University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Level 4 Project Report Template

Pieter van Tuijl
February 21, 2025

Abstract

Every abstract follows a similar pattern. Motivate; set aims; describe work; explain results.

"XYZ is bad. This project investigated ABC to determine if it was better. ABC used XXX and YYY to implement ZZZ. This is particularly interesting as XXX and YYY have never been used together. It was found that ABC was 20% better than XYZ, though it caused rabies in half of subjects."

Acknowledgements

Education Use Consent

Consent for educational reuse withheld. Do not distribute.

Contents

1 Introduction **1**

1.1 Context 1

1.1.1 Minimum Spanning Tree 1

1.1.2 Steiner Minimal Tree 1

1.2 Aims 2

1.3 Dissertation Outline 2

2 Background **3**

2.1 Historical context 3

2.2 Euclidian Steiner Minimal Tree 5

2.2.1 Formal definition 5

2.2.2 Properties 5

2.2.3 Steiner ratio 6

2.3 Rectilinear Steiner Minimal Tree 7

2.3.1 Formal definition 7

2.3.2 Properties 7

2.3.3 Steiner ratio 7

2.4 Algorithms 7

2.4.1 The Melzak algorithm 8

2.4.2 Complexity analysis for Melzak's algorithm 9

2.4.3 The GeoSteiner algorithm 11

2.5 Existing visualisation tools 11

2.5.1 Steiner-Tree-Visualisation (STV) 11

2.5.2 ESteiner-3D (E3D) 11

2.5.3 Steiner-Tree (ST) 12

3 Analysis/Requirements **13**

3.1 Guidance 13

4 Implementation **14**

4.1 Building for the web 14

4.1.1 Tech stack 14

4.1.2 Application architecture 15

4.2 Algorithm implementations 16

4.2.1 Euclidian Minimum Spanning Tree 16

4.2.2 Euclidean & Rectilinear Steiner Minimal Tree 17

4.3 WebAssembly 17

4.3.1 How WASM works 18

4.3.2 Emscripten 18
4.4 Structure of the codebase 18
4.5 Visualisation/design 18
4.6 Summary 18

5 Evaluation 19
5.1 Guidance 19
5.2 Evidence 19

6 Conclusion 21
6.1 Guidance 21
6.2 Summary 21
6.3 Reflection 21
6.4 Future work 21

Appendices 22

A Appendices 22

Bibliography 23

1 Introduction

1.1 Context

Imagine you are an engineer tasked with designing a network of roads connecting a set of cities. At most two roads may meet in a city and all cities must be connected with the least amount of total distance. This is a classic problem in graph theory, and the solution is known as the minimum spanning tree (MST).

Now, imagine that a new constraint is added to the problem. Junctions (where three or more roads meet) are allowed, provided that they are built outside the cities. You are allowed to use as many such junctions as needed, as long as the total distance of the road network is minimal. The solution to this problem is known as the Steiner minimal tree (SMT).

Algorithms drive much of our digital world. They are used in everything from search engines to social media to self-driving cars. Algorithms operate on data and both the data and the operations can be complex and difficult to visualise without good tools. This is particularly the case for spatial data.

Graph data structures are a common way to represent spatial data, where locations are modelled as nodes and connections between the locations as edges. Trees are a type of graph that connect all nodes in a single, connected network. In other words, for each node in the network, there exists exactly one path to all other nodes (Wikipedia contributors 2025*b*).
Here, we will discuss the visualisation of two types of trees: minimum spanning trees and Steiner minimal trees. Additionally, in this project, we only consider the Euclidian variants of these trees. This means that the Euclidian distance between two nodes is used as the edge weight.

1.1.1 Minimum Spanning Tree

A minimum spanning tree (MST) is a tree that connects all nodes in a network with the least amount of total distance. There exist well-established algorithms for finding the MST of a graph, such as Kruskal's or Prim's algorithm. These algorithms run in polynomial time and work by building up the MST one edge at a time.

1.1.2 Steiner Minimal Tree

A Steiner minimal tree (SMT) is a variant of the MST. Apart from connecting all nodes in the network with minimal total distance, other nodes may be added to potentially further reduce the total distance. These additional nodes are called Steiner points. A SMT with zero Steiner points is equivalent to an MST. In contrast to the MST algorithm, finding the optimal SMT is a NP-hard problem, meaning, there exists no polynomial time algorithm for finding the optimal SMT (more on this later).

TODO: show example of MST vs SMT

1.2 Aims

Many tools exist for visualising minimum spanning trees. However, few tools exist for visualising Steiner minimal trees. This is related to the earlier-mentioned fact that the optimal SMT for a given set of nodes is computationally hard to find. This has historically imposed serious limits on the instance sizes that can be solved on an average computer, thus limiting the practicality and usefulness of SMT visualisation tools. Advances in computer hardware and efficient algorithm implementations, however, have made it possible to solve instances of several orders of magnitude larger than previously possible in reasonable time (Juhl et al. 2018).

The few existing visualisation tools lack flexibility and a user-friendly interface. (see background) We aim to fill this gap by developing a user-friendly interface that allows users to visualise SMTs alongside MSTs for graph instances of arbitrary sizes. These instances can be generated randomly or imported from a file.

The interface should be able to display the MST and SMT of a graph simultaneously, allowing for direct comparison of their structures and total length.
Comparison of length will be helpful for building intuition for the *Steiner Ratio*, which is defined as the least upper bound (supremum) of the ratio between the length of the MST and the SMT. Gilbert and Pollak (1968) conjectured that this ratio is

$$\frac{2}{\sqrt{3}} \approx 1.1547$$

In other words, the *Steiner Ratio* states that the MST is at most 15% longer than the SMT in the worst case.

Lastly, the interface should be able to dynamically update the solution when the user modifies the graph.

1.3 Dissertation Outline

TODO: write at the end of the project

2 Background

2.1 Historical context

The Euclidean Steiner problem has a long history, with roots in the 17th century. The following section will provide a summary of the history as expertly given by Brazil et al. (2014).

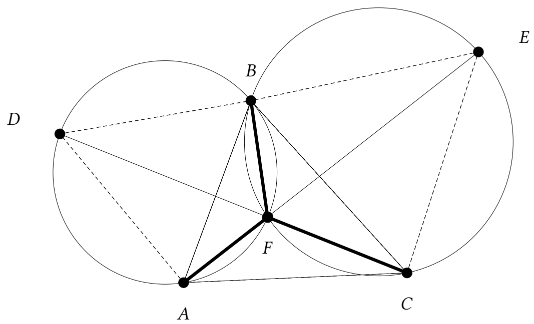
In 1643, Fermat posed the problem of given three points, finding a fourth point such that when connecting the three points to the fourth point, the sum of the distances is minimal. This problem has two cases:

- All the interior angles of the triangle formed by the three points are $\leq 120°$
- One of the interior angles is $\geq 120°$

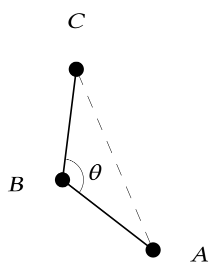
For the first case, Torricelli proposed the following construction. Given $\triangle ABC$, first draw the equilateral triangles $\triangle ABD$ and $\triangle BCE$. Then, for each of these equilateral triangles, draw the circumcircle. The position where the two circles intersect is the point where the sum of the distances is minimal, the so-called *Torricelli point* (point F in Figure 2.1a). To borrow some terminology from Winter and Zachariasen (1997), we will call the 3rd point for each of the equilateral triangles an *equilateral point* denoted as e_{XY} where X and Y are the base points from which the equilateral triangle is constructed. So, in figure 2.1a, $e_{AB} = D$ and $e_{BC} = E$.

