 4.4). This cannot happen if the points are in general position, but for triangulating big point sets (more than 1000 points) it occurs quite often.

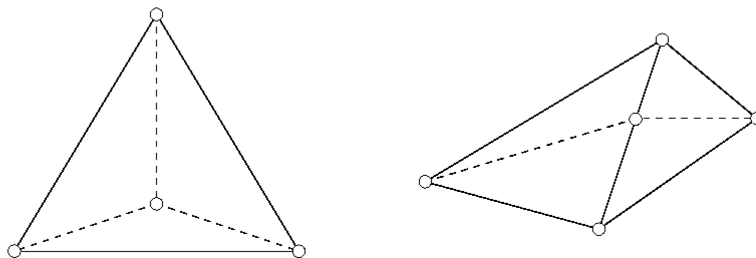

Figure 4.4: Adding a new Point and Splitting the Triangle

In Figure 4.5 we add a new point in triangle $\Delta 1$ and split this triangle into $\Delta 4, \Delta 5, \Delta 6$ which become children of $\Delta 1$. Furthermore we have to flip the edge $\overline{p_i p_j}$ and get the triangles $\Delta 7$ and $\Delta 8$. In the DAG the new triangles are leaves of $\Delta 2$ and $\Delta 6$. Another flip operation of edge $\overline{p_i p_k}$ is necessary to change the triangles $\Delta 3$ and $\Delta 8$ into pointers at $\Delta 9$ and $\Delta 10$.
So for locating a point p_r that should be inserted in the triangulation we start by the initial triangle and its three children. One of them has to contain p_r and we go on searching the triangles it points at. Continuing this way we get to the final leaf, which is the triangle in the current triangulation that contains p_r. Since any node can have at most three children, the search can be done in linear time.

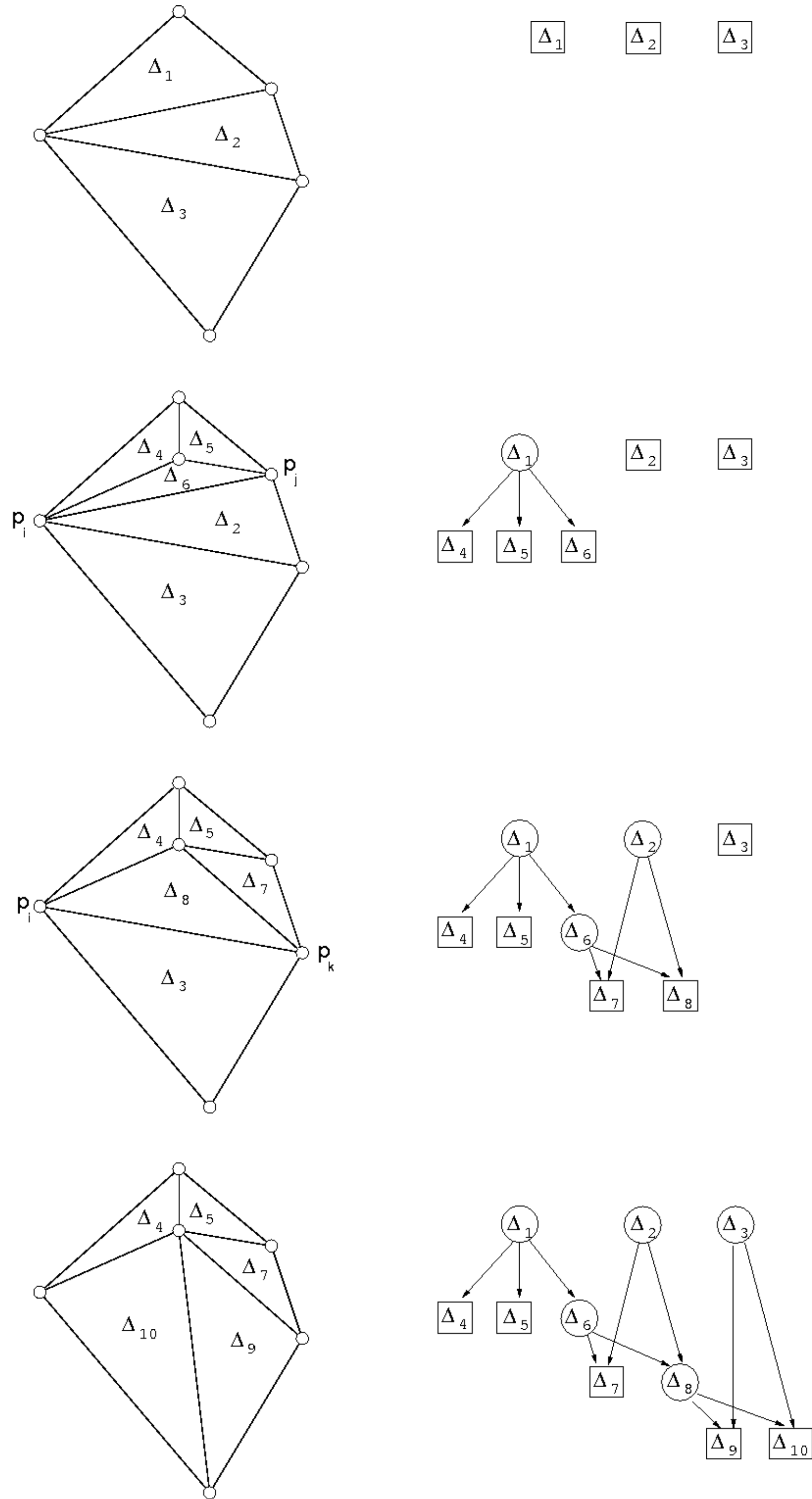

Figure 4.5: DAG Structure for Inserting a new Point

4.2.4 Tree Structure

A very well–known data structure is the tree structure. DAGs can be seen as a generalization of trees, but for some applications we prefer a tree to a DAG.
The sweepline algorithm for example uses a tree to display the moving beachline. At best we use a self-balancing binary search tree ('splay tree'), where the leaves correspond to the arcs of the beachline. If the sweepline moves from top to bottom of the point set, the leaves (from left to right) represent the arcs ordered from left to right. The internal nodes of the tree are the breakpoints which furthermore are the vertices of the Delaunay triangulation. This way we just need $O(log\ n)$ time to find an arc above a new site in the tree.
For non–uniform mesh generation the data structure may be based on quadtrees. A quadtree is a rooted tree where every internal node has four children. Representing a square each child is one of the four quadrants, which may be quartered again and again giving more layers to the tree. The three–dimensional version needs an octree where every internal node has eight children.

4.3 Applications

A very useful application of the Delaunay triangulation is geographic terrain modeling (like GIS). Avoiding acute angles makes it look real and we can imagine a three–dimensional height interpolation.
Delaunay triangulations are used for constructing three–dimensional polyhedral objects in computer graphics, so called high–quality meshes. There are several algorithms to get uniform triangles avoiding small and large angles which use the Delaunay edges as input.
Another important application in mathematics is the finite element method (FEM) for solving partial differential equations.
As mentioned above we are able to construct the Voronoi diagram out of the Delaunay triangulation in linear time. The applications of the Voronoi diagram were already handled in section 2.1.2.

4.4 Runtime Comparison

There are many worse algorithms like the naive flip approach in $O(n^2)$ time, but the optimal runtime for a two–dimensional Delaunay triangulation is $O(n\ log\ n)$. The time depends on efficient data structures too (see section 4.2) and may in practical experience have linear average execution times. But still the worst case takes $O(n\ log\ n)$ time.
According to Krämer the sweepline is the best algorithm for point sets up to 11 000 points. For more points the incremental algorithm using a quad tree performs best. Krämer states that data structures are very much influencing the runtime.

Su and Drysdale [SD95] compared several algorithms and found Dwyer's divide and conquer algorithm to perform best (even for not uniformly distributed point sets) followed by Fortune's sweepline algorithm. The (improved) incremental algorithm is only third in line. Shewchuck [She96] achieved the same results and claims that one million points can be triangulated correctly in a minute on a fast workstation.
Seidel showed that even after sorting the points by their x–coordinate computing the Delaunay triangulation requires $O(n \; log \; n)$ time. Djijev and Lingas have shown the same time requirements for constructing the Voronoi diagram.

