University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Understanding and Visualising Steiner Minimal Trees

Pieter van Tuijl

March 28, 2025

Abstract

The Steiner Minimal Tree (SMT) problem is a fundamental network optimisation problem with applications in road networks, circuit design, and telecommunications. Despite the theoretical interest in the problem and its practical applications, there is a lack of easily accessible visualisation tools since the problem's exponential nature has historically limited the feasibility and practicality of such tools. This project has aimed to develop a fast, accessible, and user-friendly web application for computing and visualizing Euclidean and rectilinear Steiner Minimal Trees and comparing them with Minimum Spanning Trees. The application leverages WebAssembly to run the C implementation of the GeoSteiner algorithms directly in the browser at near-native speeds while providing an intuitive interface for graph creation, manipulation, and analysis. The visualisation tool can compute and render Steiner minimal trees for instances of up to 200 terminals in under a second and enables real-time observation of how the Steiner ratio changes as the graph is modified by the user. This work represents the first successful open-source compilation of the GeoSteiner implementation to WebAssembly and the first publicly available online visualisation tool for the Steiner Minimal Tree problem, making the GeoSteiner algorithm more accessible across different programming languages and platforms, and providing researchers with a tool to explore the Steiner Minimal Tree problem and its properties, such as the Steiner ratio.

The web application is available at `https://steinertree.com`, and the source code is available at `https://github.com/pietert2000/visual-steiner`.

Acknowledgements

I want to heartily thank my supervisor, Dr Yiannis Giannakopoulos, for his infectious enthusiasm and support throughout this project, which has often exceeded what could be reasonably expected.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Pieter van Tuijl Date: 27 March 2025

Contents

1 Introduction **1**
1.1 Designing minimal networks 1
1.1.1 Minimum Spanning Tree vs Steiner Minimal Tree 1
1.1.2 Steiner Minimal Tree problem is NP-hard 1
1.1.3 Lack of great visualisation tools 1
1.2 Project aims and requirements 1
1.3 Our contributions 2
1.4 Dissertation outline 3

2 Background **4**
2.1 Historical context 4
2.1.1 Fermat-Torricelli (3-point) problem 4
2.1.2 Simpson lines 5
2.1.3 Generalising to *n* points 5
2.1.4 20th-century contributions 5
2.2 Euclidean Steiner Minimal Tree 6
2.2.1 Formal definition 6
2.2.2 Properties 6
2.2.3 Steiner ratio 8
2.3 Rectilinear Steiner Minimal Tree 8
2.3.1 Formal definition 8
2.3.2 Properties 8
2.3.3 Steiner ratio 8
2.4 Exact Steiner Tree algorithms 9
2.4.1 Melzak's algorithm 9
2.4.2 Complexity analysis for Melzak's algorithm 11
2.4.3 The GeoSteiner algorithm 11
2.5 Existing visualisation tools 12
2.5.1 Steiner-Tree-Visualisation (STV) 12
2.5.2 ESteiner-3D (E3D) 12
2.5.3 Steiner-Tree (ST) 12
2.6 Summary 13

3 Requirements **14**

4 Software design and implementation **15**
4.1 Technologies / Design choices 15
4.1.1 Web vs native 15

4.1.2 Tech stack 15
4.1.3 Graph drawing library 16
4.1.4 WebAssembly + Emscripten 16
4.2 Software architecture 17
4.3 Algorithm implementations 18
4.4 GeoSteiner integration 19
4.4.1 Compiling GeoSteiner to WASM 19
4.4.2 Integrating the GeoSteiner WASM module with the frontend 20
4.5 Graph State Management 21
4.5.1 Encoding multiple trees in one graph instance 21
4.5.2 Optimisations 21
4.6 Software design aspects 22
4.6.1 Immutability of the graph instance 22
4.6.2 Separation of concerns 23
4.6.3 Modularity 23
4.6.4 Type safety 23
4.6.5 Extensibility 23
4.7 Licensing 23
4.8 Summary 24

5 Visualisation and product features 25
5.1 High-level design 25
5.2 Creating and editing graphs 26
5.2.1 Creating/importing graphs 26
5.2.2 Editing graphs 27
5.3 Algorithmic visualisation 27
5.3.1 Layering 27
5.3.2 Visualising the Steiner ratio 28
5.4 Visualisation export 28

6 Analysis of the results 29
6.1 Product feasibility 29
6.1.1 Performance 29
6.1.2 Client-only architecture 29
6.2 Comparison with existing tools 30
6.3 Implications 31
6.3.1 Wider use of the GeoSteiner WASM module 31
6.3.2 Using the visualisation tool in research 31

7 Conclusion 32
7.1 Summary 32
7.2 Future work 33
7.2.1 GeoSteiner WASM module 33
7.2.2 Visualisation tool 33

Appendices 34

A Complexity derivation for Melzak's algorithm **34**

B Codebase structure **36**

Bibliography **37**

1 Introduction

1.1 Designing minimal networks

1.1.1 Minimum Spanning Tree vs Steiner Minimal Tree

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 total distance. This is a classic problem in graph theory, and the solution is known as the Minimum Spanning Tree (abbreviated as MST). Well-established and fast algorithms exist to find the MST of a set of points.

Now, imagine that a new constraint is added to the problem. Junctions (where three or more roads meet) are allowed, provided 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 (abbreviated as SMT) (Melzak 1961). Essentially, the Steiner Minimal Tree has a total length ≤ than the length of the MST by using extra points called Steiner points. It is worth noting that the Steiner problem has applications in fields other than roads and telecommunication networks, such as wire routing on printed circuit boards (see rectilinear Steiner Minimal Tree in section 2.3).

1.1.2 Steiner Minimal Tree problem is NP-hard

Though the constraint or description of the SMT problem is easy to state and understand, finding the optimal solution is complex. Since the late 1960s, effort has been put into finding an efficient algorithm for computing the SMT, but in contrast to the MST problem, no polynomial time algorithm has been found so far. The fastest and best-known algorithm for, GeoSteiner, still runs in exponential time (Winter and Zachariasen 1997).

1.1.3 Lack of great visualisation tools

Historically, the exponential nature of the SMT problem has imposed restrictions on the instance sizes that can be solved on an average computer, thus limiting the practicality and usefulness of visualisation tools for this problem.

Still, the SMT problem is a good example of the power of Moore's Law. The exponential growth in the processing power of computers over the last decades has been followed by similar increases in the size of instances that can be solved in a reasonable amount of time (Juhl et al. 2018). However, the available tools for visualising the SMT are limited in functionality, not easily accessible, and often lack an intuitive, user-friendly interface. This project aims to fill that gap.

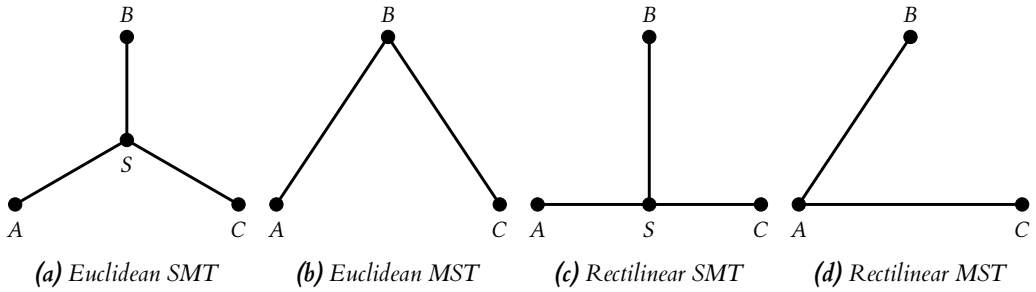
1.2 Project aims and requirements

This section provides a summary of the project's aims and requirements. Please refer to section 3 for a detailed discussion.

Aims This project aims to develop a fast, easily accessible, and user-friendly application that allows anyone to compute and visualise the Steiner Minimal Tree solution for arbitrary instance sizes and optionally compare it with the Minimum Spanning Tree solution.

Requirements Specifically, the application shall take a set of points as input, which can be generated randomly or imported using a range of established file formats, and provide the user with the ability to visualise, compare, and analyse the SMT and MST solutions. Furthermore, the application should dynamically update the solutions when the user modifies the graph by adding, moving, or removing points. Lastly, the application will provide the user with different export options using visual (e.g., .png, .jpg) and text-based formats (e.g., .gexf).

It is important to note that the Steiner Tree and Minimum Spanning Tree problems are defined for many different metric spaces. In this project, however, only the Euclidean and rectilinear (also known as the Manhattan or taxicab distance) variants are considered. Figure 1.1 shows the difference between the two variants for a 3-point example instance.



(a) *Euclidean SMT* **(b)** *Euclidean MST* **(c)** *Rectilinear SMT* **(d)** *Rectilinear MST*

Figure 1.1: *Comparison of the Euclidean and rectilinear MST and SMT solutions for a 3-point problem instance*

Access the end product here

`https://steinertree.com`

1.3 Our contributions

This project includes the following new contributions:

- The first successful (open source) compilation of the GeoSteiner implementation to WebAssembly (WASM). With this, the GeoSteiner algorithm is now easily accessible to any programming language that supports WASM.
- The first publicly available online visualisation tool for the Steiner minimal and Minimum Spanning Trees. The Euclidean and rectilinear variants of said trees can be computed and visualised for graph instances of up to 200 terminals in sub-second times. Furthermore, the application provides a dynamic tree-updating feature that lets the user observe how the Steiner ratio changes in real time as the graph is edited.
- A detailed illustration of Melzak's algorithm is provided, showing how the algorithm works step-by-step for a 5-point instance (2.3). The literature seen by the author generally contains illustrations for the 3-point case. However, it is difficult to visualise the recursive nature of the algorithm beyond the 3-point case. Our illustration meets this gap and is believed to be a new contribution.
- A derivation of the formula for the number of full Steiner topologies for a given set of terminals is provided in appendix A. The derivation is based on Winter and Zachariasen

(1997) and Gilbert and Pollak (1968) but expanded and made complete by the author for an improved understanding.

Note on the figures All the graph figures in this dissertation were created by the author using TikZ.