For the second case, Cavalieri proved that the the optimal position of the 4th point is just the point with the obtuse angle. (point B in Figure 2.1b)

In 1750, Simpson discovered an alternative construction for the first case, where a straight line is drawn from each of the equilateral points to the opposite vertex. The intersection of these lines coincides with the Torricelli point. These "Simpson lines" are demonstrated in Figure 2.1a. Later, in 1834 Heinen proved that a Simpson line has the same length as the sum of the distances to the Torricelli point.



(a) *Torricelli's and Simpson's constructions for case 1*



(b) *Case 2 where $\theta \geq 120°$*

Figure 2.1: *Solutions to the two cases of Fermat's 3-point problem. 2.1a shows two Simpson lines, EA and DC, intersecting at the Torricelli point F.*

The Fermat-Torricelli problem was later generalised to an n-point problem by Gergonne in 1811. Furthermore, Gergonne was also among the first to consider the case where more than one extra point is allowed. He further discovered that there can exist multiple local minimal solutions for the same set of points, and that all have to be constructed in order to find the global minimum. Lastly, he came up with an iterative, though incomplete, algorithm for finding a local minimum, given a full topology. We will further illustrate and discuss these concepts in 2.4.

Another contribution to the problem was made by Gauss in a letter to Schumacher in 1836. He considered the 4-point problem, and noted that using just one extra point does not result in a minimal tree (compare 2.2a with 2.2b) as in the case of the 3-point problem. He went on to illustrate the problem with the example of connecting German cities by railroad with minimal total length.

> I have on occasion considered the rail- road connection between Harburg, Bremen, Hannover and Braunschweig, and I myself have thought that this problem would be an excellent prize problem for our students

The last sentence of the quote proved correct, because two attempts were later made by Bopp in 1879 and Hoffman in 1890 to come up with a solution. Bopp considered all full topologies and used the constructions developed by Gergonne and Simpson to find the Full Steiner Tree (FST) for each full topology (if it exists). Importantly, he generalised the problem to the n-point Steiner tree problem and enumerated properties of the Steiner trees, which were later rediscovered and proven by Gilbert and Pollak (1968).

Then, in 1934, two Czech mathematicians, Jarník and Kössler, wrote a rigorous treatment on the problem and proved important properties, such as the degree property and the angle property (See 2.2.2). They further proved that even if the problem is extended to higher dimensions, the degree and angle properties still hold.

Another 20th-century contribution worth mentioning is the illustration used by Choquet in 1938, where he compares the Steiner tree problem to a network of cities connected by roads where there are no junctions available between the roads except outside the cities. (Another variant is the case where junctions are allowed inside the cities, which is the minimum spanning tree problem). In their influential book *What is Mathematics?*, Courant and Robbins (1941) later used a similar illustration, which they called the "street network problem". Interestingly, they incorrectly attributed the original Fermat-Torricelli problem to Jakob Steiner. As a consequence, the generalised Fermat-Torricelli problem (i.e. the street network problem) has since been known as the Steiner tree problem in the literature despite the fact that Steiner did not originally consider this generalisation.

Finally, Gilbert and Pollak (1968) provided a comprehensive survey of the Steiner tree problem, and derived many of its properties that have since been the basis for much of the modern literature on the topic. (more on this in 2.2.2) From the sixties onwards, the Steiner tree problem has seen a surge in popularity, partly due to the formalisation of computer algorithms for the related minimum spanning tree problem and partly due to real practical applications, such as minimum-length telephone networks and efficient routing in chips. The latter application was presciently predicted by Hanan (1966) in his paper on the rectilinear variant of the Steiner tree problem.

The tool that has been developed as part of this project supports the visualisation of both the Euclidian and rectilinear Steiner minimal trees. The following section will deal with the formal definitions of these two variants and lists some of their properties.

2.2 Euclidian Steiner Minimal Tree

2.2.1 Formal definition

The Euclidian Steiner minimal tree problem is concerned with finding the shortest tree that contains a set of vertices in the Euclidian plane.

More formally:

- Let N be a finite set of n points in $\mathbb{R}^2$ (Euclidean plane)
- Let V be the set that contains all points in N. ($V \supseteq N$)
- Let $S = V \setminus N$ be the set of additional points called *Steiner points* where $|S| \geq 0$
- Let $T = (V, E)$ be a tree that connects all points in V with exactly $E = |V| - 1$ edges

Then, the Euclidian Steiner minimal tree problem seeks to find T such that $\sum_{e \in E} |e|$ is minimised, where $|e|$ is the Euclidean length of edge $e \in E$ (Brazil et al. 2014).

2.2.2 Properties

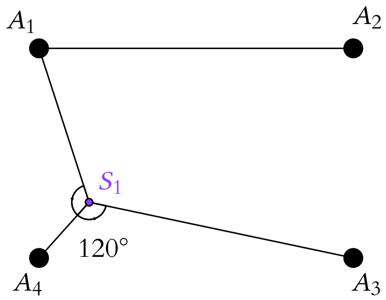
Euclidian Steiner trees have specific properties, which are enumerated in the comprehensive paper by Gilbert and Pollak (1968). We list the ones important for our discussion below and refer the reader to the mentioned paper for other properties.

1. Angle property: every pair of lines meets at $\theta \geq 120°$.
2. Degree property: every Steiner point has exactly 3 incident edges.
3. A Steiner tree with n terminals has at most $s \leq n - 2$ Steiner points. Since a tree has $n - 1$ edges, a Steiner tree with n terminals has at most $2n - 3$ edges.
4. At most one relatively minimal tree exists for a given topology.
5. A Steiner minimal tree is a union of Full Steiner Trees (FSTs). And every non-full Steiner tree can be decomposed into a union of FSTs.

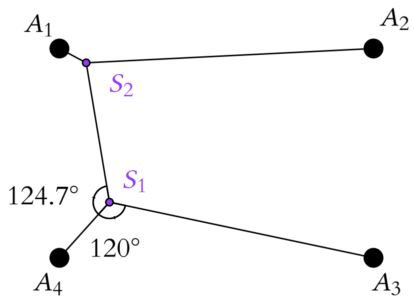
Next, we clarify the following terms, used across the literature and first defined by Gilbert and Pollak (1968):

- A topology is defined as the adjancy matrix which only specifies the connections between points (i.e. how points are interconnected). 2.2d displays three different topologies for the same set of points, where 2.2b and 2.2c share the same topology.
- A full topology has $n - 2$ Steiner points. A Full Steiner Tree (FST) is the relatively minimal tree for a full topology, where the terminals are leaves and the Steiner points are internal nodes with degree 3.
- A relatively minimal tree (RMT) is the minimal tree for a given topology. All figures in 2.2 are RMTs except for 2.2b. A relatively minimal tree is obtained when small pertubations (displacements) of the Steiner points no longer result in a smaller tree.
- A Steiner tree (ST) is always a RMT, but the converse does not hold. For instance, 2.2a is a RMT but the angle $\angle S_1 A_1 A_2 = \theta < 120°$ and therefore does not satisfy the angle property.
- A Steiner minimal tree (SMT) is a Steiner tree and is minimal for all its vertices (i.e. terminals + Steiner points). In other words, the SMT is the minimal tree across all possible topologies for a given set of points. See Figure 2.2d for an example.