5 Minimum Spanning Tree Algorithms

Cayley stated in 1889 that there are n^{n-2} spanning trees of a point set with n points. So we need an intelligent system to find the right (minimal) one.
Finding the MST is a fundamental problem in computational geometry. The MST is used for network design (like communication, system engineering, flow in pipeline systems, computer, rail and road networks) and furthermore to approximate mathematical problems like the traveling salesman problem or cluster analysis.

Theorem 5.1. *Each vertex in the MST has a maximal degree of* 6.

Proof. No two MST edges can form an angle smaller than $60°$. If they would (contradiction), the angle α between the edges must be smaller than at least one of the other angles β and γ in the triangle. Therefore the edge opposite to α becomes part of the MST instead of the other edge opposite the greatest angle β or γ, because the connection to the third vertex is shorter. To prove this we need to show, that the biggest angle of a triangle lies opposite the longest side.
The Law of Sines states that $\frac{a}{sin(\alpha)} = \frac{b}{sin(\beta)} = \frac{c}{sin(\gamma)}$. In other words $\frac{a}{b} = \frac{sin(\alpha)}{sin(\beta)}$. Let w.l.o.g. a be the longest side and $\gamma > 0$. Now we need to differ four cases:
a) $\alpha < \frac{\Pi}{2}$ and $\beta < \frac{\Pi}{2}$: Then $a > b \Rightarrow sin(\alpha) > sin(\beta) \Rightarrow \alpha > \beta$
b) $\alpha > \frac{\Pi}{2}$ and $\beta < \frac{\Pi}{2}$: Then $\alpha > \beta$
c) $\alpha < \frac{\Pi}{2}$ and $\beta > \frac{\Pi}{2}$: Then $a > b \Rightarrow sin(\alpha) > sin(\beta) \Rightarrow sin(\alpha) > sin(\Pi - \beta) \Rightarrow \alpha > \Pi - \beta \Rightarrow \alpha + \beta + \gamma > \Pi$ (contradiction)
d) $\alpha > \frac{\Pi}{2}$ and $\beta > \frac{\Pi}{2}$: Then $\alpha + \beta + \gamma > \Pi$ (contradiction) □

The minimum spanning tree was already mentioned in section 2.4. There are several methods to get the correct MST. Some of them are presented in the following section. The edge weight is in this paper according to the distance between two points.
The total length of the MST for n uniformly distributed points over a two–dimensional area A is asymptotically proportional to $\sqrt{nA}$ [CW08]. This is based on Hammersley finding an approximation to the Traveling Salesman Problem (which will not be further discussed in this paper), that may be used as an upper bound to the MST [BHH59].
The important relationship between MST and Delaunay triangulations is (like mentioned before) the MST being a subgraph of the Delaunay triangulation. So it is useful to operate the MST algorithms on the edges of the triangulation to speed up the algorithms. The MST can be found in $O(n\ log\ n)$ time by the help of Delaunay.

5.1 Greedy Algorithm

Greedy algorithms try to find the global optimum solution by getting the local optimum during the algorithm. In the case of MST most greedy algorithms can be brought down to the red and blue rule.

The algorithm marks every edge as red or blue, the blue edges being the MST in the end. It works with two rules concerning cycles and cuts. A 'cycle' is defined as a closed path starting and ending in one and the same vertex. A tree is defined as being a connected simple graph with no cycles. A 'cut' is performed by a subset S of the original point set, so that all edges with exactly one endpoint in S belong to the cut.

The two rules are:

- Red Rule – Having a cycle with no red edges, color the maximal edge of the cycle red.
- Blue Rule – Having a cut with no blue edges, color the minimal edge of the cut blue.

This rules are applied until all edges are colored or eventually may stop when $n-1$ edges are colored blue (see Figure 5.1).

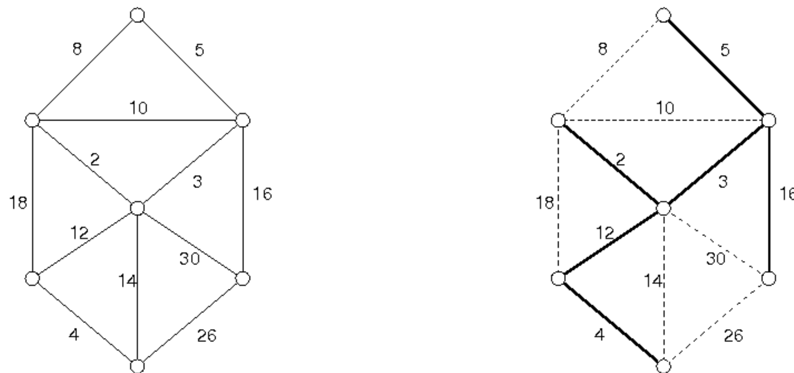

Figure 5.1: Red (dashed) and Blue (bold) Coloring

Theorem 5.2. *The greedy algorithm terminates.*

Proof. Suppose e is an uncolored edge. The blue edges cannot form a cycle so they are a forest F. Edge e either forms a cycle in F or connects two components of F. In the first case e has to be the biggest edge of the cycle and gets colored red. In the second case e is part of a cut without any blue edges, so the smallest edge of the cut gets colored blue. In both cases the set of uncolored edges is decreased by one. This process can be iterated till no uncolored edges are left. The algorithm terminates after m steps, where m is the number of edges. □

Theorem 5.3. *At each step during the algorithm there exists a minimum spanning tree T containing all blue edges and no red one.*

Proof. Induction base: When no edge is colored, the statement holds for every MST.
Induction step: a) Suppose the statement is true before applying the blue rule.
Let D be the chosen cut and e the edge to be colored blue. If $e \in T$ the statement stays true. Otherwise we must get a cycle C by adding e to T. Let f be another edge in $C \cap D$. f has to be in T and cannot be red. By applying the blue rule we got e so f cannot be blue and its weight has to be bigger or equal than e. Therefore f has to be uncolored and of bigger or equal weight than e. So $T \cup \{e\} - \{f\}$ is a MST fulfilling the statement.
b) Suppose the statement is true before applying the red rule.
Let C be the chosen cycle and e the edge to be colored red. If $e \notin T$ the statement stays true. Otherwise we must get a cut D by deleting e from T. Let f be another edge in $C \cap D$. f is not in T and therefore cannot be blue. By applying the red rule we got e so f cannot be red and its weight has to be less or equal than e. Therefore f has to be uncolored and of less or equal weight than e. So $T \cup \{f\} - \{e\}$ is a MST fulfilling the statement. □

After terminating the algorithm the blue edges form a MST.

5.2 Kruskal's Algorithm

Joseph Kruskal's algorithm (found in 1956) can be seen as a special case of the greedy algorithm. The edges are considered in ascending order. If an edge e has both endpoints in the same blue tree we apply the red rule (cycle) and color e red. Otherwise the blue rule applies (the cut being between the blue tree of one endpoint and the rest of the point set) and e is colored blue (see Figure 5.2).
As the edges need to be ordered, Kruskal's algorithm needs a total time of $O(m \log m)$, where m is the number of edges.

