FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Jens Fredskov (chw752)

Finding Steiner Minimal Trees in Euclidean d-Space

Supervisors: Pawel Winter

Autumn, 2015

Abstract

This thesis presents possible improvements to the original implementation of the algorithm for finding Steiner minimal trees proposed by Smith [13]. The suggested changes include a new data structure and method for building topologies as well as a method for ordering the terminals before the topologies. This is used to avoid double-work in the original implementation which rebuilds the entire tree every time its topology changes.

This the thesis also presents a simple iteration for optimizing full Steiner trees as an alternative to the one proposed by Smith [13]. This includes presenting an analytical solution to the Fermat-Torricelli problem as proposed by Uteshev [19] and further generalizing it from 2D to higher dimensions.

The thesis also presents experimental work performed to test the correctness and efficiency (compared to the original implementation) of the new implementation. The new suggested iteration, based on the analytical solution, shows some sub-optimal results. These are discussed and further investigated, and are concluded to probably stem from the used error-function and the if-clauses deciding when to prune a topology. A possible way of fixing the sub-optimality is then proposed.

With the sub-optimality in mind, the new simple iteration however seems to show promise as the performance of it in general is much better than the original implementation using Smith's iteration. The new implementation of Smith's iteration show performance equivalent to the original implementation. This however is ascribed to the choice of programming language, and the much larger amount of trees the new implementation optimizes before reaching optimality. The new method for ordering terminals shows very promising results (when used with either of the iterations).

Finally the thesis presents and discusses possible extensions and improvements to the new implementation.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

SMT	Steiner minimal tree
RMT	relatively minimal tree
ESTP	Euclidean Steiner tree problem
FST	full Steiner tree
MST	minimum spanning tree

1 Introduction

The Euclidean Steiner tree problem is the problem of finding the Steiner minimal tree for some given set of points (known as terminals). The Steiner minimal tree is the tree containing these terminals, and having the shortest possible length. The tree is allowed to contain extra points (known as Steiner points) if adding these to interconnect the terminals can reduce the length of the tree.

Apart from being a problem that is interesting to solve simply because it is hard, the problem is also known to have application in designing electronic circuits and a novel approach for phylogenetics have also been suggested by Brazil et al. [2].

The problem is known to be NP-hard, however in two dimensions efficient algorithms for finding Steiner minimal trees exists, some of which can solve instances with thousands of terminals. In many cases these can solve the problem much quicker than the worst-case running time. These algorithms however, do not generalize to higher dimensions, and as such we need to find other methods. The most well-known of these is the one originally presented by Smith [13]. However the algorithm is not able to solve instances with more than around 15–20 terminals. The reason for this lies with the fact that the algorithm must enumerate all possible full Steiner topologies (apart from those it is able to prune).

There have been articles to propose extensions on the algorithm to further push the boundaries of solvable instances. These have included sorting of terminals, using lower bounds, and pruning via geometric conditions. None of these have pushed the boundaries for solvable instances with more than a few terminals.

It is therefore still interesting to explore possible improvements of the algorithm, either by a more efficient implementation, extensions, or by changing parts of the algorithm (such as the iteration for optimizing a tree).

The thesis propose improvements (these have also been implemented in a new implementation) to the implementation of Smith's algorithm given in Smith [13]. Furthermore a new simple iteration based on the analytical solution for the Fermat-Torricelli problem proposed by Uteshev [19] is proposed and implemented. The thesis then presents and discusses experiments performed to test the correctness and efficiency (compared to the original implementation) of the new implementation.

The thesis also presents and discuss elements of the article and implementation in Smith [13] which are questionable.

Apart from this the thesis collects and presents preliminaries to gain an understanding of the Euclidean Steiner tree problem and the current state of the research in the field.

Finally the thesis proposes a number of possible directions in which one could continue the work of this thesis to further improve on the new implementation.

1.1 Objectives

A overview of primary and secondary objectives is provided here. The primary objectives are:

- Understand, analyze and describe the algorithm for finding Steiner minimal trees in euclidean d-space, presented by Smith [13]. A sub-objective of this is also to identify and discuss elements of the article which are questionable (e.g. the proposed error-function). It also includes identifying and discussing double-work and other questionable elements of the C-implementation given by Smith [13].
- Propose a new and possibly more efficient data structure and method for building topologies instead of the one used by Smith's implementation. Then implement the algorithm presented in Smith [13], using the new data structure.
- Present the analytical solution for the Fermat-Torricelli problem for 3 points presented for two dimensions by Uteshev [19] and generalize it to higher dimensions. Then propose and implement a simple algorithm for optimizing full Steiner topologies using the analytical solution. This is to be seen as a alternative to the algorithm given by Smith [13].
- Describe existing methods for ordering the terminals when processed by the the algorithm. Then propose and implement a new method for ordering terminals.
- Perform experiments to test the correctness and efficiency of the new implementation and compare it with the original implementation.

The secondary objectives are:

- Present preliminaries (definitions, proofs, etc.) for understanding Steiner trees and topologies in euclidean d-space.
- Present methods for finding Steiner minimal trees in two dimensions, and why these do not generalize to higher dimensions.
- Discuss possible extensions for the new implementation (e.g. concurrency).

1.2 Structural Outline

To provide the reader with an overview of what is to come, an outline of the different chapters of the thesis is presented here:

- **Chapter 2: Preliminaries** presents the Euclidean Steiner tree problem. The chapter also presents the different types of Steiner trees and topologies. The chapter afterwards presents some of the most well-known methods for finding Steiner trees in 2D, why they are effective and discusses why these do not generalize to higher dimensions. The chapter then presents some of the known related work into the field of finding Steiner trees in d-space. Finally the chapter presents some variations and applications of the Steiner tree in d-space, to get a feel of possible uses for a solution to the problem.
- **Chapter 3: Smith's Algorithm** presents the algorithm for finding Steiner minimal trees in euclidean d-space, as proposed by Smith [13]. The chapter also present a simple iteration for optimizing full Steiner trees. Finally the chapter discuss some of the more questionable elements of the implementation done by Smith.
- **Chapter 4: Data Structures and Methods** presents the improvements and changes made by the new implementation. The chapter first presents the new data structure and method proposed for building topologies, which will be used instead of the original implementation's which was to just rebuild the entire tree from scratch every time. The chapter then presents and generalizes to higher dimensions an analytical solution (originally presented by Uteshev [19]) for solving the Fermat-Torricelli problem. This is presented here, as it is used as the optimization iteration in conjunction with the simple iteration presented in Chapter 3.
- **Chapter 5: Implementation** presents the new implementation, its structure and gives a general overview of how the architecture of the new implementation is set up. The chapter also reasons for the choice of programming language.

- **Chapter 6: Experiments** presents the experimental work done to test the new implementation. The experimental work falls into two categories: correctness and performance. The chapter first presents work on correctness, and describes the sub-optimality observed with the implemented simple iteration in some instances. The chapter furthermore describes the work done to identify the cause of the sub-optimality, and discusses the possible reasons on the basis of this work. The chapter afterwards present the work done to test the performance of the implementation compared to the original implementation. The performance is measured by three different parameters: speed/run-time, number of optimized trees and number of iterations. The chapter describes and discusses the observed behavior of the two implemented optimization-iterations in comparison to the original implementation. This is done with and without terminal sorting.
- **Chapter 7: Discussion** discusses possible extensions and improvements to the work of the thesis, most of which are related to the implementation (e.g. concurrency and how one could do this).
- **Chapter 8: Conclusion** summarizes the various results and conclusions from the thesis. The suggested improvements and the implementation is evaluated.

