The Euclidean Steiner Problem

Germander Soothill

April 27, 2010

Abstract

The Euclidean Steiner problem aims to find the tree of minimal length spanning a set of fixed points in the Euclidean plane while allowing the addition of extra (Steiner) points. The Euclidean Steiner tree problem is NP-hard which means there is currently no polytime algorithm for solving it. This report introduces the Euclidean Steiner tree problem and two classes of algorithms which are used to solve it: exact algorithms and heuristics.

Contents

Chapter 1

Introduction

1.1 Context and motivation

Suppose somebody wanted to build a path in Durham connecting up all the colleges as cheaply as possible. They would want to make the path as short as possible; this is an example of the *Euclidean Steiner problem*. This problem is used in the design of many everyday structures, from roads to oil pipelines.

The aim of the problem is to connect a number of points in the Euclidean plane while allowing the addition of extra points to minimise the total length. This is an easy problem to express and understand, but turns out to be an extremely hard problem to solve.

Whilst doing an internship at a power company from July to September 2009, I did research into the optimization algorithms which are used in the design of gas turbine engines. It was here that I developed a specific interest in problems of optimization and the algorithms which can be used to solve them. It was this that lead me to chose to do this project.

The Euclidean Steiner problem is a particularly interesting optimization problem to study as it draws on ideas from graph theory, computational complexity and geometry as well as optimization. All these different themes will be explored in this report which focuses firstly on two important areas of study related to the Euclidean Steiner problem, followed by what the Euclidean Steiner problem is and finally some different ways of solving it.

1.2 Contents

The main body of this report is divided as follows. The first two chapters are introductions to two areas of study which are important when considering the Euclidean Steiner problem. Chapter 2 provides an introduction to *graph theory* and Chapter 3 provides an introduction to *computational complexity*. Both of these chapters stand pretty much alone but various ideas covered in them will be drawn on in the later three chapters. Chapter 4 focuses on explaining in detail what the Euclidean Steiner problem is and provides a brief history of the study of the problem from its first posed form until the present day. Chapters 5 and 6 focus on ways to solve the Euclidean Steiner problem. Chapter 5 discusses *exact algorithms* used to solve the problem and Chapter 6 discusses *heuristics* used to solve the problem. The final chapter of this report, Chapter 7, provides a summary of the main points raised.

The main references used will be noted at the beginning of each chapter and any other references used throughout the report will be noted when appropriate. The bibliography at the end of the report provides the details of each of the references used.

Chapter 2

Graph Theory

The *Euclidean Steiner problem* is a problem of connecting points in Euclidean space so that the total length of connecting lines is minimum. In order to formally introduce and discuss this problem, it is necessary to have a basic understanding of graph theory which is what this chapter provides. Some of the notation and equations which are used here will be refered to throughout the rest of the report, however much of the material is to provide consolidatory understanding of graph theory and as such this chapter generally stands alone.

This chapter will introduce the area of graph theory starting with basic notation and definitions including edges and vertices, paths and circuits and trees. It will then move on to looking at a well know graph theory problem, the *minimum spanning tree problem*, which is closely related to the Steiner problem. Two algorithms for solving the minimum spanning tree problem will be discussed, *Prim's algorithm* and *Kruskal's algorithm*. Finally, as this report focuses on the Steiner problem in Euclidean space, the concepts of Euclidean space and Euclidean graph theory will be formally introduced.

The main references used for this chapter are [4, 12, 5].

2.1 Vertices and Edges

A *graph* $G = (V, E)$ is a structure consisting of a set of *vertices* $V = \{v_1, v_2, ...,\}$, and a set of *edges* $E = \{e_1, e_2, ...,\}$ where each edge connects a pair of vertices. Unless otherwise stated V and E are assumed to be finite and G is called finite. Another term for a graph defined in this way is a *network*; we will use these terms interchangeably.

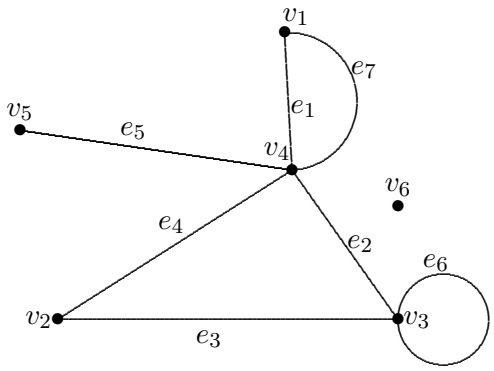


Figure 2.1: Simple Graph

In Figure 2.1 we have $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$. The edge e_5 is *incident* to the vertices v_4 and v_5, meaning it connects them. Vertices v_4 and v_5 are called *endpoints* of the edge e_5. Both endpoints of e_6 are the same (v_3) so e_6 is called a *self-loop*. Edges e_1 and e_7 are called *parallel* because both of their endpoints are the same.

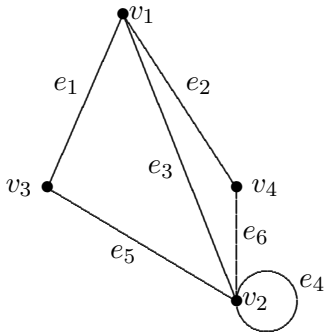


Figure 2.2: Paths

The *degree* of a vertex is the number of times it is used as the endpoint of an edge. We denote degree of a vertex v by $d(v)$. A self-loop uses its endpoint twice. In Figure 2.1, $d(v_1) = d(v_2) = 2$, $d(v_3) = 4$, $d(v_4) = 5$ and $d(v_5) = 1$. A vertex which has a degree of zero is called *isolated*. v_6 is isolated as $d(v_6) = 0$.

For a graph $G(V, E)$, $|E|$ and $|V|$ denote the number of edges and the number of vertices respectively. The number of edges in a graph is equal to the sum of the degree of each vertex divided by two, $|E| = \frac{\sum_{i=1}^{n} d(v_i)}{2}$.

2.2 Paths and Circuits

If edge e has vertices u and v as endpoints we say e *connects* u and v and that u and v are *adjacent*. A *path* is a sequence of edges $e_1, e_2, ..., e_n$ such that:

1. e_i and e_{i+1} have a common endpoint
2. if e_i is not a self-loop and is not the first or last edge then it shares one of its endpoints with e_{i-1} and the other with e_{i+1}.

Informally, what this means is that if you traced the edges of a graph with a pencil, a path is any sequence of movements you can make without taking the pencil off the paper at any point. Considering Figure 2.1; the sequence e_5, e_1, e_7, e_2 is a path as is the sequence e_4, e_3, e_6, e_2. However the sequence e_5, e_3, e_6 is not a path as there is no common endpoint of the edges e_5 and e_3.

A *circuit* is a path whose start and end vertices are the same. For example e_2, e_4, e_3 would be a circuit starting and finishing at vertex v_3. A path is *simple* if no vertex appears on it more than once. A circuit is simple if no vertex apart from the start/finish vertex appears on it more than once, and the start/finish vertex does not appear anywhere else in the circuit.

If for every two vertices in a graph (u, v) there is a path from u to v then the graph is said to be *connected*; Figure 2.2 shows a connected graph. Figure 2.1 does not show a connected graph as the isolated vertex v_6 has no path connecting it to any of the other vertices.

2.3 Trees

Let $G(V, E)$ be a graph with a set of vertices V and a set of edges E. We say G is *circuit-free* if there are no simple circuits in G. G is called a *tree* if it is both:

1. Connected
2. Circuit-free

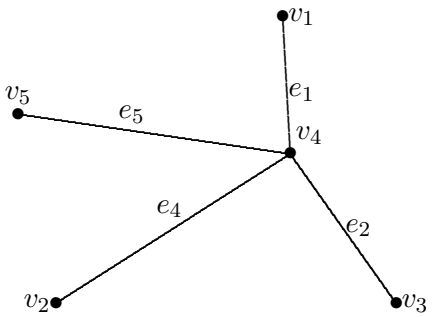


Figure 2.3: Tree

Figure 2.3 shows a graph which is both connected and circuit-free and hence is a tree. Figure 2.1 shows a graph which is neither connected nor circuit-free and Figure 2.2 shows a graph which is connected but not circuit-free, so neither of these graphs are trees.

A vertex whose degree is one is called a *leaf*. In Figure 2.3, vertices v_1, v_2, v_3, v_5 are all leaves.

Theorem. 2.1. *The following four conditions are equivalent:*

1. *G is a tree.*
2. *G is circuit free but if any new edge is added to G a circuit is formed.*
3. *G contains no self-loops and for every two vertices there is a* unique *path connecting them.*
4. *G is connected, but if any edge is deleted from G, the connectivity of G is interrupted.*

Proof. A thorough proof of this theorem is given in [4]. □

Theorem. 2.2. *Let $G(V, E)$ be a finite graph and $n = |V|$. The following are equilvalent:*

1. *G is a tree*
2. *G is circuit free and has $n - 1$ edges*
3. *G is connected and has $n - 1$ edges*

Proof. A thorough proof of this theorem is given in [4]. □

Corollary. 2.1. *A finite tree, with more than one vertex has at least two leaves.*

2.4 The Minimum Spanning Tree Problem

2.4.1 Subgraphs

A graph $G'(V', E')$ is called a *subgraph* of a graph $G(V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Arbitrary subsets of edges and vertices taken from G may not result in a subgraph because the subsets together may not form a graph. Consider the subsets $V' = \{v_1, v_2, v_3, v_4\}$ and $E' = \{e_1, e_2, e_4\}$ taken from the graph shown in Figure 2.2. As Figure 2.4(a) shows, these do from a graph. However the subsets $V'' = \{v_1, v_4\}$ and $E'' = \{e_1, e_2, e_4\}$ taken from the same graph do not form a graph as can be seen in Figure 2.4(b).

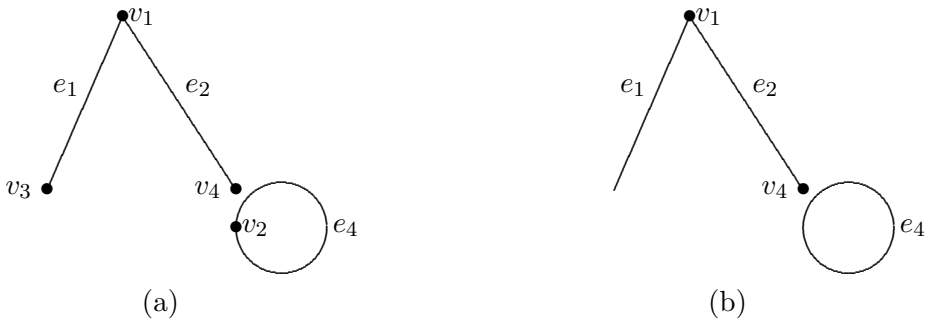


Figure 2.4: Subsets and Subgraphs

2.4.2 Weighted Graphs

The *length* of an edge, $l(e)$, is a number specified to it representing the distance travelled between the two vertices it connects. A graph which has a length specified to each of its edges is called *weighted*. Figure 2.5 shows the graph from Figure 2.1 whose edges have now been assigned lengths. If two vertices are not connected then it is as if there is edge of infinite length between them. If there is an isolated vertex, as there is in the graph shown in Figure 2.5, then this is effectively like having a length of infinity between this vertex and every other vertex in the graph.

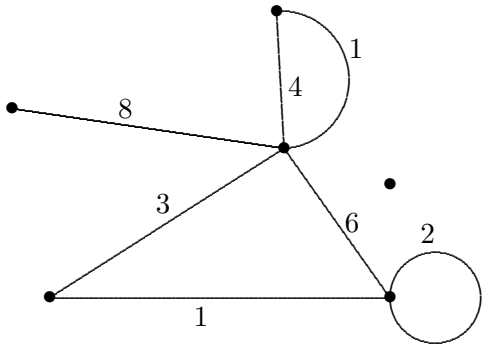


Figure 2.5: Weighted Graph

2.4.3 The Minimum Spanning Tree Problem

If $G(V, E)$ is a finite connected graph and each edge has a known length, $l(e) > 0$, then an interesting problem is to: *Find a connected subgraph, $G'(V', E')$, for which the sum of the lengths of the edges $\sum_{e \in E'} l(e)$ is a minimum.* The resulting graph is always a tree because G' must be connected and as the sum of the edges is minimum, no edges can be removed without resulting in G' being disconnected. A subgraph of $G(V, E)$ which contains *all the vertices* of G, $G'(V, E')$, and is a tree, is called a *spanning tree.*

If $G(V, E)$ is a finite connected graph and each edge has a known length , $l(e) > 0$, the *minimum spanning tree problem* is: **Find the spanning tree, $G'(V, E')$, for which the sum of the lengths of the edges $\sum_{e \in E'} l(e)$ is a minimum.**

2.4.4 Two Algorithms: Prim's and Kruskal's

As with many problems in graph theory, the best way to solve the minimum spanning tree problem is to use an algorithm. The first algorithm used to solve this problem was **Boruvka's algorithm** in 1926. The two most common algorithms used to solve this problem today are **Prim's algorithm**

and **Kruskal's algorithm** which are both *Greedy algorithms*. Greedy algorithm's are algorithms that make a locally optimal choice at each stage, meaning the best choice at that particular time.

Prim's algorithm works by starting at a vertex v and then growing a tree, T, from v. At each stage, the shortest edge is added to T which has exactly one endpoint already in T. Kruskal's algorithm considers the lengths of all the edges first. Then at each stage, the shortest edge is added to T, unless adding it creates a cycle.

Algorithm. 2.1 (Prim's). *Let $G(V, E)$ be a graph with $V = \{v_1, v_2, ..., v_n\}$. Let $l(v_i, v_j)$ be the length of the edge e, connecting vertices v_i and v_j, if there exists an edge between them and infinity otherwise. Let t be the starting vertex, T be the set of edges currently added to the tree, U be the set of vertices currently an endpoint of one of the edges in T and u and v be vertices such that $u \in U$ and $v \in V \setminus U$.*

1. *Let $t = v_1$, T start as the empty set $T \leftarrow \emptyset$, U start as the set $U \leftarrow \{v_1\}$*
2. *Let $l(t, u) = \min\{l(t, v)\}$ where $v \in V \setminus U$*
3. *Let T become the union of T and the edge e corresponding to length $l(t, u)$, $T \leftarrow T \bigcup \{e\}$*
4. *Let U become the union of U and the vertex u, $U \leftarrow U \bigcup \{u\}$*
5. *If $U = V$, STOP*
6. *For every $v \in V \setminus U$, $l(t, v) \leftarrow \min\{l(t, v), l(u, v)\}$*
7. *Go to Step 2.*

The steps of Prim's algorithm working on the weighted graph G from Figure 2.5 (after removing the isolated vertex, as G must be connected) are shown in Figure 2.6. Firstly, clearly neither the self-loop, nor the longer parallel edge will be part of the minimum spanning tree so they can be removed. Then, using Prim's algorithm starting from v_1, the edges will be added in the order shown in Figure 2.6.