1.4 Dissertation outline

This dissertation presents the development of a web-based visualisation tool for the rectilinear and Euclidean variants of the Steiner Minimal Tree and Minimum Spanning Tree problems. It is structured as follows:

- **Chapter 2**: aims to equip the reader with all the required background information related to the Steiner Tree problem to understand the problem and the purpose of the visualisation tool. First, the historical context is discussed, followed by the formal definitions of the Euclidean and rectilinear Steiner Minimal Tree problems and their properties, including the Steiner ratio. Next, exact algorithms for the Euclidean variant are discussed, such as Melzak's algorithm and the GeoSteiner algorithm, with the latter also being the fastest known exact algorithm for the Steiner Minimal Tree problem and the basis of the implementation of the visualisation tool. The chapter will end with an overview of existing visualisation tools for the Steiner Tree problem and discuss their features and limitations.
- **Chapter 3**: expands the initial project aims into focused requirements, distinguishing between core requirements and the desirable features. All the design decisions discussed in the succeeding chapter are based on these requirements.
- **Chapter 4**: gives an overview of the software design and implementation of the visualisation tool. It explains and justifies the design choices, such as the technology stack, the architecture of the application, and the chosen canvas library. This is followed by a section on the implementation of the algorithms, where focus is given to the GeoSteiner algorithm and how it was integrated into the application using WebAssembly. Next, it discusses the constraints around the state management of the graph and how these were overcome. Mention is made of various state-related optimisations that improved the application's responsiveness when using large graph instances. The chapter finishes with a discussion on software quality and design aspects that were considered to promote the extensibility and maintainability of the application, as well as a brief consideration of licensing requirements.
- **Chapter 5**: demonstrates the visualisation tool and showcases the product features alongside a discussion on how they address the requirements as set out in chapter 3.
- **Chapter 6**: provides an analysis of the end product. The feasibility of the application is discussed and demonstrated, and a comparison is made between the end product and the existing visualisation tools as listed in chapter 2. The chapter ends with a brief discussion on the potential implications of the end product and its use in other applications and research.
- **Chapter 7**: summarises the problem and the main contributions of the project. Future work is suggested for further improvements and extensions to the application.

2 Background

2.1 Historical context

The Euclidean Steiner problem has a long history, with roots in the 17th century. The following section is a summary of Brazil et al. (2014). The reader is referred to said paper for more details on the sources referenced in this section.

2.1.1 Fermat–Torricelli (3-point) problem

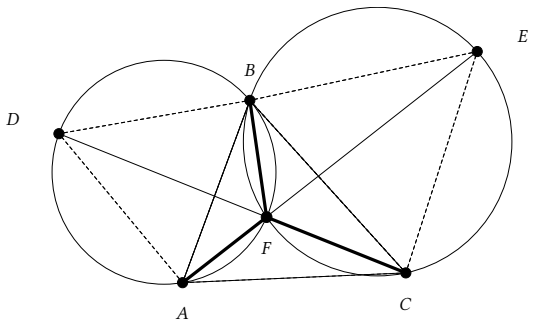
In 1643, Fermat (Fermat 1891, p. 135) posed the problem: Given three points, find a fourth point such that when connecting the three points to the fourth point, the sum of the distances is minimal. This problem has two cases:

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

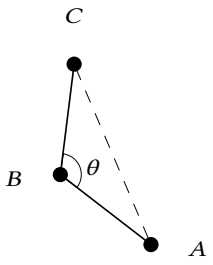
For the first case, Torricelli (1608–1647)(Torricelli 1919) proposed the following construction. Given $\triangle ABC$, first draw the equilateral triangles $\triangle ABD$ and $\triangle BCE$. Then, for each of these equilateral triangles, draw the circumcircle. The position where the two circles intersect is the point where, if it is connected to the other three points, the sum of the distances is minimal, the so-called *Torricelli point* (point F in Figure 2.1a).

Equilateral points 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. (Equilateral points will be later referenced again in our discussion on exact algorithm.) So, in figure 2.1a, $e_{AB} = D$ and $e_{BC} = E$.

For the second case, Cavalieri (1647) proved that the optimal position of the 4th point conincides with the point which has the obtuse angle. (point B in Figure 2.1b)



(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.*

2.1.2 Simpson lines

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

2.1.3 Generalising to *n* points

The Fermat-Torricelli problem was later generalised to an *n*-point problem by Gergonne in 1811. Furthermore, Gergonne was among the first to consider the case where more than one extra point is allowed. He further discovered that multiple local minimal solutions can exist for the same set of points and that all must be constructed to find the global Minimum. Lastly, he devised an iterative, though incomplete, algorithm for finding a local minimum, given a full topology (Brazil et al. 2014). We will return to the concept of local and global minima in 2.4, where Melzak's algorithm will be discussed. Melzak's algorithm is essentially an extension of Gergonne's algorithm.

Gauss made another contribution to the problem 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 famously illustrated the problem by describing the example of connecting German cities by railroad with minimal total length.

> I have on occasion considered the rail- road connection between Hamburg, 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 successful attempts were made by Bopp (1879) and Hoffmann (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). Notably, 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).

2.1.4 20th-century contributions

Then, in 1934, two Czech mathematicians, Jarník and Kössler, proved several important properties of the Steiner Tree, such as the degree property and the angle property (See 2.2.2). The latter two properties were valid even if the problem was extended to higher dimensions (Brazil et al. 2014).

Street network problem Choquet compared the Steiner Tree problem to a network of cities connected by roads where no junctions are available between the roads except outside the cities. A similar illustration was later used by Courant and Robbins (1941) in their influential book *What is Mathematics?*, in which they called the generalised Fermat-Torricelli problem the "street network problem". Interestingly, they incorrectly attributed the original Fermat-Torricelli problem to Jakob Steiner. Consequently, the generalised Fermat-Torricelli problem (i.e. the street network problem) has since been known as the Steiner Tree problem in the literature even though it is unclear whether Steiner ever came up with this generalisation (Brazil et al. 2014).

Gilbert and Pollak's seminal paper Finally, Gilbert and Pollak (1968) provided a comprehensive survey of the Steiner Tree problem. They defined much of the terminology that is used in the literature today for describing 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)

Renewed interest in the sixties From the sixties onwards, the Steiner Tree problem has grown in popularity, partly due to the development of formal algorithms for the related Minimum Spanning Tree problem and partly due to the rise of 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.

Summary In this section, we have discussed the historical context of the Steiner Tree problem and how it has evolved over the centuries. The following section will deal with the formal definitions of the Euclidean and rectilinear Steiner Minimal Tree problems and list some of their properties.

2.2 Euclidean Steiner Minimal Tree

In the present section, the Euclidean Steiner Minimal Tree is defined formally, and some of its properties are listed, including the Steiner ratio conjecture.

2.2.1 Formal definition

The Euclidean Steiner Minimal Tree problem is concerned with finding the shortest tree containing a set of vertices in the Euclidean plane.

More formally:

- Let N be a finite set of n points in $\mathbb{R}^2$
- 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 Euclidean 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 $e \in E$ (Marcus Brazil 2015).

2.2.2 Properties

Euclidean Steiner Trees have specific properties, which are enumerated in the seminal 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.

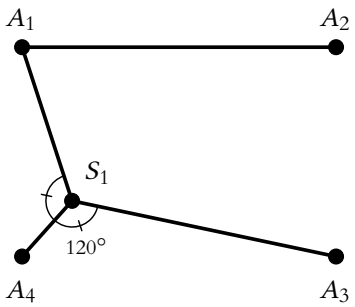
1. Angle property: every pair of lines meets at $\theta \geq 120°$.
2. Degree property: every Steiner point has exactly 3 incident edges.
3. A Steiner Tree with n terminals has $s \leq n - 2$ Steiner points. Since a tree has $n - 1$ edges, a Steiner Tree with n terminals has at most $2n - 3$ edges.
4. At most one relatively minimal tree (see 2.2.2 for a definition) 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 (see 2.2.2 for the definition of an FST).

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

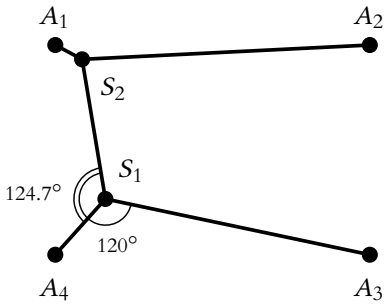
- A topology is defined as the adjacency matrix, which only specifies the connections between points (i.e. how points are interconnected). 2.2d displays three different topologies for the same set of points, where 2.2b and 2.2c share the same topology.
- A relatively minimal tree (RMT) is the minimal tree for a given topology. All figures in 2.2 are RMTs except for 2.2b. A relatively minimal tree is obtained when small perturbations (displacements) of the Steiner points no longer result in a smaller tree.

- A Steiner Tree (ST) is always an RMT, but the converse does not hold. For instance, 2.2a is an RMT, but the angle $\angle S_1A_1A_2 = \theta < 120°$ and, therefore, does not satisfy the angle property. A Steiner Tree is not necessarily minimal for all its vertices and is therefore a local minimum (e.g. compare Fig 2.2c with 2.2d).
- 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.
- A full Steiner topology has $n - 2$ Steiner points. A Full Steiner Tree (FST) is the relatively minimal tree for a full Steiner topology, where the terminals are leaves and the Steiner points are internal nodes with degree 3. Hereafter, we shall use the terms "full Steiner topology" and "full topology" interchangeably.

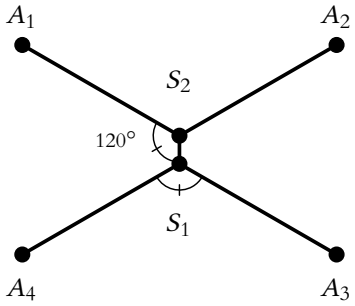
In summary, given a topology, its Steiner Tree (if it exists) can be seen as a local minimum. Only by enumerating all possible topologies and their corresponding Steiner Trees can the global Minimum (i.e. the SMT) be found. This concept forms the basis of the algorithms that will be discussed in section 2.4.



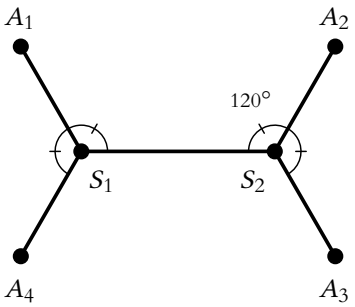
(a) *Relatively minimal tree for its topology*



(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 Euclidean Minimum Spanning Tree (EMST). The EMST is 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 Euclidean SMT and the MST has received much attention in the literature. Gilbert and Pollak (1968) famously conjectured that the lowest possible ratio is $\frac{\sqrt{3}}{2} \approx 0.866025...$ 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 invalidating the proof (Innami et al. 2010). Hence, the Steiner ratio conjecture remains an open problem to this day. Nevertheless, attempts have been made to find proofs where n is bounded. For example, Kirszenblat (2014) came up with a proof for the case where $n \leq 8$.