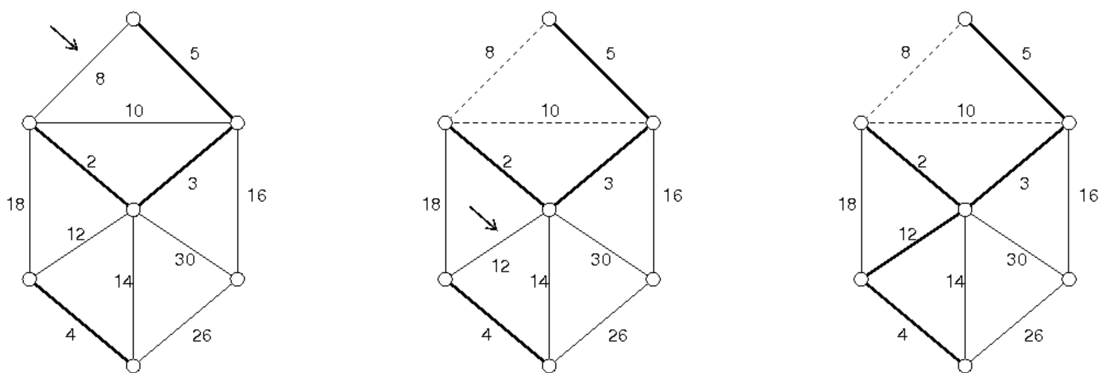

Figure 5.2: Kruskal's Algorithm

Theorem 5.4. *Kruskal's algorithm produces a spanning tree of minimal length.*

Proof. Kruskal's algorithm is a special case of the greedy algorithm where the edges are handled in ascending order. So this theorem is true because of Theorem 5.2 and 5.3. □

5.3 Reverse–Delete Algorithm

The reverse–delete algorithm is the reverse of Kruskal's algorithm and also a special case of the greedy algorithm. It starts with the original graph and deletes edges until it gets to the MST. The edges are handled in decreasing order. If deleting an edge does not disconnect the graph, the edge is deleted. According to the coloring this means the edge is part of a cycle and the maximal edge of that cycle as well (because of the decreasing order), so it must be colored red.
The algorithm starts with a connected graph and only deletes edges, if that does not disconnect the graph. The edges are deleted in decreasing order, so each deleted edge is the maximal edge of a cycle. Therefore we achieve a correct MST with the reverse–delete algorithm.
The running time with $O(m\ log\ m(log\ log\ m)^3)$ is longer than most of the other algorithms.

5.4 Prim's Algorithm

This algorithm was first found by Vojtech Jarnik in 1930, independently in 1957 by Robert Prim and once again by Edsger Dijkstra in 1959. It starts from one arbitrary point and connects its neighbors by minimal edges to a connected tree. Using optimal data structures the algorithm just takes $O(m + n\ log\ n)$ time, m being the number of edges, n being the number of vertices.

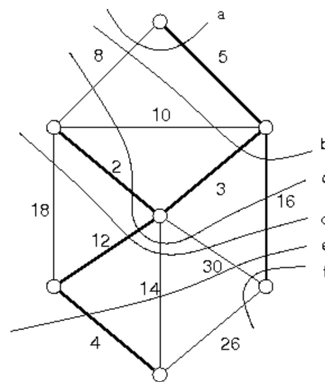

Figure 5.3: Prim's Algorithm (steps in alphabetical order)

Compared to the greedy algorithm, Prim just applies the blue rule. Starting with one vertex a cut delivers the smallest edge to the next neighbor, that has to be colored blue. For the next step we have two vertices and a cut between them and the rest of the point set (see Figure 5.3).

Theorem 5.5. *Prim's algorithm produces a spanning tree of minimal length.*

Proof. Prim's algorithm is a special case of the greedy algorithm just considering the edges in uncolored cuts. When the algorithm is finished we get $n-1$ blue edges. All remaining edges have to be colored red. According to Theorem 5.3 the blue edges form a MST. □

5.5 Boruvka's Algorithm

This algorithm was first published by Otokar Boruvka in 1926. It works by handling each vertex (or furthermore group of vertices) on its own and adding the minimal outgoing edge to it. The generated groups are joined together to build a spanning tree. According to the coloring, Boruvka applies the blue rule for each point and in the next step(s) for each group of points (see Figure 5.4).

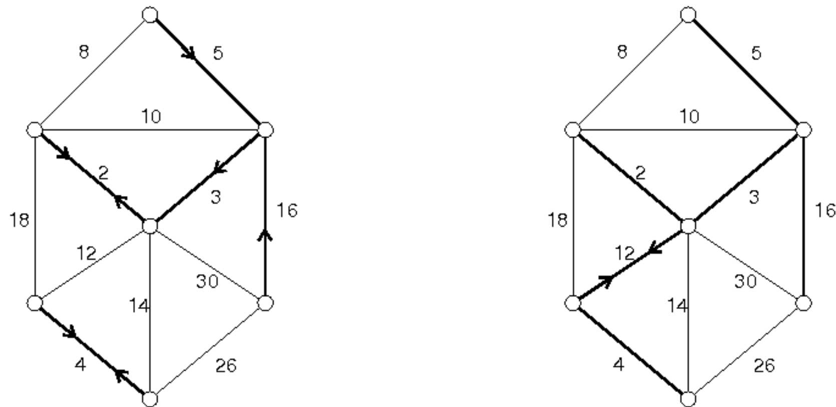

Figure 5.4: Boruvka's Algorithm (in two steps)

Boruvka's algorithm takes $O(m\ log\ n)$ time. It may be brought down to linear time by teaching the algorithm to also delete all (red) edges inside a group of vertices, that would form a cycle. Many faster attempts are based on Boruvka's algorithm and may run in $O(m)$ time. The best algorithm till now has been found by Bernard Chazelle in 2000 which works in $O(m\ \alpha(m,n))$ time, where α is the inverse of the Ackermann function (which grows very slowly and may usually be seen as a constant).

List of Figures

2.1 Delaunay Triangulation (solid) and Voronoi Diagram (dashed) 10
2.2 Connections in a Gabriel Graph 11
2.3 Connections in a Relative Neighbor Graph 12
2.4 Minimum Spanning Tree 13
2.5 Difference between DT (solid) and MWT (dotted) 14

3.1 Edge Flip 16
3.2 Delete old and Build new Edges 17
3.3 Sweepline Algorithm 18
3.4 Step by Step Algorithm (comparing angles) 18
3.5 Voronoi Algorithm (intersection of half planes) 19

4.1 DT versus CDT 20
4.2 Doubly Connected Edge List 21
4.3 Quad–Edge Structure 22
4.4 Adding a new Point and Splitting the Triangle 23
4.5 DAG Structure for Inserting a new Point 24

5.1 Red (dashed) and Blue (bold) Coloring 28
5.2 Kruskal's Algorithm 29
5.3 Prim's Algorithm (steps in alphabetical order) 30
5.4 Boruvka's Algorithm (in two steps) 31

A.1 DT and MST of 10 points 35
A.2 DT and MST of 100 points 35
A.3 DT and MST of 1000 points 35

Bibliography

[BHH59] Beardwood, Halton, and Hammersley. The shortest path through many points. In *Proceedings of the Cambridge Philosophical Society 55*, pages 299–327, 1959.

[CW08] Annabel Cartwright and Anthony Whitworth. The Statistical Analysis of Star Clusters. Monthly Notices of the Royal Astronomical Society 348, 589-598, February 2008.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry, Algorithms and Applications*. Springer, March 1997.

[Jun90] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI Wissenschaftsverlag, 1990.

[KL93] R. Klein and A. Lingas. A linear–time randomized algorithm for the bounded Voronoi diagram of a simple polygon. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, 1993.

[Mou00] Dave Mount. Spring 2003, Computational Geometry. `http://www.cs.wustl.edu/~pless/506.html`, 2000. Status 21 January 2008.

[MR06] Wolfgang Mulzer and Günter Rote. Minimum weight triangulation is NP–hard. In *Proceedings of the 22nd Annual Symposium on Computational Geometry*, pages 1–10, 2006.

[SD95] Peter Su and Robert L. Scot Drysdale. A Comparison of Sequential Delaunay Triangulation Algorithms. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, June 1995.

[She96] Jonathan Richard Shewchuck. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. Applied Computational Geometry: Towards Geometric Engineering, May 1996.