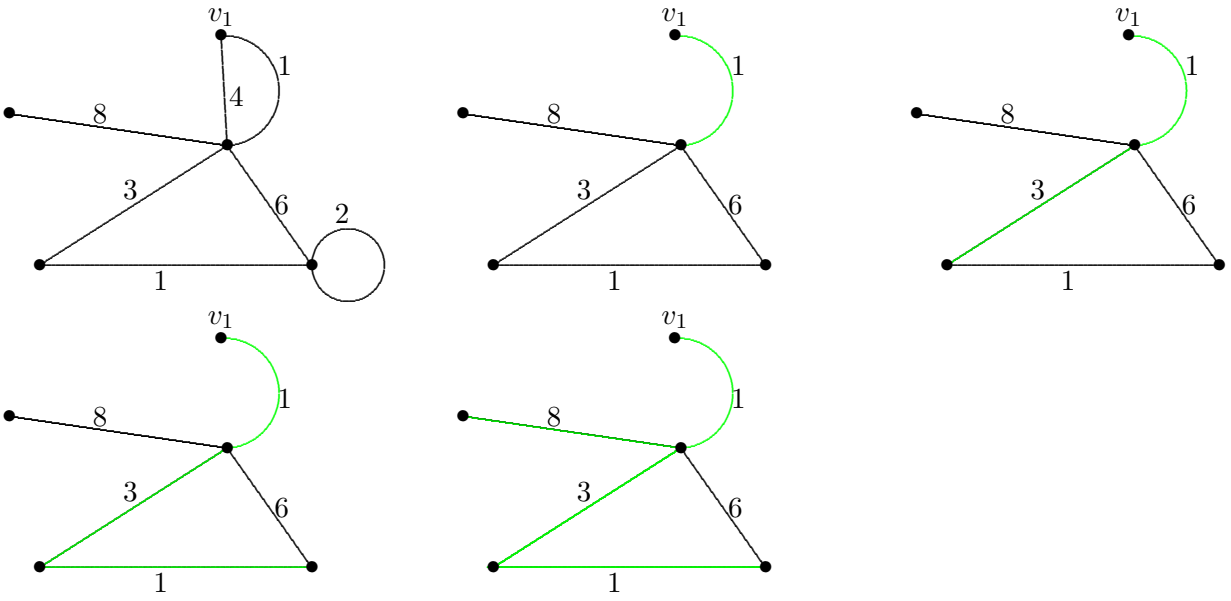


Figure 2.6: Prim's Algorithm

Algorithm. 2.2 (Kruskal's). *Let $G(V, E)$ be a graph with $V = \{v_1, v_2, ..., v_n\}$. Let $l(v_i, v_j)$ be the length of the edge e, connecting vertices v_i and v_j, if there exists an edge between them and infinity otherwise. Let T be the set of edges currently added to the tree and U be the set of vertices currently an endpoint of one of the edges in T.*

1. *Let T start as the empty set $T \leftarrow \emptyset$*
2. *Select edge e corresponding to the length $l(e) = \min\{l(u_i, v_i)\}$ not in T such that $T \bigcup \{e\}$ does not contain any cycles.*
3. *If $U = V$, STOP*
4. *Go to Step 2.*

The steps of Kruskal's algorithm working on the same weighted graph G used for the Prim's algorithm example are shown in Figure 2.7. The edges are added in order of minimum weight without a cycle being formed. The difference between this example and the example using Prim's algorithm is that with Kruskal's algorithm the bottom edge of length 1 is added before the edge of length 3, however with Prim's algorithm it was the other way around because the tree had to be built up starting from vertex v_1. With Kruskal's algorithm, when there are two edges of the same length as we have here with two edges of length 1, if both edges are allowed to be added to the graph there is no distinction between which one should be added first. Hence the order in which we add the two edges of length 1 in this case is just a matter of choice.

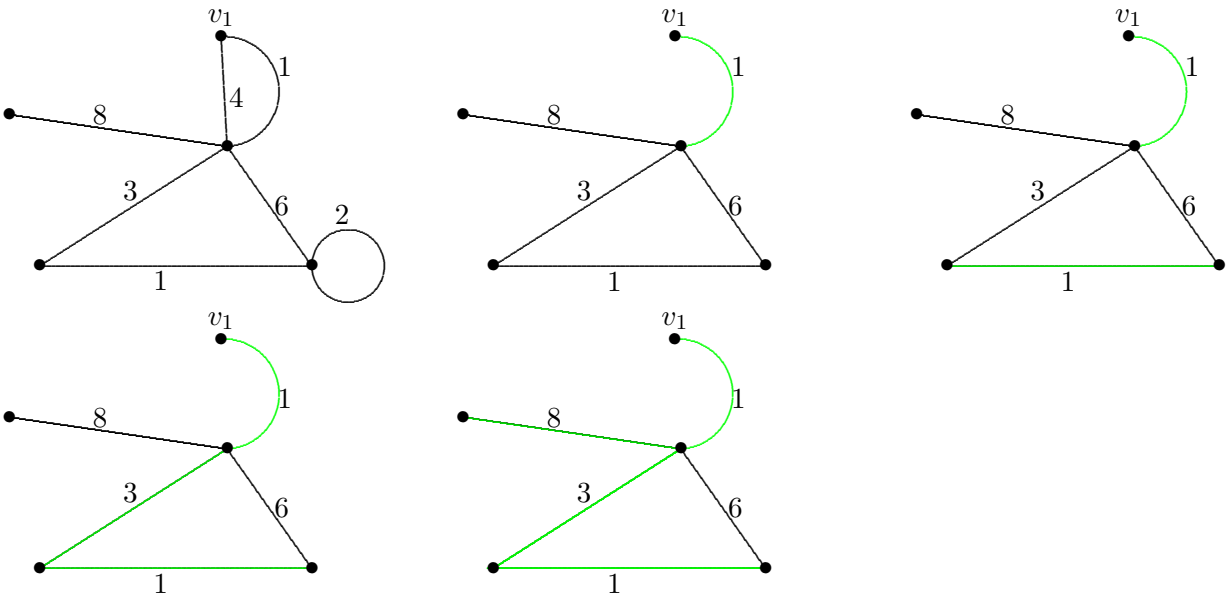


Figure 2.7: Kruskal's Algorithm

A proof of correctness for both Prim's and Kruskal's algorithms is given in [14].

2.4.5 Cayley's Theorem

For a set of vertices, $V = \{v_1, v_2, ..., v_n\}$, there are a number of different spanning trees which can connect the n vertices. Figure 2.8 shows the 3 different spanning trees which are possible for $n = 3$ vertices.

As n increases so does the number of possible spanning trees as shown in Table 2.1. *Cayley's Theorem* defines the relationship between the number of distinct vertices n and the number of possible spanning trees.

Theorem. 2.3 (Cayley's)**.** *The number of spanning trees for n distinct vertices is* n^{n-2}.

Proof. Proof given in [4]. □

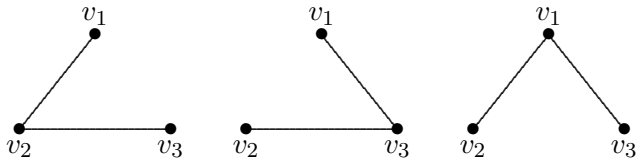


Figure 2.8: 3 Possible Trees for n=3

n	*No. Spanning Trees*
2	1
3	3
4	16

Table 2.1: How number of spanning trees increases with n

2.5 Euclidean Graph Theory

2.5.1 Euclidean Distance

The *Euclidean distance* between points x and y in Euclidean space is the length of the straight line joining them. If $x = (x_1, x_2, ..., x_n)$ and $y = (y_1, y_2, ..., y_n)$, then the Euclidean distance between x and y is given by (2.1).

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2.1}$$

In this report we shall only consider Euclidean distances where $n = 2$ and the distance is the distance between two points in the *plane* given by $d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$.

Another useful bit of notation to define is $|xy|$, which shall be used to mean $\sqrt{(x-y)^2}$, the Euclidean distance between x and y.

2.5.2 Euclidean Graphs

A *Euclidean graph* is a graph where the vertices are points in the Euclidean plane and the lengths of the edges between vertices are the Euclidean distances between the points.

Figure 2.9 shows the Euclidean plane with 7 vertices. $v_1 = (2,2)$, $v_2 = (1,6)$, $v_3 = (2,8)$, $v_4 = (6,7)$, $v_5 = (10,8)$, $v_6 = (7,4)$, $v_7 = (9,2)$. The lengths of the edges between these vertices are the Euclidean distances. Therefore, for example, the length of the edge between v_1 and v_2 is $d((2,2),(1,6)) = \sqrt{(2-1)^2 + (2-6)^2} = \sqrt{17}$.

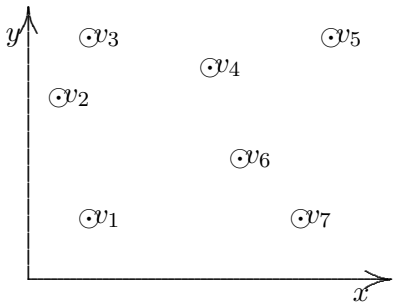


Figure 2.9: Euclidean Graph

2.5.3 The Euclidean Minimum Spanning Tree Problem

A *Euclidean spanning tree* is a spanning tree of a Euclidean graph, hence it is a circuit-free graph connecting n points in the Euclidean plane. The minimum spanning tree problem becomes the *Euclidean minimum spanning tree problem* and is: **Find the Euclidean spanning tree for which the sum of the Euclidean distances between n points is a minimum.**

The Euclidean minimum spanning tree problem can be solved just like the minimum spanning tree problem using either Prim's or Kruskal's algorithm. The Euclidean minimum spanning tree for the graph given in Figure 2.9 is shown in Figure 2.10. Using Kruskal's algorithm, the order in which the edges where added is: $\overrightarrow{v_2v_3}$, $\overrightarrow{v_6v_7}$, $\overrightarrow{v_4v_6}$, $\overrightarrow{v_1v_2}$, $\overrightarrow{v_3v_4}$, $\overrightarrow{v_4v_5}$.

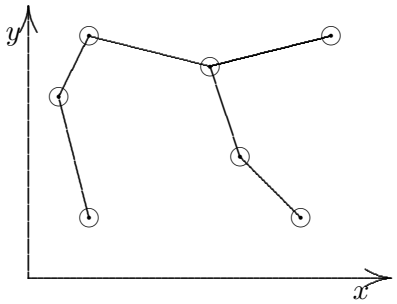


Figure 2.10: Euclidean Minimum Spanning Tree

2.6 Summary

This chapter was to consolidate the readers understanding of graph theory, focusing particulary on areas which are useful for studying the Euclidean Steiner problem.

We started with some basic notation and definitions including: edges and vertices, paths and circuits and trees. We then looked at the *minimum spanning tree problem*, a graph theory problem which is closely linked the Steiner problem. We discussed two algorithms for solving this problem, *Prim's algorithm* and *Kruskal's algorithm* and an example of the workings of both was shown. Finally the concepts of Euclidean space and Euclidean graph theory were introduced, including an example of the minimum spanning tree problem in Euclidean space.

The next chapter introduces another area of study which is important in order to consider the Euclidean Steiner problem, *computational complexity.*

Chapter 3

Computational Complexity

Computational complexity is an area of study combining Mathematics and Computer Science. It explores how difficult it is for problems to be solved algorithmically using computers and provides a means of comparing how difficult problems are. The Euclidean Steiner problem is an extremely difficult problem to solve, even using the very advanced computers we have today, falling into a class of problems called NP-hard which means that no polytime algorithm can solve it. An understanding of computational complexity it not actually necessary in order to consider the Euclidean Steiner problem, however I feel a introductory understanding of it adds a very interesting dimension to the study of the problem, which is what this chapter aims to provide.

We will start by looking at what *computational problems* are and introduce the four different types of computational problems. We will then move on to look at *complexity theory* which is the formal way of comparing the difficultly of algorithms. Here we will introduce *big O notation* which is a very useful tool used in complexity theory. Next we will discuss *complexity classes* which is a way of grouping problems depending on how difficult they are. Following this we will touch briefly on the *theory of NP-completeness* and the $NP \neq P$ problem which is one of the central challenges of all Computer Science and for which the *Clay Mathematical Institute* has offered a reward of $1000000 for a correct proof. This chapter concludes with a mention of the complexity of the Euclidean Steiner problem.

The main references used in this chapter are [18, 16, 11, 17, 13].

3.1 Computational Problems

3.1.1 Meaning of Computational Problem

Computational problems are ones that are suitable to be solved using a computer and which have a clearly defined set of results. In other words, the distinction between correct and incorrect solutions is unambiguous. For example, deciding the correct sentence to give a criminal in court is not an algorithmic problem but translating text is. Algorithmic problems are defined by the set of allowable inputs and a function which maps each allowable input to a set of results.

The inputs for a problem are called *instances*. The instances for a computational problem change but the question always remains the same. For example, a well known computational problem is:
INSTANCE: positive integer n
QUESTION: is n prime?

3.1.2 Types of Computational Problem

There are four main types of computational problem:

1. *Decision problem*: A problem where the only possible answers to the question are YES or NO. An example of a decision problem is the 3-colour problem. The instance in this case is a graph (encoded into binary), and the output is an answer to whether or not it is possible to assign one

of 3 colours to each vertex of the graph, without two adjacent vertices being assigned the same colour. Figure 3.1 shows an example of a graph which has been sucessfully 3 coloured, so the output for this instance would be YES.

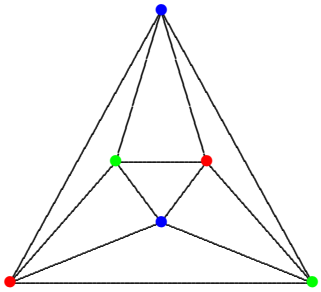


Figure 3.1: 3 Coloured Graph

2. *Search problem*: A problem where the aim is to find one of potentially many correct answers.
3. *Optimization problem*: A problem where the aim is to return a solution which is in some way optimal, i.e. the best possible answer to a particular problem. An example of an optimization problem is the minimum spanning tree problem which we looked at in the previous chapter. Again the instance is a graph, and in this case the output is a tree which is a subset of the graph and connects every vertex and for which the sum of edges is the shortest.
4. *Evaluation problem*: A problem where the value of the optimal solution is returned.

3.1.3 Algorithms

An *algorithm* is a method for solving a problem using a sequence of predefined steps. In the previous chapter we looked at two algorithms used for solving the minimum spanning tree problem: **Prim's algorithm** and **Kruskal's algorithm**. All computational problems are solved using algorithms so they are also sometimes refered to as *Algorithmic problems*.

3.2 Complexity Theory

Complexity is a measure of how difficult a problem is to solve. There are different ways in which the difficulty of a problem can be assessed. For computational problems, the two most common ways used to define the complexity of a problem are the **time** taken for an algorithm to solve the problem and the **space** (memory) used by a computer to compute the answer. We will consider a quantity based on the *computational time* taken for the best possible algorithm to solve a problem as the measure of its complexity.

3.2.1 Measuring Computational Time

Computational time for a problem depends on: the input, x; the computer, c; the programming language of the algorithm, L; and the implementation of the algorithm I. The effect input x has on computational time is clear: for larger the instances, the solution will take longer to compute than for smaller instances. The dependence of computational time on c, L and I makes it difficult to compare algorithms, as keeping these three variables the same in all circumstances is difficult to achieve. However fortunately the dependence of computational time on these three quantities is controllable.

Computational time can be simplified to a more abstract notion of *computation steps* which depends only on the algorithm A and the input x. The computation steps must be defined as a number of allowable elementary operations, these are: **arithmetic operations**, **assignment**, **memory access** and **recognition of next command to be performed**. The computation time, $t_A(x)$ of algorithm A can now be defined as a function of input x where t_A is the number of previously defined computation steps.

The formal definition of an algorithmic tool which will perform these steps is a *random access machine* (RAM). It is known that every program in every programming language for every computer can be translated into a program for a RAM, with only a small amount of efficiency lost. For each programming language and computer there is a constant k for which the translation into RAM programs increases the number of computation steps by no more than k. The *Turing Machine* is the most well known RAM used. The model dates back to an English logician, Alan Turing whose work provided a basis for building computers. This is the RAM model which we consider to be using in this chapter. We will not look at how exactly the Turing machine works, as it is not essential to our understanding of complexity theory in this report.

3.2.2 Comparing Algorithms

Computational time, $t_A(x)$ can be used to compare algorithms. Algorithm A is at least as fast as algorithm A' if $t_A(x) \leq t_{A'}(x)$ for all x. One problem with this expression is that it is only for a very simple algorithm that it would be possible to compute $t_A(x)$ for all x and hence test the relationship. Another problem is that when comparing a very simple algorithm A with a very complicated algorithm A' it is sometimes the case that $t_A(x) < t_{A'}(x)$ for small x but $t_A(x) > t_{A'}(x)$ for large x.

To resolve the first of these problems one can, instead of measuring the computational time for each input x, measure the computational time for each instance size, $n = |x|$. For example, if the problem we were considering was the minimum spanning tree problem, we would measure the computational time of **Prim's algorithm** once for each number of vertices n, instead of for all possible graphs.

A commonly used measurement for computation time is *worst-case runtime*: $t_A(n) := \sup\{t_A(x) : |x| \leq n\}$. This is a measurement of the computation time of an algorithm for the longest case up to a certain instance size. Also, $t^*_A(n) := \sup\{t_A(x) : |x| = n\}$ is often used. A *monotonically increasing* function $f(x)$ is a function such that for x increasing $f(x)$ is nondecreasing. $t^*_A = t_A$ when t^*_A is monotonically increasing, which is the case for most algorithms.

3.2.3 Big O Notation

Big O notation describes the limiting behaviour of a function as the parameter it depends on tends either towards a particular value or infinity. Big O notation allows for the simplification of functions in order to consider their growth rates. This notation is used in complexity theory when describing algorithmic use of space or time. More specifically for us, it is useful when discussing computation time, $t_A(x)$. This subsection will give an introduction to the notation.

