Honours Individual Project Dissertation

Level 4 Project Report Template

Pieter van Tuijl

February 21, 2025

Abstract

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

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

Acknowledgements

Education Use Consent

Contents

1 Introduction

1.1 Context

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

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

1.1.1 Minimum Spanning Tree

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

1.1.2 Steiner Minimal Tree

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

TODO: show example of MST vs SMT

1.2 Aims

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

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

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

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

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

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

1.3 Dissertation Outline

TODO: write at the end of the project

2 Background

Let's start with the theoretical background. Then we will proceed to talk about existing tools and how they compare to our aims.

Start with explaining why the SMT is a NP hard problem. State what algorithms exist and why they are not polynomial time algorithms.

2.1 Existing visualisation tools

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

2.1.1 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.

2.1.2 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, the tool is limited due to a lack of an interface, whether it a GUI or CLI. It also does not support the simultaneous visualisation of MSTs and SMTs and the comparison of their lengths.

2.1.3 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 provides a flexible canvas, it does not support MSTs or SMTs in the Euclidian plane (L2-norm). Furthermore, it is not possible to import a graph from an external file or export the results.

2.2 Theoretical background

2.3 Algorithms

2.4 Problem definition

Given a graph $G = (V, E)$ and a set of Steiner points $S \subseteq V$, find the SMT T of G with respect to S.‘

3 Analysis/Requirements

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

3.1 Guidance

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

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

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

4 Design

How is this problem to be approached, without reference to specific implementation details?

4.1 Guidance

Design should cover the abstract design in such a way that someone else might be able to do what you did, but with a different language or library or tool. This might include overall system architecture diagrams, user interface designs (wireframes/personas/etc.), protocol specifications, algorithms, data set design choices, among others. Specific languages, technical choices, libraries and such like should not usually appear in the design. These are implementation details.

What did you do to implement this idea, and what technical achievements did you make? Cover high level first, then cover important, or impressive details. Don't mix levels of abstraction and start with abstract thoughts and general structures and then refine into details. Keep a logical ordering.

5 Implementation

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

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

5.1 Building for the web

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

5.1.1 Tech stack

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

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

Other dependencies include:

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

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

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

5.1.2 Application architecture

The application has been developed as a client-only application, with no need for a backend API. Figure 5.1 shows the various high-level components of the application. The top-layer (i.e. the frontend) is divided into three main parts:

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

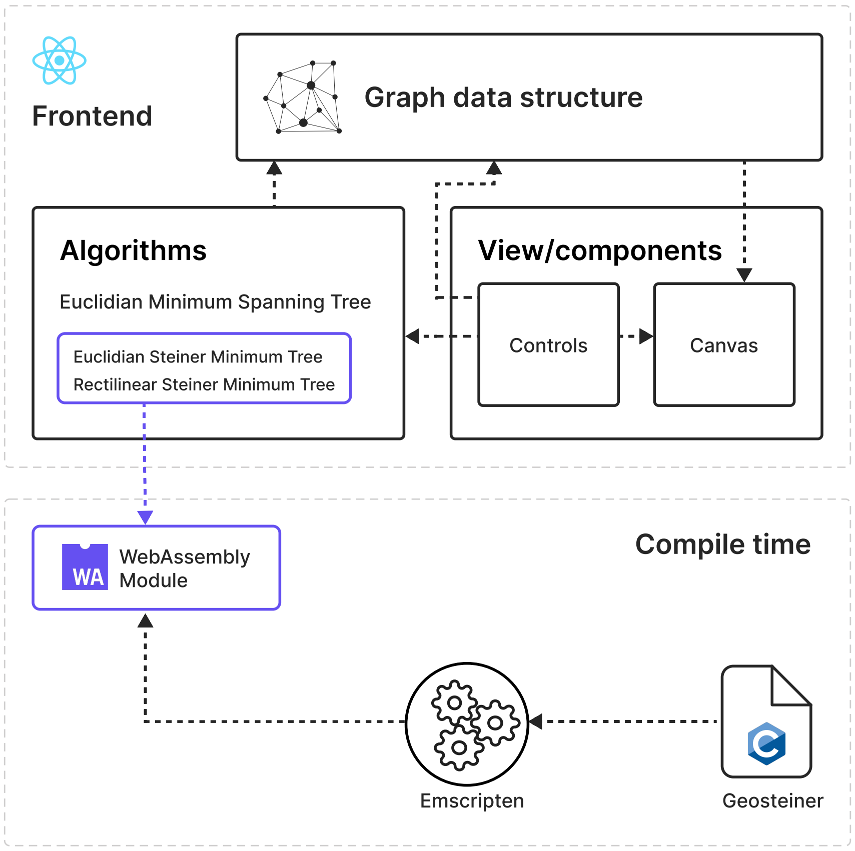


Figure 5.1: Architecture of the application. The arrows indicate the flow of data and control.