2 Preliminaries

This chapter introduces basic concepts and definitions to either help understand the problem area of the thesis, or which are necessary as they will be used later in the thesis. The most important keywords of this chapter, are:

- Topologies and trees
- Steiner trees: full Steiner trees, Steiner minimal trees
- Euclidean Steiner tree problems
- Fermat-Torricelli point/problem

The chapter will also present known methods for finding Steiner trees in 2D, why they are effective and why they do not generalize to higher dimensions. Known related owrk into the field of finding Steiner minimal trees in d-space is then presented.

Finally chapter also introduces versions of the Steiner tree problem not used directly in the thesis, but which help to give an understanding of the problem area of the thesis, and the possible applications for the Steiner tree in d-space.

2.1 The Fermat-Torricelli Problem

Before introducing the Steiner problem, it is relevant to introduce the Fermat-Torricelli problem, as this can be seen as a sort of subproblem to be solved when solving the Steiner tree problem.

The Fermat-Torricelli problem[1] in its classical two dimensonal form, is defined as follows:

Given	A set of three points $V = \{p_1, p_2, p_3\}$ in the plane.
Find	A point s such that the sum of the Euclidean distances from s to p_1, p_2 and p_3 is minimized.

The point s will be referred to as a Steiner point[2]. There are several ways to solve this problem, e.g. using the rotation-proof [1, p. 3–5].

This formulation of the problem can be seen as a specialized instance of the more general problem, where the distances are weighted. In the classical version the problem all weights are equal, i.e. all are equally important to minimize.

The generalized (weighted) two dimensional version of the problem can be formulated as in Uteshev [19]:

Given Three non-colinear points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $p_3 = (x_3, y_3)$ in the plane.

Find The point $p_* = (x_*, y_*)$ which gives a solution to the optimization problem

$$\min_{(x,y)} \sum_{j=1}^{3} m_j \sqrt{(x - x_j)^2 + (y - y_j)^2}$$

Where the weight of jth point, m_j, is a real positive number.

The generalized Fermat-Torricelli problem can be even further generalized from two dimensions to d dimensions, with $d \geqslant 2$ [7]. This simply requires the points to be d-dimensional and the optimization problem is then changed

[1]The problem is so named as it was first proposed by Fermat [6] and the earliest known solution was put forth by Torricelli [17].

[2]In the Fermat-Torricelli problem this point would normally be named the Fermat point or the Fermat-Torricelli point. However for clarity as to the connection between this problem and the Steiner problem this term is used. The name of both the Steiner problem/tree/point is named after Jakob Steiner, which might be errornous, who supposedly studied the Steiner problem for 3 terminals [3].

as follows:

$$\min_{(x_1,x_2,\dots,x_d)} \sum_{j=1}^{3} m_j \sqrt{\sum_{i=1}^{d} (x_i - x_{(j,i)})^2}$$
$$\Updownarrow$$
$$\min_{(p)} \sum_{j=1}^{3} m_j |pp_j|, \quad p \in \mathbb{R}^d$$

Solving the Fermat-Torricelli problem with even weights is equivalent with solving the smallest possible instance of the Euclidean Steiner tree problem. The analytical solution presented by Uteshev [19] and a generalization of it to $\mathbb{R}^d$ is presented in Section 4.2.

2.2 The Euclidean Steiner Tree Problem

We firstly define a graph $G = (V(G), E(G))$ where the vertices $V(G)$ are points of the graph, $V(G) \subset \mathbb{R}^d$ and the edges $E(G) \subset (\mathbb{Z}^+ \times \mathbb{Z}^+)$ are straight lines connecting the points in $V(G)$. The euclidean length of edge $e \in E(G)$ we denote $|e|$.

The Euclidean Steiner tree problem (ESTP) is then defined as follows:

Given	A set of points $R = \{p_1, p_2, \dots, p_n\}$ in $\mathbb{R}^d$.
Find	A graph $T = (V(T), E(T))$ such that $R \subseteq V(T)$, and $|T| = \sum_{e \in E(T)} |e|$ is minimized.

Note that T must be a tree, as any cycles can obviously be removed without disconnecting the graph or increasing the length. An example of the difference between the minimum spanning tree (MST) and Steiner minimal tree (SMT) can be seen in Figure 2.1.

2.3 Steiner Minimal Tree

We define a SMT as in Brazil and Zachariasen [1], which is as follows:

Definition 2.1 (Steiner minimal tree, Steiner points, terminals)**.** *A tree* $T = (V(T), E(T))$ *representing a solution to the Steiner tree problem is called a Steiner minimal tree. The given points* $R \subseteq V(T)$ *are called terminals and possible extra vertices* $S = V(T) \setminus R$ *are called Steiner points.*

The definition used by Smith [13] is simply that a SMT on n terminals is the shortest tree containing the terminals. This definition, while being a bit more

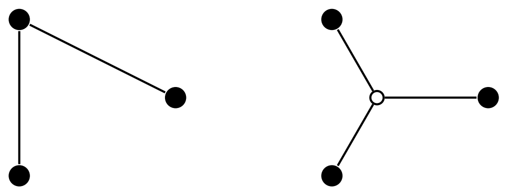


Figure 2.1: Example of the difference between the MST (left) and SMT (right) for the three points $p_1 = (0,0), p_2 = (0,2), p_3 = (2,1)$*. The MST simply interconnects the existing points with the shortest possible tree. The SMT however adds an extra point* $s_4 = (0.577, 1)$ *and interconnect these four points with the shortest possible tree. There is no way to interconnect the three original points with a shorter tree, no matter the number of extra points we add in the right tree.*

convoluted, is essentially the same as the one given by Brazil and Zachariasen [1]. We therefore use Definition 2.1 as it is more verbose and gives a definition of Steiner points and terminals as well.

2.4 Topologies

A topology is the combinatorial structure of a tree, or other form of geometric network. This means that in a topology only the adjacencies of points matter. Thus we could represent the topology of the tree $G = (V(G), E(G))$ as just the edges of the tree $E(G)$ and any graph (and tree) has an underlying topology.

We call any tree of some topology *non-degenerate* if all edges of the tree have non-zero length—otherwise the tree is called *degenerate*.

We now define a Steiner topology and a full Steiner topology as the following:

Definition 2.2 (Steiner topology, full Steiner topology)**.** *The topology of a non-degenerate SMT is called a Steiner topology. The topology of a non-degenerate SMT in which every terminal has degree 1 is called a full Steiner topology.*

Notice that a full Steiner topology must necessarily have the most possible Steiner points so $k = n - 2$. This can be seen by substituting $n_2 = n_3 = 0$ and $n_1 = n$ in Equation (2.1).

2.5 Steiner Trees

Before defining Steiner trees we define relatively minimal trees (RMTs) as in [9] as:

Definition 2.3 (relatively minimal tree)**.** *A tree is called a relatively minimal tree if it is the shortest possible tree of its underlying topology.*

As can easily be seen a SMT must also be a RMT of its underlying topology. Finally we define Steiner trees. Steiner trees currently have two definitions which are not exactly identical.

The first is the definition used by Gilbert and Pollak [9], combined with the definition for full Steiner trees (FSTs) in Smith [13], is as follows:

Definition 2.4 (Steiner tree, full Steiner tree)**.** *If a tree can be shortened no further even when splitting is allowed, the tree is called a Steiner tree. If all of the terminals of a Steiner tree have degree 1 the tree is called a full Steiner tree.*

Splitting simply means inserting an extra Steiner point into the topology. Thus it follows pretty straightforward that a Steiner tree must also be a RMT of the topology which describes it[3]. Furthermore any SMT must also be a Steiner tree, but not vice versa as illustrated by Figure 2.2.