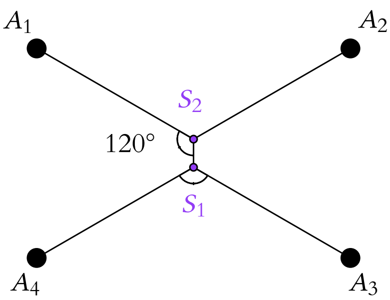
Given a topology, its Steiner tree (if it exists) can be seen as a local minimum. Only by enumerating all possible topologies and their corresponding Steiner trees can the global minimum (i.e. the SMT) be found. This concept forms the basis of the algorithms we will discuss in the algorithms section.



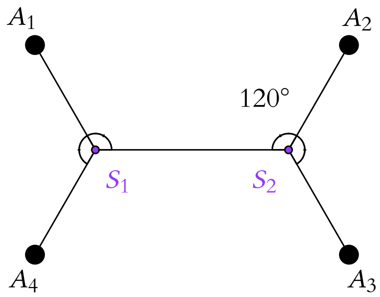
***(a)** Relatively minimal tree for its topology*



***(b)** Splitting at A_1 to add a Steiner point*



***(c)** Steiner tree obtained from 2.2b by perturbing S_2 until tree length is minimal for its topology*



***(d)** Steiner minimal tree for all its vertices*

***Figure 2.2:** Figures above illustrate how adding and perturbing Steiner points results in increasingly smaller trees. The sub figures a...d are ordered in descending order of tree length. It further shows how the minimal tree for a given topology is not necessarily a Steiner tree, and that a Steiner tree, though minimal for its topology, is not necessarily minimal for all its vertices. b, c, and d are examples of full topologies since they have $n - 2$ Steiner points.*

2.2.3 Steiner ratio

An ESMT without Steiner points is equivalent to the Euclidian minimum spanning tree (EMST). Formally, the EMST is defined as the network that connects a set of points N, such that the sum of the edge weights is minimal, where the edge weights are the Euclidean distances between the points.

SMTs are shorter or at worst equal in length to MSTs. The ratio between the length of the SMT and the MST has received much attention in the literature. Gilbert and Pollak (1968) famously conjectured that the lowest possible ratio is $\frac{\sqrt{3}}{2} \approx 0.866025\ldots$ for any set of points.

$$\text{Steiner ratio} = \frac{L_{ESMT}}{L_{EMST}} = \frac{\sqrt{3}}{2}$$

Du and Hwang (1992) submitted a proof for this conjecture which was later shown to contain a flaw which invalidated the proof (Innami et al. 2010). Hence, the Steiner ratio conjecture remains open to this day. Nevertheless, attempts have been made to find proofs where n is bounded. For example, Kirszenblat (2014) submitted a proof for the case where $n \leq 8$.

2.3 Rectilinear Steiner Minimal Tree

2.3.1 Formal definition

The Rectilinear Steiner Minimal Tree (RSMT) problem is similar to the ESMT problem in that by allowing extra Steiner points, it seeks to find the minimal tree for a given set of points, except that the rectilinear distance, also known as the Manhattan distance, is used. The rectilinear distance between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is defined as

$$|x_1 - x_2| + |y_1 - y_2|.$$

2.3.2 Properties

Marcus Brazil (2015) provides a list of properties of rectilinear Steiner trees including:

- Degree property: every Steiner point s has exactly 3 or 4 incident edges. A Steiner point of degree 3 is called a *T-point*, whilst a Steiner point of degree 4 is called a *cross*.
- Every edge has at most 1 horizontal and 1 vertical segment. An edge with exactly 1 horizontal and 1 vertical component is called a *bent edge* whilst an edge with only 1 horizontal component or 1 vertical component is called a *straight edge*.
- If s is a T-point, at most one of its incident edges is a bent edge. If s is a cross, then all of its incident edges are straight edges.

2.3.3 Steiner ratio

As before, a RSMT without Steiner points is equivalent to a rectilinear minimum spanning tree (RMST). The definition of the RMST is similar to the definition of the ESMT in 2.2.3 except that the rectilinear metric is used.

The Steiner ratio in the rectilinear case is the smallest ratio between the length of the RSMT and the RMST. It has been proven to be exactly 2/3 Marcus Brazil (2015).

$$\text{Steiner ratio} = \frac{L_{RSMT}}{L_{RMST}} = \frac{2}{3}$$

In other words, the rectilinear minimum spanning tree is at most 1.5 times the length of the rectilinear Steiner minimal tree.

2.4 Algorithms

The Steiner tree problem is easy to understand but hard to actually solve. Garey et al. (1977) proved that the Euclidian and Rectilinear Steiner tree problems are NP-hard. This means that there exist no polynomial-time algorithms for these problems (unless P = NP). However, approximation schemes can be used to find near-optimal solutions in a reasonable amount of time. In this project, we have chosen to use exact algorithms in order to demonstrate, that despite the theoretical hardness of the problem, the actual performance in practise is suprisingly good.

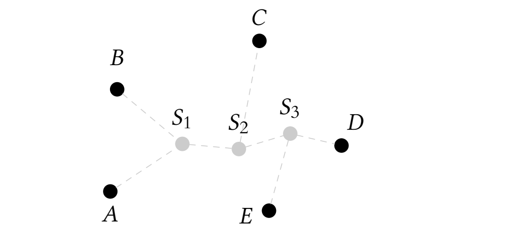
There exists two major exact algorithms for the Steiner tree problem:

- The Melzak algorithm Melzak (1961) which is largely based on the iterative algorithm by Gergonne (Brazil et al. 2014) (see 2.1). This algorithm is essentially a brute-force approach and is therefore not practical for large instances.
- The GeoSteiner algorithm developed by Winter and Zachariasen (1997). This algorithm uses ideas from the Melzak algorithm, but it is much faster and efficient due to the use of geometric properties for pruning the search space, efficient data structures, and other clever optimisations.

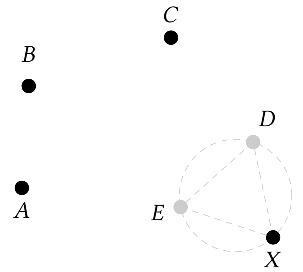
2.4.1 The Melzak algorithm

As mentioned at the end of 2.2.2 there can exists local minimal solutions for the Steiner tree problem. Hence, in order to find the global minimal solution, we need to consider all possible full (Steiner) topologies. The Melzak algorithm does not explicitly deal with this, but a full topology as inputand finds the FST if it exists.