A Programming Example

Attached is the C++ code of my program implementation. It performs the Delaunay triangulation in $O(n\ log\ n)$ time and $O(n)$ storage, n being the number of points. The MST needs $O(m\ log\ m)$ time, where m is the number of edges of the triangulation. Therefore on my personal computer (2.4 GHz, 256 MB RAM) computing 10.000 points takes 2.2 seconds.
I implemented the incremental algorithm, based on a static array of randomized points and a big starting triangle around the point set (like suggested in section 4.2.3). I also used the dynamical DAG structure that was introduced in that section, which makes it easy to find the actual point position. In most cases a new point creates 3 new DAGs, but if the new point lies on an already existing line, we get a special case of building 4 new DAGs (also already explained in section 4.2.3). Duplicate points are ignored.
If the circumcircle of the new triangles includes another point (checked by the determinant condition), one or more edges have to be flipped (see section 3.1) and 2 double–linked new DAGs are created.
The number of DAGs must be equal to

$$1 + 3 * (points - duplicates) + 2 * flips + linepoints = DAGs.$$

The number of edges can be calculated by

$$2 + DAGs - flips = edges.$$

Each edge element stores the origin and destination point (index of the point array), its length and the DAGs lying left and right to the edge. Also stored is the number of links (for deleting the edges in the end) and the information if the edge is visible in the final graph (only the tree leaves). Each DAG element stores its three children, its three edges and its three corner points in counterclockwise order.
The convex hull function at the end of the triangulation process makes sure that the outmost edges pointing to the three vertices of the starting triangle are flipped if necessary. This may happen in cases like shown in figure 2.5, where three points are almost on a line.
For computing the MST I chose Kruskal's greedy algorithm. The edges are sorted in ascending order (in this program according to the heap sort algorithm) and every point gets a group variable according to the part of the tree it belongs. If an edge is added to the MST, the connected point or points are matched to the point or points on the other side of the edge.

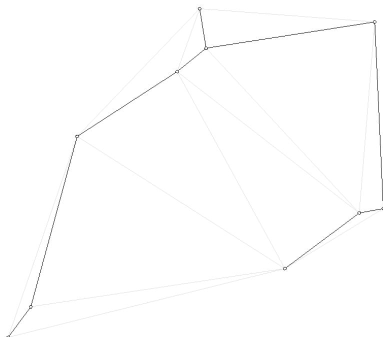

Figure A.1: DT and MST of 10 points

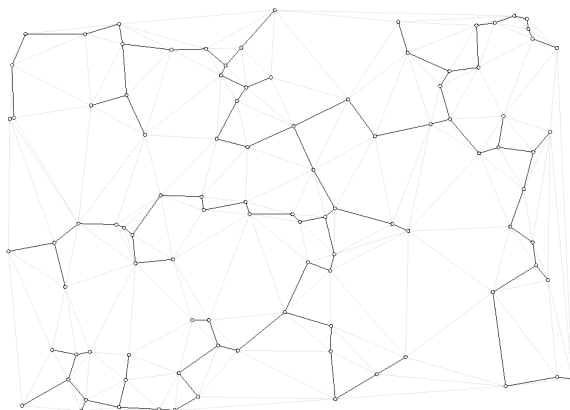

Figure A.2: DT and MST of 100 points

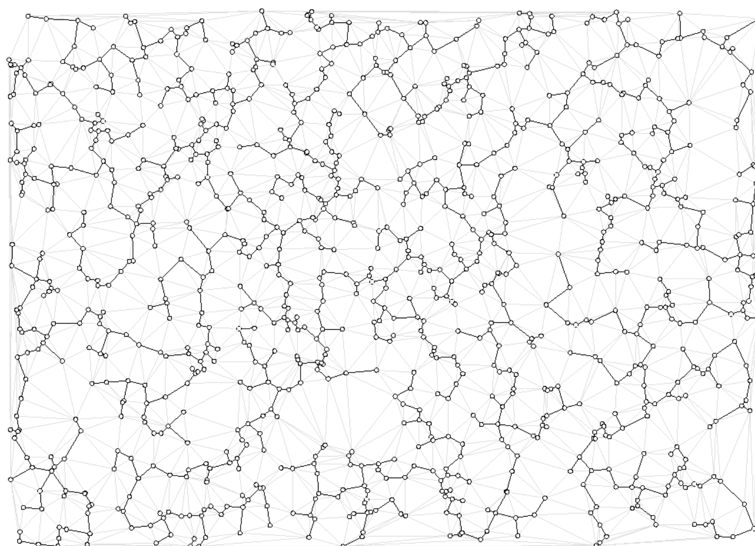

Figure A.3: DT and MST of 1000 points

The program returns the time needed to do the triangulation, the heap sort and the MST. It also counts the performed flip operations, the number of created DAGs and edges and how often the special case 'point on a line' occurs. The graphical output is written into a FIG–file, the MST one level above the Delaunay triangulation.

0.031 seconds till triangulation is done.
0.046 seconds till edges are sorted.
0.078 seconds till MST is computed.
1000 points triangulated.
2935 performed flip operations.
8871 DAG instances created.
0 ignored duplicate points.
0 points on a line.
5938 edges constructed.
122775 total MST length.

Table A.1: Sample Output for 1000 points

0.437 seconds till triangulation is done.
0.812 seconds till edges are sorted.
2.25 seconds till MST is computed.
10000 points triangulated.
29921 performed flip operations.
89847 DAG instances created.
3 ignored duplicate points.
13 points on a line.
59928 edges constructed.
384573 total MST length.

Table A.2: Sample Output for 10000 points

A.1 point.h

```
#ifndef POINT_H
#define POINT_H

class POINT {
public:
    POINT();                      //standard constructor (values set to -1)
    void Setxy (int index,int x,int y);
      //sets the coordinates and initial MST group
    void SetGroup (int index);    //sets the MST group
    int Getx();                   //returns the x coordinate
    int Gety();                   //returns the y coordinate
    int GetGroup();               //returns the MST group

private:
    int x;
    int y;
    int group;
};

#endif
```

A.2 point.cpp

```
#include "point.h"

POINT::POINT() {
    x=y=group=-1;
}

void POINT::Setxy (int index,int x,int y) {
    this->x=x;
    this->y=y;
    group=index;
}

void POINT::SetGroup (int index) {
    group=index;
}

int POINT::Getx() {
    return x;
}
```

```
int POINT::Gety() {
    return y;
}

int POINT::GetGroup() {
    return group;
}
```

A.3 edge.h

```
#ifndef EDGE_H
#define EDGE_H

class DAG;
class POINT;

class EDGE {
public:
    EDGE();                                  //standard constructor
    void SetPoints (POINT* point_array,int a,int b);
      //sets the edge endpoints and calculates the length
    void SetDags (DAG* a,DAG* b);            //sets left and right DAGs
    void SetLeftDag (DAG* a);                //sets the left DAG to the edge
    void SetRightDag (DAG* a);               //sets the right DAG to the edge
    void SetInvisible();                     //sets visibility off
    void IncLinkage();                       //increments linkage
    void DecLinkage();                       //decrements linkage
    int GetVisibility();                     //returns the state of visible
    int GetLinkage();                        //returns number of links
    int GetOrig();                           //returns the startpoint
    int GetDest();                           //returns the endpoint
    float GetLength();                       //returns the length
    DAG* GetLeft();                          //returns the left DAG
    DAG* GetRight();                         //returns the right DAG
    void CheckEdge (POINT* point_array,int point);
      //recursive function to check if the edge needs to be flipped
    void Flip (POINT* point_array,int inner_point,DAG* inside,
      DAG* outside,int from,int to);
      //flipping the edge and checking for more necessary flips

private:
    int linkage;                             //counts how often edge is linked
    int visible;                             //states visibility in the graph
```

```
    int orig;
    int dest;
    float length;
    DAG* left;
    DAG* right;
};

#endif
```