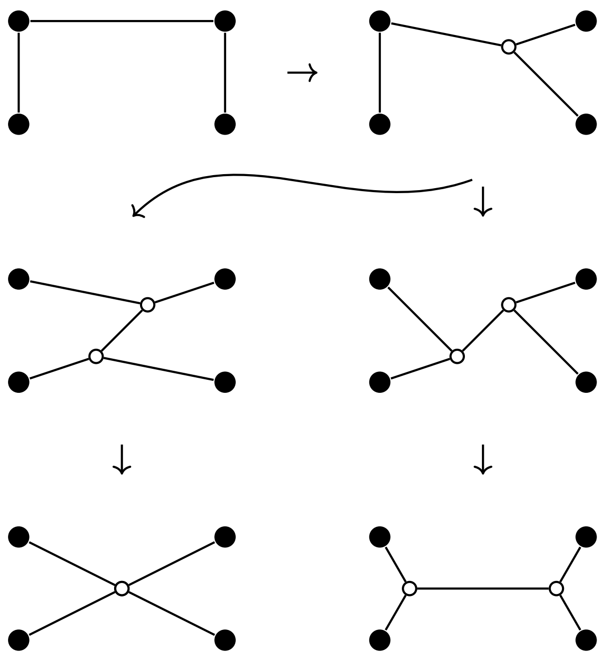


Figure 2.2: Example of how of splitting a point and inserting a new Steiner point. The flow shows two of the paths splitting could take on the shown topology down until no more splitting is possible. Optimizing the two final topologies, both are obviously RMTs of their underlying topology, but only the right one is a SMT as the other topology is degenerate (the two Steiner points lie on top of each other in the center). Thus if the original points are the terminals, only the last right tree is a SMT for the original terminals.

[3]Note that here we refer to the topology we end up with after we cannot split any more points, and not the topology we started out with.

The other, and newer definition, used by Brazil and Zachariasen [1] is as follows:

Definition 2.5 (Steiner tree, full Steiner tree)**.** *A non-degenerate (full) RMT for a Steiner topology is called a (full) Steiner tree.*

These definitions are not exactly identical as e.g. the last tree to the left in Figure 2.2 would not be a Steiner tree by Definition 2.5[4] but would be by Definition 2.4. The similarities of the two definitions should however be obvious, and it also holds for both of the definitions that any SMT must be a Steiner tree, and that any Steiner tree must be a RMT of its underlying topology. The one we will use in general is Definition 2.5.

Finally there is also the definition used by Smith [13] which defines a Steiner tree by some properties which it must also have for it to satisfy the Definitions 2.4 and 2.5. This is as follows

Definition 2.6 (Steiner tree)**.**

1. *It contains* n *terminals* $R = \{p_1, p_2, \dots, p_n\} \in \mathbb{R}^d$, *and* k *additional Steiner points* $S = \{p_{n+1}, \dots, p_{n+k}\} \in \mathbb{R}^d$.
2. *Each Steiner point has degree* 3, *the edges emanating from it are coplanar and have a mutual angles of* $120°$.
3. *Each terminal has degree at most* 3.
4. $0 \leqslant k \leqslant n-2$.

Proof. The first property is by definition and simply gives the two type of points a name.

The second and third property can be shown in the following way: First consider that any Steiner point must have at least degree 3. This should be obvious as a Steiner point with degree 1 does not connect any terminals to the rest of the tree, and thus the point can simply be removed to either shorten the tree or keep it the same[5]. Furthermore every Steiner point with a degree of 2 can be removed, and its edges be replaced with one edge that is of the same length or shorter as per the Triangle Inequality Theorem[18].

We now show that no pair of edges emanating from a Steiner point can have a mutual angle less than 120°. This can be proven in a few different ways. The one used by Gilbert and Pollak [9] is based on the mechanical model presented in the same article. A simpler approach in my own opinion however is the one used by Brazil and Zachariasen [1]. The proof is follows: Consider a point a in

[4]As it has an edge of length zero.

[5]Which could happen if a Steiner point was lying atop the point is connected to.

T, and a pair of non-zero edges $ab, ac \in E(T)$ meeting at a. Note that ab and ac must form a shortest interconnection of the points $\{a, b, c\}$ and furthermore we know that a solution the Fermat-Torricelli problem for $\{a, b, c\}$ forms a shortest interconnection as well. An immediate consequence[6] of this is that the angle between ab and ac, denoted $\angle bac$ must be at least $2\pi/3 = 120°$. To see that this is true, assume that it is smaller; we then have two cases as shown in Figure 2.3 and in neither cases is ab and ac a shortest interconnection of $\{a, b, c\}$.

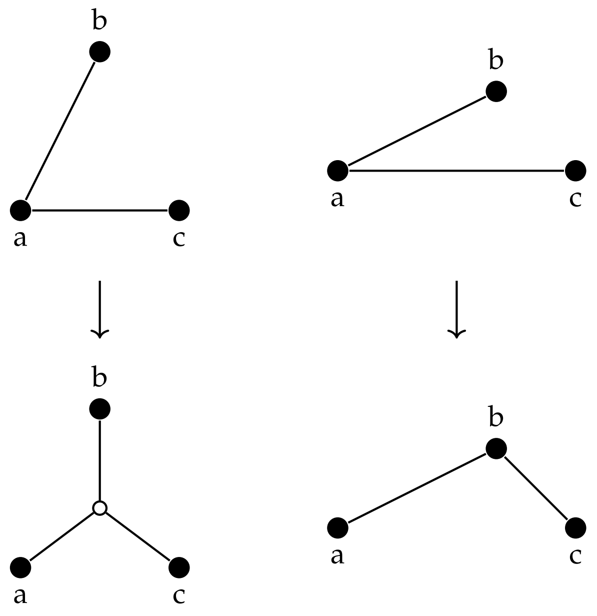


Figure 2.3: Replacing a pair of edges ab and ac for which $\angle bac < 2\pi/3$ with a shortest interconnection provided by a solution to the Fermat-Torricelli problem for $\{a, b, c\}$. The left figure has no angle greater than $120°$. The right has a angle $\angle cba \geqslant 120°$. In both cases a shorter tree is constructed. This figure is the same as in Brazil and Zachariasen [1, p. 7].

Using that the mutual angles may be no less than $120°$ it follows that any point in T can have at most degree 3[7]. It therefore follows that terminals have a degree of at most 3, which proves the third property, and that Steiner points have a degree of exactly 3 and thus also mutual angles of precisely $120°$, proving the second property.

The fourth property follows in the following way from property 3 and 4. Any

[6]This stems from the fact that the Fermat-Torricelli problem has a solution at one of the points if any of the angles is greater than $2\pi/3$ and somewhere between them otherwise. The details can either be found in Brazil and Zachariasen [1, ch. 1] or somewhat implicitly in Section 4.2.

