University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Visualising Steiner Minimal Trees

Pieter van Tuijl

February 21, 2025

Abstract

TODO: Add abstract here

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

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

Acknowledgements

TODO: Add acknowledgements here

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 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 amount of total distance. This is a classic problem in graph theory, and the solution is known as the minimum spanning tree (abbreviated as MST). There exists well-established and fast algorithms for finding 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 that they are built outside the cities. You are allowed to use as many such junctions as needed, as long as the total distance of the road network is minimal. The solution to this problem is known as the Steiner minimal tree (abbreviated as SMT) (Melzak 1961). In a nutshell, the Steiner minimal tree achieves a total length that is $\leq$ than the length of the MST, by adding extra points called Steiner points. It is worth noting that besides roads and telecommunication networks, the Steiner problem has applications in other fields, such as wire routing on printed circuit boards.

1.1.2 Steiner minimal tree problem is NP-hard

Though the constraint or the description of the SMT problem is easy to state and understand, it is actually quite hard to find the optimal solution. 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 the SMT problem 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 serious limits 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 serves as 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, despite the exponential nature of the problem (Juhl et al. 2018). However, the available tools for visualising the SMT are still very limited in functionality, not easily accessible, and often lack an intuitive, user-friendly interface. This project aims to fill that gap.

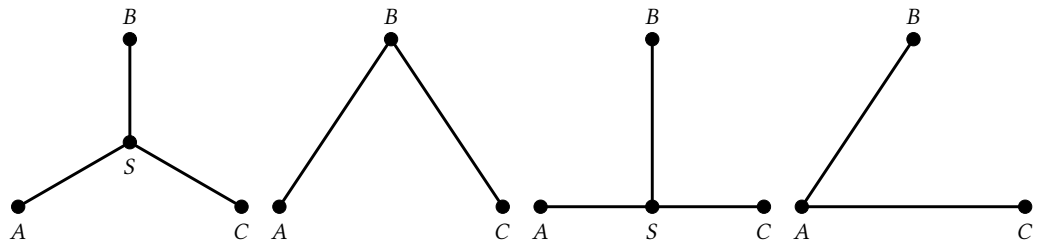
1.2 Project aims and requirements

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

Aims The aim of this project is to develop a fast, accessible, and user-friendly interface 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 both visual (e.g. .png, .jpg) and text-based formats (e.g. .txt, .csv).

It is important to note that the Steiner tree and minimum spanning tree problems are defined for any metric space, but in this project we will only consider the Euclidian and rectilinear (also known as the Manhattan or taxicab distance) variants. The interface will provide the user with the option to toggle between the two variants. Figure 1.1 shows the difference between the two variants for a 3-point example instance.



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

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

Access the end product here

`https://steinertree.com`

1.3 Dissertation outline

TODO: fill this section when all other sections are written

2 Background

2.1 Historical context

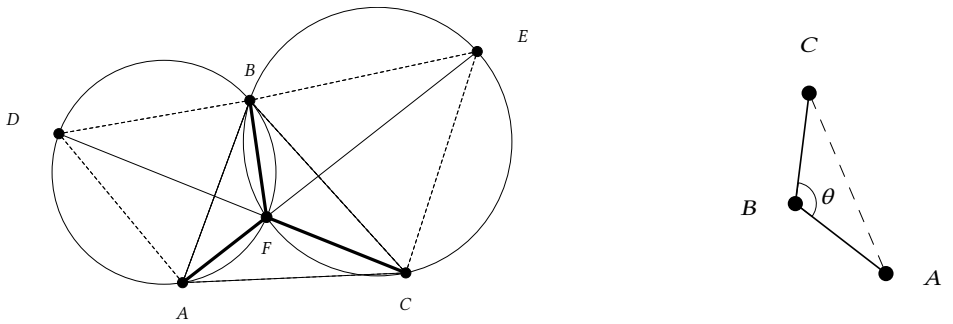
The Euclidean Steiner problem has a long history, with roots in the 17th century. The following section will provide a summary of the history as expertly given by Brazil et al. (2014).

2.1.1 Fermat–Torricelli (3-point) problem

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

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

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



(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.*

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

2.1.2 Simpson lines

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

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

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

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

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

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 are still valid even if the problem is extended to higher dimensions.

Street network problem Choquet compares the Steiner tree problem to a network of cities connected by roads where there are no junctions available

between the roads except outside the cities. (Another variant is the case where junctions are allowed inside the cities, which is the minimum spanning tree problem). 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. As a consequence, the generalised Fermat-Torricelli problem (i.e. the street network problem) has since been known as the Steiner tree problem in the literature despite the fact that it is unclear whether Steiner ever considered this generalisation.