A.4 edge.cpp

```
#include <string.h>
#include <math.h>
#include "edge.h"
#include "dag.h"
#include "point.h"
#include "functions.h"

EDGE::EDGE() {
    orig=dest=-1;
    length=-1;
    left=right=NULL;
    linkage=0;
    visible=1;
}

void EDGE::SetPoints (POINT* point_array,int a,int b) {
    orig=a;
    dest=b;
    length=sqrt(pow(abs(point_array[a].Getx()-point_array[b].Getx()),2)+
      pow(abs(point_array[a].Gety()-point_array[b].Gety()),2));
}

void EDGE::SetDags (DAG* a,DAG* b) {
    left=a;
    right=b;
}

void EDGE::SetLeftDag (DAG* a) {
    left=a;
}

void EDGE::SetRightDag (DAG* a) {
```

```
    right=a;
}

void EDGE::SetInvisible() {
    visible=0;
}

void EDGE::IncLinkage() {
    linkage++;
}

void EDGE::DecLinkage() {
    linkage--;
}

int EDGE::GetVisibility() {
    return visible;
}

int EDGE::GetLinkage() {
    return linkage;
}

int EDGE::GetOrig() {
    return orig;
}

int EDGE::GetDest() {
    return dest;
}

float EDGE::GetLength() {
    return length;
}

DAG* EDGE::GetLeft() {
    return left;
}

DAG* EDGE::GetRight() {
    return right;
}
```

```
void EDGE::CheckEdge (POINT* point_array,int point) {
    DAG* outer_dag;
    DAG* inner_dag;
    int from;
    int to;

    if(IsLeft(point_array,GetOrig(),GetDest(),point))
    {
        outer_dag=GetRight();
        inner_dag=GetLeft();
        from=GetOrig();
        to=GetDest();
    }
    else
    {
        outer_dag=GetLeft();
        inner_dag=GetRight();
        from=GetDest();
        to=GetOrig();
    }
    if(outer_dag!=NULL&&InsideCircle(point_array,outer_dag->GetCorner(2),
      outer_dag->GetCorner(1),outer_dag->GetCorner(0),point))
        Flip(point_array,point,inner_dag,outer_dag,from,to);
}

void EDGE::Flip (POINT* point_array,int inner_point,DAG* inside,DAG*
  outside,int from,int to) {
    int outer_point;
    int outer_corner;
    int inner_corner;

    for(int i=0;i<3;i++)                    //find third point of DAG
    {
        if(outside->GetEdge(i)==this)
        {
            outer_corner=(i+2)%3;
            outer_point=outside->GetCorner(outer_corner);
        }
        if(inside->GetEdge(i)==this)
            inner_corner=(i+2)%3;
    }
    SetInvisible();
    number_of_flips++;
```

```
    outside->SetTwoNewDags(point_array,from,to,inner_point,outer_point,
      outer_corner,inside,inner_corner);
}
```

A.5 dag.h

```
#ifndef DAG_H
#define DAG_H

#include "edge.h"
#include "point.h"

class DAG {
public:
    static int on_line;                          //new point lies on an edge
    DAG();                                       //standard constructor
    DAG* GetChild (int which);                   //returns a following dag
    int GetCorner (int which);                   //returns a corner point
    EDGE* GetEdge (int which);                   //returns an edge
    void SetChild (int which,DAG* dag);          //sets the following dag
    void SetCorners (int a,int b,int c);         //sets the corners
    void SetEdges (EDGE* a,EDGE* b,EDGE* c);//sets the edges
    void SetTwoNewDags(POINT* point_array,int from,int to,int
      inner_point,int outer_point,int outer_corner,DAG* inside,int
      inner_corner);
      //creates 2 new dags after flipping an edge leaving middle to NULL
    void SetThreeNewDags (POINT* point_array,int point);
      //creates 3 new dags around the new point
    void SetFourNewDags (POINT* point_array,int point);
      //creates 4 new dags around the new point leaving right to NULL

private:
    DAG* left;
    DAG* middle;
    DAG* right;
    EDGE* edge_a;
    EDGE* edge_b;
    EDGE* edge_c;
    int point_a;
    int point_b;
    int point_c;
};

#endif
```