2.3 Rectilinear Steiner Minimal Tree

In this section, the Rectilinear Steiner Minimal Tree problem is defined formally, and some of its properties are listed. It concludes with a discussion on the rectilinear Steiner ratio.

2.3.1 Formal definition

The formal definition of the Rectilinear Steiner Minimal Tree (RSMT) is very similar to the definition of the ESMT given in section 2.2.1, 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, all its incident edges are straight.

For clarity, some properties of the properties that were enumerated in 2.2.2 also apply to the RSMT, such as the maximum number of Steiner points and edges in a Steiner Tree (property 3).

2.3.3 Steiner ratio

As before, an 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 Exact Steiner Tree algorithms

The Steiner Tree problem is easy to understand but hard to solve (Marcus Brazil 2015). Garey et al. (1977) proved that the Euclidean and Rectilinear variants of the Steiner Tree problem are NP-hard. This means no exact polynomial-time algorithms exist for these problems (unless P = NP). Although 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 to demonstrate that despite the problem's theoretical hardness, the actual performance in practice is impressive on modern hardware.

This section will discuss two well-known exact algorithms for the Steiner Tree problem:

- Melzak's algorithm (Melzak 1961), which is largely based on the iterative algorithm by Gergonne (see 2.1.3). This algorithm is essentially a brute-force approach and is, therefore, not practical for large instances. Still it forms the basis of the GeoSteiner algorithm and is therefore important to understand.
- The GeoSteiner algorithm developed by Winter and Zachariasen (1997). It is much faster and efficient than any other exact algorithm for the Steiner Tree problem due to utilising the geometric properties of Steiner Trees for pruning the search space, efficient data structures, and other clever optimisations. This algorithm has been the fastest for decades.

Note: The software implementation of this project uses the GeoSteiner algorithm, but the actual implementation is not discussed until section 4.

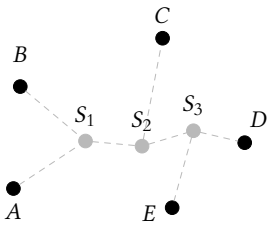
Why Melzak's algorithm is discussed Melzak's algorithm helps to understand the Steiner Tree problem, its properties, and its exponential complexity. Its ideas also form the building blocks of the GeoSteiner algorithm and understanding it is a good starting point for understanding how GeoSteiner works. Moreover, it has been found that literature is sparse on clear visualisations of the algorithm for instances with $n > 3$. One contribution of this project is to provide a practical illustrated example of the mechanics of the algorithm for an instance with $n = 5$ (see figure 2.3).

2.4.1 Melzak's algorithm

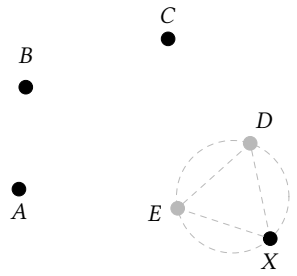
Melzak's algorithm takes a full topology as input and finds the Full Steiner Tree (FST) if it exists. At stated in 2.2.2, an FST is defined as a Steiner Tree with n terminals and $n - 2$ Steiner points, where terminals are leafs (degree 1) and Steiner points are internal nodes (degree 3).

Formal definition 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 $\triangle ABX$.

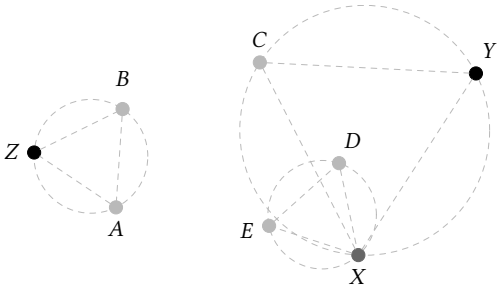
Assume that S_i is positioned on the opposite side of $\overline{AB}$ to X and that it lies on the arc $\overset{\frown}{AB}$ (Steiner arc) of the circumcircle of the equilateral triangle $\triangle ABX$, subtended by $\overline{AB}$. The Simpson segment starts at X and passes through S_i to another point v. By definition, the length of the Simpson



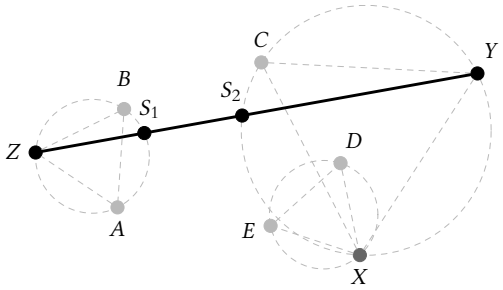
(a) *Initial full topology, Steiner points are randomly placed for visualisation purposes*



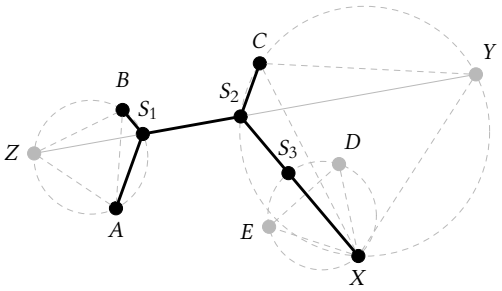
(b) *Start of merging. Replacing (E, D, S_3) with X*



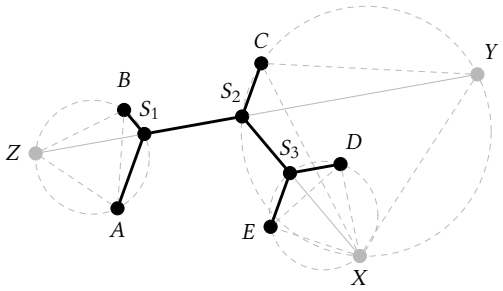
(c) *Replacing (X, C, S_2) with Y, and (A, B, S_1) with Z, results in the n = 2 base case.*



(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: *Melzak's algorithm to find the FST (Full Steiner Tree) for a full topology $\mathcal{T}_5$ (with n = 5)*

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 will also 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 $\overset{\frown}{AB}$. S_i is positioned at this point of intersection, and the FST is constructed by deleting the Simpson segment and adding the edges AS_i, BS_i, and vS_i.

Illustrating with an example 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 to visualise all the connections (topologies only encode connections,

not positions).

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, reducing 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 $\overset{\frown}{ED}$. 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 $Z = e_{AB}$ (see section 2.1.1 for the definition of an equilateral point). 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 is also the SMT. However, this is not always the case. From property 5 (see 2.2.2), it is known that the SMT is a union of FSTs, hence the SMT must be determined 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 the resulting tree is minimal. Winter and Zachariasen (1997) state that the total number of full topologies for a set of n terminals is given by

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

In short, it counts all full topologies for all subsets of n terminals. For completeness, a derivation of this formula is given in appendix A.

It is clear that $F^*(n)$ grows rapidly with n. Due to the factorial in the numerator, it can be said to grow *super-exponentially* in k (i.e. faster than 2^k). To give an idea of the scale of $F^*(n)$, $F^*(5) = 50$, $F^*(10) = 3,986,175$, and $F^*(20) = 3.93 \times 10^{20}$.

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 indicates that the exponentiality is not merely due to Melzak's algorithm but is inherent in the Steiner problem itself (Hwang and Richards 1992).

2.4.3 The GeoSteiner algorithm

The GeoSteiner algorithm has been the fastest for computing the exact Steiner Minimal Tree for 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 generation stage, further reducing the number of FSTs that need to be considered during *concatenation*. The reader is referred to Winter and Zachariasen (1997) for more details on these pruning tests. As a result of these two factors, the number of FSTs generated in the first stage is linear in n. It has been empirically shown that for the Euclidean case, on average, the number of generated FSTs is $2.5n$ with only $0.6n$ left after pruning (Juhl et al. 2018).

The process 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 has the minimal length. This results in 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 Euclidean version of the problem. It is noteworthy that the algorithm has been extended to the rectilinear case as well (Salowe and Warme 1995). It has been observed that for the rectilinear case, the generation phase is fast, but the concatenation phase is much slower compared to the Euclidean variant (Juhl et al. 2018). The software implementation of this project uses the Euclidean and Rectilinear versions of the GeoSteiner algorithm.

2.5 Existing visualisation tools

The previous section discussed the foundational ideas from Melzak's algorithm and the features of the GeoSteiner algorithm. The following section will move on from the theoretical discussion and focus on the existing visualisation tools found during the research phase.

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 Euclidean 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 and becomes unusable for instances larger than 40 nodes.
- Additionally, the GUI is not flexible enough. For example, you cannot overlay the MST and SMT simultaneously, and the canvas does not support zooming or resizing. Importing a graph from an external file or exporting the results is also not possible. And no visualisation of the Steiner ratio is provided.
- Lastly, the tool only runs on Python 2.x and would require code patching to run in modern, Python 3.x 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 Euclidean SMT of a graph and supports 2D and 3D graphs. However, the use of the software is limited by the fact that no graphical or 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 the visualisation of the MST or SMT in the Euclidean plane. And even for the rectilinear case, it is only possible to place the nodes on grid lines instead of anywhere on the canvas. Importing coordinates from an external file is not possible, nor can the visualisation results be exported. Further, no visualisation of the Steiner ratio is provided, and despite being a web app, it is not deployed publicly.

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

2.6 Summary

In this chapter, we discussed the historical context of the Steiner Tree problem, tracing the problem's evolution from Fermat's original three-point problem in the 17th century through to the modern, generalised version of the problem. The Euclidean and rectilinear variants of the Steiner Minimal Tree problem were then formally defined, and their key properties were enumerated, including the concept of the Steiner ratio. For the Euclidean case, this ratio remains an open problem with a conjectured value of $\frac{\sqrt{3}}{2}$, while for the rectilinear case, it has been proven to be exactly $\frac{2}{3}$.