Gilbert and Pollak's seminal paper Finally, Gilbert and Pollak (1968) provided a comprehensive survey of the Steiner tree problem, and derived many of its properties that have since been the basis for much of the modern literature on the topic. (more on this in 2.2.2)

Surging popularity in the sixties From the sixties onwards, the Steiner tree problem has seen a surge 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.

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 Euclidian and rectilinear Steiner minimal tree problems and lists some of their properties.

2.2 Euclidian Steiner Minimal Tree

2.2.1 Formal definition

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

More formally:

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

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

2.2.2 Properties

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

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

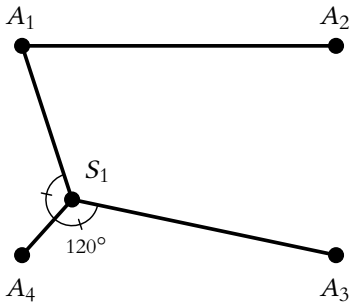
Next, we clarify the following terms, used across the literature and first defined by Gilbert and Pollak (1968):

- A topology is defined as the adjancy matrix which only specifies the connections between points (i.e. how points are interconnected). 2.2d displays three different topologies for the same set of points, where 2.2b and 2.2c share the same topology.
- A full topology has $n - 2$ Steiner points. A Full Steiner Tree (FST) is the relatively minimal tree for a full topology, where the terminals are leaves and the Steiner points are internal nodes with degree 3.
- A relatively minimal tree (RMT) is the minimal tree for a given topology. All figures in 2.2 are RMTs except for 2.2b. A relatively minimal tree is obtained when small pertubations (displacements) of the Steiner points no longer result in a smaller tree.
- A Steiner tree (ST) is always a RMT, but the converse does not hold. For instance, 2.2a is a RMT but the angle $\angle S_1A_1A_2 = \theta < 120°$ and therefore does not satisfy the angle property.
- A Steiner minimal tree (SMT) is a Steiner tree and is minimal for all its vertices (i.e. terminals + Steiner points). In other words, the SMT is the minimal tree across all possible topologies for a given set of points. See Figure 2.2d for an example.

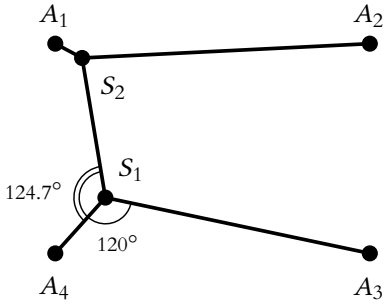
Given a topology, its Steiner tree (if it exists) can be seen as a local minimum. Only by enumerating all possible topologies and their corresponding Steiner trees can the global minimum (i.e. the SMT) be found. This concept forms the basis of the algorithms we will discuss in the algorithms section.

2.2.3 Steiner ratio

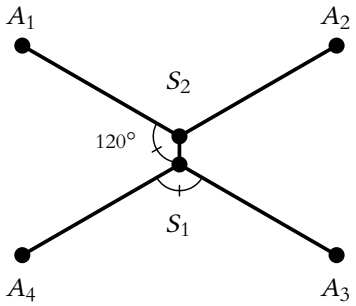
An ESMT without Steiner points is equivalent to the Euclidian minimum spanning tree (EMST). Formally, the EMST is defined as the network that connects a set of points N, such that the sum of the edge weights is minimal, where the edge weights are the Euclidean distances between the points.



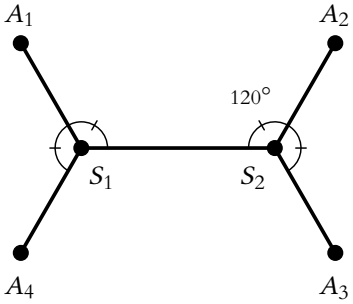
(a) *Relatively minimal tree for its topology*



(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.*

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

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

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

2.3 Rectilinear Steiner Minimal Tree

2.3.1 Formal definition

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

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

2.3.2 Properties

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

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

Some properties apply to both the ESMT and RSMT, such as the max number of Steiner points and edges in a Steiner tree (property 3).

2.3.3 Steiner ratio

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

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

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

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

2.4 Steiner tree algorithms

The Steiner tree problem is easy to understand but hard to actually solve. Garey et al. (1977) proved that the Euclidian and Rectilinear Steiner tree problems

are NP-hard. This means that there exist no polynomial-time algorithms for these problems (unless P = NP). However, approximation schemes can be used to find near-optimal solutions in a reasonable amount of time. In this project, we have chosen to use exact algorithms in order to demonstrate, that despite the theoretical hardness of the problem, the actual performance in practise is suprisingly good.