Let $f(x)$ and $g(x)$ be two functions defined on a subset of real numbers. One writes

$$f(x) = O(g(x)); x \to \infty \tag{3.1}$$

if and only if, for sufficiently large values of x, $f(x)$ is at most a constant multiplied by $g(x)$ in absolute value. That is, $f(x) = O(g(x))$ if and only if there exist a positive real number M and a real number x_0 such that

$$|f(x)| \leq M|g(x)| \bigcup x > x_0. \tag{3.2}$$

In practice, the O notation for a funtion $f(x)$ is defined by the following rules:

- If $f(x)$ is the sum of several terms, the one with the largest growth rate is kept, and all others are omitted.

- If $f(x)$ is a product of several factors any constants are omitted.

For example, let $f(x) = 9x^5 - 4x^3 + 12x - 76$. The simplification of this function using big O notation as x goes to infinity is as follows. The function is a sum of four terms: $9x^5$, $-4x^3$, $12x$ and -76. The term with the largest growth rate is $9x^5$ so discard the other terms. $9x^5$ is a product of 9 and x^5, since 9 does not depend on x, omit it. Hence $f(x)$ has a big O form of x^5, or, $f(x) = O(x^5)$.

3.2.4 Algorithmic Complexity

So far we have looked at the measuring computation time of algorithms and comparing algorithms. We now need to define the complexity of a **problem**. The *algorithmic complexity* of a problem is $f(n)$ if the problem can be solved by *any* algorithm A with a worst-case runtime of $O(f(n))$.

For example, a problem p might be solvable quickest by an algorithm with worst-case runtime given by the function, $T(n) = 5n^3 + 6n - 9$. As n gets big, the n^3 term will dominate so all other terms can be neglected and the problem has algorithmic complexity of n^3. Even if the best algorithm for p had worst-case runtime given by $T(n) = 1000000n^3$, as n grows the n^3 will far exceed the 1000000 and again the problem would have algorithmic complexity of n^3.

3.2.5 Complexity of Prim's and Kruskal's algorithms

In the previous chapter we looked at how Prim's and Kruskal's algorithms for the minimum spanning tree problem work. It turns out that Prim's algorithm has an algorithmic complexity of the order $O(|V|^2)$ where $|V|$ is the size of the set of vertices. Kruskal's algorithm has an algorithmic complexity of the order $O(|E| \log |V|)$ where $|E|$ is the size of the set of edges and $|V|$ is the size of the set of vertices. The order of complexity for Prim's algorithm is greater than that for Kruskal's algorithm so generally it will take a computer more steps to solve the minimum spanning tree problem using Prim's algorithm than using Kruskal's algorithm.

3.3 Complexity Classes

When using algorithms practically, improving computation time by a polynomial amount, logarithmic amount or even a constant might have an important effect. However in complexity theory, improvements of a polynomial amount are basically indistinguishable. Complexity theory groups problems together into *complexity classes* depending on what the function of their algorithmic complexity is. The main distinction is made between problems which have a polynomial function of complexity and those which do not. Problems which have a polynomial function of complexity are said to be solvable in *polynomial time* and are considered to be efficiently solvable for all instances. We will now introduce the main complexity classes and discuss which complexity class the Euclidean Steiner problem falls into.

3.3.1 P Complexity Class

A problem, p_1, where the best algorithm to solve is has worst-case runtime given by $T(n) = 4n^5 + 6n^4 - 9$, has algorithmic complexity of order n^5. This problem has a *polynomial* function for worst-case runtime and is therefore said to be solvable in polynomial time. A problem, p_2, where the best algorithm has a worst-case runtime given by $T(n) = 6^n + e^{2n} - 10$, has an *exponential* function. It is clear that as n gets big for both problems, the runtime for p_2 will be much longer than the runtime for p_1.

Definition. 3.1. *A computational problem belongs to the complexity class P of polynomially solvable problems and is called a P-problem, if it can be solved by an algorithm with a polynomial worst-case runtime.*

Whether or not a problem falls into complexity class P essentially decides whether or not it is considered to be efficiently solvable.

3.3.2 NP Complexity Class

An algorithm is *deterministic* if at every moment the next step is unambiguously specified. In other words the next computation step depends only on what value the algorithm is reading at that moment and the next algorithmic instruction.

A *nondeterministic* algorithm has the possibility to choose at each step between a number of actions and also try more than one path. The concept can be thought of in a number of ways, two of which are:

- All possible computation paths are tried out.
- Algorithm has the ability to guess computation steps.

Practically this idea does not make much sense but is important theoretically when considering complexity classes.

Definition. 3.2. *A computational problem belongs to the complexity class NP of nondeterministic polynomially solvable problems and is called a NP-problem if it can be solved by a nondeterministic algorithm with a polynomial worst case runtime.*

Another way of looking at NP-problems is that if you guess an answer it is possible to **verify** whether or not it is the solution in polynomial time. This is different from P-problems where, without any hint, the correct solution can be **found** in polynomial time.

3.3.3 Summary of Important Complexity Classes

There are many different complexity classes. They are defined by a combination of: the function of worst-case runtime, the concept of complexity used and which RAM model is used to model the algorithm. The two complexity classes which we have looked at both used computation **time** as the resource constraint and the **Turing machine** as the model of computation; they are summarized in Table 3.1.

Complexity class	*Model of computation*	*Resource constraint*
P	Deterministic Turing	Time poly(n)
NP	Non-deterministic Turing	Time poly(n)

Table 3.1: Some Important Complexity Classes

3.4 Theory of NP-Completeness

3.4.1 Reduction

Problems are called *complexity theoretically similar* if they belong to the same complexity class. If problems are complexity theoretically similar, they are also *algorithmically similar*. If an algorithm for one problem p_1 can be obtained from an algorithm for another problem p_2, with only a polynomially difference in steps, then the problems are algorithmically similar and so are in the same complexity class.

Reduction is the name given to turning one problem into another by showing that the an algorithm which can solve solve one problem can be converted into and algorithm which solves another problem in only a polynomial number of steps. If problems can be reduced into each other then finding an algorithm which solves one problem means that both problems can be solved.

3.4.2 Completeness

A problem p is defined to be *class-complete* if firstly it belongs to a class and secondly all problems in that class are reducible to it. Class-complete problems are the hardest problems in a class. A problem p is *class-hard* if all problems in the class can be reduced to it but p is not actually in the class. Class-hard problems are as hard as any problems in the class.

3.4.3 NP-Completeness

The first problem which was shown to be NP-complete was the *Boolen satisfiability problem*. The proof of this is very complicated and involves a complex breakdown of the steps a Turing machine has to go through in order to solve the problem; the proof is given in [17]. Now that this problem has been proved, in order to show any other problem is NP-complete, it only neccesary to shown that a problem is reducible to the Boolen satisfiability problem.

3.4.4 $NP \neq P$?

There are thousands of problems which have been shown to be NP-complete. Since they are all algorithmically similar this means that if a deterministic polytime algorithm for any of these problems is found then there will exist a polytime algorithm for all these problems. This means either all these problems are solvable in polynomial time and $NP = P$ or none of them are solvable in polynomial time and $NP \neq P$.

No one knows if $NP = P$ or $NP \neq P$ but most experts believe that $NP \neq P$. This is because there are thousands of NP-problems and people have been searching for polytime algorithms which solve them for years. It is widely thought that if they are all solvable by polytime algorithms, at least one of them would have been found by now.

The $NP \neq P$? problem is one of the central challenges for complexity theory and all Computer Science. The *Clay Mathematical Institute* has included this problem as one of the 7 most important problems connected to Mathematics and there is an award of $1000000 for the first correct proof.

3.5 Complexity of Euclidean Steiner Problem

The Euclidean Steiner problem is NP-hard meaning it is as hard as any problem in the NP complexity class. This means, at the moment, there is no known polynomial time algorithm which can solve this problem.

3.6 Summary

This chapter was to introduce the reader to the area of computational complexity which explores how difficult it is for problems to be solved algorithmically using computers. This topic is relevant to this report as the Euclidean Steiner problem is NP-hard meaning there is currently no polytime algorithm which can solve it.

We started by looking at what computational problems are and introduced the four different types of computational problems. We then looked at complexity theory which is the formal way of comparing the difficultly of algorithms. We then moved on to discuss complexity classes which is how problems are grouped depending on how difficult they are to solve. We looked mainly at two complexity classes: P-problems and NP-problems and discussed how problems which fall into the P class are considered efficiently solvable and problems which fall into the NP class are not. We discussed the $NP \neq P$? problem, one of the central challenges of Computer Science and how a proof of this would mean that no NP-problems are solvable in polynomial time.

This concludes the two introductory chapters to help with the understanding of the Euclidean Steiner problem. In the next chapter we move on to look at what the Euclidean Steiner problem actually is.

Chapter 4

What is the Euclidean Steiner Problem?

The first two chapters of this report were to introduce the reader to two important ideas of study which are useful for understanding the Euclidean Steiner problem and its intricacies. We move on now to introduce what the Euclidean Steiner problem is.

We will start by looking at the historical background of the problem. Firstly introducing the trivial cases of the problem in order for the reader to logically build up their understanding. The first interesting case of the problem is then discussed under the name of Fermat's problem which is how it was first proposed and two solutions using geometric constructions are discussed. A computational example of Fermat's problem is then given followed by a proof of Fermat's problem using calculus. The generalization of Fermat's problem to the general case will then be discussed along with the developments made to the problem and the Mathematicians who made them over the past few hundred years. The next section will move on to discuss the basic ideas surrounding the Euclidean Steiner problem. We will start by defining some notation and terminology and will then move onto discuss the three different important types of tree involved in the problem, *Steiner minimal trees*, *Steiner trees* and *relatively minimal trees*. The next section will discuss the basic properties of one of these types of tree, Steiner trees. Finally in this chapter we will look at the number of Steiner topologies that there are for a given problem which is the main reason why this problem is so difficult to solve.

The main references used for the historical background section are [6, 19, 2, 13] and the main references used for the rest of this chapter are [13, 9].

4.1 Historical Background

4.1.1 Trivial Cases ($n = 1, 2$)

The easiest way to approach the Euclidean Steiner problem is to first consider the simplest cases and then build the concept up. The first case of the problem is the $n = 1$ case, which is to minimize the lengths connecting a single point, v_1, in the Euclidean plane. Clearly for this case the connecting distance is zero; see Figure 4.1(a).

The next case is the $n = 2$ case where problem is to connect two points, v_1 and v_2, in the Euclidean plane with minimum length. The minimum distance connecting two points is just a straight line between the points; see Figure 4.1(b).

4.1.2 Fermat's Problem ($n = 3$)

The origins of the Euclidean Steiner problem date back to the mathematician Fermat (1601-1655) who is most famous for posing another problem, commonly known as Fermat's Last Theorem: no three positive integers a, b, and c can satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than two. Like Fermat's last theorem, Fermat's problem, sparked up many years of mathematical

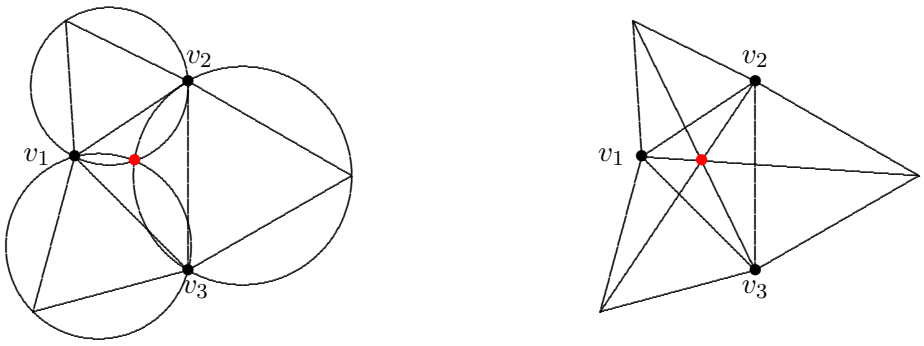
(a) (b)

Figure 4.1: Trivial Cases

research and discovery. Fermat proposed his problem in the form: *Find in the plane a point whose total distance from three given points is minimal.* This is the $n = 3$ case of the Euclidean Steiner problem.

Two mathematicians, Torricelli and Simpson, proposed geometric solutions to this problem over the next hundred years. These ruler and compass constructions, to find the point which minimises the distance to three other given points in the plane, are as follows. Denote the three points in the plane as v_1, v_2, v_3. Both the Torricelli and Simpson methods start by first joining the three points to form a triangle $\Delta v_1 v_2 v_3$ and then constructing three equilateral triangles, one on each edge of the original triangle, $\Delta v_1 v_2 v_3$. The Torricelli method then constructs three circles circumscribing each equilateral triangle. The point at which all the circles intersect is called the *Torricelli point*, p, this is shown in red in Figure 4.2(a). The Simpson method works by drawing a *Simpson line* between each vertex of the equilateral triangles not in $\Delta v_1 v_2 v_3$ with the opposite vertex which is in $\Delta v_1 v_2 v_3$. The intersection of the Simpson lines is the same as the Torricelli point which is shown in red in Figure 4.2(b).

The point which is found using the Torricelli or Simpson method, which solves Fermat's problem, makes three angles of 120° with each of the other two sets of points v_1, v_2, v_3.



(a) (b)

Figure 4.2: Torricelli and Simpson Methods

It turns out that both these geometric methods fail to construct a minimising point when any of the angles of $\Delta v_1 v_2 v_3$ are larger than 120°. If this is the case, the Torricelli method constructs a point outside the triangle $\Delta v_1 v_2 v_3$ and not all three lines in the Simpson method intersect in the same place. When one of the angles is larger than 120°, the point which minimises the total distance from the three given points is actually the point with the obtuse angle. Figure 4.3 shows three points v_1, v_2, v_3

where the interior angle $v_1v_2v_3$ is 130°. The point which minimises the total distance connecting the three points in this case is the point v_2.

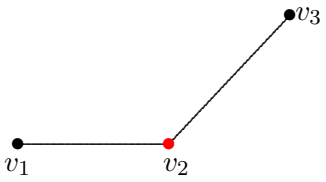


Figure 4.3: Angle greater than 120°

It was a mathematician, Heinen, who brought all these ideas together to formulate the first complete solution to Fermat's Problem in 1834:

1. If one of the interior angles formed from three points v_1, v_2, v_3 is greater than 120°, then the point defined by Fermat's problem coincides with the vertex where the angle is greater than 120°.

2. If all the interior angles formed from three points v_1, v_2, v_3 are less than 120°, then the point defined by Fermat's problem is the intersection of Simpson lines or the Torricelli point and is the point which makes 3 angles of 120° with points and v_1, v_2, v_3.

Fermat's problem is the Euclidean Steiner problem for the case where $n = 3$. This problem and its solution will be useful when discussing the problem for greater values of n. Proofs of correctness for both the Torricelli method and Simpson method are given in *Interactive Mathematics Miscellany and Puzzles*, [3].

4.1.3 An Example of Fermat's Problem

We will now look at an example of finding the point which solves Fermat's problem using the Torricelli method. We want to find the point which minimises the total distance between $v_1 = (2, 6)$, $v_2 = (5, 3)$ and $v_3 = (5, 8)$.

a) First we join the points $(2, 6)$, $(5, 3)$ and $(5, 8)$ to form a triangle as shown in Figure 4.4(a).

b) The next step is to find the three points to construct three equilateral triangles one on each of the edges. To find the third vertex of the equilateral triangle with vertices $(2, 6)$ and $(5, 3)$ call the third vertex (x_1, y_1) and solve the two simultaneous equations: $\sqrt{(x_1 - 2)^2 + (y_1 - 6)^2} = \sqrt{(2 - 5)^2 + (6 - 3)^2} = \sqrt{18}$ and $\sqrt{(x_1 - 5)^2 + (y_1 - 3)^2} = \sqrt{18}$. Solving gives the equation $y = x + 1$ and the quadratic $2x^2 - 14x + 11 = 0$ which gives $x = [14 \pm \sqrt{14^2 - 4 \times 2 \times 11}]/4 = 0.902, 6.098$. This gives us two possible points for the third point of the triangle $(0.902, 1.902)$ and $(6.0981, 7.0981)$. As we want the equilateral triangle to be formed on the outside of to original triangle we can select the point $(x_1, y_1) = (0.902, 1.902)$ as the point we want. The third point of the equilateral triangle with vertices $(2, 6)$ and $(5, 8)$ can be found in the same way by solving $\sqrt{(x_2 - 2)^2 + (y_2 - 6)^2} = \sqrt{(2 - 5)^2 + (6 - 8)^2} = \sqrt{13}$ and $\sqrt{(x_2 - 5)^2 + (y_2 - 8)^2} = \sqrt{13}$. The two possible points are $(5.232, 4.402)$ and $(1.768, 9.598)$ and considering the diagram we select $(x_2, y_2) = (1.768, 9.598)$. The third point for the equilateral triangle with vertices $(5, 3)$ and $(5, 8)$ is found in the same way giving $(x_3, y_3) = (6.443, 5.5)$. Points (x_1, y_1), (x_2, y_2) and (x_3, y_3) are shown in blue in Figure 4.4(b).

c) The midpoint of each equilateral triangle is found by finding the midpoint of the three vertcies. Call the midpoints of the three equilateral triangles m_1, m_2 and m_3. $m_1 = ([2 + 5 + 0.902]/3, [6 + 3 + 1.902]/3) = (2.634, 3.634)$, $m_2 = ([2 + 1.768 + 5]/3, [6 + 9.598 + 8]/3) = (2.923, 7.866)$ and $m_3 = (6.443, 5.5)$. The points m_1, m_2 and m_3 are shown in red in Figure 4.4(c). The midpoint of an equilateral triangle is the same as the midpoint of the circle which circumscribes it. The Torricelli point which we are searching for is the point at which the three circles circumscribing the three equilateral

triangles intersect. It is found by solving the three simultaneous equations which are the equations of the three circles. $C_1 : (X - 2.634)^2 + (Y - 3.634)^2 = 6$, $C_2 : (X - 2.923)^2 + (Y - 7.866)^2 = 4.332$ and $C_3 : (X - 6.443)^2 + (Y - 5.5)^2 = 8.335$. Solving for X, Y gives $(X, Y) = (3.583, 5.893)$ which is shown in green in Figure 4.4(c).