[7]As $3 \cdot 120° = 360°$ meaning that there room for no more than at most three edges around a point.

tree T has $|V(T)|-1$ edges. Thus a Steiner tree has $n+k-1$ edges. Every edge has two endpoints meaning there are $2n+2k-2$ edges. As every Steiner point has a degree of 3, they account for $3k$ of the endpoints. For the terminals we must split them in three groups: n_1, those that have a degree of 1. n_2, those that have a degree of 2 and, n_3, those that have a degree of 3. The terminals thus account for the rest of the endpoints, written as $n_1 + 2n_2 + 3n_3 \geqslant n$. We can then write the equation as

$$\begin{aligned} 3k + n_1 + 2n_2 + 3n_3 &= 2n + 2k - 2 \\ k &= 2n - 2 - (n_1 + 2n_2 + 3n_3) \\ 0 \leqslant k &\leqslant n - 2 \end{aligned} \tag{2.1}$$

Thus proving the fourth property. □

2.6 Finding Steiner Trees in 2D

In general the problem of finding SMTs is NP-hard. This however is only a measure of the worst-case upper bound. Thus in some cases it may indeed be possible to find them much faster.

Two-dimensional algorithms that in many cases can find the SMTs much faster than the worst-case scenario exist.

This section will give a short introduction to the Hwang-Melzak and GeoSteiner algorithm. Afterwards the section will discuss why these algorithms do not generalize to dimensions higher than two.

2.6.1 The Hwang-Melzak Algorithm

The Hwang-Melzak algorithm was first presented by Melzak [12] and later improved by Hwang and Weng [11]. For a proof of correctness see Hwang [10] and Melzak [12].

The algorithm is used for optimizing the tree T of a pre-specified full Steiner topology $\mathcal{T}$ with at least $n \geqslant 5$ terminals, assuming that T is non-degenerate. The algorithm can be implemented with a worst case run-time of $\mathcal{O}(n)$. The algorithm is very well-known, and is also presented in Brazil and Zachariasen [1] and Smith [13].

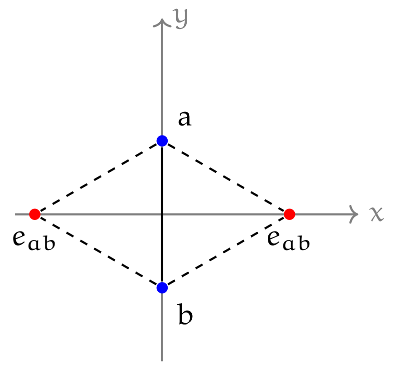
Before describing the algorithm, we define an equilateral point.

Definition 2.7 (Equilateral point)**.** *An equilateral point e_{ab}, of the points a and b, is a point such that the triangle $\triangle abe_{ab}$ is equilateral. It is clear that in two dimensions such a point has two unique placements, one on each side of the line passing through a and b.*

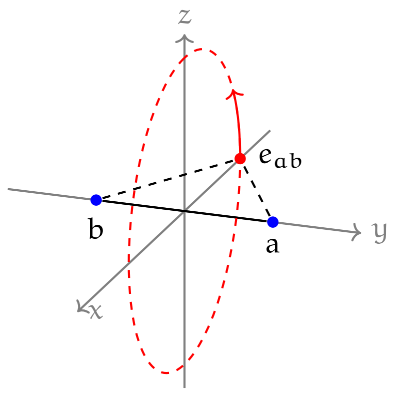
The algorithm consists of two steps. The first, the *merging* step, works as follows: Let u be a Steiner point connected to a, b and s, where a and b are terminals[8]. Replace a, b and u with a the equilateral point e_{ab} of a and b. Do this $n-2$ times until the topology consists of only 2 points. The second step, the *reconstruction* step, works a follows: Let $\mathcal{T}'$ be the full Steiner topology on the $n-1$ new terminals[9]. Then the coordinates of u can be found by taking the intersection between the Steiner arc $\widehat{ab}$[10] and the edge $e_{ab}s$. Do this on the 2 point tree $n-2$ times, reinserting the points represented by the equilateral points, until we again have a tree of n terminals with the correct coordinates for the $n-2$ Steiner points.

It may well be that $\widehat{ab}$ and $e_{ab}s$ do not intersect, if this is the case, the algorithm halts, and no solution exists for the given full Steiner topology.

The biggest issue of the above algorithm is deciding on which side of the line ab we should place the equilateral point e_{ab}. However it turns out that by choosing the order in which we perform the merging steps we are always able to find the correct side. Thus this algorithm runs in $\mathcal{O}(n)$.



(a) When $d = 2$ we have exactly two unique possible placements.



(b) When $d \geqslant 3$ we have a circle in d-space of possible placements. Here $d = 3$ is shown.

Figure 2.4: The case where $d = 2$ only has the two unique possible placements of the equilateral point e_{ab} for the two points a and b, one of which can be eliminated when running the algorithm. However when $d \geqslant 3$ this is no longer the case, and we are unable to determine a unique placement of the equilateral point.

[8]It can be proven, that such a pair can always be found. However intuitively it also makes sense, by simply thinking about the number of terminals and Steiner points in a full Steiner topology, and realizing that no Steiner point can be connected solely to other Steiner points, and thus we must be able to find such a point.

[9]I.e. the same set of terminals as before, but where a and b has been replace with e_{ab}

[10]The arc between a and b on the circle circumscribing the triangle $\triangle abe_{ab}$.

The proof of the algorithm and how to decide the side for the equilateral point can be found in Brazil and Zachariasen [1, ch. 1].

The algorithm unfortunately does not generalize from two dimensions to higher dimensions. The reason for this is, that while the edges emanating from a Steiner point are indeed coplanar, the plane the points lie in is not known in advance if there are more than 3 terminals. This means that we no longer have a unique pair of equilateral points for each cherry[11], but instead a continuous range of equilateral points lying on a circle in d-space, as illustrated by Figure 2.4. Thus in contrast to the two-dimensional case where we can select one of the two unique equilateral points, in higher dimensions we are no longer able to do this, and the algorithm breaks down.

2.6.2 The GeoSteiner Algorithm

The GeoSteiner algorithm was originally proposed by Winter [23]. Later improvements of the implementation have allowed for the computation of SMTs for several thousand terminals [1].

The GeoSteiner algorithm is pretty involved, and will thus not be described in greater detail here. In general the algorithm has two phases. In the first (*generation*) phase, FSTs for all full Steiner topologies of all subsets of terminals are generated. This enumeration can be said to be done implicitly[12] by simulating the Hwang-Melzak algorithm, and using geometric properties to prune parts of the Steiner arcs on which the Steiner points can lie. In this way the algorithm can prune FSTs of full Steiner which cannot be part of the final SMT, and if all FSTs of a full Steiner topology is pruned, i.e. if no feasible part of a Steiner arc is left, the entire full Steiner topology can be pruned. The strength of the algorithm lies partly in generating FSTs with similar full Steiner topologies in a single pass and partly in being able to prune partially constructed full Steiner topologies. In the second, (*concatenation*) phase, the algorithm selects a subset of not pruned FSTs that span all terminals and has the minimum total length. This concatenation can be formulated as the minimum spanning tree problem in a hypergraph, which is indeed also NP-hard. However using a branch-and-cut algorithm most instances can be solved quite effectively[13]. For a fuller description of the algorithm, see e.g. Brazil and Zachariasen [1, sec. 1.4].

While the algorithm is the most effective currently known for $d = 2$, it is unfortunately not applicable when $d \geqslant 3$. First, the generation of FSTs for all subsets of terminals in $\mathbb{R}^d$, $d \geqslant 3$, is much more difficult than in $\mathbb{R}^2$. Optimizing a

[11] A pair of terminals adjacent to a common Steiner point in a Steiner topology is called a cherry for the topology.

[12] as opposed to explicitly defining each FST one at a time.

[13] For an implementation of GeoSteiner see e.g. `http://www.diku.dk/hjemmesider/ansatte/martinz/geosteiner/`.