A.6 dag.cpp

```
#include <string.h>
#include "dag.h"
#include "functions.h"

DAG::DAG() {
    left=middle=right=NULL;
    edge_a=edge_b=edge_c=NULL;
    point_a=point_b=point_c=-1;
}

//--------------------------------------------------------------------------//

DAG* DAG::GetChild (int which) {
    if(which==0)
        return left;
    if(which==1)
        return middle;
    if(which==2)
        return right;
    return NULL;
}

//--------------------------------------------------------------------------//

int DAG::GetCorner (int which) {
    if(which==0)
        return point_a;
    if(which==1)
        return point_b;
    if(which==2)
        return point_c;
    return -1;
}

//--------------------------------------------------------------------------//

EDGE* DAG::GetEdge (int which) {
    if(which==0)
        return edge_a;
    if(which==1)
        return edge_b;
```

```
    if(which==2)
        return edge_c;
    return NULL;
}

//--------------------------------------------------------------------------//

void DAG::SetChild (int which,DAG* dag) {
    if(which==0)
        left=dag;
    if(which==1)
        middle=dag;
    if(which==2)
        right=dag;
}

//--------------------------------------------------------------------------//

void DAG::SetCorners (int a,int b,int c) {
    point_a=a;
    point_b=b;
    point_c=c;
}

//--------------------------------------------------------------------------//

void DAG::SetEdges (EDGE* a,EDGE* b,EDGE* c) {
    edge_a=a;
    edge_a->IncLinkage();
    edge_b=b;
    edge_b->IncLinkage();
    edge_c=c;
    edge_c->IncLinkage();
}

//--------------------------------------------------------------------------//

void DAG::SetTwoNewDags(POINT* point_array,int from,int to,int
  inner_point,int outer_point,int outer_corner,DAG* inside,int
  inner_corner) {
    DAG* left;
    DAG* right;
    EDGE* new_edge;
```

```
EDGE* actual_edge;

new_edge=new EDGE();
number_of_edges++;
left=new DAG();
right=new DAG();

left->SetCorners(from, outer_point, inner_point);
left->SetEdges(GetEdge((outer_corner+2)%3),new_edge, inside->
  GetEdge(inner_corner));
right->SetCorners(inner_point, outer_point, to);
right->SetEdges(new_edge, GetEdge(outer_corner), inside->
  GetEdge((inner_corner+2)%3));
new_edge->SetDags(left, right);
new_edge->SetPoints(point_array, outer_point, inner_point);

inside->SetChild(0, left);                //set links to the new DAGs
inside->SetChild(2, right);
SetChild(0, left);
SetChild(2, right);

actual_edge=GetEdge((outer_corner+2)%3);
if(IsLeft(point_array, actual_edge->GetOrig(), actual_edge->GetDest(),
  inner_point))
    actual_edge->SetLeftDag(left);
else
    actual_edge->SetRightDag(left);
actual_edge=GetEdge(outer_corner);
if(IsLeft(point_array, actual_edge->GetOrig(), actual_edge->GetDest(),
  inner_point))
    actual_edge->SetLeftDag(right);
else
    actual_edge->SetRightDag(right);
actual_edge=inside->GetEdge((inner_corner+2)%3);
if(actual_edge->GetDest()==inner_point)
    actual_edge->SetLeftDag(right);
else
    actual_edge->SetRightDag(right);
actual_edge=inside->GetEdge(inner_corner);
if(actual_edge->GetOrig()==inner_point)
    actual_edge->SetLeftDag(left);
else
    actual_edge->SetRightDag(left);
```

```
    //legalize edges
    GetEdge((outer_corner+2)%3)->CheckEdge(point_array,inner_point);
    GetEdge(outer_corner)->CheckEdge(point_array,inner_point);
}

//--------------------------------------------------------------------------//

void DAG::SetThreeNewDags (POINT* point_array,int point) {
    int a=GetCorner(0);
    int b=GetCorner(1);
    int c=GetCorner(2);
    EDGE* actual_edge;
    EDGE* edgea,*edgeb,*edgec;
    DAG* daga,*dagb,*dagc;

    edgea=new EDGE();
    edgeb=new EDGE();
    edgec=new EDGE();
    number_of_edges=number_of_edges+3;

    daga=new DAG();
    dagb=new DAG();
    dagc=new DAG();

    SetChild(0,daga);
    daga->SetEdges(GetEdge(0),edgeb,edgea);
    daga->SetCorners(a,b,point);
    SetChild(1,dagb);
    dagb->SetEdges(edgeb,GetEdge(1),edgec);
    dagb->SetCorners(point,b,c);
    SetChild(2,dagc);
    dagc->SetEdges(edgea,edgec,GetEdge(2));
    dagc->SetCorners(a,point,c);

    edgea->SetDags(GetChild(2),GetChild(0));
    edgea->SetPoints(point_array,GetCorner(0),point);
    edgeb->SetDags(GetChild(0),GetChild(1));
    edgeb->SetPoints(point_array,GetCorner(1),point);
    edgec->SetDags(GetChild(1),GetChild(2));
    edgec->SetPoints(point_array,GetCorner(2),point);

    for(int j=0;j<3;j++)                    //set links to the new DAGs
    {
```

```
        actual_edge=GetEdge(j);
        if(IsLeft(point_array,actual_edge->GetOrig(),actual_edge->
          GetDest(),point))
            actual_edge->SetLeftDag(GetChild(j));
        else
            actual_edge->SetRightDag(GetChild(j));
    }
    for(int j=0;j<3;j++)                        //legalize edges
        GetEdge(j)->CheckEdge(point_array,point);
}

//---------------------------------------------------------------------------//

void DAG::SetFourNewDags (POINT* point_array,int point) {
    DAG* inside,*outside;
    DAG* daga,*dagb,*dagc,*dagd;
    EDGE* actual_edge;
    EDGE* edgea,*edgeb,*edgec,*edged;
    int inner_corner,outer_corner;

    actual_edge=GetEdge(on_line-1);
    actual_edge->SetInvisible();
    inside=actual_edge->GetLeft();
    outside=actual_edge->GetRight();
    for(int j=0;j<3;j++)                        //find third point of DAG
    {
        if(inside->GetEdge(j)==actual_edge)
            inner_corner=(j+2)%3;
        if(outside->GetEdge(j)==actual_edge)
            outer_corner=(j+2)%3;
    }

    edgea=new EDGE();
    edgeb=new EDGE();
    edgec=new EDGE();
    edged=new EDGE();
    number_of_edges=number_of_edges+4;

    daga=new DAG();
    dagb=new DAG();
    dagc=new DAG();
    dagd=new DAG();
```

```
inside->SetChild(0,daga);
daga->SetEdges(edgeb,inside->GetEdge((inner_corner+2)%3),edgea);
daga->SetCorners(point,inside->GetCorner((inner_corner+2)%3),inside->
  GetCorner(inner_corner));
inside->SetChild(1,dagb);
dagb->SetEdges(edgea,inside->GetEdge(inner_corner),edgec);
dagb->SetCorners(point,inside->GetCorner(inner_corner),inside->
  GetCorner((inner_corner+1)%3));
outside->SetChild(0,dagc);
dagc->SetEdges(edged,outside->GetEdge(outer_corner),edgeb);
dagc->SetCorners(point,outside->GetCorner(outer_corner),outside->
  GetCorner((outer_corner+1)%3));
outside->SetChild(1,dagd);
dagd->SetEdges(edgec,outside->GetEdge((outer_corner+2)%3),edged);
dagd->SetCorners(point,outside->GetCorner((outer_corner+2)%3),
  outside->GetCorner(outer_corner));

//set the new edges and DAGs
edgea->SetDags(inside->GetChild(0),inside->GetChild(1));
edgea->SetPoints(point_array,inside->GetCorner(inner_corner),point);
if(inside->GetEdge((inner_corner+2)%3)->GetOrig()==inside->
  GetCorner(inner_corner))
    inside->GetEdge((inner_corner+2)%3)->SetRightDag(inside->
      GetChild(0));
else
    inside->GetEdge((inner_corner+2)%3)->SetLeftDag(inside->
      GetChild(0));
if(inside->GetEdge(inner_corner)->GetOrig()==inside->
  GetCorner(inner_corner))
    inside->GetEdge(inner_corner)->SetLeftDag(inside->GetChild(1));
else
    inside->GetEdge(inner_corner)->SetRightDag(inside->GetChild(1));

edgeb->SetDags(outside->GetChild(0),inside->GetChild(0));
edgeb->SetPoints(point_array,inside->GetCorner((inner_corner+2)%3),
  point);
edgec->SetDags(inside->GetChild(1),outside->GetChild(1));
edgec->SetPoints(point_array,outside->GetCorner((outer_corner+2)%3),
  point);
edged->SetDags(outside->GetChild(1),outside->GetChild(0));
edged->SetPoints(point_array,outside->GetCorner(outer_corner),point);
if(outside->GetEdge((outer_corner+2)%3)->GetOrig()==outside->
  GetCorner(outer_corner))
```

```
            outside->GetEdge((outer_corner+2)%3)->SetRightDag(outside->
              GetChild(1));
        else
            outside->GetEdge((outer_corner+2)%3)->SetLeftDag(outside->
              GetChild(1));
        if(outside->GetEdge(outer_corner)->GetOrig()==outside->
          GetCorner(outer_corner))
            outside->GetEdge(outer_corner)->SetLeftDag(outside->GetChild(0));
        else
            outside->GetEdge(outer_corner)->SetRightDag(outside->
              GetChild(0));

        //legalize edges
        inside->GetEdge(inner_corner)->CheckEdge(point_array,point);
        inside->GetEdge((inner_corner+2)%3)->CheckEdge(point_array,point);
        outside->GetEdge(outer_corner)->CheckEdge(point_array,point);
        outside->GetEdge((outer_corner+2)%3)->CheckEdge(point_array,point);
}
```

A.7 functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
#include <string.h>
#include "dag.h"

extern int number_of_flips;
extern int number_of_dags;
extern int number_of_edges;
extern int online_points;
extern int duplicate_points;
extern int edge_index;

//generating a big starting triangle including the whole pointset
//and random x y coordinates (fitting into the A4 page)
extern void FillPointArray (POINT* point_array,int number_of_points);

//recursive function to find the point in the DAG structure
extern DAG* FindPoint (POINT* point_array,int point,DAG* dag);

//checking if the point is inside the triangle
//setting value DAG::on_line if the point is on an edge of the triangle
```

```
extern int InsideTriangle (POINT* point_array ,int a,int b,int c,int
  point);

//checking if point c lies on the left side of the line through ab
//returning -1 if point is on the line
extern int IsLeft (POINT* point_array ,int a,int b,int c);

//checking if the point is inside the circumcircle
//triangle points have to be commited in clockwise order
extern int InsideCircle (POINT* point_array, int a,int b,int c,
  int point);

//creating the header of the FIG file
extern void XfigStart (FILE* fptr);

//creating a point in the FIG file
extern void XfigPoint (FILE* fptr ,POINT point);

//creating a line from point a to point b in the FIG file
extern void XfigLine (FILE* fptr ,POINT* point_array ,int niveau ,int color ,
  int a,int b);

//recursive function to create the convex hull by flipping edges
//in clockwise order through the DAGs around the pointset
extern int ConvexHull (POINT* point_array ,DAG* first_dag ,int last_point);

//recursive function to delete the dynamic DAG tree structure ,
//write edges into the file and fill the edge array
extern void HandleDagsAndEdges (FILE* fptr ,POINT* point_array ,EDGE**
  edge_array ,DAG* startdag);

//sorting the edges according to the heap sort algorithm
extern void HeapSort (EDGE** edge_array);

//creating the heap structure
extern void DownHeap (EDGE** edge_array ,int i ,int n);

//computing the MST according to Kruskal's algorithm from the sorted edge
//array and writing it to the file , returning the length of the MST
extern float ComputeMST (FILE* fptr ,POINT* point_array ,EDGE** edge_array ,
  int number_of_points);