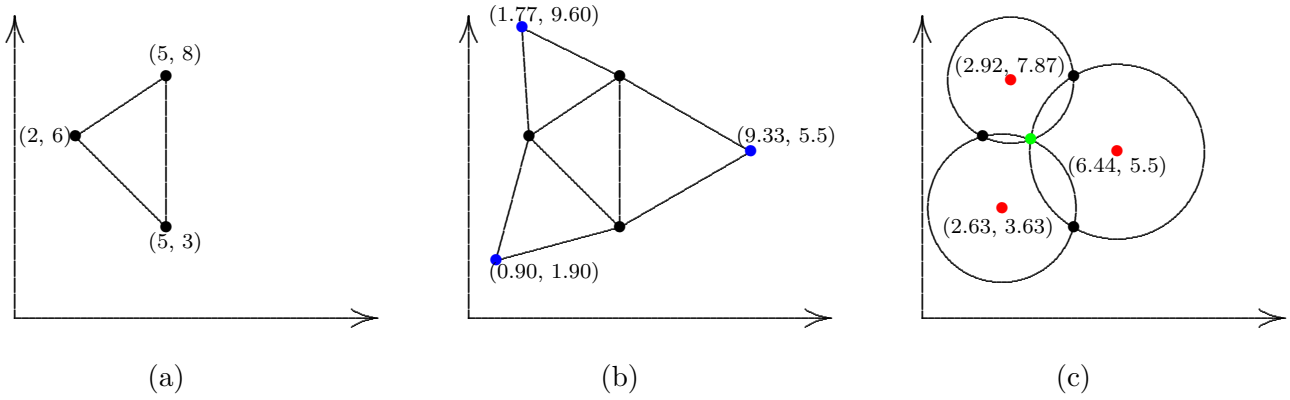


Figure 4.4: Fermat's problem: An example

The point we found using the Torricelli construction, $(3.583, 5.893)$, is the point which minimises the total distance to the three given points $(2, 6)$, $(5, 8)$ and $(5, 3)$. We also know that it should make three angles equal to $120°$ with each pair of points from $(2, 6)$, $(5, 8)$ and $(5, 3)$. We shall check this condition now.

The *cosine rule* is a rule for the relationship between the angles and lengths of non right-angled triangles. If the lengths of the edges of the triangle are denoted by a, b and c and the angles are denoted by A, B and C, where the angle is opposite the edge of the same letter, then the cosine rule is given by (4.1).

$$2bc \cos A = b^2 + c^2 - a^2 \tag{4.1}$$

Denote the angles which the Torricelli point, $(3.583, 5.893)$, makes with the three points as θ_1, θ_2 and θ_3. The values of θ_1, θ_2 and θ_3 can be found using the Cosine rule. The values for θ_1, θ_2 and θ_3, as shown in Figure 4.5, are calculated as follows. $2 \times 2.539 \times 3.221 \cos\theta_1 = 2.539^2 + 3.221^2 - 5^2 \Leftrightarrow \cos\theta_1 = -0.50003 \Leftrightarrow \theta_1 = 120.0$. $2 \times 3.221 \times 1.587 \cos\theta_1 = 3.221^2 + 1.587^2 - 4.243^2 \Leftrightarrow \cos\theta_2 = -0.49980 \Leftrightarrow \theta_2 = 120.0$. $2 \times 1.587 \times 2.529 \cos\theta_3 = 1.587^2 + 2.539^2 - 3.606^2 \Leftrightarrow \cos\theta_3 = -0.5012 \Leftrightarrow \theta_3 = 120.1$

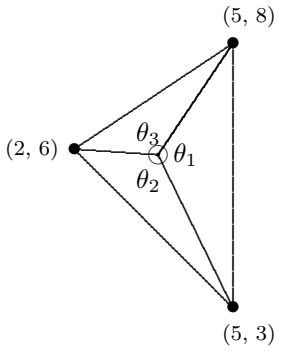


Figure 4.5: Three angles of 120°

4.1.4 Proving Fermat's Problem using Calculus

Given a function $f(x)$, the minimum or maximum points of $f(x)$ are either found at:

1. a *stationary point* of $f(x)$, which is a point at which $\frac{df(x)}{dx} = 0$
2. a *rough point* of $f(x)$, which are either points at which $\frac{df(x)}{dx}$ does not exist or at the end points of $f(x)$, a, b.

If the maximum or minimum point is at a stationary point, p, the nature of the point is decided by the second derivative of the funtion at p, $\frac{d^2f(p)}{dx^2}$.

- If $\frac{d^2f(p)}{dx^2} < 0$ then p is a maximising point.
- If $\frac{d^2f(p)}{dx^2} > 0$, p is a minimising point.

If f is a function of more variables, $f(x_1, x_2, ..., x_i)$, the *partial derivatives* of f must be considered. The stationary points are found at the points where $\frac{\partial f(x_1)}{\partial x} = \frac{\partial f(x_2)}{\partial x} = ... = \frac{\partial f(x_i)}{\partial x} = 0$. For example, to find the stationary points of $f(x, y) = x^2 + 4xy + 2y^2 + 3x - y + 5$, first calculate the partial derivatives,

$$\frac{\partial f(x,y)}{\partial x} = 2x + 4y + 3$$
$$\frac{\partial f(x,y)}{\partial y} = 4x + 4y - 1.$$

Stationary points are then found by solving the equations

$$2x + 4y + 3 = 0$$
$$4x + 4y - 1 = 0$$

Solving for x and y gives the point $(2, -7/4)$. This is a minimising point for $f(x, y)$.

A new way to consider Fermat's problem is by expressing the total length joining the given points and the point we are searching for as a function and then differentiating the function to find the value which minimises it. This idea can be used to prove the result of Fermat's problem which is what we shall do here. This proof is an embellishment of a proof given in [15]. Denote the three points as (a_1, b_1), (a_2, b_1), (a_3, c_3) and the point which minimises the total length connecting them as (x, y). The distances between the point (x, y) and the three other points are given by

$$d_1 = \sqrt{(x - a_1)^2 + (y - b_1)^2}$$
$$d_2 = \sqrt{(x - a_2)^2 + (y - b_2)^2}$$
$$d_3 = \sqrt{(x - a_3)^2 + (y - b_3)^2}$$

Two possible problems to consider are **(A):** to minimise $d_1+d_2+d_3$ and **(B):** to minimise $d_1^2+d_2^2+d_3^2$. Problem **(A)** is Fermat's problem. We shall consider problem **(B)** first as it is easier to solve.

Problem **(B)** is to minimise $f(x, y) = d_1^2 + d_2^2 + d_3^2 = (x - a_1)^2 + (y - b_1)^2 + (x - a_2)^2 + (y - b_2)^2 + (x - a_3)^2 + (y - b_3)^2$. Differentiating with respect to x and y we obtain

$$\frac{\partial f(x,y)}{\partial x} = 2[x - a_1 + x - a_2 + x - a_3]$$
$$\frac{\partial f(x,y)}{\partial y} = 2[y - b_1 + x - b_2 + y - b_3]$$

Solving $\frac{\partial f(x,y)}{\partial x} = 0$, $\frac{\partial f(x,y)}{\partial y} = 0$ gives $x = 1/3[a_1 + a_2 + a_3]$ and $y = 1/3[b_1 + b_2 + b_3]$. This is the *centeroid* of the triangle (a_1, b_1), (a_2, b_2), (a_3, b_3) as shown in Figure 4.6(a).

(a) (b)

Figure 4.6: Centriod and Torricelli point

Problem **(A)** is to minimise

$$f(x,y) = d_1 + d_2 + d_3 = \sqrt{(x-a_1)^2 + (y-b_1)^2} + \sqrt{(x-a_2)^2 + (y-b_2)^2} + \sqrt{(x-a_3)^2 + (y-b_3)^2}.$$

Differentiating with respect to x and y we obtain

$$\frac{\partial f(x,y)}{\partial x} = \frac{(x-a_1)}{\sqrt{(x-a_1)^2 + (y-b_1)^2}} + \frac{(x-a_2)}{\sqrt{(x-a_2)^2 + (y-b_2)^2}} + \frac{(x-a_3)}{\sqrt{(x-a_3)^2 + (y-b_3)^2}}$$
$$\frac{\partial f(x,y)}{\partial y} = \frac{(y-b_1)}{\sqrt{(x-a_1)^2 + (y-b_1)^2}} + \frac{(y-b_2)}{\sqrt{(x-a_2)^2 + (y-b_2)^2}} + \frac{(y-b_3)}{\sqrt{(x-a_3)^2 + (y-b_3)^2}}$$

Setting the partial derivative to zero and solving for x and y in terms of a_1, a_2, a_3, b_1, b_2 and b_3 turns out to be a very difficult system of equations to solve. Fortunately there is another way to approach it.

For a function of two variables, $f(x,y)$, the vector of the partial derivatives, $(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$, is called the *gradient* of $f(x,y)$ and is denoted $\mathbf{grad} f(x,y)$. Taking *implicit derivatives* of $d_1 = \sqrt{(x-a_1)^2 + (y-b_1)^2}$ $\Leftrightarrow d_1^2 = (x-a_1)^2 + (y-b_1)^2$ gives $2d_1(\frac{\partial d_1}{\partial x}) = 2(x-a_1)$ and $2d_1(\frac{\partial d_1}{\partial y}) = 2(y-b_1)$. Dividing through by $2d_1$ gives

$$\frac{\partial d_1}{\partial x} = \frac{x-a_1}{d_1}$$
$$\frac{\partial d_1}{\partial y} = \frac{y-b_1}{d_1}$$

Hence the gradient of d_1 is equal to $G_1 = (\frac{x-a_1}{d_1}, \frac{y-b_1}{d_1})$. Length of the gradient is: $\frac{(x-a_1)^2}{d_1^2} + \frac{(y-b_1)^2}{d_1^2} = \frac{d_1^2}{d_1^2} = 1$ meaning the gradient is a *unit vector*. The gradient of d_1 is a unit vector G_1 from center point (x,y). Similarly, distances d_2 and d_3 have unit vectors G_2 and G_3. The unit vectors, G_1, G_2, G_3 add to zero in order for the function $f(x,y)$ to be minimised, $G_1 + G_2 + G_3 = 0$ This only occurs when the vectors G_1, G_2, G_3 have equal angles between them. Hence the point (x,y) is at the point which makes three angles of 120° with points (a_1, b_1), (a_2, b_2), (a_3, c_3). This point is the Torricelli point of the triangle as shown in Figure 4.6(b).

4.1.5 Generalization to the Euclidean Steiner Problem

Fermat's problem: given three points in the plane, v_1, v_2, v_3, find point p such that $|v_1p| + |v_2p| + |v_3p|$ is minimal, can be generalised in a number of ways.

The first generalisation to be seriously considered during the next few hundred years after Fermat posed his problem was: *Given n points in the plane* $v_1, v_2, ..., v_n$*, find in the plane point p such that*

$\sum_{j=1}^{n} |v_j p|$ *is minimal.* This problem can be thought of as, given a polygon in the plane, trying to find the point inside the polygon so that the total length between each of the vertices and the point is minimal. This generalisation produces a *star* with p connected by an edge to each of the vertices. It is in fact this generalization of the problem which was looked into by the mathematician Jacob Steiner (1796-1863) who was a professor at the University of Berlin specializing in Geometry and whose name is still attributed to a more interesting generalisation of Fermat's problem.

This more interesting generalisation of Fermat's problem was actually made by Jarnik and Kossler around 1930, it is: *Given n points in the plane, construct the shortest tree whose vertices contain these n points.* Courant and Robbins discussed Fermat's problem and this generalization of it in their book *What is Mathematics?* which was published in 1941. It was they that called the problem the *Euclidean Steiner problem* and expressed it in the form: Given n points in the plane, find a shortest network that interconnects them. It is not known whether Steiner made any particular contributions to this generalization of the problem, or even if he was aware if it.

Melzak developed the first algorithm for solving the Euclidean Steiner problem, published in his paper *On The Problem Of Steiner*, [1], which will be looked at in more detail in the next chapter on exact algorithms for solving the Euclidean Steiner problem. His algorithm builds on the geometric solution for Fermat's problem.

It was Gibert and Pollak who first gave the name *Steiner minimal tree* to the solution of the Euclidean Steiner probem, the shortest network connecting the n points and *Steiner points* for vertices of the Steiner minimal tree which are not one of the n original vertices. They discussed the problem in their paper *Steiner Minimal Trees* [9], which is the main reference along with the book *The Steiner Tree Problem* [13], for the rest of this chapter.

4.2 Basic Ideas

4.2.1 Notation and Terminology

In Chapter 2 a *network* was defined, as was the Euclidean minimum spanning tree problem: given a Euclidean graph $G(V)$, find a subgraph such that total sum of the Euclidean distances between the vertices is minimum. The difference between the Euclidean minimum spanning tree problem and the Euclidean Steiner problem is, the solution of the Euclidean minimum spanning tree problem must include only the vertices, V, however the solution of the Euclidean Steiner problem must include the original vertices, V, but can also include other points in the plane as extra vertices in order to make the total connecting length smaller.

Possible solutions for the Euclidean Steiner problem on n points are networks, T, in the Euclidean plane interconnecting the n given points and also some extra points from the plane. The vertices of T are made up of the n original points which we denote as *terminals*, t_i, $i = 1, .., n$ as well as extra vertices added in the plane. Any vertex of T which is not a terminal is called a *Steiner point*, s_j.

The name given to the solution of the Euclidean Steiner tree problem, that is: *the network T interconnecting n, with the addition of s_j, for which the total distance between the n point is minimal*, is a *Steiner minimal tree.*

4.2.2 Kinds of Trees

A Steiner minimal tree, as its name suggests, must be a tree. Recall from Chapter 2 that a minimum spanning graph for a set of vertices must be a tree, as removing the edge which forms the circuit results in a shorter graph without the vertices becoming disconnected. Hence when searching for the solution to the Euclidean Steiner problem, only possible trees for the given network need to be investigated. Therefore when we refer to network T, we shall refer to it as a *tree.*

Steiner minimal trees for a given input n are difficult to find. This is because a network T which is *locally minimal*, meaning it is minimal over a section of the possible solutions, is not necessarily *globally minimal*, the absolute minimum out of all possible solutions.