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

- The Melzak algorithm Melzak (1961) which is largely based on the iterative algorithm by Gergonne (Brazil et al. 2014) (see 2.1.3). This algorithm is essentially a brute-force approach and is therefore not practical for large instances.
- The GeoSteiner algorithm developed by Winter and Zachariasen (1997). This algorithm uses ideas from the Melzak algorithm, but it is much faster and efficient due to the use of geometric properties for pruning the search space, efficient data structures, and other clever optimisations.

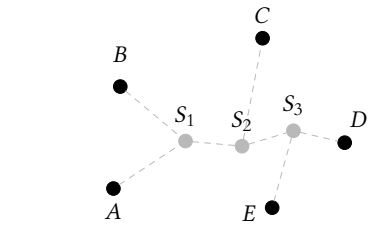
2.4.1 The Melzak algorithm

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

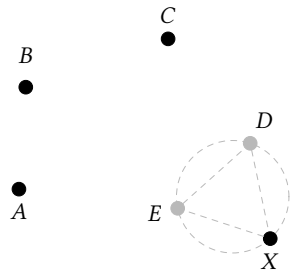
Formal definition Before illustrating the algorithm with a practical example, we use the following definition from (Marcus Brazil 2015): Let $\mathcal{T}_n$ be a full topology with n points and $n-2$ Steiner points. Let S_i be a Steiner point in $\mathcal{T}_n$ which is adjacent (connected by an edge) 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 some other point v. By definition, the length of the Simpson segment between X and S_i is $|S_iA| + |S_iB|$. Hence, A, B, and S_i can be replaced by X with a direct line segment to v, without increasing the tree length. This process of *merging* is repeated until the base case, $\mathcal{T}_2$, is reached.

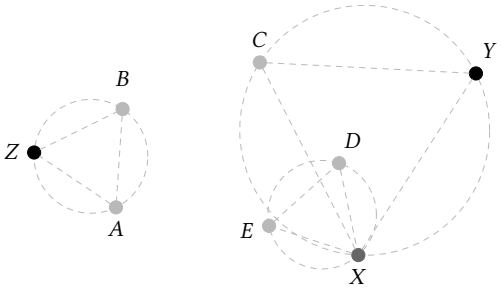
Next, the FST can be *reconstructed* by backtracking the merging steps. Suppose the FST of $\mathcal{T}_{n-1}$ is defined. This means that the location of v also will be known. There exists an FST for $\mathcal{T}_n \iff$ there exists an FST for $\mathcal{T}_{n-1}$ and the Simpson segment between v and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



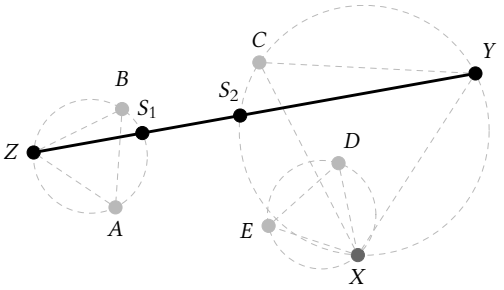
(a) *Initial full topology, Steiner points are randomly placed for visualisation purposes*



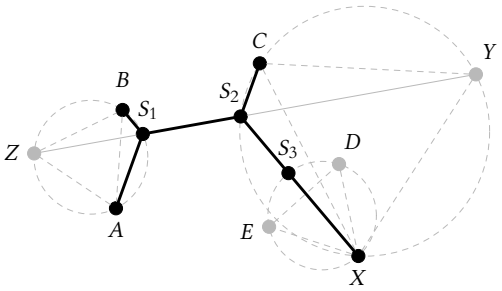
(b) *Start of merging. Replacing (E, D, S_3) with X*



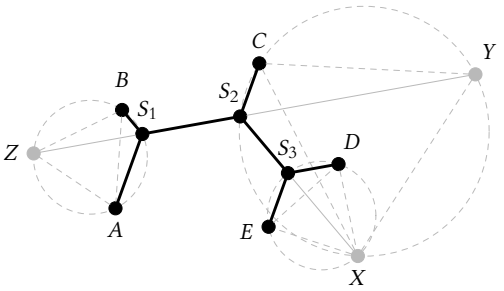
(c) *Replacing (X, C, S_2) with Y, and (A, B, S_1) with Z, results in the n = 2 base case.*



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



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



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

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

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, in order to visualise all the connections.

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

The FST is subsequently built up by backtracking the merging steps. By drawing a Simpson segment $\overline{ZY}$, the location of S_1 and S_2 are found (figure 2.3d), after which $\overline{ZY}$ is deleted and the edges AS_1, BS_1, S_1S_2, CS_2, with the Simpson segment $\overline{XS_2}$ are added (figure 2.3e). Subsequently, the position of S_3 is the intersection between $\overline{XS_2}$ and the Steiner arc $\overset{\frown}{ED}$. 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 $Z = e_{AB}$. The original Melzak algorithm, however, accounts for the fact that each equilateral point e_{xy} can be placed on either side of the segment $\overline{xy}$. Hence, the algorithm has a worst-case complexity of $O(2^n)$. An improved, $O(n)$ version of the algorithm was proposed by Hwang (1986), which picks the correct side for each equilateral point e_{xy} in constant time.