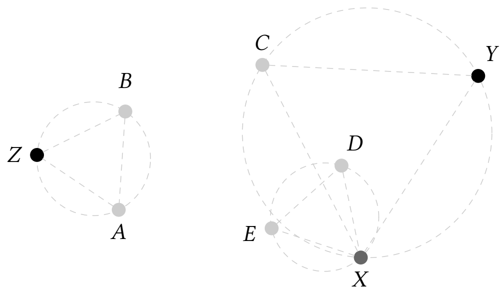
Before illustrating the algorithm with a practical example, we use the following definition from (Marcus Brazil 2015): Let $\mathcal{T}_n$ be a full topology with n points and $n-2$ Steiner points. Let S_i be a Steiner point in $\mathcal{T}_n$ which is adjacent (connected by an edge) to A and B. $\overline{AB}$ shall denote the segment between A and B. Let X be the 3rd (equilateral) point of the equilateral triangle $\triangle ABX$.



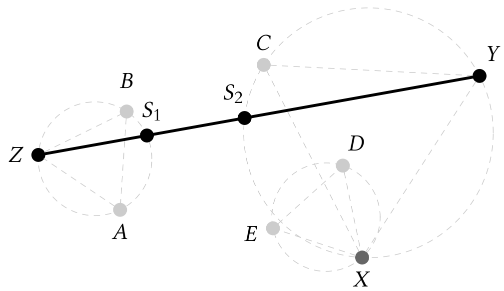
(a) Initial full topology, Steiner points are randomly placed for visualisation purposes



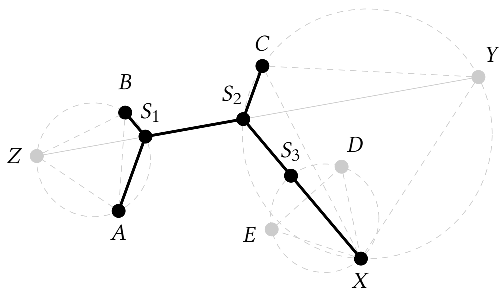
(b) Start of merging. Replacing (E, D, S_3) with X



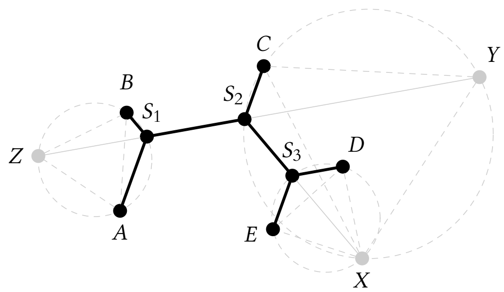
(c) Replacing (X, C, S_2) with Y, and (A, B, S_1) with Z, results in the $n = 2$ base case.



(d) Start of reconstruction. The FST is the segment between Z and Y. Positions for S_1 and S_2 are now found.



(e) FST for $\mathcal{T}_4$



(f) Final FST for $\mathcal{T}_5$

Figure 2.3: *The Melzak algorithm to find the FST for a full topology $\mathcal{T}_5$ (with $n = 5$)*

Assume that S_i is positioned on the opposite side of $\overline{AB}$ to X and that it lies on the arc $\overset{\frown}{AB}$ (Steiner arc) of the circumcircle of the equilateral triangle $\triangle ABX$, subtended by $\overline{AB}$. The Simpson segment starts at X and passes through S_i to some other point v. By definition, the length of the Simpson segment between X and S_i is $|S_iA| + |S_iB|$. Hence, A, B, and S_i can be replaced by X with a direct line segment to v, without increasing the tree length. This process of *merging* is repeated until the base case, $\mathcal{T}_2$, is reached.

Next, the FST can be *reconstructed* by backtracking the merging steps. Suppose the FST of $\mathcal{T}_{n-1}$ is defined. This means that the location of v also will be known. There exists an FST for $\mathcal{T}_n \iff$ there exists an FST for $\mathcal{T}_{n-1}$ and the Simpson segment between v and X intersects the Steiner arc $\widehat{AB}$. S_i is positioned at the point this intersection and and the FST is constructed by deleting the Simpson segment and adding the edges AS_i, BS_i, and vS_i.

The algorithm is visualised in figure 2.3. Figure 2.3a shows an instance of the full topology $\mathcal{T}_5$ (with $n = 5$). Please note that the Steiner points are randomly (and incorrectly) positioned, in order to visualise all the connections.

First, E, D and S_3 are replaced by X which reduces the topology by 1 terminal and 1 Steiner point (figure 2.3b). It is not yet possible to draw a Simpson segment between X and S_2 since the location of S_2 is unknown. In the next iteration, C, X and S_2 are replaced by Y which reduces the topology to size $n = 3$. As the location of S_1 is also yet unknown, A, B and S_1 are replaced by Z, which reduces the topology to the base case (figure 2.3c).

The FST is subsequently built up by backtracking the merging steps. By drawing a Simpson segment $\overline{ZY}$, the location of S_1 and S_2 are found (figure 2.3d), after which $\overline{ZY}$ is deleted and the edges AS_1, BS_1, S_1S_2, CS_2, with the Simpson segment $\overline{XS_2}$ are added (figure 2.3e). Subsequently, the position of S_3 is the intersection between $\overline{XS_2}$ and the Steiner arc $\widehat{ED}$. Finally, $\overline{XS_2}$ is deleted and the edges S_2S_4, ES_3, and DS_3 are added (figure 2.3f).

2.4.2 Complexity analysis for Melzak's algorithm

In figure 2.3, we have chosen specific sides for the equilateral points $X = e_{DE}$, $Y = e_{CX}$, and $Z = e_{AB}$. The original Melzak algorithm, however, accounts for the fact that each equilateral point e_{xy} can be placed on either side of the segment $\overline{xy}$. Hence, the algorithm has a worst-case complexity of $O(2^n)$. An improved, $O(n)$ version of the algorithm was proposed by Hwang (1986), which picks the correct side for each equilateral point e_{xy} in constant time.

In the example in figure 2.3, the final FST happens to also be the SMT. However, this is not always the case. From property 5, we know the the SMT is a union of FSTs, hence the SMT can be found by enumerating all possible full topologies, applying the Melzak algorithm to each, and taking the union of the resulting FSTs such that the length of minimal. Winter and Zachariasen (1997) state that the total number of full topologies for a set of n points is given by

$$F^*(n) = \sum_{k=2}^{n} \binom{n}{k} \frac{(2k-4)!}{2^{k-2}(k-2)!}$$

We will now show a derivation of this formula based on Property 7 in Gilbert and Pollak (1968).

Let $f(s)$ be the number of full topologies for the case of s Steiner points. Let $F(n, s)$ be the number of (non-full) topologies for the case of n terminals and s Steiner points. $F(n, s)$ can be written in terms of $f(s)$ as follows.

If we assume that Steiner points are unlabelled, there are $\binom{n}{s+2}$ ways to choose the terminals from which the full topology will be constructed. For each chosen set of terminals, there are $f(s)$ possible full topologies. Hence, there are a total of$\binom{n}{s+2}f(s)$ possible full topologies.