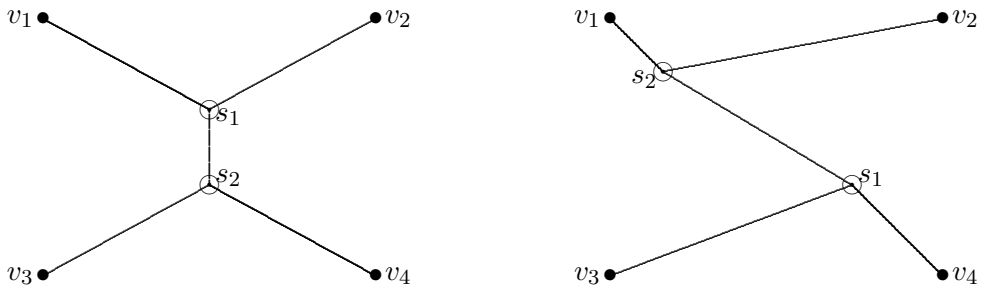


Figure 4.7: Equivalent topologies

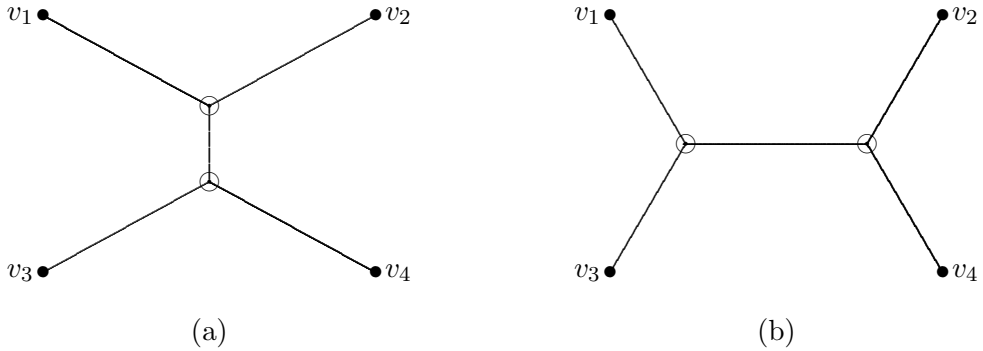


Figure 4.8: Different topologies

The *topology* of a tree T, is a description of the topologically features of T. This means a description of which pairs of vertices from all possible terminals and Steiner points $t_1, ..., t_n, s_1, ...s_j$ are connected. The topology only specifies the connections, not the positioning of the Steiner points and thefore not the lengths of the edges. The positioning of the n terminals is known as this is the input for the problem. Figure 4.7 shows two graphs which are topologically identical. Both have four terminals, $v_1, ..., v_4$, and two Steiner points, s_1, s_2 whose positions change between Figure 4.7(a) and Figure 4.7(b). The black points are the terminals and the unfilled circles are the Steiner points. It is clear that for both trees, terminals v_1 and v_2 are connected to the same Steiner point, as are v_3 and v_4. However the location of the Steiner points s_1 and s_2 are different in the two graphs and hence the total length of the edges connecting the terminals are different. Figure 4.8 shows two networks which are topologically different. In Figure 4.8(a), terminals v_1 and v_2 are connected through the same Steiner point as are v_3 and v_4 but in Figure 4.8(b), terminals v_1 and v_3 are connected through the one Steiner point and v_2 and v_4 through the other. A tree which is shorter than any other tree with the same topology is called a *relatively minimal tree* for that topology. 4.7(a) is actually the relatively minimal tree for its topology.

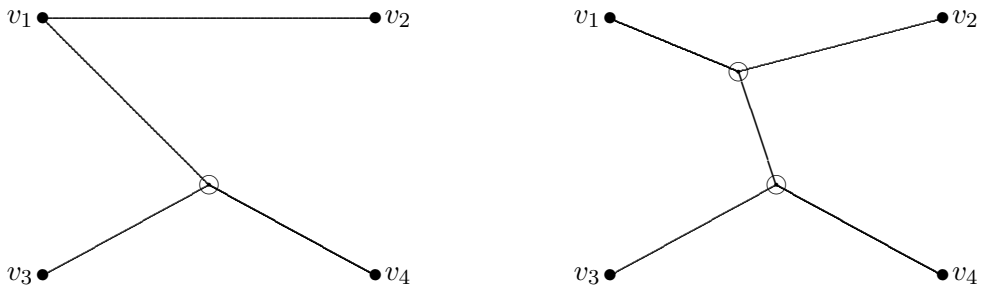
Relatively minimal trees are not allowed to have any edges of zero length. Due to this condition, some topologies have no relatively minimal tree. Consider Figure 4.9(a); the length of the tree with this topology is minimal as $|s_1 - s_2| \to 0$. The tree which is approached is called a *degenerate tree*. Figure 4.9(b) is the degenerate tree for this topology.

Two operations which can be used to act on a tree T are called *shrinking* and *splitting*. Deleting an edge and collapsing its two end points is called *shrinking*. The opposite of shrinking is *splitting* which consists of disconnecting the edges, $[v_1, u]$ and $[v_2, u]$, which are connecting vertex u to v_1 and v_2 from vertex u, and then forming three new edges, $[v_1, u']$, $[v_2, u']$, $[u, u']$, connecting vertices v, u_1, u_2 to a newly formed vertex v'. Figure 4.10(a) shows the resulting graph when you shrink edge $[v_1, s_1]$

(a) (b)

Figure 4.9: Degenerate tree

of Figure 4.7(a). Figure 4.10(b) shows the resulting graph when you re-split edges $[v_1, v_2]$ and $[v_1, s_2]$ of Figure 4.10(a).



(a) (b)

Figure 4.10: Shrinking and Splitting

If a tree cannot be shortened by a small permutation, *including* splitting and shrinking, then the tree is called a *Steiner tree*. A Steiner tree is always a relatively minimal tree for its topology and a Steiner minimal tree is always a Steiner tree.

Steiner minimal tree $\overset{always}{\rightarrow}$ **Steiner tree** $\overset{always}{\rightarrow}$ **relatively minimal tree**

Relatively minimal trees are useful because at most one relatively minimal tree exists for a topology and if one does exists then it is possible to construct it. It is possible to obtain Steiner minimal trees by first constructing the relatively minimal tree for each possible topology and then finding out which one has the shortest length, this will be the Steiner minimal tree. As we will see, this is not an easy thing to do as there are thousands of different relatively minimal trees even for a network with $n = 6$.

4.3 Basic Properties of Steiner Trees

We have looked at some basic ideas surrounding the Euclidean Steiner problem including three types of tree which are important when trying to solve it. This section looks at some basic properties of Steiner trees. These properties make it easier to search for all the Steiner trees for a given problem, hence narrowing down the search for the Steiner minimal tree.

4.3.1 Angle Condition

If two edges of T meet at an angle which is less than 120° then it is possible to shorten the tree in the following way. Call the vertex at which two edges, e_1 and e_2, meet at an angle less than 120°, v.

Next find two points p_1 and p_2 on e_1 and e_2 respectively such that $|p_1 - v| = |p_2 - v|$. A new Steiner point, s, can then be located at the Torricelli point of $\Delta p_1 v p_2$ (recall the $n = 3$ case of the Euclideam Steiner problem). A new shorter tree is formed by connecting v and the other two endpoints of e_1 and e_2 to Steiner point s; see Figure 4.11. Hence, a condition for all Steiner trees is: ***no edges can meet at an angle of less than 120°***. This is known as the *angle condition* for Steiner trees. The angle condition means that no vertex of a Steiner tree (Steiner point or terminal) can have degree greater than 3 as this would result in an angle less than 120°. The angle condition also means that no edges of a Steiner minimal tree can cross because crossing edges results in two angles less than or equal to 90°.

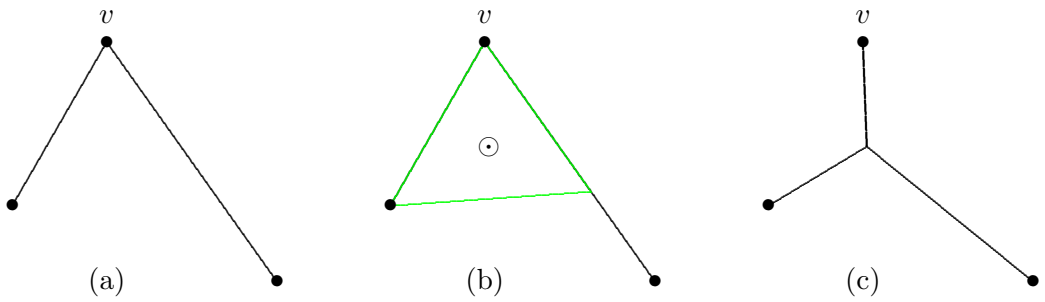


Figure 4.11: Angle condition

4.3.2 Degrees of Vertices

Steiner points are not required to be in a Steiner tree T, their purpose is only to reduce the total length of connecting lines. As such, it is clear that all Steiner points of degree 1, $d(s_j) = 1$ can be removed from T, along with their connecting edges, making the total length of the tree smaller. Furthermore, all Steiner points of degree 2, $d(s_j) = 2$, can be removed, along with their two connecting edges, and be replaced by an edge connecting the two vertices (whether they be terminals or Steiner points) adjacent to the Steiner point, s_j. From this we can deduce that all Steiner points must have degree of at least 3, $d(s_j) \geq 3$.

A result of the angle condition is that all vertices of a Steiner tree, terminals or Steiner points, must have degree less than or equal to 3. Hence from this we can deduce that all Steiner points must have degree of exactly 3, $d(s_j) = 3$, with three edges meeting at 120°. Also, that all terminals must have degree of less than or equal to 3, $d(v_i) \leq 3$, with edges meeting at angles of 120° or more.

4.3.3 Number of Steiner Points

Each Steiner tree T has n vertices, $v_1, ..., v_n$ and k Steiner points, $s_1, ..., s_k$. Recall the results from Chapter 2: (i) every tree with $|V|$ vertices has $|E| = |V| - 1$ edges, (ii) for all graphs, $|E| = \frac{\sum_{i=1}^{n} d(v_i)}{2}$.

Theorem. 4.1. *A Steiner tree has at most $n - 2$ Steiner points.*

Proof. A Steiner tree has $n + k$ vertices, therefore by (i), it has $n + k - 1$ edges. Since each Steiner point has degree 3, $d(s_j) = 3$ and each terminal has *at least* degree 1, $d(v_i) \geq 1$, by (ii), the number of edges must be *at least* $[3k + n]/2$. It follows that

$$n + k - 1 \geq [3k + n]/2 \tag{4.2}$$

$$n - 2 \geq k \tag{4.3}$$

□

4.3.4 Summary of Geometric Properties

Let T be a Steiner tree with n terminals, v_i and k Steiner points, s_j:

1. T has vertices $v_1, ..., v_n, s_1, ..., s_k$
2. T has no crossing edges
3. $d(s_i) = 3$, $1 \leq i \leq k$
4. Each s_j, $1 \leq j \leq k$ is the Steiner point for the triangle made up of the 3 vertices adjacent to s_j
5. $d(v_i) \leq 3$, $1 \leq i \leq n$
6. $0 \leq k \leq n - 2$

4.3.5 Convex hull

In Euclidean space, a shape S is called *convex* if, for every pair of points within it, p_i and p_j, every point on the straight line, $\overrightarrow{p_i p_j}$ is also within S. For example a circle is convex but a cresent is not.

The *convex hull* for a set of points $v_1, ..., v_n$ in Euclidean space is the minimal convex shape containing $v_1, ..., v_n$, i.e. the shape for which the total area is minimum. This concept can be thought of as the points $v_1, ..., v_n$ being posts, the convex hull is then the shape created if you put a big elastic band around the outside of the posts.

In a Steiner tree, all the Steiner points lie in the convex hull of the terminals $v_1, ..., v_n$.

4.3.6 Full Steiner Trees

A topology with the maximum number of Steiner points, $k = n - 2$, is called a *full topology*. The corresponding relatively minimal tree for this topology is called a *full Steiner tree*. A full Steiner tree is a Steiner tree because there is no possibility of splitting it and forming new Steiner points. In a full Steiner tree, every terminal has degree of exactly 1, $d(v_i) = 1$. Every full Steiner tree has $n + k - 1 = n + n - 2 - 1 = 2n - 3$ edges.

The definition of a *subgraph* was given in Chapter 2. A Steiner topology can be uniquely broken down into edge disjoint subgraphs, (edge disjoint meaning none of the subgraphs share any edges) each of which is a full Steiner topology. Full Steiner trees are easier to construct than Steiner trees so one way to find a Steiner tree is to build it up from its full Steiner tree components.

A Steiner tree which is not full can be decomposed into a union of full trees in the following way: Replace each terminal v_i which has degree $d(v_i) = d$ where $d \geq 2$ by d new terminals $v_{i1}, ..., v_{id}$ all locacted at the same position, v_i, but disconnected. Connect each of the d edges which were connected to v_i to a different new terminal, $v_{i1}, ..., v_{id}$. This will result in several smaller full Steiner trees which are called *full components* of the original Steiner tree. Conversely, when a specified topology is not full, it is possible to find the topologies of the full components and then construct the full components seperately before joining them to produce the desired Steiner tree.

The fact that a Steiner topology can be broken down into edge disjoint full Steiner topologies leads to the following two corollaries. An explaination of how being able to spilt Steiner trees into full components leads to the uniquness of relatively minimal trees and Steiner trees is given in [9].

Corollary. 4.1. *There exists at most one relatively minimal tree for a given Steiner topology.*

Corollary. 4.2. *There exists at most one Steiner tree for a given Steiner topology.*

4.4 Number of Steiner Topologies

One way of finding a Steiner minimal tree for n is to construct all possible Steiner trees and find which one has the shortest length. Corollary 4.2 states that there exists at most one Steiner tree for a given Steiner topology so in order to exhaust all possible Steiner trees it is sufficient to find all possible Steiner topologies.

Clearly the number of possible Steiner topologies increases with the number of terminals, n. We shall now consider what the actual relationship between the number of possible Steiner topologies and number of terminals is. Since to find a minimal Steiner tree we must consider all Steiner topologies, the way the number of different Steiner topologies increases with n gives us an indication of how much harder it is to find Steiner minimal trees as n increases.

Let $f(n)$, $n \geq 3$ be the number of full Steiner topologies with for a given n. Since they are *full* Steiner topologies we know they will have $n-2$ Steiner points. We want first to derive a formula for the number of full Steiner topologies, $f(n)$.

Recall that in a full Steiner topology, every terminal has degree 1 and is adjacent to a Steiner point. Recall also that every full Steiner topology has $2n-3$ edges. Let $f(n+1)$ be a full Steiner tree with $n+1$ terminals. If we remove terminal v_{n+1} and its adjacent Steiner point then we obtain a full Steiner topology with n terminals. Every full Steiner topology with $n+1$ terminals can be obtained from a full Steiner topology with n terminals by adding a Steiner point, s_j, in the middle of one of the $(2n-3)$ edges and adding an edge connecting s to the new terminal v_{n+1}. Hence $f(n+1) = (2n-3)f(n)$.

Theorem. 4.2. *Let $f(n)$, $n \geq 2$ be the number of full Steiner topologies for n terminals. $f(n) = (2n-4)!/[2^{n-2}(n-2)!]$.*

Proof. We prove this by *induction*:

Prove true for $n = 3$. $f(3) = 1$ as there is only one possible full Steiner tree for $n = 3$ which is each terminal connected by one edge to the same Steiner point. $f(3) = \frac{(2\times 3-4)!}{2^{3-2}(3-2)!} = 1$. So proved true for $n = 3$.

Assume true for n. Assume that $f(n) = \frac{(2n-4)!}{2^{n-2}(n-2)!}$.

Prove true for $n+1$. (Want to show that $f(n+1) = \frac{(2(n+1)-4)!}{2^{(n+1)-2}((n+1)-2)!} = \frac{(2n-2)!}{2^{(n-1)}(n-1)!}$)

$$\begin{aligned}
f(n+1) &= (2n-3)\cdot f(n) \\
&= (2n-3)\cdot \frac{(2n-4)!}{2^{n-2}(n-2)!} \\
&= \frac{(2n-3)!}{2^{n-2}(n-2)!} \\
&= \frac{(2n-2)\times(2n-3)!}{(2n-2)\times 2^{n-2}(n-2)!} \\
&= \frac{(2n-2)!}{2\times 2^{n-2}\cdot(n-1)\times(n-2)!} \\
&= \frac{(2n-2)!}{2^{n-1}\cdot(n-1)!}
\end{aligned}$$

□

Let $F(n,k)$, $n \geq 3$ be the number of Steiner topologies with n terminals and k Steiner points, where there are no terminals of degree 3, $d(v_i) = 1, 2$. $F(n,k)$ can be obtained from the number of full Steiner topologies with $n = k$, $f(k)$. This is done by first selecting $k+2$ terminals and a full Steiner topology on it and then adding the remaining $n-k-2$ terminals one at a time at interior points on one of the $(k+2)+k-1 = 2k+1$ edges. The first terminal can go to one of $2k+1$ edges, the second to one of $2k+2$ edges and the $(n-k-2)^{th}$ to one of $2k+n-k-2 = k+n-2$ edges. So

$$F(n,k) = \binom{n}{k+2} f(k) \frac{(n+k-2)!}{(2k)!}$$

This is the number of Steiner topologies with terminals of degree 1 and 2, $d(v_i) = 1, 2$. Lets now consider Steiner topologies with terminals of degree 3, $d(v_i) = 3$ as well. Let n_3 be the number of terminals with degree 3 in the topology. A Steiner topology with n_3 terminals of degree 3 can be obtained from a Steiner topology with $n - n_3$ terminals of degree 1 and 2 and $k + n_3$ Steiner points, by labelling n_3 of the Steiner points as terminals.