#endif
```

A.8 functions.cpp

```
#include "functions.h"

void FillPointArray (POINT* point_array,int number_of_points) {
    srand(time(0));
    point_array[0].Setxy(0,-21000,-21000);
    point_array[1].Setxy(1,0,21000);
    point_array[2].Setxy(2,21000,0);
    for(int i=3;i<number_of_points+3;i++)
        point_array[i].Setxy(i,rand()%7000+1,rand()%5000+1);
}

//------------------------------------------------------------------------------//

DAG* FindPoint (POINT* point_array,int point,DAG* dag) {
    int inside;
    DAG* left=dag->GetChild(0);
    DAG* middle=dag->GetChild(1);
    DAG* right=dag->GetChild(2);

    if(left==NULL&&middle==NULL&&right==NULL)
        return dag;
    if(left!=NULL)
    {
        inside=InsideTriangle(point_array,left->GetCorner(0),left->
          GetCorner(1),left->GetCorner(2),point);
        if(inside==1)
            return FindPoint(point_array,point,left);
        if(inside==-1)
            return NULL;
    }
    if(middle!=NULL)
    {
        inside=InsideTriangle(point_array,middle->GetCorner(0),
          middle->GetCorner(1),middle->GetCorner(2),point);
        if(inside==1)
            return FindPoint(point_array,point,middle);
        if(inside==-1)
            return NULL;
    }
    if(right!=NULL)
    {
```

```
            inside=InsideTriangle(point_array,right->GetCorner(0),
              right->GetCorner(1),right->GetCorner(2),point);
            if(inside==1)
                return FindPoint(point_array,point,right);
            if(inside==-1)
                return NULL;
        }
}

//--------------------------------------------------------------------------//

int InsideTriangle (POINT* point_array,int a,int b,int c,int point) {
    int ab=IsLeft(point_array,a,b,point);
    int bc=IsLeft(point_array,b,c,point);
    int ca=IsLeft(point_array,c,a,point);

    if((ab==1&&bc==1&&ca==1)||(ab==0&&bc==0&&ca==0))
    {
        DAG::on_line=0;
        return 1;
    }

    int xa=point_array[a].Getx();
    int ya=point_array[a].Gety();
    int xb=point_array[b].Getx();
    int yb=point_array[b].Gety();
    int xc=point_array[c].Getx();
    int yc=point_array[c].Gety();
    int xd=point_array[point].Getx();
    int yd=point_array[point].Gety();

    //checking if the point lies on another point
    if((ab==-1&&bc==-1)||(bc==-1&&ca==-1)||(ca==-1&&ab==-1))
        return -1;

    //checking if the point lies on a line between start end end point
    if(ab==-1&&((xa<=xd&&xb>=xd)||(xa>=xd&&xb<=xd))&&((ya<=yd&&yb>=yd)||
      (ya>=yd&&yb<=yd)))
    {
        DAG::on_line=1;
        return 1;
    }
    if(bc==-1&&((xb<=xd&&xc>=xd)||(xb>=xd&&xc<=xd))&&((yb<=yd&&yc>=yd)||
```

```
    (yb>=yd&&yc<=yd)))
  {
      DAG::on_line=2;
      return 1;
  }
  if(ca==-1&&((xc<=xd&&xa>=xd)||(xc>=xd&&xa<=xd))&&((yc<=yd&&ya>=yd)||
    (yc>=yd&&ya<=yd)))
  {
      DAG::on_line=3;
      return 1;
  }
  return 0;
}

//------------------------------------------------------------------------------//

int IsLeft (POINT* point_array,int a,int b,int c) {
  int xa=point_array[a].Getx();
  int ya=point_array[a].Gety();
  int xb=point_array[b].Getx();
  int yb=point_array[b].Gety();
  int xc=point_array[c].Getx();
  int yc=point_array[c].Gety();

  if((xc-xa)*(yb-ya)-(yc-ya)*(xb-xa)>0)
      return 1;
  if((xc-xa)*(yb-ya)-(yc-ya)*(xb-xa)==0)
      return -1;
  return 0;
}

//------------------------------------------------------------------------------//

int InsideCircle (POINT* point_array,int a,int b,int c,int point) {
  double xa=point_array[a].Getx();
  double ya=point_array[a].Gety();
  double xb=point_array[b].Getx();
  double yb=point_array[b].Gety();
  double xc=point_array[c].Getx();
  double yc=point_array[c].Gety();
  double xd=point_array[point].Getx();
  double yd=point_array[point].Gety();
```

```
    double control=(xa*xa+ya*ya)*(xb*(yc-yd)+xc*(yd-yb)+xd*(yb-yc))+
      (xb*xb+yb*yb)*(xa*(yd-yc)+xc*(ya-yd)+xd*(yc-ya))+
      (xc*xc+yc*yc)*(xa*(yb-yd)+xb*(yd-ya)+xd*(ya-yb))+
      (xd*xd+yd*yd)*(xa*(yc-yb)+xb*(ya-yc)+xc*(yb-ya));
    if(control>0)
        return 1;
    return 0;
}

//------------------------------------------------------------------------//

void XfigStart (FILE* fptr) {
    fprintf(fptr,"#FIG 3.2\nLandscape\nCenter\nMetric\nA4 \n100.00");
    fprintf(fptr,"\nSingle\n-2\n600 2\n");
    //for explanation see http://epb.lbl.gov/xfig/fig-format.html
}

//------------------------------------------------------------------------//

void XfigPoint (FILE* fptr,POINT point) {
    fprintf(fptr,
      "1 3 0 1 0 7 48 -1 20 0.000 1 0.0000 %d %d 20 20 %d %d %d %d\n",
      point.Getx(),point.Gety(),point.Getx(),point.Gety(),point.Getx()
      +20,point.Gety());
}

//------------------------------------------------------------------------//

void XfigLine (FILE* fptr,POINT* point_array,int niveau,int color,int a,
  int b) {
    fprintf(fptr,"2 1 0 1 %d 0 %d -1 -1 0.000 0 0 -1 0 0 2\n",
      color,niveau);
    fprintf(fptr,"\t%d %d %d %d\n",point_array[a].Getx(),point_array[a].
      Gety(),point_array[b].Getx(),point_array[b].Gety());
}

//------------------------------------------------------------------------//

int ConvexHull (POINT* point_array,DAG* first_dag,int last_point) {
    int first_corner;
    int pre_first_point,first_point,second_point,third_point;
    EDGE* pre_first_edge,*first_edge,*second_edge,*third_edge;
    DAG* pre_first_dag,*second_dag,*third_dag;
```

```
int side_points=0;

for(int i=0;i<3;i++)
{
    if(first_dag->GetCorner(i)<3)
        side_points++;
    else if(first_dag->GetCorner(i)!=last_point)
    {
        first_point=first_dag->GetCorner(i);
        first_edge=first_dag->GetEdge(i);
        first_corner=i;
    }
}
if(side_points==3)                    //only one DAG existing
    return 0;
side_points=0;
second_dag=first_edge->GetLeft();
for(int i=0;i<3;i++)
{
    if(second_dag->GetCorner(i)<3)
        side_points++;
    else if(second_dag->GetCorner(i)!=first_point)
    {
        second_point=second_dag->GetCorner(i);
        second_edge=second_dag->GetEdge(i);
    }
}
if(side_points==2)                    //less than 2 DAGs in a row
    return 0;
side_points=0;
third_dag=second_edge->GetLeft();
for(int i=0;i<3;i++)
{
    if(third_dag->GetCorner(i)<3)
        side_points++;
    else if(third_dag->GetCorner(i)!=second_point)
    {
        third_point=third_dag->GetCorner(i);
        third_edge=third_dag->GetEdge(i);
    }
}
if(side_points==2)                    //row finished
    return 0;
```