The chapter then focused on exact algorithms. Melzak's algorithm was discussed in detail, not only because it forms the basis for understanding the GeoSteiner algorithm but also because it helps explain the problem's exponential complexity. The GeoSteiner algorithm, which remains the fastest known exact algorithm for decades, was then discussed, highlighting the key optimisations that make it so fast in practice.

Finally, existing visualisation tools were reviewed, which highlighted limitations in in terms of speed, visualisation options, import/export capabilities, and platform independence. These limitations, combined with advances in the development of an efficient exact algorithm for the Steiner Tree problem, motivate the development of our visualisation tool, whose requirements will be discussed in the next chapter.

3 Requirements

The visualisation requirements stated in this section expand the project aims laid out in the introduction. This section is important because:

- all design decisions and implementation choices discussed in the following chapters derive from these requirements.
- it demonstrates how the shortcomings of existing tools (as highlighted in section 2.5) are addressed and also forms the basis for evaluating the end product.

As stated in the introduction, the project aimed to develop a fast, easily accessible, user-friendly application for visualising Steiner Minimal Trees and some of their properties. This aim has been translated into the following **essential requirements**:

E1 The software must feature a well-designed, clean, and intuitive user interface.

E2 The software must support the visualisation of the Steiner Minimal Tree (SMT) and the Minimum Spanning Tree (MST), specifically the Euclidean and rectilinear variants.

E3 Include a canvas for graph visualisation that supports zooming and panning and can render large graphs.

E4 The computation of the SMT and MST must be fast and snappy for instances of up to 200 nodes (< 1 second).

E5 The user can create a graph instance of any size by importing from an external file, drawing on the canvas, or generating a random instance.

E6 The user can add, move, or remove nodes of an existing graph instance by directly manipulating the canvas.

E7 If the user updates a graph instance after the SMT and MST have been computed, both trees must be dynamically recalculated and rerendered on the canvas.

E8 The SMT and MST can be directly compared in terms of length (Steiner ratio) and tree structure.

E9 The software must be deployed as a web app to be accessible and platform-independent.

In addition, the following **desirable requirements** have been identified:

D1 The user can customise the appearance of the visualisation, such as the node size, edge thickness, and colours.

D2 The results are exportable in useful formats, such as image (PNG, JPG) or text-based formats (GEXF).

D3 The user can toggle the display of algorithm outputs.

D4 The software must be modular and extensible, allowing for straightforward integration of other graph algorithms.

4 Software design and implementation

This section serves as a guide to the implementation of the software.

- First, justification is given for the selected tech stack and design choices.
- Then, the application's big-picture architecture is discussed.
- After that, the implementation of algorithms is discussed, focusing on the integration with the GeoSteiner library.
- Then, aspects of the software design, such as modularity, type-safety, separation of concerns, extensibility, and immutability, are highlighted.
- Lastly, there is a brief mention of licensing requirements.

4.1 Technologies / Design choices

This section presents and justifies the key technologies and design choices used in developing the software. First, we discuss the choice of a web-based architecture over a native application, followed by an examination of the selected technology stack and its components. Special attention is given to the graph drawing library selection process, where we compare different options based on their features and performance characteristics. Finally, background information on WebAssembly and Emscripten is provided, which will be relevant for the disucssion of the implementation of the GeoSteiner algorithm in 4.4.

4.1.1 Web vs native

The first implementation decision was to build a web app instead of a native app (see requirement E9 in section 3). Browsers are ubiquitous nowadays and provide a cross-platform way to run and distribute applications. Updates can easily be deployed without requiring users to install them manually. Further, differences between browser vendors have become less pronounced due to the development of standards and the rise of frameworks that abstract away the jarring differences.

4.1.2 Tech stack

The web app is built using **Typescript**[1] (TS) as the primary programming language, which is a statically-typed superset of JavaScript (JS) that compiles to JS (Bierman et al. 2014). Type-safety makes the application more robust and easier to reason about. In addition, Typescript has a great developer experience (e.g. for refactoring and code completion). Empirical evidence has also suggested that it is effective at detecting bugs and that it improves code maintainability (Gao et al. 2017).

React[2] has been used for the core UI logic. React is a popular library for building interactive frontends and has a large ecosystem of component libraries behind it. React code is declarative (i.e., you only define what the UI should look like, not how to achieve it) and, therefore, easier to reason about and maintain.

[1] `https://www.typescriptlang.org/`
[2] `https://react.dev/`

Other dependencies include:

- **Shadcn/ui**[3]: a React component library that provides a set of accessible components, such as buttons, inputs, drop downs, colour-pickers, etc.
- **Sigma.js**[4]: a popular canvas library for visualising graphs.
- **Graphology**[5]: a library that provides a graph data model, which is required by Sigma.js.

	Supported features							Quality metrics			
Library	Typescript support	React support	Interactive	Large graphs	Dynamic updates	Customizable	Flexible data model	Comprehensive docs	Maintained	Size (kB)	GitHub stars
Vivagraph	✗	✗	✗	✗	✓	✓	✓	✗	✗	58.6	3.8k
Cytoscape.js	✗	✓	✓	✗	✓	✓	✗	✓	✓	406.5	10.3k
Sigma.js	✓	✓	✓	✓	✓	✓	✓	✗	✓	89.2	11.5k

Table 4.1: *Comparison of some popular JavaScript graph rendering/drawing libraries.*

4.1.3 Graph drawing library

As specified above, the app uses the open source library Sigma.js for graph drawing, with Graphology as its graph data model. Table 4.1 shows how Sigma.js compares to other popular JavaScript graph drawing libraries. Sigma.js was chosen because it ticked most boxes regarding required features and quality metrics. Sigma.js is one of the most performant graph drawing libraries[6] because it uses WebGL for rendering instead of SVG or the HTML 2D Canvas. Besides, it uses a separate, lightweight, and flexible data model (Graphology) for storing and manipulating graph data, which helps keep the views separate from the data manipulation logic. Lastly, it satisfies the essential requirement of supporting direct interaction with the graph through the canvas (see E6 in section 3) by exposing user-interaction events, such as the clicking or dragging nodes.

4.1.4 WebAssembly + Emscripten

As this project uses the GeoSteiner C library to compute the Steiner Minimal Tree, it is necessary to use WebAssembly to run the algorithm in the browser. The browser cannot natively execute C code, but WebAssembly provides a way to run languages other than JS directly in the browser at near-native performance.

WebAssembly WebAssembly[7] (WASM) is a binary instruction format for a stack-based virtual machine that can be executed in the browser. It is designed to be a portable target for the compilation of languages, such as C and C++, into a compact binary format. Due to its compact format, executing WASM code is much faster than executing optimised JS code (Nyaga 2025). WASM is supported by most browsers, except for Internet Explorer and Opera Mini [8].

[3] `https://ui.shadcn.com/`
[4] `https://sigmajs.org/`
[5] `https://Graphology.github.io/`
[6] `https://www.youtube.com/watch?v=JGN9dejWjAk&ab_channel=DominikMaszczyk`
[7] `https://webassembly.org/`
[8] `https://caniuse.com/wasm`

Emscripten Emscripten[9] is a compiler toolchain that can compile C and C++ code to WASM. It comprises a designated compiler frontend (emcc) and an LLVM back end optimised for generating WASM binaries. The output from the Emscripten compiler is a .wasm file (module) and a .js file containing "glue code". This code serves as an interface between the WASM binary and the JavaScript environment (Selvatici 2018). The front end only needs to load the glue code, which loads and instantiates the WASM module.

Communication between JS and WASM For JS to call functions in the WASM module, the WASM module must explicitly export these functions (by default, no functions from a compiled C/C++ library are exported). Moreover, it is not possible to pass complex data structures to a WASM function or return such data from it since WASM only supports 4 basic data types (i.e., 32/64-bit ints and floats). Instead, the shared contiguous memory space between JS and WASM can be used to pass pointers to memory locations that contain the data. The glue code provides special "views" of the same memory space. For example, depending on the type of data that needs to be accessed or written, it can be viewed as an array of 64-bit floats or 8-bit unsigned chars. This avoids the need to manually handle different offsets and strides.

4.2 Software architecture

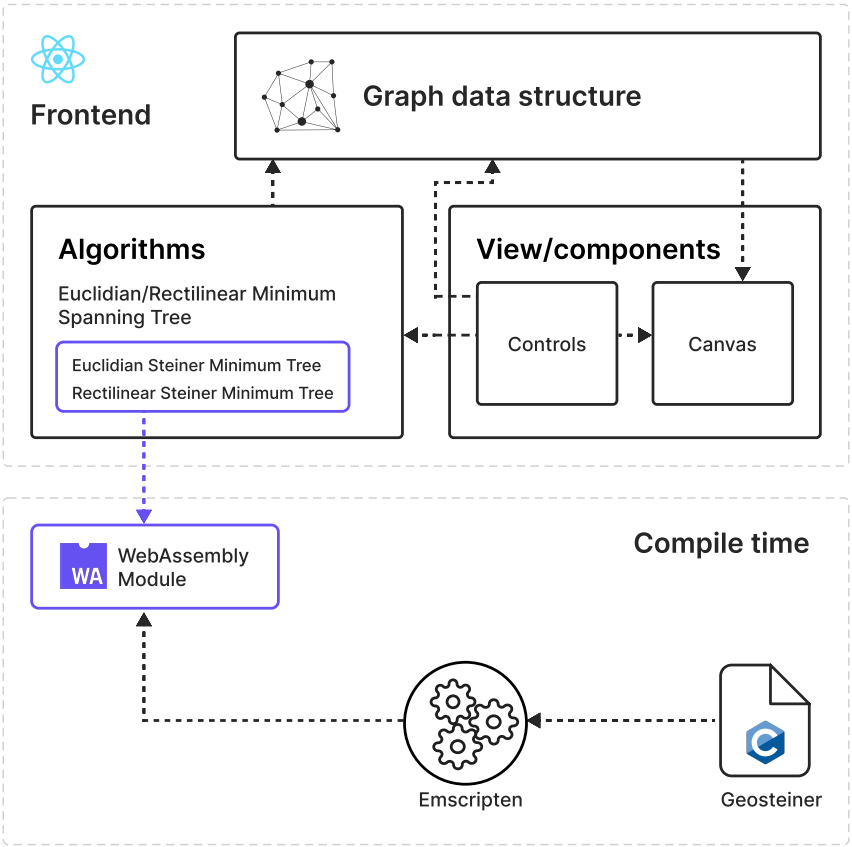


Figure 4.1: Architecture of the web application. It comprises the frontend (top layer) and the WebAssembly module (bottom layer) compiled with Emscripten from the GeoSteiner C library. The WASM module is loaded by the front end at runtime. The arrows indicate the flow of data and control.