Let $F(n)$, $n \geq 3$ be the number of Steiner topologies with n terminals.

$$F(n) = \sum_{k=0} \sum_{n_3=0} \binom{n}{n_3} \frac{F(n-n_3, k+n_3)(k+n_3)!}{k!} \tag{4.4}$$

Both functions $f(n)$ for full Steiner topologies and $F(n)$ for Steiner topologies are *superexponential*. This means they increase even faster than an exponetial function. The first values of $f(n)$ and $F(n)$ are given in Table 4.1.

n	2	3	4	5	6	7
$f(n)$	1	1	3	15	105	945
$F(n)$	1	4	31	360	5625	110800

Table 4.1: How number of full Steiner topologies and Steiner topologies increases with n

4.5 Summary

This chapter introduced what the Euclidean Steiner problem is.

We started by looking at the historical background of the problem. We looked in depth at the $n = 3$ case of the problem, known as Fermat's problem and we discussed two solutions to this problem using geometric construction. We also looked at a proof Fermat's problem using calculus. We then discussed the generalization of Fermat's problem to the general case which is the Euclidean Steiner problem. The next section looked at some basic ideas surrounding the Euclidean Steiner problem. We introduced some notation and terminology most importantly: terminals, Steiner points and Steiner toplogies and also the three types of tree: Steiner minimal trees, Steiner trees and relatively minimal trees. The next section discussed the basic properties of Steiner trees which are the optimal tree for a given Steiner topology. These properties included the angle condition, the degrees of vertices and the number of Steiner points. Finally in this chapter we looked at the number of Steiner topologies for a problem with n terminals and how it is given by a superexponential function of n.

Now we have looked in depth at what the Euclidean Steiner problem is, in the next chapter we move on to look at ways of solving the problem. The next chapter explores *exact algorithms* which are algorithms which search for the exact solution to the problem, these are different from *heuristics* which are a type of *approximation algorithm* which we shall consider in the penultimate chapter of this report.

Chapter 5

Exact Algorithms

The aim of this chapter is to draw together some of the ideas from the previous chapter and introduce some algorithms used to solve the Euclidean Steiner tree problem. The algorithms introduced in this chapter are called *exact algorithms* because their object is to find the exact solution to the problem. This objective is different from that of *approximate algorithms* or *heuristics* which aim to find an approximately good solution to the problem, but not necessarily the exact solution. Heuritics are introduced in the next chapter

We start by returning to Fermat's problem and introduce an algorithm called the *3 point algorithm* which is used for solving this $n = 3$ case of the Euclidean Steiner problem. The 3 point algorithm also forms the basis for the first algorithm ever used to solve the Euclidean Steiner problem which is called the *Melzak algorithm*, so this leads us on to discuss the Melzak algorithm. An numerical algorithm known as *Smith's numerical method* is then introduced. An example of the workings of the Melzak algorithm and Smith's numerical method are given; as the same example is used for both algorithms this allows for a comparison of the methods. This chapter concludes with a brief mention on the best exact algorithm for solving the Euclidean Steiner problem known today which is the *GeoSteiner algorithm.*

The main references used for this chapter are [1, 13, 19].

5.1 The 3 Point Algorithm

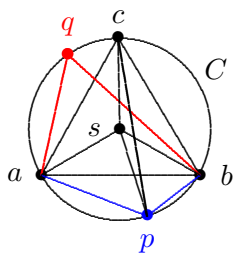
In the last chapter we introduced Fermat's problem which was to find a point p such that the total distance to three other points a, b and c is minimised. We saw if the points a, b and c make three angles which are less than 120°, then the solution is the point, p, which for which the edges ap, bp and cp make three angles of 120° and that this point can be found using either the Torricelli method or Simpson method. We saw that if the points a, b and c make an angle which is greater than 120°, then the solution is the vertex of the obtuse angle. In this section we shall only consider three points a, b and c such that all angles of Δabc are less than 120°.

We shall now return to Fermat's problem to consider a few more properties which will allow us to construct an algorithm for the $n = 3$ case. We start by introducing a lemma from Euclidean geometry which is neccessary for the proof of the working of the algorithm. The lemma and algorithm are taken from the book, *The Steiner Tree Problem - A Tour through Graphs, Algorithms, and Complexity by Hans Jurgen Promel and Angelika Steger*, [13], the proofs are also based on the proofs in this book but more computational steps have been added and they have been more thoroughly explained.

5.1.1 Lemma from Euclidean Geometry

Lemma. 5.1. *Let Δabc be an equilateral triangle with circumscribing circle C. Then all points p on the smaller segment of C between a and b make an angle of 120° between a and b and all points q on the larger segment between a and b make an angle of 60° between a and b.*

Proof. Let the notation θabc mean the angle between the edges ab and bc. Let p be any point on the smaller segment of C. θapc is equivalent to θaqb due to symmetry, so it is enough to show that θapb equals 120° and θapc equals 60°, to prove the lemma. s is the centre of the circle C. The triangles, Δasp and Δbsp are both isosceles because the edges as, sp, bs and sb are all the equal to the radius of the circle, C, and hence are all equal. This means $\theta apb = 1/2(180° - \theta asp) + 1/2(180° - \theta bsp) = 180° - 1/2(\theta asb)$. Because Δabc is equilateral, $\theta asb = 120°$, so $180° - 1/2(\theta asb) = 120°$. By a similar argument, the triangle Δcsp is isosceles so, $\theta apc = 1/2(180° - \theta asp) + 1/2(180° - \theta csp) = 180° - 1/2(\theta asp + \theta csp) = 180° - 1/2(120° + 120°) = 60°$.



□

5.1.2 The Algorithm

The input for this algorithm is three points a, b and c such that all the angles of the triangle Δabc are less than 120°. The output is the Torricelli point p for a, b and c.

Algorithm. 5.1 (The 3 point algorithm)**.**

1. *Construct an equilateral triangle, Δabd such that d is on one side of the line ab and c is on the other.*
2. *Construct a circle C circumscribing Δabd.*
3. *The Torricelli point is the intersection of the line cd with the circle C.*

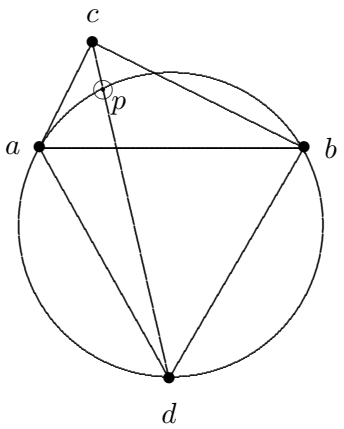


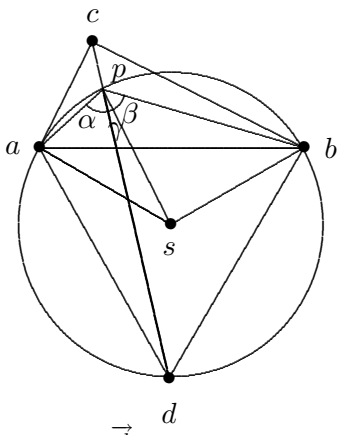
Figure 5.1: 3 Point Algorithm

Theorem. 5.1. *Algorithm 5.1 does find the Torricelli point for points a, b and c.*

Proof. We saw in the previous chapter that if the point p makes three angles of 120° with a, b and c, then p is is the Torricelli point. Therefore, in order to check whether p is the Torricelli point, it is enough to check whether angles θapb, θbpc and θcpa are all 120°. From Lemma 5.1 we know that $\theta apb = 120°$. By the same reasoning, if we drew a circle circumscribing the equilateral triangle $\Delta bcd'$ and the line ad' we would deduce $\theta bpc = 120°$. From the circle circumscribing $\Delta cad''$ and bd'', $\theta cpa = 120°$. □

Theorem. 5.2. *The p found in Algorithm 5.1 satifies* $|ap| + |bp| = |dp|$.

Proof. Let the points a, b, c, d and p be as explained in Algorithm 5.1 and shown in Figure 5.1 and let point s be the centroid of the triangle Δabd. Let the angle θaps be denoted α, θdps be denoted γ and θspb be denoted β.



By connecting s to the midpoint of pb, $\frac{\overrightarrow{pb}}{2}$, the midpoint of pa, $\frac{\overrightarrow{pa}}{2}$ and the midpoint of pd, $\frac{\overrightarrow{pd}}{2}$, we construct three right-angle triangles $\Delta ps\frac{\overrightarrow{pb}}{2}$, $\Delta ps\frac{\overrightarrow{pa}}{2}$ and $\Delta ps\frac{\overrightarrow{pd}}{2}$. Recall that in a right-angled triangle

$$\cos\alpha = \frac{|adjacent|}{|hypotenuse|} \tag{5.1}$$

Therefore we can deduce:

$$\cos\alpha = \frac{1/2|ap|}{|ps|}$$
$$\cos\beta = \frac{1/2|bp|}{|ps|}$$
$$\cos\gamma = \frac{1/2|dp|}{|ps|}$$

If we show that $\cos\alpha + \cos\beta = \cos\gamma$, it will follow that $\frac{1/2|ap|}{|ps|} + \frac{1/2|bp|}{|ps|} = \frac{1/2|dp|}{|ps|}$ which is equivalent to $|ap| + |bp| = |dp|$ which is what we wish to show.

From Lemma 5.1, $\theta apb = 120°$ and $\theta apd = \theta dpb = 60°$. Hence, $\beta = 120° - \alpha$ and $\gamma = \alpha - 60°$. Recall the double angle formula

$$\cos(x \pm y) = \cos x \cos y \mp \sin x \sin y \tag{5.2}$$

So therefore,

$$\cos(\beta) = \cos(120° - \alpha) = \cos 120 \cos\alpha + \sin 120 \sin\alpha = -\frac{1}{2}\cos\alpha + \frac{\sqrt{3}}{2}\sin\alpha$$
$$\cos(\gamma) = \cos(\alpha - 60°) = \cos\alpha \cos 60 + \sin\alpha \sin 60 = \frac{1}{2}\cos\alpha + \frac{\sqrt{3}}{2}\sin\alpha$$

Subtracting the first equation from the second we obtain $\cos\gamma - \cos\beta = \frac{1}{2}\cos\alpha + \frac{\sqrt{3}}{2}\sin\alpha - (-\frac{1}{2}\cos\alpha + \frac{\sqrt{3}}{2}\sin\alpha) = \cos\alpha$. Hence $\cos\alpha + \cos\beta = \cos\gamma$ and therefore $|ap| + |bp| = |dp|$.

□

5.1.3 An Example

If the points as shown in Figure 5.1 are $a = (1, 5)$, $b = (6, 5)$ and $c = (2, 7)$. The point d is found by finding the point which is the same distance from a and b, hence by solving $\sqrt{(x-1)^2 + (y-5)^2} = \sqrt{25}$

and $\sqrt{(x-6)^2+(y-5)^2} = \sqrt{25}$. d turns out to be $(3.5, 0.670)$. The midpoint of the circle is found to be $(3.5, 3.557)$. Recall, the equation of a circle is given by

$$C : (x-x_1)^2 + (y-y_1)^2 = r^2 \tag{5.3}$$

where (x_1, y_1) is the midpoint of the circle and r is the radius of the circle. Hence the equation of our circle C is, $C : (x-3.5)^2 + (y-3.557)^2 = 8.332$. Recall, the equation of a line through two points (x_1, y_1) and (x_2, y_2) is given by

$$y = y_1 + [(y_2 - y_1)/(x_2 - x_1)](x - x_1) \tag{5.4}$$

So the equation of the line $\overrightarrow{cd}$ is: $y = 7 + [(0.670-7)/(3.5-2)] \cdot (x-2)] = -4.22x + 15.44$. The point p is the point at which $\overrightarrow{cd}$ intersects C and turns out to be $(2.205, 6.137)$. By Theorem 5.2, this point p should satify

$$|ap| + |bp| = |dp| \tag{5.5}$$

$|ap| = \sqrt{(1-2.205)^2 + (5-6.137)^2} = 1.657$ and $|bp| = \sqrt{(6-2.205)^2 + (5-6.137)^2} = 3.962$, so $|ap| + |bp| = 5.62$ and $|dp| = \sqrt{(3.5-2.205)^2 + (0.607-6.137)^2} = 5.68$.

5.2 The Melzak Algorithm

Melzak was the first mathematician to propose an algorithm for the Euclidean Steiner problem. The algorithm works by finding the Steiner tree for every possible topology. By Corollary 4.2, there is a unique Steiner tree for every topology. Melzak's algorithm focuses on how to find the Steiner tree for a given topology, and then once they are all found, the one which is shortest will be the Steiner minimal tree. The working of the algorithm will be explained here and an example will be given. The algorithm, along with it derivation by Melzak, can be seen in its complete form in the paper *On the problem of Steiner*, [1].

The algorithm is based on the 3 point algorithm we have just seen for the $n = 3$ case. Assume we know the topology of our Steiner tree, T. We therefore know the number of Steiner points in T and all the connections between the terminals and the Steiner points $v_1, ..., v_n, s_1, ..., s_k$. But we do not know the positions of the Steiner points.

Every Steiner tree with at least one Steiner point will also have at least two terminals, call them a and b, which are connected to the same Steiner point, call it p. We say that terminals a and b are *siblings*. We know that all Steiner points have degree 3, so there will be a third vertex adjacent to p, call it c. We know that p is the Torricelli point of Δabc. Due to Theorem 5.2, the tree T' obtained by removing edges ap and bp and adding edge pd has the **same length** as tree T (where d is the third point of the equilateral triangle Δabd constructed on the edge ab of triangle Δabc). Therefore, finding the Steiner tree for n with a given topology is equivalent to finding the Steiner tree for T with terminals a and b and their connecting edges to c replaced by a new terminal d connected to c.

The location of the Steiner point p is determined by constructing the equilateral triangle Δabd on the edge ab of the triangle Δabc. If c is a terminal, then which side to construct the equilateral triangle on is clear, however if c is a Steiner point, its location is not known so it is neccessary to check **both** possibilities.

Melzak's algorithm consists of two stages. In the first stage, which is known as the *merging stage*, the topology is reduced as the number of terminals is reduced from n down to 2 and original terminals are replaced by new terminals by the process described above. Every original terminal, except at most one, is replaced during this stage. During this stage a number of subproblems are generated due to the fact that if c is a Steiner point, there are two possible choices for it's location. Once the number if terminals has been reduced to 2 and no more Steiner points remain in the altered topology, the second stage takes over. The second stage is the *reconstruction stage* or *expansion stage*. Is starts by connecting the two remaining terminals by a straight line. Then a Steiner tree is built up by replacing the orginal terminals with Steiner points in the position of the Torricelli point for 3 of the vertices.

Describing the algorithm in words does not make it obvious how it works. A formal definition of the algorithm is given next, followed by an example in which the workings of the algorithm become a lot clearer.

5.2.1 The Algorithm

The input for the algorithm is a set of n terminals and a topology. The output of the algorithm is the Steiner tree for this topology.

Algorithm. 5.2 (Melzak's algorithm)**.**

1. **while** *there exists one Steiner point* **do**

   Choose two terminals a and b which are both adjacent to the same Steiner point, p. Call the third adjacent point to p, c. Compute the two points d_1 and d_2 which form an equilateral triangle with a and b. Remove a and b from the topology.

   If *c is a terminal then choose from d_1 and d_2 the point such that the triangle Δabd_i lies* outside *of Δabc and replace p with d_i.*

   Otherwise *generate two new problems, one for d_1 and one for d_2. In both of them replace the Steiner point p by a terminal with coordinates d_i. Handle the generated problems recursivly.*

2. **while** *tree T does not contain all original terminals* **do**

   if *there exists no Steiner points* **then**

   Connect the terminals by edges according to the current topology.

   else *[This is the backtracking step. Let a and b be the two terminals removed at this corresponding step in STAGE 1, d be the inserted terminal, and c the adjacent vertex to d in the current topology of the tree]*

   If *the circle circumscribing Δabd intersects the line dc (except at d itself) then connect a and b with the intersection point p and delete the line segement dp.*

   Otherwise *STOP: the given topology admits no valid Steiner tree.*

5.2.2 An Example

The workings of Melzak's algorithm are much easier to understand by considering an example. We shall now work through an example of Melzak's algorithm with **terminals**: $v_1 = (1, 3), v_2 = (2, 1), v_3 = (5, 4)$ and $v_4 = (4, 0.6)$, as shown in Figure 5.2(a) and **topology**: 2 Steiner points s_1 and s_2 with v_1 and v_2 connected to s_1 and v_3 and v_4 connected to s_2, as shown in Figure 5.2(b).