```
    //check for edges to be flipped
    if(!IsLeft(point_array,first_point,third_point,second_point))
    {
        second_edge->Flip(point_array,third_point,third_dag,second_dag,
          second_edge->GetOrig(),second_edge->GetDest());

        pre_first_edge=first_dag->GetEdge((first_corner+1)%3);
        pre_first_dag=pre_first_edge->GetRight();
        if(pre_first_dag!=NULL)          //start over from DAG pre_first
        {
            for(int i=0;i<3;i++)
                if(pre_first_dag->GetEdge(i)==pre_first_edge)
                    pre_first_point=pre_first_dag->GetCorner((i+2)%3);
            return ConvexHull(point_array,pre_first_dag,pre_first_point);
        }
        else                             //back at the start of the row
            return ConvexHull(point_array,first_dag,first_dag->
              GetCorner((first_corner+2)%3));
    }
    else                                 //convex hull fine, going on
        return ConvexHull(point_array,second_dag,first_point);
}

//------------------------------------------------------------------//

void HandleDagsAndEdges (FILE *fptr,POINT* point_array,EDGE** edge_array,
  DAG* startdag) {
    DAG* left=startdag->GetChild(0);
    DAG* middle=startdag->GetChild(1);
    DAG* right=startdag->GetChild(2);
    EDGE* actual_edge;

    for(int i=0;i<3;i++)
    {
        actual_edge=startdag->GetEdge(i);
        if(actual_edge->GetLinkage()==1)
        {
            edge_array[edge_index]=actual_edge;
            edge_index++;
            //output without start triangle
            if(actual_edge->GetOrig()>2&&actual_edge->GetDest()>2&&
              actual_edge->GetVisibility())
```

```
                XfigLine(fptr,point_array,50,0,actual_edge->GetOrig(),
                  actual_edge->GetDest());
        }
        else
            actual_edge->DecLinkage();
    }

    if(middle==NULL&&left!=NULL&&right!=NULL&&left->GetCorner(0)!=-1)
    {
        left->SetCorners(-1,-1,-1);        //avoid double delete after flip
        right->SetCorners(-1,-1,-1);
    }
    else
    {
        if(left!=NULL)
            HandleDagsAndEdges(fptr,point_array,edge_array,left);
        if(middle!=NULL)
            HandleDagsAndEdges(fptr,point_array,edge_array,middle);
        if(right!=NULL)
            HandleDagsAndEdges(fptr,point_array,edge_array,right);
    }
    delete startdag;
    number_of_dags++;
}

//-----------------------------------------------------------------------//

void HeapSort (EDGE** edge_array) {
    EDGE* save_edge;
    for(int i=number_of_edges/2-1;i>=0;i--)
        DownHeap(edge_array,i,number_of_edges);
    for(int n=number_of_edges-1;n>0;n--)
    {
        save_edge=edge_array[n];
        edge_array[n]=edge_array[0];
        edge_array[0]=save_edge;
        DownHeap(edge_array,0,n);
    }
}

//-----------------------------------------------------------------------//

void DownHeap (EDGE** edge_array,int i,int n) {
```

```
    EDGE* save_edge;
    int j=2*i+1;
    if(j<n)
    {
        if(j+1<n)
            if(edge_array[j]->GetLength()<edge_array[j+1]->GetLength())
                j++;
        if(edge_array[i]->GetLength()<edge_array[j]->GetLength())
        {
            save_edge=edge_array[i];
            edge_array[i]=edge_array[j];
            edge_array[j]=save_edge;
            DownHeap(edge_array,j,n);
        }
    }
}

//----------------------------------------------------------------------------//

float ComputeMST (FILE* fptr,POINT* point_array,EDGE** edge_array,int
  number_of_points) {
    int group_a,group_b;
    int mstedges=0;
    float mstlength=0;

    for(int i=0;mstedges<number_of_points-1-duplicate_points;i++)
    {
        group_a=point_array[edge_array[i]->GetOrig()].GetGroup();
        group_b=point_array[edge_array[i]->GetDest()].GetGroup();
        if(group_a!=group_b)
        {
            XfigLine(fptr,point_array,49,4,edge_array[i]->GetOrig(),
              edge_array[i]->GetDest());
            mstedges++;
            mstlength=mstlength+edge_array[i]->GetLength();
            for(int i=3;i<number_of_points+3;i++)
                if(point_array[i].GetGroup()==group_b)
                    point_array[i].SetGroup(group_a);
        }
    }
    return mstlength;
}
```

A.9 main.cpp

```
#include <iostream>
#include <string.h>
#include <time.h>
#include "dag.h"
#include "functions.h"

using std::cout;
using std::cin;

const int number_of_points=10000;
int DAG::on_line=0;
int number_of_flips=0;
int number_of_dags=0;
int number_of_edges=0;
int online_points=0;
int duplicate_points=0;
int edge_index=0;

int main()
{
    clock_t time=clock();
    FILE* fptr;
    char filename[20];
    float mstlength=0;
    EDGE* edgea,*edgeb,*edgec;
    EDGE** edge_array;
    DAG* actual_dag;
    DAG* startdag;                          //dynamic tree of DAGs
    POINT *point_array;                     //points + starting triangle
    point_array=new POINT[number_of_points+4];      // + extra point

    if(point_array==NULL)
    {
        cout << "ERROR: out of memory";
        return 1;
    }
    FillPointArray(point_array,number_of_points);
    startdag=new DAG();                     //set the starting triangle
    edgea=new EDGE();
    edgeb=new EDGE();
    edgec=new EDGE();
```

```
number_of_edges=number_of_edges+3;

edgea->SetDags(startdag,NULL);
edgea->SetPoints(point_array,0,1);
edgeb->SetDags(startdag,NULL);
edgeb->SetPoints(point_array,1,2);
edgec->SetDags(startdag,NULL);
edgec->SetPoints(point_array,2,0);
startdag->SetEdges(edgea,edgeb,edgec);
startdag->SetCorners(0,1,2);

for(int i=3;i<number_of_points+3;i++)    //insert points incremental
{
    actual_dag=FindPoint(point_array,i,startdag);
    if(actual_dag==NULL)
        duplicate_points++;
    else
    {
        if(DAG::on_line==0)
            actual_dag->SetThreeNewDags(point_array,i);
        else
        {
            online_points++;
            actual_dag->SetFourNewDags(point_array,i);
        }
    }
}

sprintf(filename,"triangulation.fig");
fptr=fopen(filename,"w");
XfigStart(fptr);
for(int i=3;i<number_of_points+3;i++)
    XfigPoint(fptr,point_array[i]);

//extra point in leftmost DAG
point_array[number_of_points+3].Setxy(number_of_points+3,-10000,0);
actual_dag=FindPoint(point_array,number_of_points+3,startdag);
ConvexHull(point_array,actual_dag,0);
//extra point in rightmost DAG
point_array[number_of_points+3].Setxy(number_of_points+3,10000,
  -5000);
actual_dag=FindPoint(point_array,number_of_points+3,startdag);
ConvexHull(point_array,actual_dag,2);
```

```
    //extra point in downmost DAG
    point_array[number_of_points+3].Setxy(number_of_points+3,10000,
      10000);
    actual_dag=FindPoint(point_array,number_of_points+3,startdag);
    ConvexHull(point_array,actual_dag,1);

    cout << (double)(clock()-time)/CLOCKS_PER_SEC
      << " seconds till triangulation is done.\n";
    edge_array=new EDGE*[number_of_edges];
    HandleDagsAndEdges(fptr,point_array,edge_array,startdag);
    HeapSort(edge_array);
    cout << (double)(clock()-time)/CLOCKS_PER_SEC
      << " seconds till edges are sorted.\n";
    mstlength=ComputeMST(fptr,point_array,edge_array,number_of_points);

    fclose(fptr);
    for(int i=0;i<number_of_edges;i++)
        delete edge_array[i];
    delete[] edge_array;
    delete[] point_array;

    cout << (double)(clock()-time)/CLOCKS_PER_SEC
      << " seconds till MST is computed.\n\n";
    cout << number_of_points << " points triangulated.\n";
    cout << number_of_flips << " performed flip operations.\n";
    cout << number_of_dags << " DAG instances created.\n\n";
    cout << duplicate_points << " ignored duplicate points.\n";
    cout << online_points << " points on a line.\n";
    cout << number_of_edges << " edges constructed.\n";
    cout << mstlength << " total MST length.\n";

    cin.get();
    return 0;
}
```