[9] https://emscripten.org/

The application is developed as a static, client-only application without a back-end API. A great benefit of this architecture is that it is secure by default as no communication with a server backend is required. Figure 4.1 shows the various high-level components of the application. The top layer (i.e. the front end) is divided into three main parts:

- The views part, which handles user interaction (controls) and the visual appearance of the application.
- The algorithms part, which handles the computation of the Minimum Spanning Tree (MST) and Steiner Minimal Tree (SMT) algorithms.
- The graph data structure as a global state, which can be read and updated by the views and algorithms. The state containing the graph instance is immutable (see section 4.6.1).

Through the relevant UI controls, the user can 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. The algorithm's output is merged with the existing graph data structure in the latter case. This is important because it keeps the graph instance immutable (see section 4.6.1). Updates to the graph data structure are reflected in the canvas component.

The lower layer contains the WebAssembly module, which is compiled with Emscripten from the GeoSteiner C library and dynamically imported by the front end during runtime.

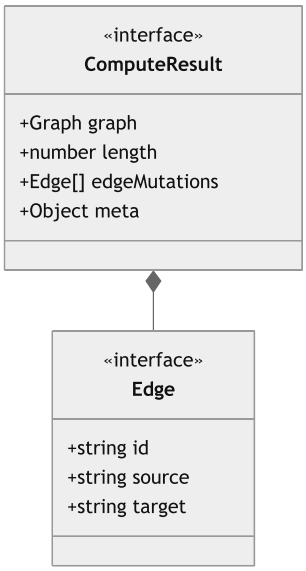
See appendix B for the organisation and code distribution of the codebase.

4.3 Algorithm implementations

This section briefly discusses which algorithms were implemented and how they were integrated into the application.

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. Each algorithm returns a tree and its length. The list of edge mutations keeps track of all the edges 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 node identifiers. By coding to an interface rather than 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 data structure by a *merge* operation. This 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 (see section 4.5).

The following algorithms have been used:

- Prim's $O(n^2)$ algorithm for Euclidean and rectilinear Minimum Spanning Trees (using (Skiena 2008, pp. 194–195))

- The GeoSteiner algorithms for Euclidean and rectilinear Steiner Minimal Trees.

Since Steiner Minimal Trees are the focus of this project, the details of Prim's algorithm are not discussed here. Section 4.4 discusses the integration of the GeoSteiner library in more detail.

4.4 GeoSteiner integration

This section explains how the GeoSteiner library was integrated within the web application. This section is crucial because, to the author's knowledge, this integration is the first of its kind, and it is expected to open doors for new levels of interest in exact algorithms for the Steiner Tree problem. Due to WASM's portable format, it will now be possible to leverage the GeoSteiner algorithms in any language with WASM support, including Python[10], Java [11], server-side Javascript runtimes like Node.js[12], and many more. Please refer to section 6 for more discussion on the possible implications.

Outline This section consists of two parts:

- The first part will explain how the GeoSteiner library was modified and compiled into a WASM module using Emscripten.
- The second part will discuss the integration of the WASM module and the calling code in the front end.

4.4.1 Compiling GeoSteiner to WASM

The GeoSteiner library consists of the main algorithm implementation and an optimised version of the open-source linear solver, lp_solve v2.3[13], that is used during the FST concatenation stage. Compiling any library to WASM requires compiling all its dependencies to WASM.

It was therefore first verified that the GeoSteiner library and the solver dependency would be portable to WASM using the portability guidelines in Emscripten (2025). Despite satisfying all requirements, several problems were encountered during the first compilation attempts.

Floating point control register (fpcr) settings GeoSteiner uses different floating point control register settings depending on the processor architecture. The two settings common across all architectures, however, were found to be:

- The rounding mode is set to "round to nearest"
- Floating-point exceptions for overflow, divide-by-zero, and invalid operations are trapped.

The rounding mode for the WASM virtual machine is already set to "round to nearest" by default, voiding the need to set the rounding mode in WASM environments. However, floating point traps are not implemented in WASM (WebAssembly 2025). The GeoSteiner code relies on floating-point traps because most systems will generate and propagate NaNs during floating-point exceptions, which may compromise the numerical stability of the algorithm. At present, this feature is only supported on x86 and ARM64 architectures (Warme et al. 2022). Due to this limitation, the GeoSteiner code was modified not to rely on floating-point traps in WASM environments. This was achieved by using preprocessor directives to conditionally apply the floating-point traps.

[10] https://github.com/wasmerio/wasmer-python
[11] https://github.com/wasmerio/wasmer-java
[12] https://nodejs.org/
[13] https://sourceforge.net/projects/lpsolve/files/lpsolve/2.3/

Makefile modifications The GeoSteiner library ships with an auto-generated Makefile. The following changes were made to ensure compatibility with the Emscripten toolchain.

- A flag was set to create a static library instead of a dynamic one, according to Emscripten's recommendation for the best performance[14].
- The build process was modified to use the *emcc* compiler frontend instead of the default GCC compiler.
- In order to create a static library, the normal compilation process depends on a locally installed version of the GNU libtool[15]. Libtool itself uses the *ar* and *ranlib* utilities. Emscripten provides a custom version of these utilities that is compatible with WASM, called *emar* and *emranlib*, respectively. Unfortunately, it was not possible to use environment variables to set custom paths for the *ar* and *ranlib* utilities used by libtool, so, direct changes to the local libtool installation had to be made instead.

With these changes, a WASM-compatible libgeosteiner.a static library archive file was created and used to compile the WASM module.