There are still $n-(s+2)$ points left, though, which can be added to the currently full topology to make it non-full. A tree has $n-1$ edges, hence the current full topology has $((s+2)+s)-1 = 2s+1$

edges. The leftover points can be added by splitting existing edges into two, resulting in the net addition of 1 edge.

For the first leftover point, we have $2s + 1$ possible edges to split, for the second $2s + 2$, and so on. This is repeated until the last leftover point is added, at which point our topology has $n + s - 1$ edges. This can be written as

$$\begin{aligned} F(n, s) &= \binom{n}{s+2} f(s)(2s+1)(2s+2)\cdots(n+s-2) \\ &= \binom{n}{s+2} f(s) \frac{(n+s-2)!}{(2s)!} \end{aligned}$$

The formula for $f(s)$ can be defined recursively as follows:

$$\begin{aligned} f(0) &= 1 \\ f(s+1) &= (2s+1)f(s) \end{aligned}$$

In other words, when all $f(s)$ full topologies are found, adding one more Steiner point by the same edge-splitting technique to each of the $f(s)$ topologies, results in $(2s+1)f(s)$ new topologies. This recurrence relation can be solved by using the double-factorial form for an odd integer $n = 2k - 1$, with $k \geq 0$ (Wikipedia contributors 2025*a*).

$$(2k-1)!! = \frac{(2k)!}{2^k k!}$$

Rewriting $f(s)$ in terms of the double-factorial, we get

$$f(s) = \frac{(2s)!}{2^s (s)!}$$

Finally,

$$\begin{aligned} F(n, s) &= \binom{n}{s+2} \frac{\cancel{(2s)!}}{2^s (s)!} \frac{(n+s-2)!}{\cancel{(2s)!}} \\ &= \binom{n}{s+2} \frac{(n+s-2)!}{2^s (s)!} \end{aligned}$$

In the case of the Melzak algorithm, only full topologies are considered. We therefore let $n = k$ and $s = k - 2$, and rewrite $F(n, s)$ to $F(n)$.

$$F(n) = \binom{n}{k} \frac{(2k-4)!}{2^{k-2}(k-2)!}$$

Since the full topologies are considered for all subsets of n terminals, starting from 2 terminals as the minimum, we arrive at the original expression.

$$F^*(n) = \sum_{k=2}^{n} \binom{n}{k} \frac{(2k-4)!}{2^{k-2}(k-2)!}$$

The reason for showing this derivation is that it gives some insight into the complexity of the problem. $F^*(n)$ is huge even for small values of n. Due to the factorial in the numerator, it can be said to grow *super-exponentially* in k (i.e. faster than 2^k). Even though the fast GeoSteiner algorithm is not super-exponential, it is still exponential in the number of FSTs as we will see in the next section. This shows that the exponentiality is not merely due to Melzak's algorithm, but inherent in the Steiner problem itself Marcus Brazil (2015).

2.4.3 The GeoSteiner algorithm

The GeoSteiner algorithm has been the fastest algorithm for computing the exact Steiner minimal tree for multiple decades. It consists of the same two steps as the brute-force algorithm above.

1. Enumerate all FSTs (i.e. *FST generation*)
2. Find the subset of FSTs that form the SMT (i.e. *concatenation*).

However, instead of enumerating all possible full topologies for all subsets of n terminals, it only enumerates the equilateral points that are used in the construction of the FSTs (Winter and Zachariasen 1997). Equilateral points are often repeated across different full topologies of subsets, so this avoids large redundancies. Secondly, using clever *pruning tests*, many equilateral points are discarded at an early stage of the generation stage, further reducing the number of FSTs that need to be considered in the end. The reader is referred to Winter and Zachariasen (1997) for more details on these pruning tests. As a result, the number of FSTs generated in the first stage are linear in n. Juhl et al. (2018) note that on average, the number of generated FSTs is $2.5n$ with only $0.6n$ left after pruning.

The process itself of generating and pruning is quadratic in n, but the last stage of concatenation is a major bottleneck. During concatenation, the algorithm has to check every possible subset of FSTs to see which one has the minimal length, which has exponential complexity in the number of FSTs. Hence, it is critical that as few FSTs as possible are generated in the first stage.

The discussion on the GeoSteiner algorithm above has so far assumed the Euclidian version of the problem. It is noteworthy, however, that the algorithm has been extended to the rectilinear case as well (Salowe and Warme 1995). Interestingly, the generation phase is very fast but the concatenation phase is much slower compared to the Euclidian variant ((Juhl et al. 2018)).

2.5 Existing visualisation tools

During the research phase, we found three related projects on Github whose features and limitations will be discussed in this section.

2.5.1 Steiner-Tree-Visualisation (STV)

STV is a Python-based GUI tool developed by Keydrain (2015). It provides a simple interface for visualising Euclidian MSTs and SMTs, displaying their lengths and allowing for direct comparison. However, the tool is limited by a few factors.

- First, a brute-force approximation algorithm is used to find the SMT, and despite the $O(n^4 \log(n))$ complexity, the tool freezes for instances larger than 40 nodes.
- Additionally, the GUI is not very flexible. For example, you cannot overlay the MST and SMT simultaneously and the canvas does not support zooming or resizing. It is also not possible to import a graph from an external file or export the results.
- Lastly, the tool does not work out of the box and requires code patching to run in modern Python environments.

2.5.2 ESteiner-3D (E3D)

E3D is another Python-based tool developed by Abd (2024). It is a program that can be used to find the Euclidian SMT of a graph and supports 2D and 3D graphs. However, use of the software is limited by the fact that no graphical nor command-line interface is provided. It also does not support the simultaneous visualisation of MSTs and SMTs and the visualisation of the Steiner ratio.

2.5.3 Steiner-Tree (ST)

ST is a Javascript-based tool developed by Dawkey (2019). It is a web page that can be used to find the Rectilinear SMT of a graph. Although the tool has a nice interface and uses an interactive canvas, it does not support visualisation of the MST or SMT in the Euclidian plane. And even for the rectilinear case, it is only possible to place the nodes on grid lines instead of anywhere on the canvas. This makes it possible to use a polynomial-time algorithm, but it makes the tool unsuitable for imported graph instances. Exporting the visualisation results is also not possible.

Overall, we have found that the existing tools are limited by a lack of smooth, user-friendly interfaces, visualisation options, flexible import/export capabilities, and platform independence.

3 Analysis/Requirements

What is the problem that you want to solve, and how did you arrive at it?

3.1 Guidance

Make it clear how you derived the constrained form of your problem via a clear and logical process.

The analysis chapter explains the process by which you arrive at a concrete design. In software engineering projects, this will include a statement of the requirement capture process and the derived requirements.

In research projects, it will involve developing a design drawing on the work established in the background, and stating how the space of possible projects was sensibly narrowed down to what you have done.