FST with more than 3 terminals when $d \geqslant 3$ requires solving high-degree polynomials [13]. As a consequence of this we can never use an algorithm such as Hwang-Melzak, but must use numerical approaches, such as Smith's optimization routine described in Chapter 3. Such numerical approaches seem to block the generation of FSTs across various subsets of terminals. Furthermore many of the geometrical properties used to prune seems to be much weaker when $d \geqslant 3$ [8].

Finally Smith [13, p. 142] describes that it seems when $d \geqslant 3$, that the topology of a SMT often consists of only a single large FST involving all $n - 2$ Steiner points, instead of many small edge-disjoint FSTs. Whether this is true, is not really qualified by Smith [13], but it does seem likely that the FSTs will tend to grow larger when we get to higher dimensions, as we have more space in which the Steiner point can lie. Thus we will not be able to leverage the benefits of the two phases in GeoSteiner, as the FSTs we generate in the first phase will span a large portion of the set terminals, or possibly even the entire set, making that phase very large and the second phase trivial.

2.7 Finding Steiner Trees in d-Space

As described in Section 2.6.2 determining FSTs in $\mathbb{R}^d$, $d \geqslant 3$ requires solving high-degree polynomials, and thus we need to resort to numerical approaches.

The most well-known of these is Smith's algorithm, proposed by Smith [13]. This algorithm is described in much greater detail in Chapter 3, but in general it works by enumerating all full Steiner topologies and then optimize these using an iteration which converges to the FST of that full Steiner topology. The algorithm can be implemented with a branch-and-bound approach[14], such that the found FSTs can be used to prune full Steiner topologies before optimizing them. The algorithm however cannot solve problem instances with more than around 15 to 20 terminals within a feasible amount of time.

There have been several attempts at optimizing Smith's algorithm to improve its speed and thus increase the number of terminals feasible to solve. Some of these are

- Fampa and Anstreicher [5] proposed using lower bounds on partial Steiner topologies[15] to prune FSTs. Roughly the method works in the following way: Imagine we have a FST not connecting all n terminals.

[14]The approach is not completely classic branch-and-bound as we are only able to bound, after having generated a complete solution. We do however generate small sub-problems of fewer terminals, and using the found bounds are able to prune these and their entire tree descendants/solutions. Thus in this way the algorithm can indeed be seen as branch-and-bound. The details of this can be found in Chapter 3.

[15]I.e. topologies that do not include all n terminals.

To select the next terminal to connect to the topology we compute the lower bounds $z^*(\bar{D}_l^+)$ of every terminal not yet connected, and select the terminal for which the largest number of descending topologies can be pruned. In this way the lower bound allow Fampa and Anstreicher to both prune some descendants of topologies, and also vary the order in which they connect terminals to reduce the number of topologies they need to optimize. The conic formulation and the actual lower bound will not be discussed here as they are rather extensive. The method shows some improvement on both the number of topologies enumerated and the actual running time. However as pointed out by Fonseca et al. [8] while a smaller number of FSTs are generated, the time spent on computations increase significantly, and thus the speed gained from computing these lower bounds is minimal when compared to e.g. distance-based sorting of the terminals.

- Van Laarhoven and Anstreicher [20] proposed both a method for sorting the terminals by their distance to the centroid and a set of geometric criteria used to prune non-optimal FSTs. The use of geometric criteria is partly what makes the 2D algorithms so successful, and thus it seems natural to explore these for the d-space case as well. However again as noted by Fonseca et al. [8] these computations give very little when compared to simply sorting the terminals based on distance in terms of speed vs. time spent on computations.

- Fonseca et al. [8] proposed a way of sorting the terminals, such that the first three terminals maximized the sum of their pairwise distance, and all terminals afterwards maximize the sum of their distances to the already spanned terminals.

Finally Fonseca et al. [8] has also proposed a new branch enumeration algorithm, instead of Smith's algorithm. This algorithm draws its inspiration from the GeoSteiner algorithm. Instead of enumerating full Steiner topologies and optimizing their respective trees it creates branches of subsets of terminals. The algorithm utilizes that upon removing a Steiner point from a full Steiner topology, 3 branches are created, and that one can always find a Steiner point such that the branches have at most $\lfloor \frac{n}{2} \rfloor$ terminals, which reduces the number of branches that need to be generated. The algorithm then consists of three phases. First, it computes SMTs for subsets with up to 8 terminals. Second, it generates branches containing up to $\lfloor \frac{n}{2} \rfloor$ terminals, and the SMTs found in the first phase are used to prune away branches that cannot be part of the final SMT. Finally it generates full Steiner topologies with n terminals by concatenating three disjoint branches. The shortest tree found is then outputted. Fonseca et al. noted that the branch enumeration algorithm was able to solve many of the tested topologies with 15 terminals within the given time of 12 hours which Smith's, even with terminal sorting, was not.

There has also been some work into heuristics and approximation algorithms. This is however somewhat limited, and out of scope for this thesis as we here mainly focus on Smith's algorithm and exact algorithms.

2.8 Variations on the Steiner Tree Problem and Applications

Steiner trees can of course be modified in several different ways. One could think of changing both the metric, the constraints, the cost function and so on. Some of the more well-known, are presented here with reference to where one could read more about the type of Steiner tree if so desired. The section also discuss some of the real world applications of Steiner trees, i.e. their usefulness outside the academic world as more than a mathematical curiosity.

Some of the types of Steiner trees to mention are:

- **Rectilinear Steiner Tree** A version of the Steiner tree in which edges must consist of vertical and horizontal line segments. It can be proven, that an edge always only needs to consist of at most a single horizontal, and a single vertical line segment, and thus we calculate the edge length using the rectilinear $\mathcal{L}_1$ norm. See e.g. Brazil and Zachariasen [1, ch. 3].
- **Fixed Orientation Steiner Tree** A version the of Steiner tree, which can be seen as a generalization of the rectilinear Steiner tree. In this version the metric used is a fixed orientation metric, meaning that edges are only allowed to have some fixed orientations, the rectilinear version can be seen as a fixed orientation version where we have two perpendicular orientations. See e.g. Brazil and Zachariasen [1, ch. 2].

Of possible applications, some more mentionable are:

- **Electronic circuits/VLSI Design** Both rectilinear and fixed orientation Steiner trees have use in VLSI design and the realization of electronic circuits as described by Brazil and Zachariasen [1, sec. 2.7,sec 3.6]. When designing electronic circuits the wires between components can normally only be drawn in a rectilinear, or lately fixed orientation, matter. Thus Steiner trees can be used to pack the components closer. This also has uses for higher dimensions than 2. As electronic components normally consists of several layers in which the wires can run between, thus crossing each other without shorting, the problem also has use in 3 dimensions.

- **Phylogenetics** A method using d-space Euclidean Steiner trees has been proposed to be used in the determination of phylogenetic trees by Brazil et al. [2]. Phylogenetic trees are used to determine the relationship between different species, these are represented as the leafs of the tree, and the interior nodes are then their ancestors. In this way one can represent the genetic difference of species, as their distance in the phylogenetic tree.

Brazil and Zachariasen [1] cover several more possible variations on Steiner trees, and applications of Steiner trees for the interested reader.

3 Smith's Algorithm

This chapter presents the different parts that go into the combined algorithm as presented by Smith [13]. Initially the representation and generation of topologies is described. The reasoning behind the representation is also given, by describing how we can use the representation to prune topologies which cannot become the Steiner minimal tree. A simple iteration which can be used to optimize the tree of a given topology is then presented. This is done to both approach the iteration proposed by Smith somewhat softer, and as this iteration will be used later in the thesis. The chapter then presents the iteration proposed by Smith for optimizing the tree of a given topology. Finally the chapter presents and discusses elements of the article by Smith [13] that was found questionable or odd. This includes both elements of the article and the actual implementation he gives.