GeoSteiner facade As mentioned in section 4.1.4, WASM-defined functions that are to be called from JS must be explicitly exported, which is done by adding their names to the *EXPORTED_FUNCTIONS emcc* compile flag. Since the complexity of interacting with the WASM module grows with the number of exported functions, it was decided only to export a **simplified API**, using the *facade*[16] design pattern. This facade handles all the underlying GeoSteiner-specific logic required for the computation of Steiner Minimal Trees but irrelevant for the front end. The facade's interface is shown in listing 4.1.

```
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 facade functions make heavy use of pointers because arrays are non-primitive data types which cannot be directly passed to WASM functions. Instead, JS and WASM share a special, contiguous memory space that can be used to share complex data types such as arrays (see section 4.1.4). Data can be passed to the WASM module by allocating memory in the memory space and passing a pointer to the memory location to the WASM module. Due to this explicit memory management, 'malloc' and 'free' are also explicitly exported from the WASM module in addition to the facade functions.

4.4.2 Integrating the GeoSteiner WASM module with the frontend

The output from the WASM compilation process is a .wasm file and a .js file containing "glue code" (see section 4.1.4). The .js file is loaded like any other JavaScript file, which completely hides the loading and setup process of the WASM module. It is worth noting that the JavaScript environment is unaware of any functions defined in the WASM module. Therefore, the glue code provides a special *cwrap* function to make them available and callable from the JavaScript environment. The *cwrap* function takes the name of the function to wrap, the return type, and the argument types.

[14] https://emscripten.org/docs/compiling/Dynamic-Linking.html
[15] https://www.gnu.org/software/libtool/
[16] https://en.wikipedia.org/wiki/Facade_pattern

As noted before, explicit memory management is needed for the passing and returning of non-primitive data. From the properties of the Steiner minimum tree (section 2.2.2 and 2.3.2), it is known that the maximum number of Steiner points is $n - 2$, and the maximum number of edges is $2n - 3$. Using these constraints, memory is allocated for the Steiner points and edges, and their pointers are passed to the facade functions.

4.5 Graph State Management

Besides the GeoSteiner integration, another core feature is how the graph state is managed. As mentioned in section 4.2 and justified in section 4.6.1, the graph instance is immutable, but its internal state is mutable. As each algorithm outputs a tree as a new graph instance, its nodes and edges must somehow be merged with the existing graph instance without losing track of which nodes and edges belong to which algorithm. This section will describe the merging process and the optimisations that were used to make it efficient.

4.5.1 Encoding multiple trees in one graph instance

The graph's internal state is an undirected graph, which is represented as a map of nodes and edges mapped by their identifier (key). Each node and edge has a set of attributes that describe its properties. The main attributes are:

- *algorithm* -> `string`[]: list of algorithms for which the node or edge was part of the output tree.
- *isSteiner* -> `boolean`: indicates if the node is a Steiner node.

In an undirected graph, at most one edge can exist between any two nodes. Edges can be queried by their key or by the keys of the two nodes they connect.

The graph's internal state can be mutated through the following user interactions:

1. Adding, moving, or removing nodes (creating a new graph in the UI corresponds to removing all previous nodes and adding the new ones)
2. Triggering an algorithm's computation

1 is trivial. 2 is more complex and requires *merging* the algorithm's output tree (denoted as *source*) with the existing graph instance (denoted as *target*). The following cases must be considered:

- The *source* has an edge that already exists in the *target*. In this case, the edge attributes of the *source* are merged with the ones of the *target*. Specifically, the *algorithm* attribute is updated to include the algorithm that generated the source tree, and the original key of the *target* edge is replaced with the *source* edge's key.
- The *source* has an edge that does not exist in the *target*. In this case, the edge is added to the *target*, and its *algorithm* attribute is updated to include the algorithm that generated the source tree.
- The *source* has a node that already exists in the *target*. In this case, the *source* node's attributes are updated similarly as case 1.
- The *source* has a node that does not exist in the *target*. In this case, the node is added to the *target*, similar to case 2.

By keeping track of which edges/nodes are part of which algorithm output tree, the visibility of each tree can be toggled separately in the UI while using the same graph instance (see requirements E8 and D3 in section 3).

4.5.2 Optimisations

This section discusses several graph-state-related optimisations to improve the application's responsiveness per requirements E3 and E4 in section 3.

Optimising canvas re-renders It was observed that the initial implementation of the merging process described above was a major bottleneck. Our investigation revealed this was due to how updates to the graph's internal state were handled. For each operation, such as updating node or edge attributes, Graphology will trigger an event, which will cause a re-render of the entire graph on the canvas. This makes sense, except for quick successive operations in loops. In the latter case, the browser will freeze due to the number of re-renders. An issue was raised in the Sigma.js GitHub repository[17].

In the meantime, a workaround was developed that prevents events from being triggered for updates to node and edge attributes. Events will still get triggered for adding or removing nodes and edges in batches, but at least no more freezes were noticed.

Edge optimisation Another optimisation was to eliminate unnecessary edges from the graph instance. The input to the MST algorithm is a complete graph with $\frac{n(n-1)}{2}$ edges. Hence, the global graph instance was initially maintained as a complete graph by explicitly adding edges between all nodes. These edges were hidden from the user by setting a visibility attribute. However, for graphs with more than 300 nodes, rendering started to slow down significantly. It was discovered that the number of edges and their attributes were the culprits. Attributes such as edge colour and thickness were duplicated across many edges. Moreover, the render function would still iterate over all edges, including the hidden ones.

To address this, redundant edge attributes were removed, and the global graph instance was modified to store only the edges that are visible to the user at a given time. Consequently, the MST algorithm was modified to take sparse graphs as input and compute edge weights on the fly.

4.6 Software design aspects

This section discusses how best practices were applied to the application's software design. Aspects such as immutability, separation of concerns, modularity, type safety, and extensibility will be discussed.

4.6.1 Immutability of the graph instance

As stated before in section 4.2, the state that contains the graph instance is immutable. In other words, the graph instance itself can be modified by adding or removing nodes and edges, but not the state that contains the graph instance (e.g. by creating a new graph instance). Even though this is generally not recommended in React (React.dev 2024), it is a constraint that arises from using Sigma.js with React (react sigma 2024). Yet, for our scenario, this approach has the following benefits:

- Sigma.js handles its own synchronisation of the view and the state instead of relying on the React framework. Consequently, only the canvas, not the whole application, is re-rendered when the graph's internal state changes. This is a major benefit as re-rendering the canvas is significantly less expensive than re-rendering the DOM.
- It guarantees that all parts of the application will always have a consistent view of the graph instance's internal state (since the reference of the graph instance is never changed). In React, in particular, an inconsistent state due to batch updates can be a major source of bugs for applications with complex user interactions.

The only drawback is that components (other than the canvas) that depend on the graph's internal state are not automatically re-rendered when the graph state changes. As a workaround, a publish/subscribe design pattern[18] has been implemented for the components that need to react to changes in the graph's internal state.

[17] https://github.com/jacomyal/sigma.js/issues/1046
[18] https://en.wikipedia.org/wiki/Publish-subscribe_pattern

4.6.2 Separation of concerns

The code has been structured using well-defined boundaries to keep the application's code maintainable, readable, and testable. For example, there exists a clear separation between algorithm implementations, control logic, and the presentation logic (i.e. UI components). This makes it straightforward to test each part in isolation and to change implementations later if needed. On top of that, we use a React pattern called "hooks", which provides a way to separate UI and control logic. Hooks are written as normal functions that can be used to abstract and potentially reuse state-related logic across components. In other words, they can be "hooked" to a component to provide a defined state and functionality.

4.6.3 Modularity

The code for this project extensively uses modules. Each module is essentially a file that contains related functions, variables, and other resources. Modularity is further facilitated by React, a component-based framework. Each React component can be viewed as a Lego block that can be combined with other components to build a progressively complex system.

4.6.4 Type safety

The application is written in Typescript for compile-time type safety. Typescript facilitates the use of interfaces to define the data structure passed around the codebase. This makes it easier to reason about the code and see what happens in a given part of the codebase. The reader is referred to section 4.1.2 for a more extensive discussion on Typescript.

4.6.5 Extensibility

We are very well aware that the application has the potential to be extended with new algorithms and features, even becoming a general-purpose tool for visualising and analysing graphs. Therefore, the application has been designed with extensibility in mind from the beginning. An example is given in section 4.3 of how the application uses generic interfaces to abstract the algorithm implementations. Moreover, by using technologies such as WebAssembly, it is now possible to extend the application with new algorithms in languages other than JavaScript. C, C++, and Rust are a few examples of languages that can be compiled to WebAssembly.

4.7 Licensing

Except for the GeoSteiner library, all software dependencies are licensed with *permissive* open-source licenses, such as MIT, ISC, Apache 2, and BSD. These are non-restrictive licenses that only require attribution. In our case, this is satisfied by using a so-called *package.json* file, which specifies the location and version of each dependency. However, the GeoSteiner library is not distributed with a typical open-source license (CC BY-NC 4.0), and neither is it available in package managers or a public repository such as GitHub. Instead, it can be downloaded from the website as a zip file. The license[19] states that technical modifications are allowed as long as the authors are attributed, an indication of modification is provided, and a link to GeoSteiner's license is included. Section 2(a)(4) of the license also states that technical modifications do not constitute adapted material, thus voiding the need to use a similar license in our project.

Attribution for the GeoSteiner library has been added both to the project's GitHub repository[20] and the web application[21].