4 Implementation

Having given a high-level design of the proposed solution in the previous chapter, this chapter will discuss the practicalities of our implementation and address topics such as:

- Why we chose to build a web app over a native app.
- What tech stack we used and why.
- What the big picture architecture is and how different parts are connected.
- How the codebase is structured, in order to support extensibility and maintainability.
- How we implemented the algorithms and integrated them with the UI, including a in-depth look at the use of WebAssembly (WASM).

4.1 Building for the web

The first implementation decision was to build a web app instead of a native app. As stated in the problem description, our aim was to make a visually-appealing and accessible tool that does not require technical expertise or custom tooling for use. Browsers are ubiquitous nowadays and provide a platform-independent way to run and distribute applications. Updates can easily be deployed without the need for users to install them manually. Futhermore, differences between browser vendors have become less pronounced due to the development of standards and the rise of frameworks that abstract away some of the most jarring differences.

4.1.1 Tech stack

The application is built using Typescript[1] (TS) as the main programming language, which is a statically-typed superset of Javascript (JS) that compiles to JS (Bierman et al. 2014). JS, the web's main language, is dynamically typed and has some loose features that can make it prone to bugs. For example, invalid references are not detected until runtime, in contrast to TS which adds compile-time type checking. Empirical evidence has shown that TS is effective at detecting bugs and improving code maintainability (Gao et al. 2017).

Interactive web applications often rely on something called DOM manipulation. This is the process of mutating HTML elements in the DOM, the tree of HTML elements that make up the UI. DOM manipulation is costly and is a major performance bottleneck for web apps. We have used React[2] (which makes use of a virtual DOM) to build an interactive UI whilst still ensuring that the real DOM gets mutated as little as possible. Moreover, React code is declarative (i.e. we tell what the UI should look like, not how to achieve it) and therefore easier to reason about and maintain.

Other dependencies include:

- Shadcn/ui[3] for base UI components, such as buttons, inputs, dropdowns, etc.

[1] https://www.typescriptlang.org/
[2] https://react.dev/
[3] https://ui.shadcn.com/

- Sigma.js[4] for drawing graphs on the canvas.
- Graphology[5] for the graph data model

4.1.2 Application architecture

The application has been developed as a client-only application, with no need for a backend API. Figure 4.1 shows the various high-level components of the application. The top-layer (i.e. the frontend) is divided into three main parts:

- The views part, which handles user interaction (controls) and the visual appearance
- The algorithms part, which handles the computation of the algorithms supported by the application.
- The graph datastructure as a global state, which can be read and mutated by the views and algorithms.

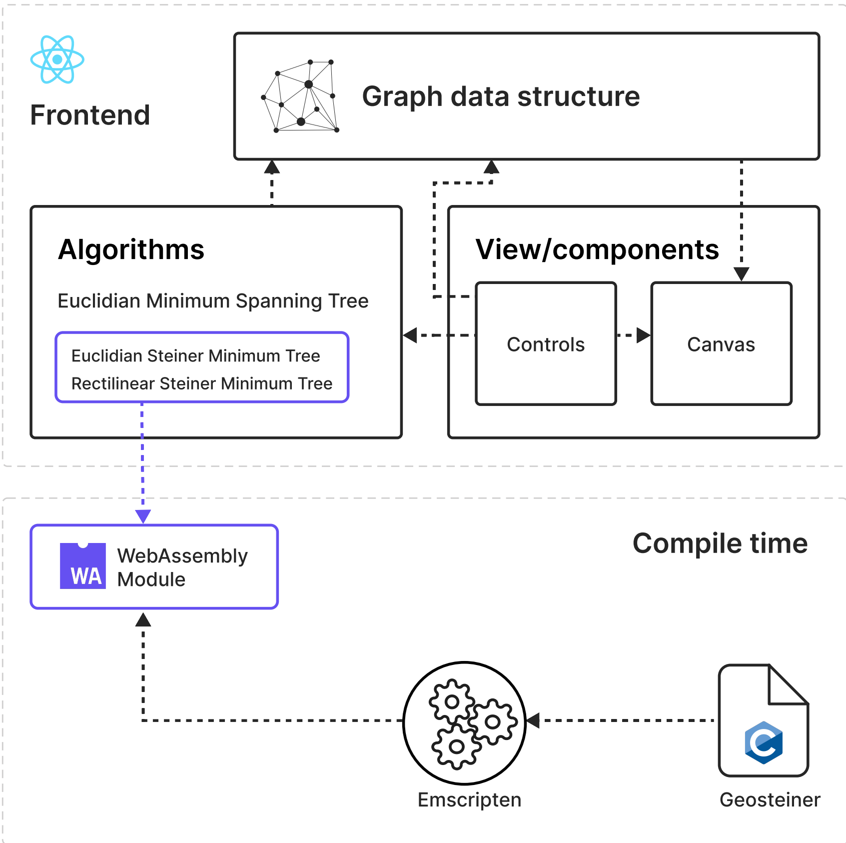


Figure 4.1: Architecture of the application. The arrows indicate the flow of data and control.

The controls allow the user to modify the graph instance, either directly by adding a new graph or by editing the existing one, or indirectly by triggering the computation of an algorithm. Updates to the graph datastructure are reflected in the the canvas component.

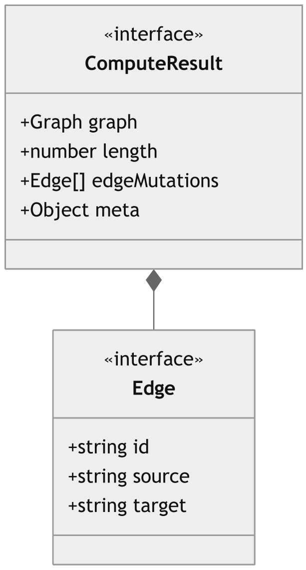
The lower layer contains the WebAssembly module that contains the WASM byte code of the compiled Geosteiner C-library, which is used for computing the Euclidian and Rectilinear variants of the Steiner minimal tree. The WebAssembly module is dynamically imported by the frontend during runtime. We discuss the use of WebAssembly in more detail in section 4.3.

[4]https://sigmajs.org/
[5]https://graphology.github.io/

4.2 Algorithm implementations

Each algorithm is implemented as an isolated, pure function that takes an undirected graph as input and returns a computation. Figure 4.2 specifies the interface of the computational output. The tree is returned as new graph instance, alongside the length of the tree. The list of edge mutations keeps track of all the edges that were added to the tree in order of insertion. Some algorithms, such as the ESMT and the RSMT also pass additional metadata, such as the list of Steiner nodes. By coding to an interface as opposed to a concrete implementation, algorithm implementations can be swapped out and new algorithms can be added easily.

Figure 4.2: Interface of the algorithm computational output.



The output from the algorithm is used to update the global graph datastructure by a *merge* operation. We can't directly mutate the global graph because of limitations in Sigma.js (the library used for canvas drawing). Merging the tree with the existing graph is done by adding all the tree edges and nodes into the existing graph. If an edge or node already exists, its attributes are updated to indicate that it is part of the algorithm-generated tree. By keeping track of which edges/nodes are generated by which algorithm, we can toggle the visibility of each tree separately whilst still using the same graph instance.