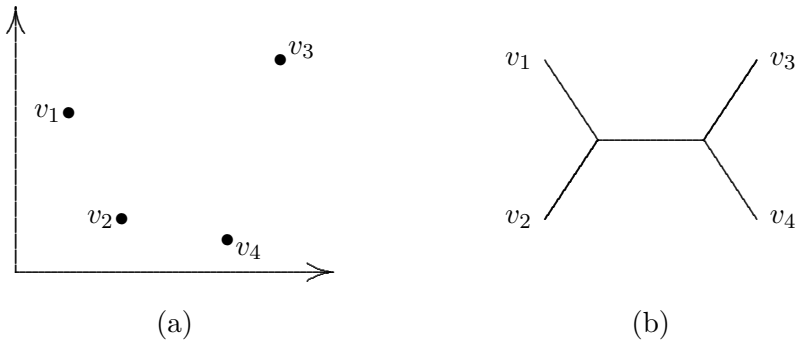


Figure 5.2: Melzak Algorithm: starting terminals and topology

STEP 1: Choose $a = v_1$ and $b = v_2$ which are both adjacent to Steiner point, p. The third point adjacent to p is also a Steiner point, so two subproblems are generated; the first with d_1 replacing p

and the second with d_2 replacing p, as shown in Figure 5.3(a) and Figure 5.3(b). The new topology for the subproblems is shown in Figure 5.3(c).

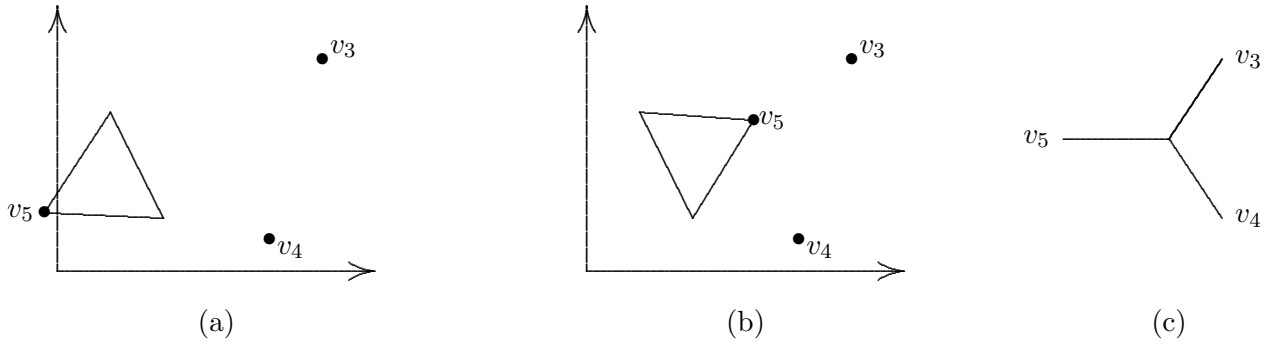


Figure 5.3: Melzak Algorithm: STEP1

STEP 2: For each subproblem, as shown in Figure 5.3(a) and Figure 5.3(b), choose $a = v_5$ and $b = v_4$ which are both adjacent to a Steiner point, p. The third point adjacent to p in each each subproblem is the terminal v_3. For each subproblem, either d_1 or d_2 is selected so that Δabd_i lies outside of Δabc. Therefore for the first subproblem the graph as shown in Figure 5.4(a) is selected and for the second problem the graph as shown in Figure 5.4(c) is chosen. New topology has $a = v_5$ and $b = v_4$ removed and replaced by v_6 connected to v_3 as shown in Figure 5.4(e).

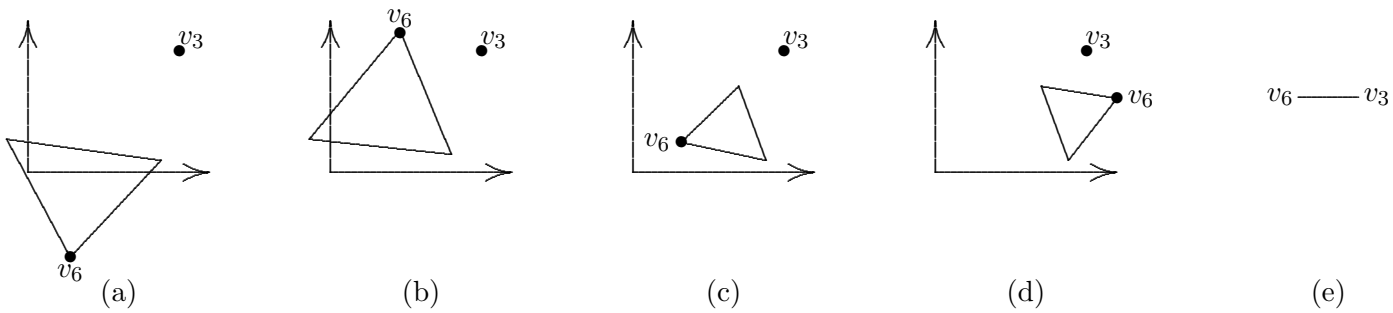


Figure 5.4: Melzak Algorithm: STEP2

STEP 3: Now we proceed to the *expansion phase*. We start by joining the remaining two vertices v_3 and v_6 for both of the subproblems as shown in Figure 5.5.

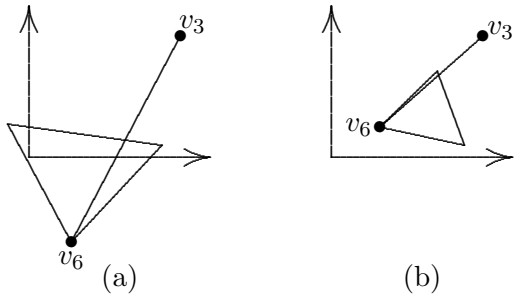


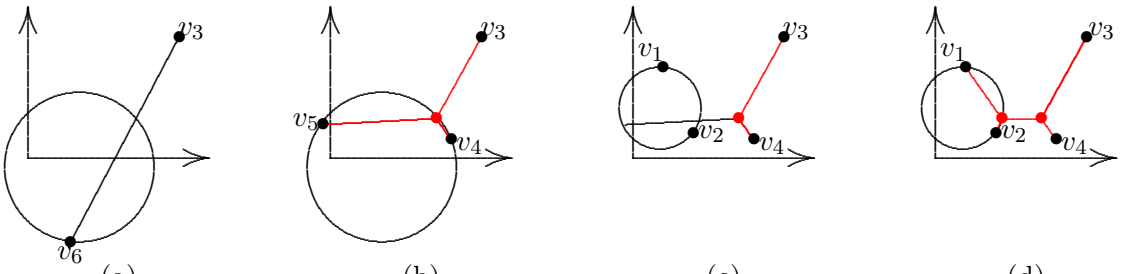
Figure 5.5: Melzak Algorithm: STEP3

STEP 4: Using the subproblem as shown in Figure 5.5(b), after the second expansion phase, the STOP criteria is met as one of the circumscribing circles, Δabd does not intersect the line dc.

Hence we shall consider the subproblem as shown in Figure 5.5(a). The last two terminals removed were v_4 and v_5 when they were replaced by terminal v_6 and point c was v_3. The circle circumscribing $\Delta v_4 v_5 v_6$ intersects the line $v_3 v_6$ at a point which is not v_6 so we make this interesection point p and connect it to v_4 and v_5 and delete the line segment pv_6. This step is shown in the progression from Figure 5.6(a) to Figure 5.6(b).

The two terminals removed at the previous step were v_1 and v_2 when they were replaced by terminal v_5 and point c was the Steiner point p. The circle circumscribing $\Delta v_1 v_2 v_5$ intersects the line $v_5 p$ at a point which is not v_5 so we call this intersection point p' and connect p' to v_1 and v_2 and remove the line segment $p' v_5$.

The resulting network is the Steiner tree for the given topology which is shown in Figure 5.6(d).



(a) (b) (c) (d)

Figure 5.6: Melzak Algorithm: STEP4

5.2.3 Complexity of Melzak's Algorithm

The complexity of Melzak's algorithm for finding the Steiner tree for a given topology is bounded by the number of Steiner points to the power of two because each Steiner point, depending on the nature of its adjacent vertices, can generate up to two subproblems which need to be explored. Since the number of Steiner points is at most $n-2$, an upper bound for the complexity of Melzak's algorithm is $O(2^n)$.

To find the minimal Steiner tree of a network using Melzak's algorithm, it is necessary to find all possible Steiner topologies for the network, then apply Melzak's algorithm on each topology to find all the Steiner trees and then finally calculate the length of each Steiner tree and the Steiner minimal tree is the one which is the shortest.

5.3 A Numerical Algorithm

A function of one variable, $f(x)$, is called *convex* if for any two points in its domain, x_1 and x_2, and for any λ where $0 < \lambda < 1$ then, $f[\lambda x_1 + (1-\lambda)] \leq \lambda f(x_1) + (1 - \lambda f(x_2))$. If the inequality is scrict then $f(x)$ is called *strictly convex*.

If a function is convex then the second derivative $f''(x) \geq 0$ for all x. The function will have only one stationary point, where $f'(x) = 0$ and this point will be a minimum. A *locally minimal point* is one which is a minimum over a subset of the domain. A *globally minimal point* is one that is the minimal point over the entire domain of the function, hence it is the smallest of all locally minimal points. If a function is strictly convex, this means that is has only one minimal point, so if a minimal point is found it is the global minimal point. It is found by finding the point at which $f'(x) = 0$.

If a function of many variables $f(x_1, ..., x_n)$ is convex, then the same concept applies and the minimal point is found by finding the point at which the first partial derivatives are all equal to zero.

5.3.1 The Algorithm

The length of a tree for a given topology is a convex function of the location of the Steiner points. Therefore one way of finding the positions of Steiner points for a given topology is by setting the first partial derivatives to zero. This results in a system of nonlinear equations as we saw for the $n = 3$ case is the previous chapter, when we used derivatives to solve Fermat's problem. Since it is a system of nonlinear equations, a numerical method is required. It turns out that the function for the tree length has complicated behaviour when the Steiner points get near to the optimum points which means that usual numerical methods are inefficient. We will now consider an iterative procedure which

was proposed by a mathematician called Smith, where at each step the positions of the Steiner points are updated.

Algorithm. 5.3 (Smith's Numerical Method)**.** *Let T be a Steiner topology with n terminals and k Steiner points. Let $s_k = (x_k, y_k)$ be the k^{th} Steiner point with Euclidean coordinates (x_k, y_k). Let the coordinates of the points adjacent to $s_k = (x_k, y_k)$ be given by $w_j = (u_j, v_j)$. The summations below are over j where $[w_j, s_k]$ is an edge of the tree T. At the i^{th} step, $i = 0, 1, ...$ solve the system of $2n-4$ linear equations.*

$$x_k^{i+1} = \frac{\sum \frac{u_j^{i+1}}{\sqrt{[u_j^i - x_k^i]^2 + [v_j^i - y_k^i]^2}}}{\sum \frac{1}{\sqrt{[u_j^i - x_k^i]^2 + [v_j^i - y_k^i]^2}}}$$

$$y_k^{i+1} = \frac{\sum \frac{v_j^{i+1}}{\sqrt{[u_j^i - x_k^i]^2 + [v_j^i - y_k^i]^2}}}{\sum \frac{1}{\sqrt{[u_j^i - x_k^i]^2 + [v_j^i - y_k^i]^2}}}$$

Smith proved that for all initial choices of Steiner point coordinates, (x_k^0, y_k^0), this method converges to the unique optimum Steiner point coordinates. He also proved that the algorithm converges in a way such that the total length of the tree after each iteration is always less than the previous iteration.

5.3.2 An Example

We will consider the same example as used with Melzak's algorithm. Recall the coordinates of the terminals are: $v_1 = (1, 3), v_2 = (2, 1), v_3 = (5, 4)$ and $v_4 = (4, 0.6)$, as shown in Figure 5.2(a) and the starting topology is: 2 Steiner points; v_1 and v_2 connected to s_1 and v_3 and v_4 connected to s_2, as shown in Figure 5.2(b).

We will start with approximations of the two Steiner points: $s_1^0 = (x_1^0, y_1^0) = (3, 2)$, $s_2^0 = (x_2^0, y_2^0) = (4, 2)$ which gives a starting tree as shown in Figure 5.7. This starting tree has total length $L^0 = \sqrt{(1-3)^2 + (3-2)^2} + \sqrt{(2-3)^2 + (1-2)^2} + \sqrt{(3-4)^2 + (2-2)^2} + \sqrt{(5-4)^2 + (4-2)^2} + \sqrt{(4-4)^2 + (0.6-2)^2} = 8.286$.

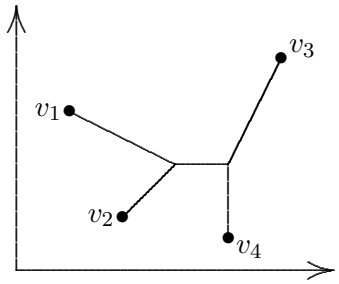


Figure 5.7: Smith's Numerical Algorithm: starting network

ITERATION 1: We start by deducing the $2n-4 = 2\times4-4 = 4$ linear equations. The Steiner point s_1 is adjacent to two terminals v_1 and v_2 and the other Steiner point s_2 so the points $w_j = (u_j, v_j)$ for $s_1^0 = (3, 2)$ are: $w_1 = v_1 = (1, 3)$, $w_2 = v_2 = (1, 3)$ and $w_1 = s_2^0 = (4, 2)$.

$$x_1^1 = \frac{\sum \frac{u_j^1}{\sqrt{[u_j^0 - x_1^0]^2 + [v_j^0 - y_1^0]^2}}}{\sum \frac{1}{\sqrt{[u_j^0 - x_1^0]^2 + [v_j^0 - y_1^0]^2}}} = \frac{\frac{1}{\sqrt{[1-3]^2+[3-2]^2}} + \frac{2}{\sqrt{[2-3]^2+[1-2]^2}} + \frac{x_2^1}{\sqrt{[4-3]^2+[2-2]^2}}}{\frac{1}{\sqrt{[1-3]^2+[3-2]^2}} + \frac{1}{\sqrt{[2-3]^2+[1-2]^2}} + \frac{1}{\sqrt{[4-3]^2+[2-2]^2}}} = 0.864044 + 0.464184x_2^1$$

$$y_1^1 = \frac{\sum \frac{v_j^1}{\sqrt{[u_j^0 - x_1^0]^2 + [v_j^0 - y_1^0]^2}}}{\sum \frac{1}{\sqrt{[u_j^0 - x_1^0]^2 + [v_j^0 - y_1^0]^2}}} = \frac{\frac{3}{\sqrt{5}} + \frac{1}{\sqrt{2}} + \frac{y_2^1}{\sqrt{1}}}{\frac{1}{\sqrt{5}} + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{1}}} = 0.950995 + 0.464184y_2^1$$

The Steiner point s_2 is adjacent to two terminals v_3 and v_4 and the Steiner point s_1 so the points $w_j = (u_j, v_j)$ for $s_2^0 = (4, 2)$ are: $w_1 = v_3 = (5, 4)$, $w_2 = v_4 = (4, 0.6)$ and $w_3 = s_1^0 = (3, 2)$.

$$x_2^1 = \frac{\sum \frac{u_j^1}{\sqrt{[u_j^0 - x_2^0]^2 + [v_j^0 - y_2^0]^2}}}{\sum \frac{1}{\sqrt{[u_j^0 - x_2^0]^2 + [v_j^0 - y_2^0]^2}}} = 2.356333 + 0.462642x_1^1$$

$$y_1^1 = \frac{\sum \frac{v_j^1}{\sqrt{[u_j^0 - x_2^0]^2 + [v_j^0 - y_2^0]^2}}}{\sum \frac{1}{\sqrt{[u_j^0 - x_2^0]^2 + [v_j^0 - y_2^0]^2}}} = 1.025874 + 0.462642 y_1^1$$

This gives us our system of 4 linear equations to solve:

$x_1^1 = 0.864044 + 0.464184 x_2^1$

$y_1^1 = 0.950995 + 0.464184 y_2^1$

$x_2^1 = 2.356333 + 0.462642 x_1^1$

$y_2^1 = 1.025874 + 0.462642 y_1^1$

The solution of which is: $s_1^1 = (x_1^1, y_1^1) = (2.493242, 1.817499)$ and $s_2^1 = (x_2^1, y_2^1) = (3.509812, 1.866725)$. This network is shown in Figure 5.8. Our tree after 1 itertaion has total length $L^1 = 7.838 \leq L^0 = 8.286$.