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

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

For the curious reader, a derivation of this formula is given in appendix A.

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

2.4.3 The GeoSteiner algorithm

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

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

However, instead of enumerating all possible full topologies for all subsets of n terminals, it only enumerates the equilateral points that are used in the construction of the FSTs (Winter and Zachariasen 1997). Equilateral points are often repeated

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

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

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

2.5 Existing visualisation tools

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

2.5.1 Steiner-Tree-Visualisation (STV)

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

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

2.5.2 ESteiner-3D (E3D)

E3D is another Python-based tool developed by Abd (2024). It is a program that can be used to find the Euclidian SMT of a graph and supports 2D and 3D graphs. However, use of the software is limited by the fact that no graphical nor

command-line interface is provided. It also does not support the simultaneous visualisation of MSTs and SMTs and the visualisation of the Steiner ratio.

2.5.3 Steiner-Tree (ST)

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

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

3 Requirements analysis

The visualisation requirements given in this section are an expansion of 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 forms the basis for evaluation of the end product.

As earlier stated in the introduction, the project's aim was to develop a fast, accessible, user-friendly interface for visualising Steiner minimal trees and some of their properties. To satisfy this aim, the software must have the following **essential requirements**:

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

E2 The software must support visualisation of the Steiner minimal tree (SMT) and the minimum spanning tree (MST), specifically the Euclidian 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 by generating a random instance.

E6 The user can add, move, or remove nodes of an existing graph instance by direct manipulation of 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 reflected 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 in order 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 colors.

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

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 along with the codebase's structure.
- After that, the implementation of core features is discussed, with a focus on the integration with the GeoSteiner algorithms library.
- Lastly, aspects of the software design, such as modularity, type-safety, separation of concerns, extensibility, and immutability, are highlighted.

4.1 Technologies / Design choices

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 the need for 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 some of the most jarring differences.

4.1.2 Tech stack

The web app is built using **Typescript**[1] (TS) as the main programming language, which is a statically-typed superset of Javascript (JS) that compiles to JS (Bierman et al. 2014). 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 suggests that it is effective at detecting bugs and improving 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. 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.

Other dependencies include:

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

- **Shadcn/ui**[3]: a React component library that provides a set of accessible components, such as buttons, inputs, dropdowns, color-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 of the boxes in terms of required features and quality metrics. Sigma.js is one of the most performant graph drawing libraries[6] because it uses WebGL instead of SVG or the HTML 2D Canvas for rendering. Besides, it makes use of a separate, lightweight and very flexible data model (Graphology) for storing and manipulating graph data, which helps with keeping 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 of nodes.

4.1.4 WebAssembly + Emscripten

As this project uses the GeoSteiner C library for the computation of the Steiner minimal tree, it is necessary to use WebAssembly in order 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.

[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

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, with the exception of Internet Explorer and Opera Mini [8].

Empscripten Emscripten[9] is a compiler toolchain that can compile C and C++ code to WASM. It consists of a designated compiler frontend (emcc) and LLVM backend optimised for the generation of WASM binaries. The output from the Empscripten 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 frontend only needs to load the glue code, which in turn loads and instantiates the WASM module.

Communication between JS and WASM In order 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 instance, it can be viewed as an array of 64-bit floats or an array of 8-bit unsigned chars depending on the type of data that needs to be accessed or written. This avoids the need to manually handle different offsets and strides.

4.2 Software architecture

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

- The views part, which handles user interaction (controls) and the visual appearance
- The algorithms part, which handles the computation of the Minimum Spanning Tree (MST) and Steiner Minimal Tree (SMT) algorithms.
- The graph datastructure 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. In the latter case, the algorithm's

[7] https://webassembly.org/
[8] https://caniuse.com/wasm
[9] https://emscripten.org/