4.2.1 Euclidian Minimum Spanning Tree

The EMST is implemented using Prim's algorithm (Skiena 2008, pp. 194–195). Since the algorithm operates in Euclidian space, the input graph is first preprocessed to be a complete undirected graph. See pseudocode 1 for the implementation.

```
Input: Graph G = (V, E)
Output: Tree T = (V', E')
initialize tree T, parent vector p, distance vector d;
Set v to startNode;
while v not in T do
    E = v.edges();
    foreach edge e ∈ E do
        u ← edge.target();
        w ← edge.weight();
        if distance[u] > w and u not in T then
            distance[u] = w;
            parent[u] = v;
        end
    end
    fringe = V \ V';
    v = node u in fringe with smallest distance to v;
    T.addNode(v);
    T.addEdge(p[v], v);
end
return T;
```

Algorithm 1: Prim's $O(n^2)$ algorithm for the EMST

4.2.2 Euclidean & Rectilinear Steiner Minimal Tree

For the ESMT and RSMT, we make use of the Geosteiner[6] C-library implementation. As highlighted in the background section, efficient exact SMT algorithms are hard to implement and the GeoSteiner library is a well-established library that makes use of many heuristics and clever techniques to find the exact solution in a reasonable time. From the frontend perspective therefore, we treat the library as a black box and only care about the input and the output, whilst leaving the implementation details to the library.

We have implemented a wrapper around the library that provides a simplified interface, only exposing the functions that are needed by the frontend. Listing 4.1 shows the wrapper functions that are exposed to the frontend.

```
void calc_esmt(int n, double *terms, double *length, int *nsps, double *sps, int
    *nedges, int *edges);

void calc_rsmt(int n, double *terms, double *length, int *nsps, double *sps, int
    *nedges, int *edges);
```

Listing 4.1: *Wrapper functions for the Geosteiner library*

The browser, however, cannot execute C code directly. Therefore, we need to compile the C code to WebAssembly (WASM) byte code. We do this using the Emscripten[7] toolchain. We discuss the details of the compilation process and integration of the WASM module with the frontend in section 4.3.

With WASM, we have to explicitly manage memory allocation and deallocation. From the properties of the Steiner minimum tree, we know that the maximum number of Steiner points is $n-2$ and the maximum number of edges is $2n-3$. Using these constraints, we pre-allocate the memory for the Steiner points and edges and pass the pointers to the wrapper functions defined above.

4.3 WebAssembly

Previously, Javascript was the only language that could run in the browser. This meant that native code could not be executed and always had to be manually ported to JS. This limitation let to attempts to compile native code to JS, such as ASM.js. It was found that ASM.js was more performant than code that had been manually rewritten in JS (Nyaga 2025). In 2015, WebAssembly[8] (WASM) was introduced as new compilation target for the browser. WASM is a binary instruction format that is more performant than JS and can be used to run native code at near native speeds. WASM is supported by most browsers, with the exception of Internet Explorer and Opera Mini [9].

WASM is designed to be portable (i.e. write once, run everywhere), similar to C in a sense. Code compiled to WASM can run on any platform, both web and off-web (e.g. Node.js). The current application is a client-only application, but if we were to decide to run the Geosteiner algorithms on a Node.js backend, WASM would allow us to do so seamlessly.

[6]http://www.geosteiner.com/
[7]https://emscripten.org/
[8]https://webassembly.org/
[9]https://caniuse.com/wasm

4.3.1 How WASM works

The browser features a virtual machine (VM) that can execute JS and WASM binary code. A WASM binary is called a module and can be loaded into the VM by calling the WebAssembly Browser APIs from a JS script. In order for JS to call WASM functions, the WASM module must explicitly export these functions. WASM itself only supports 4 basic data types: 32/64 bit ints/floats. This means that we can't pass complex data structures to a WASM function or return such data from it. Instead, we can utilise the shared contiguous memory space between JS and WASM to pass pointers to memory locations that contain the data.

4.3.2 Emscripten

As highlighted in the figure 4.1, the Geosteiner library is compiled to WASM using the Emscripten toolchain. The Emscripten toolchain consists of a designated compiler frontend (emcc) and LLVM backend optimised for the generation of WASM binaries. Apart from the compiler, the toolchain also exposes a set of "glue code" that aims to simplify the boilerplate code required to both load the WASM module and interface between the WASM module and JS code (Selvatici 2018).

The glue code also provides an ergonomic memory management API. We can 'view' the same memory buffer in different ways depending on the data type, such as an array of 32-bit integers or 64-bit floats. This avoids the need to manually handle different offsets and strides when accessing the data.

The output from the Empscripten compiler is a .wasm file and a .js file. The .js file is imported by the frontend and contains the glue code and is used to asynchronously load and instantiate the WASM module.

4.4 Structure of the codebase

4.5 Visualisation/design

4.6 Summary

5 Evaluation

How good is your solution? How well did you solve the general problem, and what evidence do you have to support that?

5.1 Guidance

- Ask specific questions that address the general problem.
- Answer them with precise evidence (graphs, numbers, statistical analysis, qualitative analysis).
- Be fair and be scientific.
- The key thing is to show that you know how to evaluate your work, not that your work is the most amazing product ever.

5.2 Evidence

Make sure you present your evidence well. Use appropriate visualisations, reporting techniques and statistical analysis, as appropriate. The point is not to dump all the data you have but to present an argument well supported by evidence gathered.

If you use numerical evidence, specify reasonable numbers of significant digits; don't state "18.41141% of users were successful" if you only had 20 users. If you average *anything*, present both a measure of central tendency (e.g. mean, median) *and* a measure of spread (e.g. standard deviation, min/max, interquartile range).

You can use `siunitx` to define units, space numbers neatly, and set the precision for the whole LaTeX document.

For example, these numbers will appear with two decimal places: 3.14, 2.72, and this one will appear with reasonable spacing 1 000 000.00.

If you use statistical procedures, make sure you understand the process you are using, and that you check the required assumptions hold in your case.

If you visualise, follow the basic rules, as illustrated in Figure 5.1:

- Label everything correctly (axis, title, units).
- Caption thoroughly.
- Reference in text.
- **Include appropriate display of uncertainty (e.g. error bars, Box plot)**
- Minimize clutter.

See the file `guide_to_visualising.pdf` for further information and guidance.