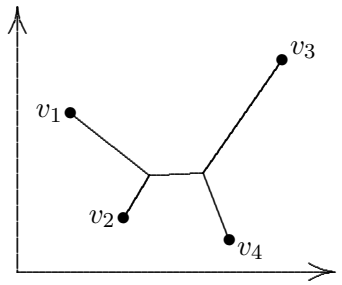


Figure 5.8: Smith's Numerical Algorithm: after 1 iteration

5.3.3 Generalizing to Higher Dimensions

Smith's numerical method does not offer any signficant advantage over the Melzak algorithm in the Euclidean plane. However its advantage is that it can be easily generalized to higher dimensions, whereas the Melzak algorithm cannot.

5.4 The GeoSteiner Algorithm

The fasted algorithm known today which can be used to find solutions to the Euclidean Steiner tree problem is called the *GeoSteiner algorithm*. The difficultly with exact algorithms, which we discussed when considering Melzak's algorithm, is that they are exponential; as the Steiner tree for every possible Steiner topology has to found. This means that normally they can only be used to solve problems for a small number of terminals. What is incredible about the GeoSteiner algorithm is that it can solve problem with up to 2000 terminals; this is amazing as 18 years ago the maximum number of terminals which was feasible to solve for was 29. The algorithm was derived by Warme, Winter and Zachariasen. I will not explain how the algorithm works as it is very involved but Winter and Zachariasen explain the algorithm in their book *Large Euclidean Steiner Minimal Trees in an Hour*, [8].

5.5 Summary

This chapter introduced a number of exact algorithms used to solve the Euclidean Steiner tree problem.

We started by looking at the 3 point algorithm which is used solve the $n = 3$ case of the Euclidean Steiner problem. This then lead us on to consider Melzak's algorithm which was the first ever exact algorithm used to solve the Euclidean Steiner problem. We then looked at Smith's numerical method which is a numerical algorithm for solving the problem and has the advantage that it can be generalized to higher dimensions of Euclidean space. We finished by mentioning the GeoSteiner algorithm which is the best exact algorithm for solving the Euclidean Steiner problem known today.

The algorithms introduced in this chapter are exact algorithms because their object is to find the exact solution to the problem. We have discussed how, due to the number of different possible

Steiner topologies for a given input of terminals, the exact algorithms all run in exponential time. The best exact algorithm (the GeoSteiner algorithm) can solve for up to 2000 terminals but if there is a greater number of terminals to solve for, a different approach needs to be used. The next and penultimate chapter of this report will discuss *approximation algorithms* or *heuristics* which aim to find an approximately good solution to the problem, but not the exact solution.

Chapter 6

Heuristics

This chapter introduces a different approach for solving the Euclidean Steiner problem known as *heuristics* or *approximation algorithms*. A heuristic is an algorithm which aims to get as close to the optimal solution as possible without actually having to reach the optimum solution, in other words it seeks to find an acceptably good solution.

In this chapter we will introduce the most common form of heuristics used to solve the Euclidean Steiner problem which are those that are based on first finding the Euclidean minimum spanning tree and then improving the solution. This chapter starts by introducing the *Steiner ratio* which is a ratio comparing the lengths of the Euclidean minimum spanning tree and the Euclidean Steiner tree for given terminals. We will then go on to look at one of two heurstics which were introduced in 1998 in a paper by Dreyrer and Overton, *Two Heuristics for the Steiner Tree Problem*, [10].

The main references for this chapter are [10, 19, 6].

6.1 The Steiner Ratio

6.1.1 STMs and MSTs

Recall that the *minimum spanning tree* (MST) for n vertices $v_1, ..., v_n$ in the Euclidean plane is the shortest spanning tree interconnecting all vertices v_i using **only** edges $[v_i, v_j]$, whereas the *Steiner minimal tree* (SMT) is the shortest spanning tree interconnecting all vertices v_i using edges connecting both the vertices v_i and also any other points in the Euclidean plane.

Figure 6.1: The STM and MST for an equilateral triangle

If $|E|$ denotes the total length of the edges connecting the set E of edges, then for any input of terminals $|SMT| \leq |MST|$. This is clear because Steiner minimal trees are chosen from a set of trees which includes minimum spanning trees as a subset. We know there exists an algorithm, *Kruskal's algorithm*, which finds the minimum spanning tree of a given set of terminals in time of the order $O(|E|log|V|)$. As we are working with Euclidean minimum spanning trees, if $n = |V|$ is the number of vertices, this simplifies to $O(nlogn)$.

A heursitic for finding the Steiner minimal tree for n terminals is to find the minimal spanning tree. This is clearly much computationally easier as the complexity for finding the minimal spanning tree is $O(nlogn)$ whereas the complexity for finding the Steiner minimal tree is exponential. One important

aspect of a heuritic is that it is fast which we known to be true, another important quality of a good heuristic is that the solution is close to the true optimal solution. We know that $|SMT| \leq |MST|$, but we are interested in how much longer the MST is than the SMT. In other words we want to know the ratio $r = \frac{|SMT|}{|MST|}$.

In Figure 6.1, lets say that the length between each of the three vertices is 1. The the ratio r is computed as follows. The length $|MST| = 1 + 1 = 2$. The length of the $|SMT| = 3x$ where x is the length from the Steiner point to one of the terminals. $cos(30) = \frac{1/2}{x} \Rightarrow x = \frac{1/2}{cos(30)} = \frac{1/2}{\frac{\sqrt{3}}{2}} = \frac{1}{\sqrt{3}}$. Hence $|SMT| = 3x = \sqrt{3}$ and therefore $r = \frac{|SMT|}{|MST|} = \frac{\sqrt{3}}{2}$.

6.1.2 The Steiner Ratio

The *Steiner ratio* (SR) is the largest ratio of length of minimum spanning tree, $|MST|$ to length of Steiner minimal tree, $|SMT|$ for all possible problems. If $|MST(N)|$ denotes the minimum spanning tree for the set of terminals N and $|SMT(N)|$ denotes the Steiner minimal tree for the same set of terminals N for all possible problems. Then we know for all possible N:

$$|SMT(N)| \leq |MST(N)| \tag{6.1}$$

It was conjectured in 1968 by Gilbert and Pollak that the largest ratio occurs for the $n = 3$ case shown above where $|MST|/|SMT| = \sqrt{3}/2$, so therefore $SR = \sqrt{3}/2$. The conjecture is proved in Gibert and Pollak's paper *Steiner minimal trees*, [9]. Hence, $|MST|$ never exceeds $|SMT|$ by more than $\sqrt{3}/2 - 1 \simeq 15.5\%$.

6.2 The Steiner Insertion Algorithm

Dreyrer and Overton introduce two polynomial time heuristics for solving the Euclidean Steiner problem in their paper, *Two Heuristics for the Steiner Tree Problem*, [10]. Their first heuristic, the *Steiner Insertion Algorithm*, is the one we shall look at as it uses the idea of improving the minimal spanning tree. It starts by finding the minimal spanning tree for the terminals and then makes the tree shorter by inserting Steiner points between edges of the minimal spanning tree meeting at edges less than 120°.

The algorithm has a step in it where a *local optimization algorithm* is run. This algorithm finds the optimal positions for the Steiner points given the topology of the tree. The local optimization alogrithm used by Dreyer and Overton for their Steiner Insertion Algorithm is a *primal-dual interior point method for minimizaing a sum of Euclidean vector norms* which is not discussed here but is explained in [7].

Let t_x, t_y, t_z denote termials and let s_n denote an inserted Steiner point then the Steiner Insertion Algorithm is as follows.

Algorithm. 6.1 (The Steiner Insertion Algorithm)**.**

1. *Find the minimal spanning tree.*
2. *FOR each edge connecting fixed points* (t_x, t_y) *DO*
   (a) *Find the edge* (t_y, t_z) *that meets* (t_x, t_y) *at the smallest angle, where* t_z *can be either a fixed point or a Steiner point.*
   (b) *IF this angle is less than* 120° *THEN*
      i. *Place a new Steiner point* s_n *on top of* t_y*.*
      ii. *Remove the edges* (t_x, t_y) *and* (t_y, t_z)*. These edges will no longer be considered for the loop of Step 2.*
      iii. *Add the edges* (t_x, s_n)*,* (t_y, s_n) *and* (t_z, s_n)*.*

(c) Run the local optimization algorithm on the tree with its new toplogy

In order to see how this algorithm works we shall work through a simple case with 4 terminals. We start with the unit square, so we have terminals $t_1 = (0, 0)$, $t_2 = (0, 1)$, $t_3 = (1, 0)$, $t_4 = (1, 1)$. One of the 4 possible MSTs is given by the edges (t_1, t_2), (t_2, t_4), (t_4, t_3) as shown in Figure 6.2(b).

Starting with the edge (t_1, t_2) we see the edge (t_2, t_4) meets it at an angle of 90°. Since 90° is less than 120° an insertion of a Steiner point occurs so the edges (t_1, t_2) and (t_2, t_4) are discarded and replaced with the Steiner point s_1 on top of terminal t_2 and three edges (t_1, s_1), (t_2, s_1) and (t_4, s_1) as shown in Figure 6.2(c). Next the algorithm will insert a Steiner point between edges (t_4, s_1) and (t_4, t_3) and replace edge (t_4, t_3) by (t_3, s_1) as shown in Figure 6.2(d). Finally the local optimization algorithm will optimize this toplogy resulting in the Steiner tree as shown in Figure 6.2(e).

It is important to note here that it is not the Steiner Insertion algorithm which places the Steiner point in their final position. The Steiner Insertion algorithms determines the topology of the SMT and the local optimization algorithm finds the position of the Steiner points. In this example I just chose the positioning of the Steiner points by eye.

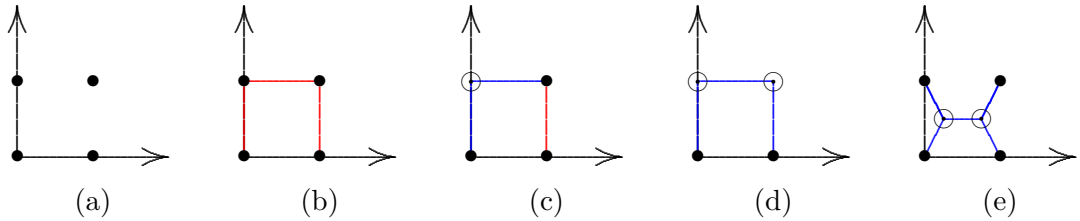


Figure 6.2: The Steiner Insertion Algorithm

6.3 Summary

This chapter introduced heuristics for solving the Euclidean Steiner problem.

We started by discussing how heuristics are different from the exact algorithms discussed in the previous chapter as they aim to find an acceptably good solution in a quicker amount of time instead of the exact solution. We then introduced the idea of the Steiner ratio, which is a ratio comparing the lengths of Euclidean minimum spanning trees and Euclidean Steiner trees. Finally we looked at a heuristic called the Steiner Insertion algorithm which is based on first finding the Euclidean minimum spanning tree and then improving the solution by replacing edges meeting at less than 120° by Steiner points.

The next and final chapter contains a summary of the ideas discussed in this report.

Chapter 7

Conclusion

The Euclidean Steiner problem is a fascinating problem, the study of which draws on ideas from graph theory, computational complexity and geometry as well as optimization. The aim of the Euclidean Steiner problem is to find the tree of minimal length spanning a set of fixed points in the Euclidean plane while allowing the addition of extra (Steiner) points. The Euclidean Steiner tree problem is NP-hard which means there is currently no polytime algorithm for solving it.

The solution of the Euclidean Steiner problem is called a Steiner minimal tree. For every Steiner topology there is a locally optimal solution known as a Steiner tree. The Steiner minimal tree is the globally optimal solution, so finding it requires finding the shortest of all Steiner trees. Many basic properties of Steiner trees are known which helps in the search for them. The main difficultly in solving the problem is that the number of different Steiner topologies increases superexponentially with n, where n is the number of terminals, to the extent that for just $n = 6$ there are thousands of topologies to consider.

Exact algorithms for the Euclidean Steiner problem are algorithms which aim to find the exact solution. The first exact algorithm to be derived was Melzak's algorithm which was published in 1968. Melzak's algorithm finds the Steiner tree for each Steiner topology using geometric construction and then selects the shortest one. Smith's numerical method is similar to Melzak's algorithm in that it finds the Steiner tree for each Steiner topology so that the shortest one can be selected. The advantage of Smith's numerical method over Melzak's algorithm is that it can be easily generalized to higher dimensions whereas Melzak's algorithm cannot. The difficultly with exact algorithms is that they run in exponential time, as the Steiner tree for every possible Steiner topology has to found. Up until 18 years ago the maximum number of terminals which the problem could be solved for was 29. The best exact algorithm today is the GeoSteiner algorithm which can solve the Euclidean Steiner problem with up to 2000 terminals.

A different approach to solving the Euclidean Steiner problem is to use heuristics which are algorithms that aim to get as close to the optimal solution as possible without actually having to reach the optimum solution i.e. they seek to find acceptably good solutions quickly. The simplest heuristic for the Euclidean Steiner problem is to find the Euclidean minimum spanning tree. The Steiner ratio is the largest ratio of minimum spanning tree to Steiner minimal tree and its value is $\frac{\sqrt{3}}{2}$ which means the minimum spanning tree is never more than 15.5% longer than the Steiner minimal tree. More advanced heuristics start by finding the minimum spanning tree and then aim to improve the solution. One heuristic used to solve the Euclidean Steiner problem is the Steiner Insertion algorithm which works by first finding the minimum spanning tree and then replacing edges which make angles of less than 120° by Steiner points.

In recent years the Euclidean Steiner problem has been used in many diverse applications ranging from *VLSI-layout* which is the process of making integrated circuits otherwise known as chips to *phylogentic trees* which are branching diagrams showing inferred evolutionary relationships amongst species. Therefore interest in this problem is sure to continue to rise, motivating many more different algorithms and interesting ways of solving it.

Acknowledgements

I would like to express my thanks to Dr James Blowey for agreeing to take on the supervision of this project and for all his help and advice regarding it.

Bibliography

[1] Melzak Z A. On the problem of steiner. *Canad. Math. Bull*, 4(2), 1961.

[2] Bob Bell. Steiner minimal tree problem. 'http://cse.taylor.edu/~bbell/steiner/', publisher = CSS Senior Seminar Project, year = 1999.

[3] Alexander Bogomolny. The fermat point and generalizations. 'http://www.cut-the-knot.org/Generalization/fermat_point.shtml, year = 2010.

[4] Shimon Even. *Graph algorithms.* Rockville : Computer Science Press, 1979.

[5] Alan Gibbons. *Algorithmic graph theory.* Cambridge University Press, 1985.

[6] Michael Herring. The euclidean steiner tree problem. 'http://denison.edu/mathcs/projects/cs272S04/herring.pdf', publisher = Denison University, year = 2004.

[7] Conn A R & Overton M L. A primal-dual algorithm for minimizing a sum of euclidean norms. *Journal of Computational and Applied Mathematics*, 138(1), 2002.

[8] Winter P & Zachariasen M. *Large Euclidean Steiner Minimal Trees in an Hour.* ISMP, 1998.

[9] Gilbert E N & Pollak N O. Steiner minimal trees. *Siam*, 16(1), 1968.

[10] Derek R. Dreyer & Michael L. Overton. Two heuristics for the steiner tree problem. *Journal of Global Optimization*, 13(1), 1996.

[11] Christos H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[12] Beasley J R. Or - notes. 'http://people.brunel.ac.uk/~mastjjb/jeb/or/graph.html', publisher = Imperial College.

[13] Hans Jurgen Promel & Angelika Steger. *The Steiner Tree Problem - A Tour through Graphs, Algorithms, and Complexity.* Vieweg, 2002.

[14] James A. Storer. *An introduction to Data Structures and Algorithms.* Springer, 2002.

[15] Gil Strang. Calculus: Chapter 13 - partial derivatives. 'ocw.mit.edu/ans7870/textbooks/Strang/Edited/Calculus/13.pdf-2008-08-19, publisher = Massachusetts Institute of Technology.

[16] Professor Luca Trevisan. Computational complexity: Notes for lecture 1. 'http://www.cs.berkeley.edu/~luca/cs278-08/lecture01.pdf', publisher = Berkeley University, year = 2008.

[17] Ingo Wegener. *Complexity theory : exploring the limits of efficient algorithms.* Springer, 2005.

[18] Alexander K. Hartmann & Martin Weigt. *Phase transitions in combinatorial optimization problems : basics, algorithms and statistical mechanics.* Wiley-VCH, 2005.

[19] Frank K. Hwang & Dana S. Richards & Powel Winter. *The Steiner Tree Problem.* North Holland, 1992.