Apart from the already used n and k, for clarity we define the following symbols

- R, the set of terminals.
- S, the set of Steiner points.
- $\mathcal{T} \equiv E(T)$, the pre-specified topology. Defined as all edges which the tree T contains in the form (i, j) where p_i and p_j are contained in either S or R and an edge exists in the topology from p_i to p_j or vice versa.[1]
- L, the length of the tree. Defined as the sum of the euclidean lengths of all edges in the tree.

3.1 Overview

In general the algorithm follows the following form, proposed by Gilbert and Pollak [9].

[1]Notice that in Chapter 2 we mainly represented and edge between the points p_a and p_b as ab as this is really simple. However this notation is not very good if e.g. one point is labeled p_{i+1}. Thus we adopt this extra notation as well, and may use these interchangeably.

1. Enumerate all Steiner topologies on n terminals and k, $0 \leqslant k \leqslant n-2$ Steiner points.

2. Optimize the coordinates of the Steiner points for each topology, to find the shortest possible Euclidean embedding (tree) of that topology.

3. Select, and output, the shortest tree found.

In other words this approach might be seen as an exhaustive search of the solution space, and the approach described is simply to "try all possibilities".

Smith however avoids necessarily having to enumerate and optimize all topologies, by doing the following:

Firstly he only looks at FSTs as it turns out we can regard any topology which is non-full, as a FST where some points overlap.

Secondly the algorithm builds the topologies by an enumeration process. First it builds the topology for 3 fixed terminals[2] (and 1 Steiner point). From this it then builds all descendants having 4 fixed terminals (and 2 Steiner points). Then again from each of these it builds the descendants having 5 fixed terminals (and 3 Steiner points), etc., until it has built all descendants having n fixed terminals (and $n-2$ Steiner points). The idea behind this way of building the FSTs is to allow for a branch-and-bound approach in which we prune all descendants of a topology if we have an upper bound with a shorter length than the length of the RMT of the topology. This process will be described more thoroughly in Section 3.2.2.

The optimization of a tree is done by an iterative process that updates all Steiner points of the tree every iteration. Smith describes the iteration as "analogous to a Gauss-Seidel iteration" [13, p. 145]. The equations, using the coefficients generated by each iteration, are solved using Gaussian elimination. This iteration and how to solve it is described in Section 3.3.

3.2 Topologies

The first step of the algorithm is to generate topologies. It is therefore natural that we need some way of representing and generating these topologies.

The algorithm only considers FSTs, where $k = n-2$. This simplification is allowed, as we can simply regard any Steiner tree with $k \leqslant n-2$ as a FST where some edges have length zero and thus some points have "merged".

[2]By "fixed terminals" we mean that the order in which the terminals are added is fixed, i.e. even though we where we add the next terminal, the ith terminal added to the topology will always be the same.

An incentive for only looking at FSTs can be found in Gilbert and Pollak [9] which presents a table, here presented in Table 3.1, containing the number of possible Steiner topologies for different numbers of terminals and Steiner points. As can clearly be seen the number of FSTs (the diagonal) is a lot smaller than the total number of topologies. Thus not having to optimize the non-full topologies is clearly desirable.

	n				
k	3	4	5	6	7
0	3	12	60	360	2520
1	**1**	12	120	1200	12600
2		**3**	75	1350	22050
3			**15**	630	17640
4				**105**	6615
5					**945**

Table 3.1: The number of possible topologies for a Steiner tree with n terminals and k Steiner points.

Note however that even with this simplification the number of FSTs is still exponential in N, which is clear from Corollary 3.1.1.

3.2.1 Representation

It turns out that every FST can be represented using a vector, in particular we utilize the following theorem put forth by Smith [13]

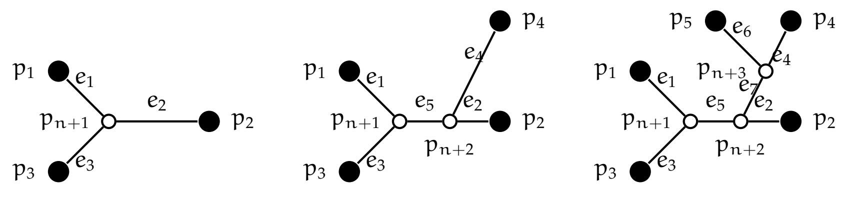
Theorem 3.1. *There is an 1–1 correspondence between FSTs with $n \geqslant 3$ terminals, and $(n-3)$-vectors $\vec{a}$, whose ith entry a_i is an integer between $1 \leqslant a_i \leqslant 2i+1$.*

Here terminals have indicies $1, 2, \ldots, n$, and Steiner points have indicies $n + 1, n + 2, \ldots, 2n - 2$.

Proof. The proof of this theorem is done constructively by induction on n. It is clear that the smallest FST[3] we can construct must have $n = 3$ as the number of Steiner points is $n - 2 = 1$. Thus we start with the initial null-vector $\vec{a} = ()$ corresponding to the unique FST for the terminals $\{p_1, p_2, p_3\}$ connected through the respective edges $\{e_1, e_2, e_3\}$ and one Steiner point p_{n+1} as seen in Figure 3.1a. After this first step, each entry of the topology vector is considered, one at a time, where the ith entry of the topology vector describes the insertion of the $(n + 1 + i)$th Steiner point on the edge e_{a_i} and its connection to the

[3]This holds not only for FSTs but for any Steiner topology

$(i+3)$th terminal. Thus for the ith insertion we will have $2i+1$[4] different edges on which we can insert Steiner point p_{n+1+i} and connect it to the regular point p_{i+3}.



(a) The initial null-vector. *(b) Connecting point p_4 on edge e_2.* *(c) Connecting point p_5 on edge e_4.*

Figure 3.1: Construction of the FSTs corresponding to the vector $\vec{a} = (2, 4)$.

□

Furthermore we get a corollary saying that the number of FSTs is exponential in n

Corollary 3.1.1. *The number of FSTs on n terminals is*

$$1 \cdot 3 \cdot 5 \cdots (2n-7) \cdot (2n-5) = \prod_{i=0}^{n-3} 2i+1$$

I.e. the number of FSTs is exponential in n.

Which is clear as we must insert $n-2$ Steiner points, where the null-vector is the 0th iteration. Thus the last iteration must be $n-3$, and for each iteration we have $2i+1$ different insertions.

Something which might not be completely obvious from the above proof is that we generate all possible topologies exactly once. This is important as duplicate topologies could significantly increase the run time of the algorithm. That we do not at any point construct duplicate topologies can be shown as a proof by contradiction.

Proof. Assume that we have generated the same topology by two different procedures (topology vectors). Both topologies must have started from the same three terminals and single Steiner point (the smallest Steiner topology possible). We now begin splitting the topologies. At some point the two topologies must have split different edges, as if this was not the case, the two

[4]At the first iteration we clearly have 3 edges to insert on, and as each subsequent insertion generates two new edges we have, after i insertions, $3 + \underbrace{2 + \cdots + 2}_{2i} = 2i + 1$.

topologies would have the exact same procedure. Thus we consider the first step i where two different edges are split. At this point we insert Steiner point p_{n+i}. Now find in the first topology a pair of terminals p_1 and p_2 such that p_{n+i} after the split lies on the path from p_1 to p_2, but does not lie on the path between the same two points in the second topology. This is always possible, and thus the two topologies are different, and since we cannot remove the Steiner point from the path in the first topology, or add it to the path in the second topology, these two topologies can never become the same again. □

The way Smith chooses to enumerate the edges is not explained outright, but only in the form of a visual example [13, p. 143]. Exactly how one enumerates the edges on a split is actually not so important, more so is it important to keep it consistent. The way Smith does this, is as in Figure 3.1. That is when inserting Steiner point p_{n+1+i} on the edge e_{a_i}[5] going from point p_{a_i} to point p_j with $n < j < n+i+1$, split it such that we get the following three edges[6]: $e_{a_i} = (a_i, n+1+i)$, $e_{2i+2} = (i+3, n+1+i)$ and $e_{2i+3} = (j, n+1+i)$. This is also the way the new implementation enumerates the edges.