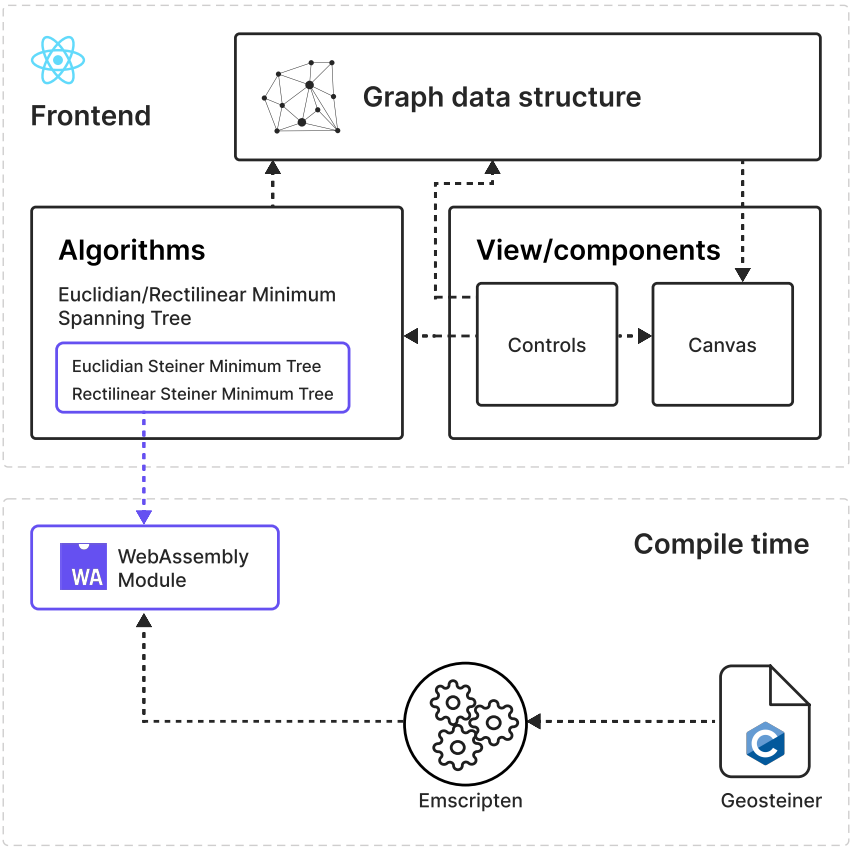


Figure 4.1: *Architecture of the web application. It consists of the frontend (top layer) and the WebAssembly module (bottom layer) compiled with Emscripten from the GeoSteiner C library. The WASM module is loaded by the frontend at runtime. The flow of data and control is indicated by the arrows.*

output is merged with the existing graph datastructure. This is important because it keeps the graph instance immutable (see section 4.6.1). Updates to the graph datastructure are reflected in the the canvas component.

The lower layer contains the WebAssembly module that is compiled with Emscripten from the GeoSteiner C library. The WebAssembly module is dynamically imported by the frontend during runtime.

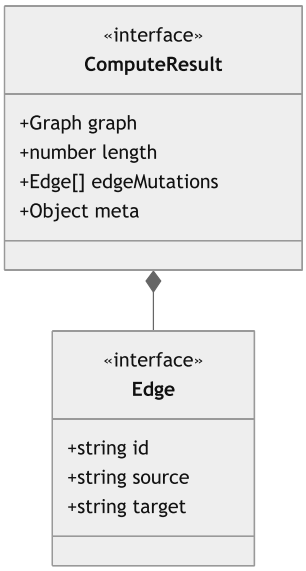
See appendix B for the organisation and code distribution of the codebase.

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 that were added to the tree in order of insertion. Some algorithms, such as the ESMT and the RSMT also pass additional metadata, such as the list of Steiner node identifiers. By coding to an interface as opposed to a concrete implementation, algorithm implementations can be swapped out and new algorithms can be added easily.

Figure 4.2: *Interface of the algorithm computational output.*



The output from the algorithm is used to update the global graph datastructure by a *merge* operation. 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 is dedicated to explaining 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 the field of Steiner minimal trees. 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 implications.

[10]https://github.com/wasmerio/wasmer-python
[11]https://github.com/wasmerio/wasmer-java
[12]https://nodejs.org/

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 frontend.

4.4.1 Compiling GeoSteiner to WASM

The GeoSteiner library consists of the main algorithm implementation and an optimised version of the linear solver, lp_solve v2.3[13], that is used during the FST concatenation stage. In order to compile any library to WASM, it is necessary to also compile all its dependencies to WASM.

It was first verified, that the GeoSteiner library and the solver dependency would be portable to WASM but none of the portability guidelines in Emscripten (2025) were found to be violated. Unfortunatly, several problems were encountered during the first compilation attempts.

Floating point control register (fpcr) settings Depending on the processor architecture, GeoSteiner uses different settings for the floating point registers. However, two settings are common to all architectures:

- 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 set to "round to nearest" by default, so no special rounding mode settings were needed. 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 WASM's limitation, the GeoSteiner code was modified to not rely on floating-point traps. This was achieved by using preprocessor directives to conditionally apply the floating-point traps. TODO: Add a link to the modified code here.

Makefile modifications The Geosteiner library ships with an autogenerated 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, 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.