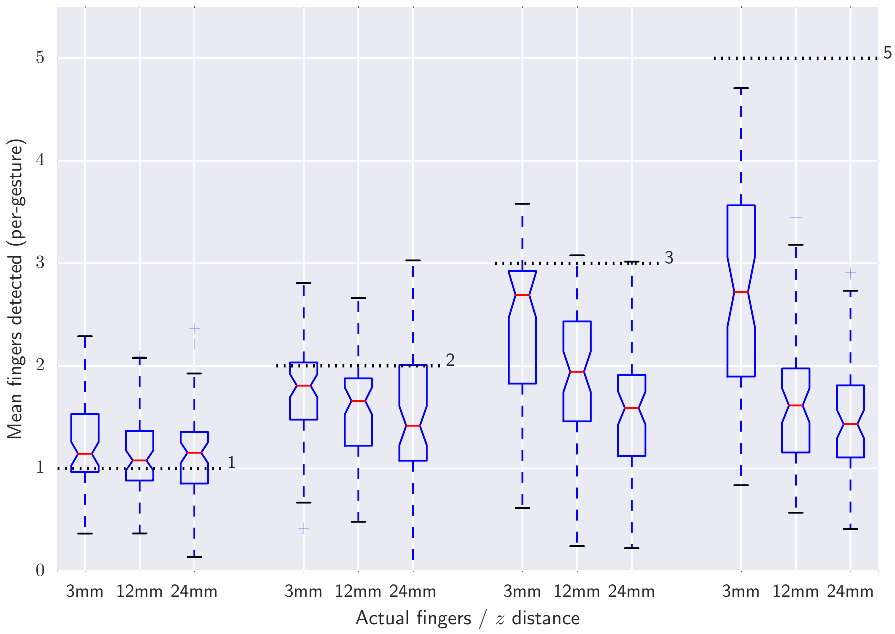


Figure 5.1: *Average number of fingers detected by the touch sensor at different heights above the surface, averaged over all gestures. Dashed lines indicate the true number of fingers present. The Box plots include bootstrapped uncertainty notches for the median. It is clear that the device is biased toward undercounting fingers, particularly at higher z distances.*

6 Conclusion

Summarise the whole project for a lazy reader who didn't read the rest (e.g. a prize-awarding committee). This chapter should be short in most dissertations; maybe one to three pages.

6.1 Guidance

- Summarise briefly and fairly.
- You should be addressing the general problem you introduced in the Introduction.
- Include summary of concrete results ("the new compiler ran 2x faster")
- Indicate what future work could be done, but remember: **you won't get credit for things you haven't done**.

6.2 Summary

Summarise what you did; answer the general questions you asked in the introduction. What did you achieve? Briefly describe what was built and summarise the evaluation results.

6.3 Reflection

Discuss what went well and what didn't and how you would do things differently if you did this project again.

6.4 Future work

Discuss what you would do if you could take this further – where would the interesting directions to go next be? (e.g. you got another year to work on it, or you started a company to work on this, or you pursued a PhD on this topic)

A Appendices

Use separate appendix chapters for groups of ancillary material that support your dissertation. Typical inclusions in the appendices are:

- Copies of ethics approvals (you must include these if you needed to get them)
- Copies of questionnaires etc. used to gather data from subjects. Don't include voluminous data logs; instead submit these electronically alongside your source code.
- Extensive tables or figures that are too bulky to fit in the main body of the report, particularly ones that are repetitive and summarised in the body.
- Outline of the source code (e.g. directory structure), or other architecture documentation like class diagrams.
- User manuals, and any guides to starting/running the software. Your equivalent of `readme.md` should be included.

Don't include your source code in the appendices. It will be submitted separately.

Bibliography

Abd, K. (2024), 'Esteiner-3d'.
URL: `https://github.com/Kallel-Abd/ESteiner-3D/tree/main`

Bierman, G., Abadi, M. and Torgersen, M. (2014), Understanding typescript, *in* 'European Conference on Object-Oriented Programming', Springer, pp. 257–281.

Brazil, M., Graham, R. L., Thomas, D. A. and Zachariasen, M. (2014), 'On the history of the euclidean steiner tree problem', *Archive for History of Exact Sciences* **68**(3), 327–354.
URL: *https://doi.org/10.1007/s00407-013-0127-z*

Dawkey (2019), 'Steiner-tree'.
URL: `https://github.com/Dawkey/Steiner-Tree/`

Du, D.-Z. and Hwang, F. K. (1992), 'A proof of the gilbert-pollak conjecture on the steiner ratio', *Algorithmica* **7**(1), 121–135.
URL: *https://doi.org/10.1007/BF01758755*

Gao, Z., Bird, C. and Barr, E. T. (2017), To type or not to type: Quantifying detectable bugs in javascript, *in* '2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)', pp. 758–769.

Garey, M. R., Graham, R. L. and Johnson, D. S. (1977), 'The complexity of computing steiner minimal trees', *SIAM Journal on Applied Mathematics* **32**(4), 835–859.
URL: `https://www.jstor.org/stable/2100193`

Gilbert, E. N. and Pollak, H. O. (1968), 'Steiner minimal trees', *Siam Journal on Applied Mathematics* **16**, 1–29.
URL: `https://api.semanticscholar.org/CorpusID:123196263`

Hwang, F. (1986), 'A linear time algorithm for full steiner trees', *Operations Research Letters* **4**(5), 235–237.
URL: `https://www.sciencedirect.com/science/article/pii/0167637786900088`

Innami, N., Kim, B. H., Mashiko, Y. and Shiohama, K. (2010), 'The steiner ratio conjecture of gilbert-pollak may still be open', *Algorithmica* **57**(4), 869–872.
URL: *https://doi.org/10.1007/s00453-008-9254-3*

Juhl, D., Warme, D., Winter, P. and Zachariasen, M. (2018), 'The geosteiner software package for computing steiner trees in the plane: an updated computational study', *Mathematical Programming Computation* **10**(4), 487–532.

Keydrain (2015), 'Steiner-tree-visualisation'.
URL: `https://github.com/Keydrain/Steiner-Tree-Visualisation`

Kirszenblat, D. (2014), The steiner ratio conjecture for eight points.
URL: *https://api.semanticscholar.org/CorpusID:125510711*

Marcus Brazil, M. Z. (2015), *Optimal Interconnection Trees in the Plane: Theory, Algorithms and Applications*, Algorithms and Combinatorics, 1 edn, Springer.

Melzak, Z. A. (1961), 'On the problem of Steiner', *Canadian Mathematical Bulletin* **4**(2), 143–148. Received January 3, 1961.
URL: *https://www.cambridge.org/core/services/aop-cambridge-core/content/view/S0008439500050803*

Nyaga, M. (2025), 'Understanding webassembly — the fundamentals'.
URL: *https://drs.software/blog/understanding-webassembly---the-fundamentals/*

Salowe, J. S. and Warme, D. M. (1995), 'Thirty-five-point rectilinear steiner minimal trees in a day', *Networks* **25**(2), 69–87.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230250206*

Selvatici, M. (2018), 'Emscripten tutorial'.
URL: *https://marcoselvatici.github.io/WASM_tutorial/*

Skiena, S. S. (2008), *The Algorithm Design Manual*, 2nd edn, Springer.

Wikipedia contributors (2025*a*), 'Double factorial — Wikipedia, the free encyclopedia', https://en.wikipedia.org/w/index.php?title=Double_factorial&oldid=1278154632. [Online; accessed 26-March-2025].

Wikipedia contributors (2025*b*), 'Tree (graph theory) — Wikipedia, the free encyclopedia'.
URL: *https://en.wikipedia.org/w/index.php?title=Tree_(graph_theory)&oldid=1270850413*

Winter, P. and Zachariasen, M. (1997), 'Euclidean steiner minimum trees: An improved exact algorithm', *Networks* **30**(3), 149–166.
URL: *https://doi.org/10.1002/(SICI)1097-0037(199710)30:3<149::AID-NET1>3.0.CO;2-L*