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

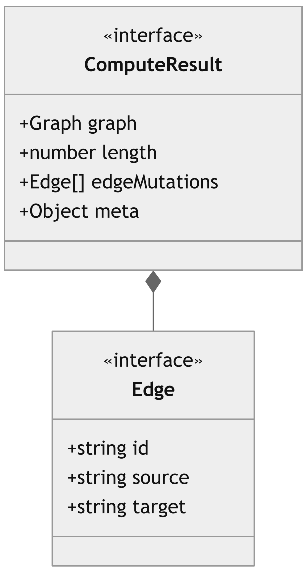
The lower layer contains the WebAssembly module that contains the WASM byte code of the compiled Geosteiner C-library, which is used for computing the Euclidian and Rectilinear variants of the Steiner minimal tree. The WebAssembly module is dynamically imported by the frontend during runtime. We discuss the use of WebAssembly in more detail in section 5.3.

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

5.2 Algorithm implementations

Each algorithm is implemented as an isolated, pure function that takes an undirected graph as input and returns a computation. Figure 5.2 specifies the interface of the computational output. The tree is returned as new graph instance, alongside the length of the tree. The list of edge mutations keeps track of all the edges that were added to the tree in order of insertion. Some algorithms, such as the ESMT and the RSMT also pass additional metadata, such as the list of Steiner nodes. By coding to an interface as opposed to a concrete implementation, algorithm implementations can be swapped out and new algorithms can be added easily.

Figure 5.2: Interface of the algorithm computational output.



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

5.2.1 Euclidian Minimum Spanning Tree

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

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

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

5.2.2 Euclidean & Rectilinear Steiner Minimal Tree

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

We have implemented a wrapper around the library that provides a simplified interface, only exposing the functions that are needed by the frontend. Listing 5.1 shows the wrapper functions that are exposed to the frontend.

```c
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 5.1: *Wrapper functions for the Geosteiner library*

The browser, however, cannot execute C code directly. Therefore, we need to compile the C code to WebAssembly (WASM) byte code. We do this using the Emscripten[7] toolchain. We discuss the details of the compilation process and integration of the WASM module with the frontend in section 5.3.

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

5.3 WebAssembly

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

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

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

5.3.1 How WASM works

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

5.3.2 Emscripten

As highlighted in the figure 5.1, the Geosteiner library is compiled to WASM using the Emscripten toolchain. The Emscripten toolchain consists of a designated compiler frontend (emcc) and LLVM backend optimised for the generation of WASM binaries. Apart from the compiler, the toolchain also exposes a set of "glue code" that aims to simplify the boilerplate code required to both load the WASM module and interface between the WASM module and JS code (Selvatici 2018).

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

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

5.4 Structure of the codebase

5.5 Visualisation/design

5.6 Summary

6 Evaluation

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.

6.2 Evidence

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

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

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

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

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

If you visualise, follow the basic rules, as illustrated in Figure 6.1:

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

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

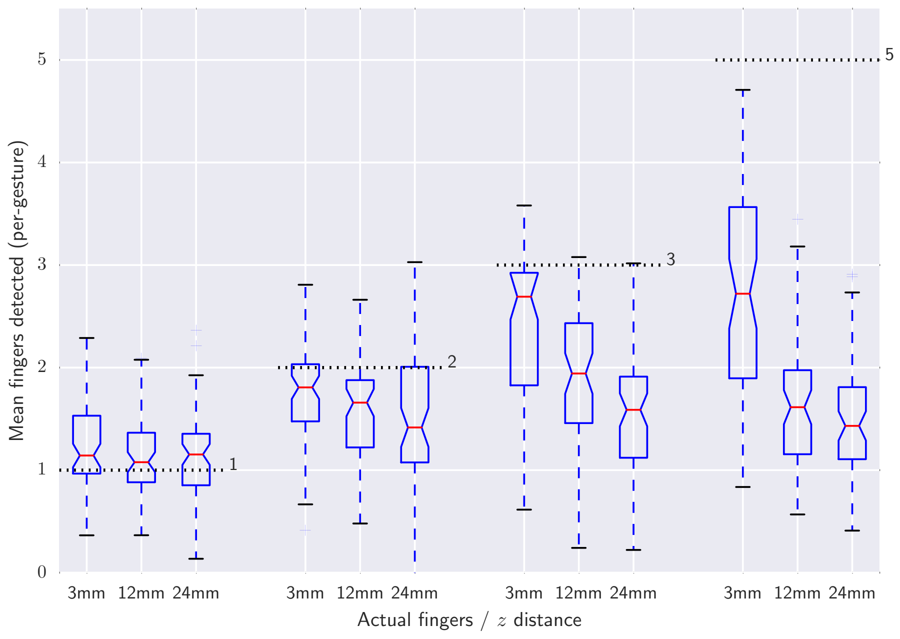


Figure 6.1: *Average number of fingers detected by the touch sensor at different heights above the surface, averaged over all gestures. Dashed lines indicate the true number of fingers present. The Box plots include bootstrapped uncertainty notches for the median. It is clear that the device is biased toward undercounting fingers, particularly at higher z distances.*

7 Conclusion

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

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 Appendices

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

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

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

Bibliography

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

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

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

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.

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*

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*

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

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

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

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