[13] https://sourceforge.net/projects/lpsolve/files/lpsolve/2.3/
[14] https://emscripten.org/docs/compiling/Dynamic-Linking.html

- The normal compilation process depends on a locally installed version of the GNU libtool[15]. Libtool uses the *ar* and *ranlib* utilities for creating static libraries. Emscripten provides a custom version of these tools that is compatible with WASM, called *emar* and *emranlib* respectively. Unfortunately, it is not possible to use environment variables to set custom paths for the *ar* and *ranlib* utilities used by libtool. Therefore, direct changes to the local libtool installation had to be made.

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

GeoSteiner facade As mentioned in section 4.1.4, functions that are to be called from JS must be must be explicitly exported from the WASM module, 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 to only export a **simplified API**, using the *facade*[16] design pattern. This facade handles all the underlying GeoSteiner-specific logic that is required for the computation of Steiner minimal trees but irrelevant for the frontend. 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*

JS and WASM can share non-primitive data such as arrays by using a shared contiguous memory space (see section 4.1.4). Hence, the facade functions make heavy use of pointers. This requires explicit memory management, which is why 'malloc' and 'free' also had to be explicitly exported besides 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 not aware of any functions that are 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.

As noted before, explicit memory management is needed for the passing and returning of non-primitive data. From the properties of the Steiner minimum

[15] https://www.gnu.org/software/libtool/
[16] https://en.wikipedia.org/wiki/Facade_pattern

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 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 a 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.

At most one edge can exist between any two nodes in an undirected graph. Edges can be queried for 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 in a similar manner 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 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 responsiveness of the application as per requirements E3 and E4 in section 3.

Optimising canvas rerenders It was observed that the initial implementation of the merging process described above was a major bottleneck. Our investigation revealed that 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 rerender of the entire graph on the canvas. This makes sense, except in the case of quick successive operations in loops. In the latter case, the browser will freeze due to the number of rerenders. An issue was raised in the Sigma.js GitHub repository[17].

In the mean time, 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 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 an complete graph by explicitly adding edges between all nodes. These edges were hidden from the user by setting a visibility attribute. For graphs of more than 300 nodes, however, rendering started to slow down significantly. It was discovered that the number of edges and their attributes were the culprit. Attributes, such as edge color and thickness, were duplicated across large numbers of edges. Moreover, the render function would still iterate over all edges, including the hidden ones.

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

4.6 Software design aspects

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

[17] https://github.com/jacomyal/sigma.js/issues/1046

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, there are notable benefits with this approach:

- Unnecessary React renders are avoided as the graph's internal state is only tied to the canvas which is handled by Sigma.js. This is partly why Sigma.js is so efficient and performant.
- It provides a guarantee that all parts of the application will always have a consistent view of the graph instance's internal state. Especially in React applications with complex user interactions, inconsistent state due to batch updates can be a major source of bugs.

The only drawback is that components (other than the canvas) that depend on the graph's internal state are not automatically rerendered 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 internal state.

4.6.2 Separation of concerns

4.6.3 Modularity

4.6.4 Type safety

4.6.5 Extensibility

[18] https://en.wikipedia.org/wiki/Publish-subscribe_pattern

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 how graphs can be created by the user and how existing graphs can be edited directly in the canvas.
- Algorithmic visualisation: this will discuss how algorithmic results are visualised and how the user can customise these visualisations.
- Exporting results: this will discuss how the results can be exported from the application for use in other applications.

5.1 High-level design

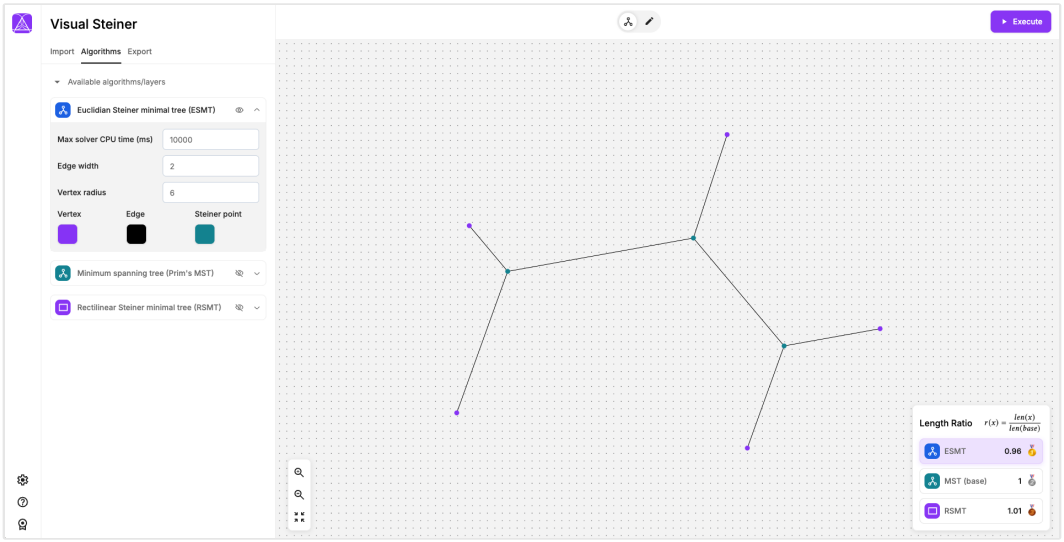


Figure 5.1: *High-level design of the application.*

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 the majority of the screen width since it is