3.2.2 Generation

Using the representation just described the problem of generating all topologies can now be done as a backtracking problem generating all $(n-3)$-topology vectors. An example of how the generation works can be seen in Figure 3.2

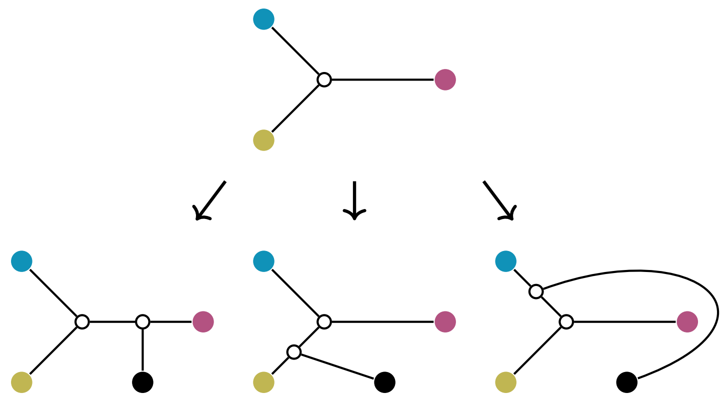


Figure 3.2: Upon splitting an edge and connecting a new terminal to the topology the algorithm can perform the split on any of the existing edges. Thus in this case the algorithm branches into three different topologies. In general the splitting of a topology results in the topology branching into as many new topologies, as there are edges in the topology on which the split is perfomed.

[5]Here a_i is the i-th entry in the topology vector.

[6]Remember that an edge is represented as (i, j) meaning that it runs from point p_i to point p_j.

To further speed up the generation of topologies, or rather to avoid generating unnecessary topologies, Smith also utilizes the following theorem

Theorem 3.2. *For any set of n distinct terminals in any Euclidean space, the length of the shortest tree, interconnecting $n-1$ points, with topology vector $a_1 \cdots a_{n-4}$ is no greater than the length of the shortest tree, interconnecting n points, with topology vector $a_1 \cdots a_{n-3}$.*

Proof. Consider deleting the edge incident to terminal p_n in the RMT on n terminals. Replace the Steiner point previously adjacent to p_n and its edges with a straight edge, connecting the other two points adjacent to the Steiner point. This tree is not longer than before p_n was removed, and furthermore it has the topology of the tree on the $n-1$ first terminals. This new tree currently has some length shorter than the previous, and per the definition of RMT, the RMT of the new tree must either be the length or the new tree or shorter. Thus we have proven that the RMT of this topology cannot be greater than the previous topology. $\square$

The algorithm utilizes this to prune in the following way. Imagine we have found some upper bound for the SMT of all terminals. If we then optimize any tree with a generated topology vector which does not yet include all the terminals, and it turns out to have length greater than the upper bound, we can prune all topologies that would be generated from this vector. This is possible as the length of the larger trees with more terminals cannot become any smaller than the length of the current tree, and thus cannot become smaller than the length of the upper bound.

In general we need to generate topologies and optimize their respective trees depth-first. The reason for this is two-fold: Firstly going breadth-first would mean that we generate all FSTs as the very last topologies to be generated. Thus we would not get an upper bound before this stage, and would therefore be unable to prune any significant branches, as all that is left to optimize at this point are leaves, i.e. the FSTs. We could of course find some upper bound using a heuristic, e.g. a MST. This would possibly allow us to prune some branches. Thus going depth-first is advantageous as it allows us to not only get an upper bound without the need of a heuristic, but also allows us to update the upper bound and use this, should we find a new better one.

Secondly is also the concern of memory usage of the actual implementation. If we were to perform the algorithm breadth-first it would mean keeping track of earlier topologies, which would drastically increase memory usage, in contrast to depth-first we one only need to remember the current way down.

The actual implementation of the backtracking is not described by the article part of Smith [13], but only by the source code in the appendix. The details of the new implementation will be discussed in Chapter 5. The original implementation faces some problems, one of which is the fact that the entire topology is rebuilt from scratch every time we need to either add of remove a point. These will be discussed in Section 3.4.2.

In actuality the way in which we optimize the trees is a mix between the depth- and breadth-first search, which could be considered a best-first approach. This is described more thoroughly in Section 3.4.2.

3.3 Optimization of a Prespecified Topology

When a topology has been generated, we need some way of optimizing the positions of the Steiner points. The approach taken by Smith is in general to create an iteration with $i - 2$ equations[7], one for each Steiner point, and incrementally optimize all of them every iteration. This is done by solving the equations using Gauss elimination.

This section will be concerned with describing the iteration, the proof of correctness and convergence, the speed of convergence and the error function for deciding convergence.

3.3.1 Simple Iteration

To get a sense of how we might go about optimizing the tree in a simple manner we first look at an iteration which only optimizes one Steiner point at a time. This is a simpler approach to the one proposed by Smith [13], but according to Smith will probably also take longer time to converge. The iteration is described by Figure 3.3. The algorithm is intentionally a bit vague, as even the simple iteration has several choices we can make.

Firstly we should choose how a single Steiner point is optimized. As earlier described this is the same as the Fermat-Torricelli problem, and thus we can go about finding the Steiner point in all the same ways. Secondly we need to decide some way of calculating the error of the current Steiner point. This could e.g. be done as in Smith [13][8].

The problem Smith poses for the simple iteration, described here, is that while it quickly becomes optimal *locally* it may still converge slowly as it does not regard the *global* structure.

In the new implementation a version of the simple iteration (using an analytical solution presented in Section 4.2) exists. The above claim will therefore be discussed further in Chapter 6 when we look at the performance of the new implementation.

[7]Where $i = 3, 4 \ldots n$ is the number of terminals currently in the topology.

[8]Note that we would not sum the error as is done by Smith, as we only need the error of the current Steiner point and not the entire tree.

```
Data:
- A full Steiner topology with:
  - the Steiner points $S$,
  - the terminals $R$,
  - and edges $E$.
- A small positive number $\epsilon$
Result: The RMT, $T$, for the given full Steiner topology
1 error ← calculate error for $T$;
2 while error $> \epsilon$ do
3     foreach $s \in S$ do
4         optimize $s$ with respect to its neighbors in the given topology;
5     error ← recalculate error for $T$;
6 return $T$
```

Figure 3.3: Pseudo code describing the optimization strategy using the simple iteration. The algorithm describes running a single optimization. In general one would have to run the algorithm several times, until the error of the entire tree falls below some threshold.

3.3.2 Smith's Iteration

Now that we have looked at a simple way of optimizing the topology, by finding the coordinates of one Steiner point at a time, we can progress to the iteration proposed by Smith [13].