[19] `http://www.geosteiner.com/LICENSE`
[20] `https://github.com/pietert2000/visual-steiner`
[21] `https://steinertree.com`

4.8 Summary

This chapter has presented the software design and implementation of the visualisation tool. We began by justifying the choice of a web-based architecture and the selected technology stack, including the use of WebAssembly to run the GeoSteiner C library in the browser at near-native performance. The architecture of the application was then discussed, showcasing the different components and parts of the application and how they interact with each other.

The implementation of the algorithms was examined, with particular focus on the integration of the GeoSteiner library through WebAssembly. This integration represents a significant contribution, as it is the first successful compilation of the GeoSteiner implementation to WebAssembly, making the algorithm accessible to any programming language that supports WebAssembly. The chapter also discussed how the graph state is managed and the various optimisations that were implemented to improve the application's responsiveness.

Finally, we considered licensing and important software design aspects such as immutability, separation of concerns, modularity, type safety, and extensibility, which were crucial for developing a maintainable and extensible application. The next chapter will demonstrate the UI and its features in detail.

5 Visualisation and product features

The core problem solved by the application[1] is visualising Steiner Minimal Trees and Minimum Spanning Trees together. In this section, a high-level overview of the application's design is given, followed by a discussion of the design and interaction details of specific features, such as:

- Creating and editing graphs: This will discuss the ways in which a user can create or import graphs, and how the canvas can be used for editing existing graphs.
- Algorithmic visualisation: This will discuss how algorithmic results are visualised and how the user can customise these visualisations.
- Visualisation export: This will discuss how the results can be exported from the application for use in other applications.

5.1 High-level design

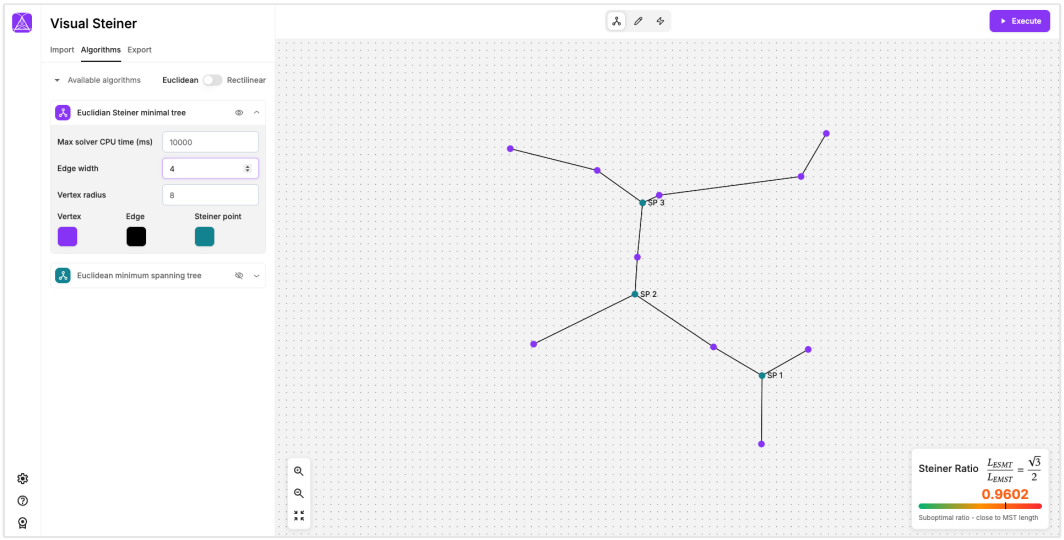


Figure 5.1: High-level design of the application.

Figure 5.1 shows the application's design. It features a visual separation between the canvas and the controls (i.e. the sidebar), according to the architecture shown in figure 4.1. The canvas takes up most of the screen width since it is expected that most of the user's attention and time will be spent there. The "controls" on the left are partitioned into three sections:

- **Import**: This section displays information about the currently active graph and allows the user to create a new graph, either by importing from a file or by generating a random graph with a specified number of terminals (see section 5.2).

[1] `https://steinertree.com`

- **Algorithms**: This section contains customisation options for the visualisation of the algorithms (see section 5.3).
- **Export**: This section allows the user to export the visualisation results for use in other applications. (see section 5.4)

As per requirement E1 (see section 3), we have focused on a clean and intuitive design. This is achieved by using a consistent contrastive colour schema, incorporating white space, and using familiar icons to indicate the purpose of the controls.

For example, the three buttons in the centre-top of figure 5.1 are used to switch between the three "modes" of the canvas. The left-most "mode" is to view the graph, where no graph editing is allowed, the middle button is for the edit mode. In there, users can manipulate the displayed graph by adding, moving, or removing nodes. The difference with this mode and the live edit mode (3rd button) is that in the case of the latter, the optimal trees are immediately recomputed and rerendered and the Steiner ratio is updated when changes are made to any of the terminals.

Care has also been taken to ensure the primary action is always visible to the user. For instance, the primary action in figure 5.1 is the *Execute* button on the top right. Clicking it will recompute the SMT and MST of the currently active graph and display the results on the canvas.

5.2 Creating and editing graphs

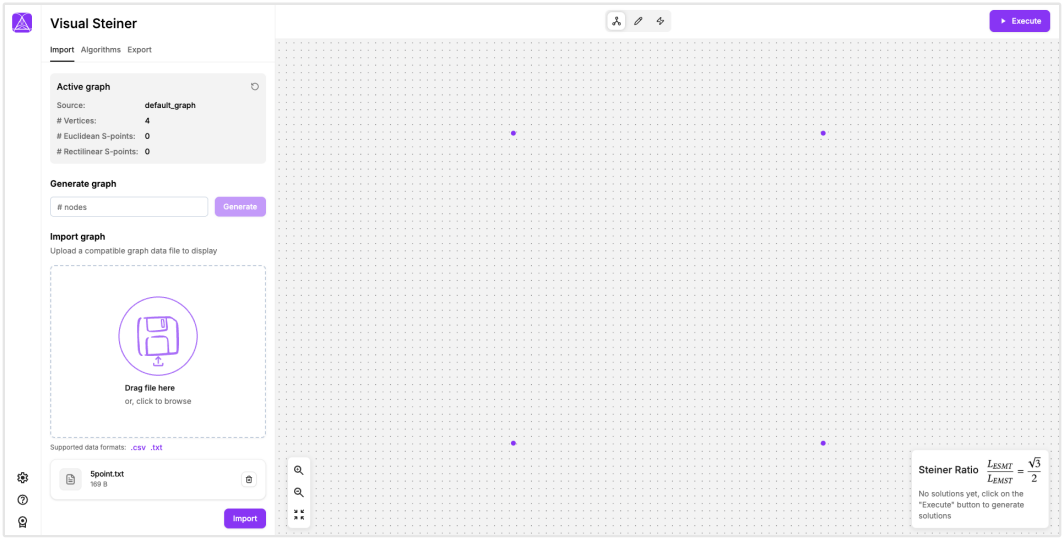


Figure 5.2: *Creating a graph instance by importing from a file or generating a random graph.*

As per requirements E5 and E6 (see section 3), the user can create or edit graphs without navigating to a different page. The contents of the import tab can be seen in figure 5.2. The top section displays information about the active graph instance. The application comes with a default 4-point example graph selected by default.

5.2.1 Creating/importing graphs

New graphs can be created by importing from a file or by generating a random graph with a specified number of terminals. File formats, such as CSV and TXT, containing a list of 2D coordinates are supported. The user can also download an example file to familiarize themselves with the format. Again, care has been taken to ensure the primary action is visible. In the example

in figure 5.2, the *Import* button is highlighted as a file has been uploaded and is ready to be imported.

5.2.2 Editing graphs

The user can edit the active graph by toggling the *Edit* mode in the header above the canvas, which will make the graph in the canvas editable. As all supported algorithms take an undirected, complete graph as input, there is no need for supporting edge drawing. Hence, the available edit actions have been limited to adding, moving, and removing nodes.

5.3 Algorithmic visualisation

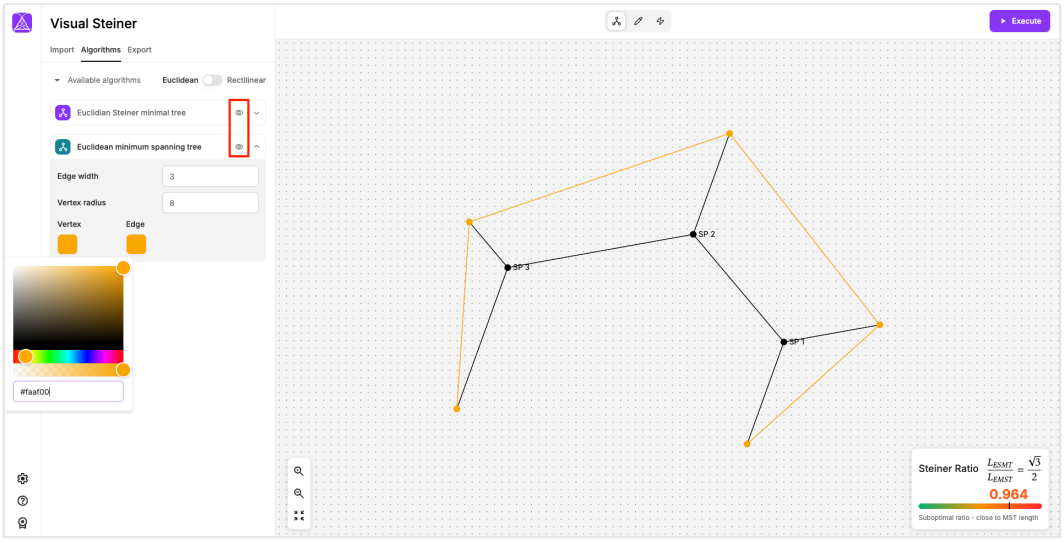


Figure 5.3: *Algorithmic visualisation customisation options showcasing the "layering" approach.*

The software supports the visualisation of the Euclidean and rectilinear variants of the SMT and MST. The user can seamlessly toggle between the Euclidean and rectilinear metrics by a mere button press. (see the toggle just above the red rectangular selection in figure 5.3)

5.3.1 Layering

As is stated in requirement E8, being able to compare the tree structure and length of SMTs and MSTs is an essential requirement. This led us to consider a "layering" approach for visualising the algorithmic results. Different layers can be toggled on and off independently, and each layer can be customised with its own set of options (requirements D1 and D3 in 3). Figure 5.3 shows how each algorithm is modelled as a layer and each layer has its own set of customisation options. In this case, both the MST and SMT trees are toggled on, but each has different styles applied. The yellow edges are the MST edges, and the black edges belong to the SMT.

Sometimes, edges or nodes of the two trees overlap. When this happens, the hierarchical order of the algorithm layers is used to determine which styles take precedence. In figure 5.3, the terminals of the SMT and MST overlap (excluding the Steiner points), but as the STM layer is hierarchically above the MST layer, the terminals are painted with the color that was set in the SMT layer.

5.3.2 Visualising the Steiner ratio

As a reminder, the Steiner Ratio is defined as the smallest ratio of the length of the Steiner Minimal Tree to the length of the Minimum Spanning Tree. A canvas widget has been added (see bottom left of figure 5.3) to visualise the Steiner ratio of the currently active graph instance. This feature is helpful to get an intuition (or potentially find counter-examples) for the Steiner ratio conjecture given by Gilbert and Pollak (1968). By dynamically updating the tree and Steiner ratio when the user edits the graph, the user can quickly see how the Steiner ratio changes as the tree changes.

5.4 Visualisation export

The application's usefulness is further enhanced by the ability to export the active graph instance for use in other applications. The export view, shown in figure 5.4, allows the user to export the currently active graph to either an image, like PNG or JPG, or text format, like GEXF[2].

Image-based exports are useful for embedding the visualisations in other documents. When exporting to an image, the user can choose between the customised style set by the user or the default style inspired by how trees are typically visualised in literature. Further, each algorithm's output tree can be exported separately, with the option to render only the edges, nodes, or both. It is also possible to export to GEXF which is an XML-based encoding for graphs. It is supported by a lot of graph-based software and its format is easy to understand.

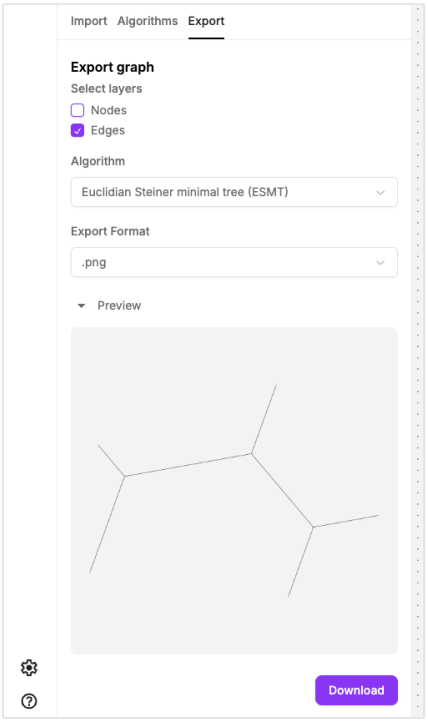


Figure 5.4: *Export options in the visualisation tool*

[2]https://gexf.net/

6 Analysis of the results

As stated in the introduction, the project aimed to develop a fast, accessible, user-friendly interface for visualising Steiner Minimal Trees and some of their properties. So far, only the design and implementation of the product have been discussed (see sections 4 and 5), but in this section, the product result is evaluated for its feasibility, how it compares to existing tools, and its potential implications.

6.1 Product feasibility

When the technology stack and the client-only architecture initially were chosen, it was based on the assumption that GeoSteiner would be able to run fast enough through WebAssembly to meet the performance requirements and that loading the WASM module on each application page load would not be a problem in terms of bandwidth usage. The following section will show that these assumptions were correct and that the approach was feasible.

6.1.1 Performance

Requirement E4 states that the product should be able to compute the (Euclidean or rectilinear) Steiner Minimal Tree and Minimum Spanning Tree in under 1 second for up to 200 terminals. By testing the product[1] with multiple random graphs of 200 terminals (using the "generate graph" button in the "import" tab), it was found that the app was consistently able to compute and render the Steiner Minimal Tree and Minimum Spanning Tree of a graph in under a second, meeting the initial design constraints. Naturally, the exact timing will vary depending on the hardware used. Instead, the reader is encouraged to test the product for themselves. Moreover, most use cases are expected to be with graphs much smaller than 200 terminals. So, the product should perform well for most use cases, even on less capable hardware. For example, one use case could be checking counter-examples to the Euclidean Steiner ratio conjecture (see section 6.3.2). As the instance size increases, it becomes increasingly hard to get anywhere near the conjectured lower bound (as shown in Table 3 in Winter and Zachariasen (1997)). Hence, instances with 3-10 terminals are most useful in this scenario.

The impressively fast performance of the GeoSteiner algorithms has also enabled dynamic tree re-computation when the user edits the graph, meeting requirement E7 (section 3). This feature is only usable and responsive if the computation happens fast enough since long feedback loops will make the UI feel laggy.

6.1.2 Client-only architecture

In determining the feasibility of the client-only architecture, the size of the GeoSteiner WASM module was also considered. For example, if the module is too large, it might take too long to load the application, thus failing to meet the requirement of a easily accessible tool. As it turns out, the GeoSteiner WASM module and its required glue code is around 400kB and with Gzip

[1] `https://steinertree.com`

compression (which is enabled by default in most browsers), it is around 170kB. It is clear that this size is reasonable and does not pose a problem for the product's accessibility.

From the two sections above, it can be concluded that:

- The GeoSteiner algorithm despite its exponential complexity, is fast enough in practise to be used for visualisation purposes.
- Running the GeoSteiner algorithm in the browser through WebAssembly is feasible and combines the benefits of native performance with platform independence.

6.2 Comparison with existing tools