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

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 to 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).
- **Algorithms**: This section contains visualisation customisation options for output of the algorithms (see section ??).
- **Export**: This section allows the user to export the currently active graph instance for use in other applications. (see section 5.4)

As per requirement E1 (see section 3), focus is given to a clean and intuitive design. This is achieved by using a consistent constrastive color schema, incorporating white space, and using familiar icons to indicate the purpose of the controls. For example, the two buttons in the center-top of figure 5.1 indicate two "modes" of the canvas. As the right button has a pencil icon, which is commonly used for editing or drawing actions, it is clear that the other button must be for the "non-editing" mode.

Care has also been taken to ensure that the primary action is always visible and obvious 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 lead to the results being displayed on the canvas.

5.2 Creating and editing graphs

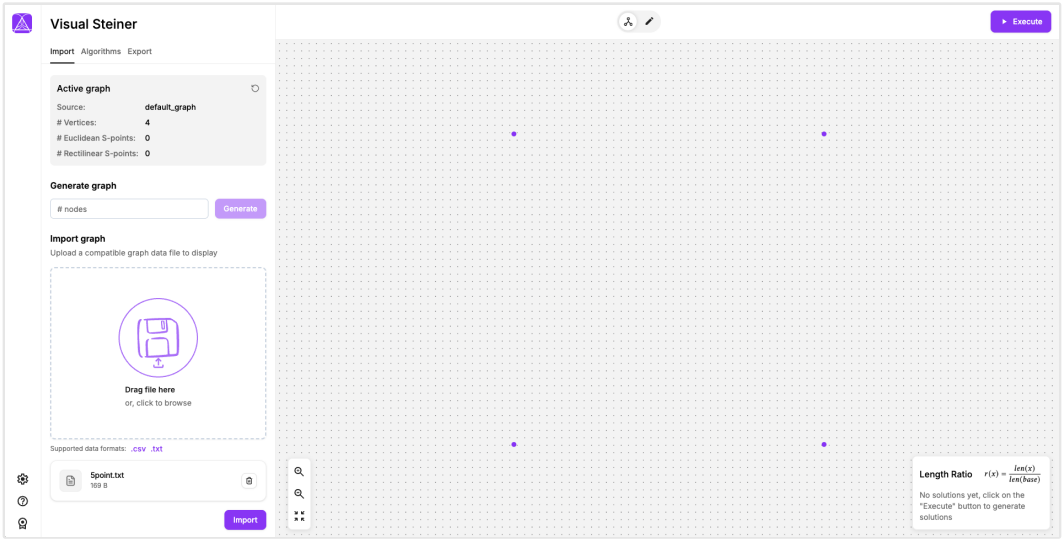


Figure 5.2: *Creating a graph instance by importing from a file or generating a random graph.*

As per requirements E5 and E6 (see section 3), flexibility is given to the user to create or edit graphs at any time in the same view. At the top of the import tab

section in the sidebar (figure 5.2), the currently active graph instance is displayed. The application comes with a default 4-point example graph selected by default.

5.2.1 Creating/importing graphs

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 get familiar with the format. Again, care has been taken to make sure the primary action is visible and obvious. 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* button in the header above the canvas, which will make the graph in the canvas editable. As all supported algortihms take a undirected, complete graph as input, there was no need for supporting edge drawing, so the available edit actions have been limited to adding, moving, and removing nodes.

5.3 Algorithmic visualisation

Being able to compare the trees of the MST and SMT algorithms is among the application's main features as per requirement E8 (see section 3). Hence, "layering" was naturally a core design aspect for the visualisation of 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).

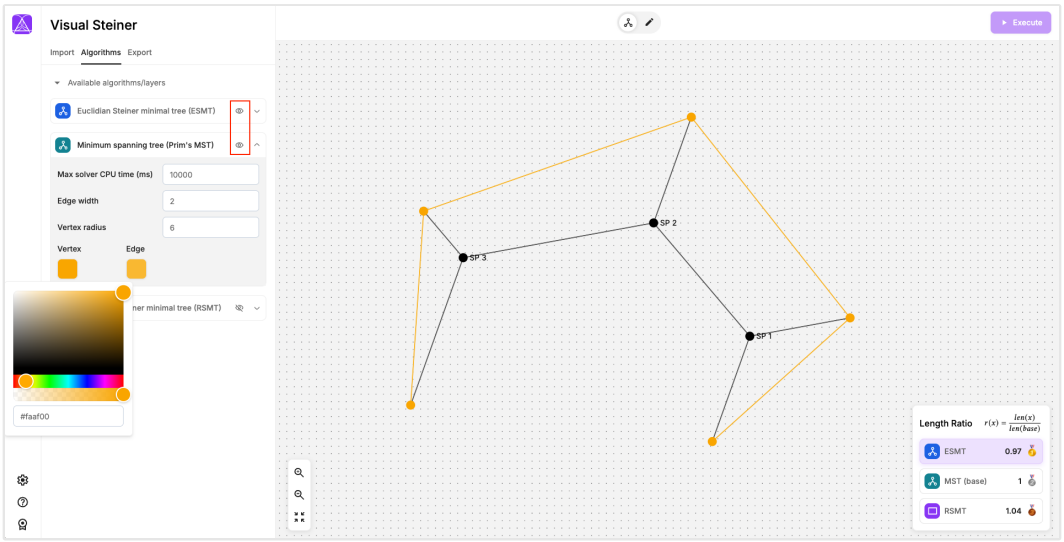


Figure 5.3: *Algorithmic visualisation customisation options showcasing the "layering" approach.*

5.3.1 Layering

Figure 5.3 shows an example of the algorithmic visualisation customisation options. Both the MST and SMT trees are visible but their styles can be customised independently. The yellow edges are the MST edges and the black edges belong to the SMT.

Sometimes, edges or nodes of the two trees overlap. In this case, the hierarchical order of the algorithm layers is used to determine which styles take precedence. In figure 5.3, the SMT layer is above the MST layer. The terminals are therefore painted with the color set in the SMT layer (in this case the color is the same for both layers).

5.3.2 Visualising the Steiner ratio

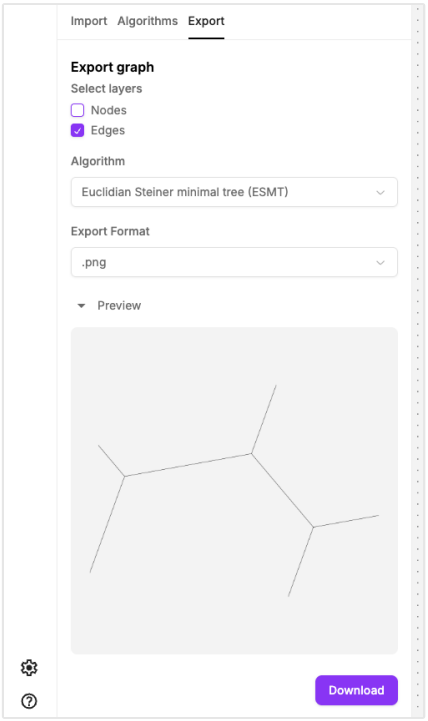
As a reminder, the Steiner Ratio is defined as the smallest ratio of the length of the Steiner minimal tree to the length of the minimum spanning tree. A canvas widget has been added (see bottom left of figure 5.3) to visualise the Steiner ratio of the currently active graph instance. This feature is useful to get an intuition (or potentially find counter-examples though this is unlikely) 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 Exporting results

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].

The image-based exports are useful for embedding the visualisation in other documents, while the text-based exports are useful for using the graph in other applications. GEXF has been the export format of choice due to its popularity and support for a wide range of applications.

When exporting to an image, it is possible to choose between the customised style set by the user or the default style which is inspired by how trees are typically visualised in literature. Furter, each algorithm's output tree can be exported separately with the option to render only the edges or nodes or both.



[2] https://gexf.net/

6 Discussion of the results

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

6.1 Guidance

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

TODO: fill this chapter

7 Conclusion

TODO: fill this chapter

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

7.1 Guidance

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

7.2 Summary

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

7.3 Reflection

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

7.4 Future work

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

A Melzak algorithm complexity derivation

The derivation of the formula for the number of full topologies $F^*(n)$ based on Property 7 in Gilbert and Pollak (1968) is given below.

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

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

There are still $n - (s + 2)$ points left, though, which can be added to the currently full topology to make it non-full. A tree has $n - 1$ edges, hence the current full topology has $((s + 2) + s) - 1 = 2s + 1$ edges. The leftover points can be added by splitting existing edges into two, resulting in the net addition of 1 edge.

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

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

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

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

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

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

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

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

Finally,

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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/`

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

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*