The previous section demonstrated the feasibility of our approach to meeting the performance and accessibility requirements. This section will discuss how the product compares to existing tools.

Section 2.5 listed existing tools and concluded that they lacked a smooth and user-friendly interface, visualisation options, flexible import/export capabilities, and platform independence. A summary of the comparison of features between our product and the existing tools is given in table 6.1. For a full discussion on the product requirements and features, the reader is referred to sections 3 and 5. It is furthermore worth noting that, besides comparing the product to the existing tools, table 6.1 also implicitly makes clear that all requirements have been met by the product.

Req.	**Feature**	STV[2]	E3D[3]	ST[4]	**Our product**
E1	Well-designed, clean, and intuitive interface	✗	✗	✓	✓
E2	Support for the euclidean and rectilinear SMT and MST	✓	✗	✗	✓
E3	Canvas with large graph, zoom, and pan support	✗	✗	✗	✓
E4	Computation of 200 nodes < 1 sec	✗	✗	✗	✓
E5	Support for flexible graph creation options, including random generation and external files	✗	✗	✗	✓
E6	Graph can be manipulated on the canvas	✓	✗	✓	✓
E7	Dynamic tree updates	✓	✗	✓	✓
E8	Support for comparison of MST/SMT tree structure and length (Steiner Ratio)	✓(partial)	✗	✗	✓
E9	Deployed as public web app	✗	✗	✗	✓
D1	Visualisation can be customised	✗	✗	✗	✓
D2	Export	✗	✗	✗	✓
D3	Toggeability of algorithms	✓(partial)	✗	✗	✓
D4	Modular and extensible software architecture	✗	-	✗	✓

Table 6.1: *Feature comparison between the existing tools and the end product. See the requirements section 3 for more details.*

[2] `https://github.com/Keydrain/Steiner-Tree-Visualisation`
[3] `https://github.com/Kallel-Abd/ESteiner-3D`
[4] `https://github.com/Dawkey/Steiner-Tree`

6.3 Implications

In this section, the implications of the result will be discussed. As shown in figure 4.1, the application is split into two main components, the GeoSteiner WASM module and the UI. Each component has its own set of implications, which will be discussed in the following sections.

6.3.1 Wider use of the GeoSteiner WASM module

As mentioned in section 4.4, the GeoSteiner WASM module is a portable byte code representation of the GeoSteiner algorithms library implemented in C. Due to WASM's portability and highly efficient byte code organisation, it is now feasible to use the GeoSteiner algorithm in any language that supports WASM. In other words, integrating with the GeoSteiner algorithms became much easier. This opens the door for usage of the GeoSteiner algorithms in other applications, such as network analysis libraries[5]. These libraries often rely on approximate algorithms as it is hard to implement an efficient version of an exact Steiner Tree algorithm.

Moreover, as the GeoSteiner WASM module is entirely independent of the UI, it could be used in other (open source) graph visualisation tools, such as GeoGebra [6].

6.3.2 Using the visualisation tool in research

The visualisation tool may also have several uses in research. First, its features will be useful for gaining a better intuition for the Steiner Tree and its properties. For instance, our implementation can smoothly recompute the Steiner Minimal Tree in response to user input, such as adding, moving, or removing points. This enables the user to observe how the Steiner ratio changes as a function of the configuration of the terminals. It also allows verifying potential counter-examples to the (Euclidean) Steiner ratio conjecture in real time.

[5] https://networkx.org
[6] https://www.geogebra.org/geometry

7 Conclusion

7.1 Summary

Problem Statement The Steiner Minimal Tree (SMT) problem and the Minimum Spanning Tree (MST) problem belong to the class of network optimisation problems. The SMT problem is similar to the MST problem but allows for the addition of Steiner points to further reduce the tree length. Despite being simple to state, the SMT problem is NP-hard. Consequently, its associated exponential complexity has limited the feasibility and practicality of visualisation tools, in contrast to the MST problem, which has efficient algorithms and plenty of visualisation options. This lack has also limited the ability of researchers and students to build an intuition for the problem, particularly in investigating the Steiner ratio (still an open problem in the Euclidean but not in the rectilinear case), which is the smallest ratio of the length of the Steiner Minimal Tree to the length of the Minimum Spanning Tree.

This dissertation presented the development of a web-based visualisation tool for the Euclidean and rectilinear variants of the Steiner Minimal Tree and Minimum Spanning Tree. The tool leverages GeoSteiner, the fastest known algorithm for the SMT problem, and exploits state-of-the-art techniques for optimal performance. The result is a publicly available client-only web app[1] that can be run on any device with a modern web browser.

The **key contributions** of this work include:

- The first successful (open source) compilation of the GeoSteiner implementation to WebAssembly. With this, the GeoSteiner algorithm is now easily accessible to any programming language that supports WASM.
- The first publicly available online visualisation tool for the Steiner minimal and Minimum Spanning Trees.
- A fast and interactive interface allows the user to compute and visualise the Steiner Minimal Tree and Minimum Spanning Tree of any graph instance and compare them in structure and length. Despite the theoretical exponential complexity of the SMT problem, the tool can compute and display the results for graphs with up to 200 terminals in sub-second times.
- A dynamic tree-updating feature enables the user to observe how the Steiner ratio changes in real time as the graph is edited.

Implications The abstraction of the GeoSteiner implementation to a WebAssembly module makes integration into new programming languages and applications straightforward. This opens new possibilities and could lead to the broader adoption of exact algorithms instead of approximations in open source network analysis libraries and other geometric computing or visualisation tools. Moreover, researchers could use the visualisation tool to explore the Steiner Tree and its properties in a new and interactive way. In particular, the dynamic tree-updating feature has dramatically reduced the feedback loop for investigating potential counter-examples to the Steiner ratio conjecture.

[1] `https://steinertree.com`

7.2 Future work

This section discusses potential future work for the GeoSteiner WASM module and the visualisation tool.

7.2.1 GeoSteiner WASM module

The GeoSteiner WASM module could be extracted into a separate package and distributed as a stand-alone library in the JavaScript ecosystem through NPM. This would make it easier to integrate it into other web-based applications, such as GeoGebra. Different language bindings could also be created for the GeoSteiner WASM module and distributed as separate packages in the package manager of choice for each programming language.

7.2.2 Visualisation tool

The visualisation tool has the potential to be extended with new graph algorithms and could serve as a general-purpose tool for visualising and analysing graphs. In addition, several optimisations could be applied to improve the tool's performance even further. For example, the algorithms' computation could be moved to another Web-worker thread to avoid blocking the browser's main thread. The UI itself could also benefit from improvements, such as adding a responsive design for mobile devices and canvas support for bent edges, which is especially relevant for visualising the rectilinear Steiner Minimal Tree.

A Complexity derivation for Melzak's algorithm

Below, we derive the formula given by Winter and Zachariasen (1997) for the number of full topologies $F^*(n)$, by using Property 7 in Gilbert and Pollak (1968). However, the literature skips some of the steps and the author's contribution therefore is to provide an expanded derivation of the formula, adding extra details where felt necessary.

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 unlabeled, there are $\binom{n}{s+2}$ ways to choose the terminals from which the full topology will be constructed. Each chosen set of terminals has $f(s)$ 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, 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 this 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).

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

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

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

Finally,

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

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.

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

B Codebase structure

Asset	Type	Description	LOC
src/components/*	React	Collection of reusable React components based on Shadcn/UI.	1030
src/features/*	React	Features are independent modules that have a specific scope and their own set of components, hooks, types, and tests. Features include the canvas and the sidebar.	2210
src/hooks/*	React	Collection of global custom function hooks for functionality such as pub-sub, debouncing, sorted lists, etc.	68
src/providers/*	React	Collection of scoped state management providers	231
index.html, src/main.tsx, src/App.tsx	React	Entry point for the web application.	53
src/lib/*	Typescript	Contains all business logic for the application, such as algorithms, data structures, utilities, input parsers, etc.	585
geosteiner/*	C	Contains a copy of the open source GeoSteiner library with some modifications for compatibility with WASM.	n/a
wasm/*	WASM	Contains the setup for compiling the GeoSteiner library into a WASM module and the simplified GeoSteiner API for the calling code.	79
**/tests/*	Typescript	Integration/unit tests across the codebase.	301
public/*	Static assets	Static assets such as images, icons, examples, and the WASM module.	N/A

Table B.1: *Repository structure and distribution of code. Only the important folders are listed. The LOC values are accurate as of 28/03/2025.*

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.

Bopp, K. (1879), 'Üeber das kürzeste verbindungssystem zwischen vier punkten.'.

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*

Cavalieri, B. (1647), 'Exercitationes geometricae sex.'.

Courant, R. and Robbins, H. (1941), 'What is mathematics?'.

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*

Emscripten (2025), 'Emscripten portability'.
URL: *https://emscripten.org/docs/porting/guidelines/portability_guidelines.html*

Fermat, P. (1891), 'Fermat, p. 1891. oeuvres, vol. 1.'.

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*

Hanan, M. (1966), 'On steiner's problem with rectilinear distance'.

Heinen, F. (1834), 'Über systeme von kräften.'.

Hoffmann, E. (1890), 'Über das kürzeste verbindungssystem zwischen vier punkten der ebene.'.

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

Hwang, F. K. and Richards, D. S. (1992), 'Steiner tree problems', *Networks* **22**(1), 55–89.
URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230220105

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'.

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/

react sigma (2024), 'Introduction | react sigma'. [Online; accessed 28-March-2025].
URL: https://sim51.github.io/react-sigma/docs/start-introduction/

React.dev (2024), 'Updating objects in state'. [Online; accessed 28-March-2025].
URL: https://react.dev/learn/updating-objects-in-state

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/

Simpson, T. (1750), 'The doctrine and applications of fluxions.'.

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

Torricelli, E. (1919), 'De maximis et minimis. in opere di evangelista torricelli, ed. g. loria, and g. vassura. faenza, italy.'.

Warme, D. M., Winter, P. and Zachariasen, M. (2022), 'Geosteiner manual'.
URL: http://www.geosteiner.com/geosteiner-5.3-manual.pdf

WebAssembly (2025), 'Webassembly faqs'.
URL: https://webassembly.org/docs/faq/

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

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*