To optimize the topology Smith proposes the following

Theorem 3.3. *As the ith step in an iterative process, $i = 0, 1, \ldots$, solve the $n-2$ equations*[9],[10]

$$p_l^{(i+1)} = \frac{\sum_{j:(j,l)\in\mathcal{T}} \frac{p_j^{(i+1)}}{|p_l^{(i)} - p_j^{(i)}|}}{\sum_{j:(j,l)\in\mathcal{T}} \frac{1}{|p_l^{(i)} - p_j^{(i)}|}}, \qquad n+1 \leqslant l \leqslant 2n-2 \tag{3.1}$$

which can be rewritten as follows, to show that forces on p_l is in equilibrium

$$\vec{0} = \sum_{j:(j,l)\in\mathcal{T}} \frac{p_l^{(i+1)} - p_j^{(i+1)}}{|p_l^{(i)} - p_j^{(i)}|}, \quad n+1 \leqslant l \leqslant 2n-2$$

[9]Note that in the second sum of Equation (3.1) Smith [13] gives the index $j : jk \in T, j \in S$. For the rewrite to make sense, this should one have been $j : jk \in T$. Thus I believe this is just a typo and have progressed as such.

[10]Note that since the unknowns in the iteration are vectors we actually have d independent equation systems of $n-2$ equations.

for the unknowns $p_l^{(i+1)}$ *where* $p_l \in S$ *represents the ith Steiner point, being a d-vector. Then from all initial choices of Steiner point coordinates* $p_l^{(0)}$*, except for a set of measure zero in* $\mathbb{R}^{(n-2)d}$*, the iteration converges to the unique optimum Steiner point coordinates*[11] *that minimize the total tree length* L*. This convergence happens in such a way that the sequence of tree lengths* $L(S^{(i)})$ *is monotonically decreasing.*

Here L is defined as

$$L(S) = \sum_{j,l:(j,l)\in\mathcal{T}} |p_l - p_j|$$

Which is simply the Euclidean norm. Smith [13] also presents a proof of the convergence of the iteration. A presentation and discussion of this can be found in Appendix A.

Solving the Equations

The equation system of each iteration can be solved using Gaussian elimination as described by Smith [13, p. 148–149]. Roughly the Gaussian elimination described and implemented by Smith[12] is done by identifying an equation that has at most one unknown on the right hand side. This corresponds to a Steiner point being connected to two or more terminals, which is always guaranteed to exist. Working from this point we can then eliminate a variable by substitution in the other equations, which will result in at least one equation having no unknowns on the right hand side. In the end we just do backward substitution. Doing the elimination from this specific point, instead of just some random point means that we can a perform all elimination in $\mathcal{O}(N)$. Since we have d linear systems we can then solve them all in $\mathcal{O}(nd)$. The details can be found in [13, p. 148–149].

Speed of Convergence

According to Smith [13, p. 150] the iteration has a good speed of convergence. In most cases geometric convergence will occur. However some degenerate configurations of terminals can cause sublinear convergence.

3.4 Questionable Elements of Smith's Article

There are a number of elements in the article and the given implementation [13] which are somewhat questionable. These I will try to discuss here.

[11]Note that if we have an edge of length zero, the iteration is not actually defined in the way we have written it, as we get a division by zero. Smith [13, p. 147–148] argues as to why this is not a problem, and how to resolve such a situation. This argument will be discussed later in this section. Thus the proof of convergence, for now, assumes this situation does not happen.

[12]and also the new implementation.

3.4.1 Choice of Error Function

To detect when convergence has happened Smith proposes using a special error function instead of the one often used, where one stops when the improvements from iteration to iteration falls below some ϵ. The function proposed by Smith is

$$E^2 = \sum_{\substack{i \in S \\ (i,j) \in \mathcal{T} \\ (i,l) \in \mathcal{T} \\ j \neq l}} \operatorname{pos}(2(\vec{x}_j - \vec{x}_i) \cdot (\vec{x}_l - \vec{x}_i) + |\vec{x}_j - \vec{x}_i| \cdot |\vec{x}_l - \vec{x}_i|) \tag{3.2}$$

Here $\operatorname{pos}(x) = \max(x, 0)$. This error function sums together all angles which are smaller than 120°. The error function is derived from calculating the angle between edges in the following way:

$$\cos v = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$$

If we then wish to find the cosine for 120°

$$\begin{aligned}
\cos 120^\circ = -\frac{1}{2} = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|} \Leftrightarrow \\
0 = \vec{a} \cdot \vec{b} + \frac{1}{2}(|\vec{a}| \cdot |\vec{b}|) \Leftrightarrow \\
0 = 2(\vec{a} \cdot \vec{b}) + |\vec{a}| \cdot |\vec{b}|
\end{aligned}$$

Thus any angle less than 120° gives a positive number when inserted in the function, and any angle below will give a negative number (which is set to zero by $\operatorname{pos}(x)$).

Why this error function is supposed to be better is not explained very thoroughly by Smith. It is only referred to that doing some experimentation which led him to believe that this would be a good function. The idea behind the function seems to be that as the final tree should only have angles of 120° any angle less than this must still need to be optimized.

There are however some problems with this choice of error function. Consider the configuration as shown in the first tree of Figure 3.4. Here there are two pairs of terminals, very far apart. These are connected by two Steiner points. Running Smith's iteration the two Steiner points will move together, and up towards the center of the rectangle of the four terminals. However suppose the two points reach each other before they reach the center. In this case the angles on the left and right will be more than 120° and thus not add to the error. Furthermore the edge between the Steiner points becomes zero (or practically

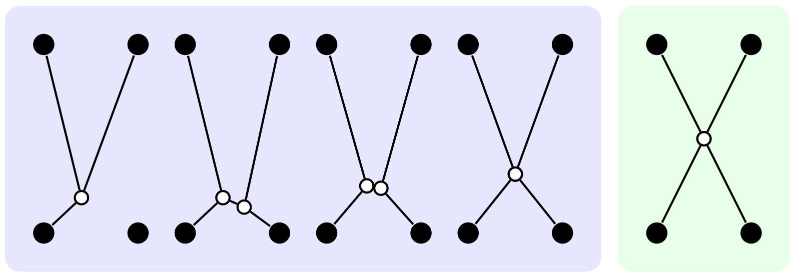


Figure 3.4: The left part of the figure shows how the use of the current error function might result in a sub-optimal tree. The right part of the figure shows the optimal tree. Notice that this topology does not result in the SMT for these terminals.

zero). This means, that even though the angles using this edge all are less than 120° and add to the error, the error they contribute with is zero—which is clear by looking at the definition of the error.

This means that the algorithm will stop before it should with a tree which has not been minimized.

Note however that the topology used in Figure 3.4 is not the one which gives the SMT for the four terminals. The optimal topology would have the top pair connected with one Steiner point, the two bottom terminals connected with another Steiner point, and those two Steiner points connected. When optimized, this topology will not face the same problem, as the two Steiner points will not move towards each other as illustrated in Figure 3.5. Thus it is still unclear whether this problem can occur with a topology which optimizes to a SMT. Should this be the case then we can for sure say that the algorithm is only a approximation algorithm, and not an exact algorithm[13].

3.4.2 The Implementation

The implementation by Smith contains several unexplained variables and conditionals not directly related to the algorithm as explained in the article by Smith [13].

SCALE

The variable `SCALE` is used by Smith when calculating the initial placement of a Steiner point. Steiner points are initially placed at the centroid of its three neighbors. However the point is not placed exactly at the centroid, but is instead pertubed slightly. To each coordinate of the centroid is added

[13]Some of the other problems in the article also casts doubt on whether the algorithm is actually exact, e.g. the earlier described uncertainty about the convergence of the iteration function.

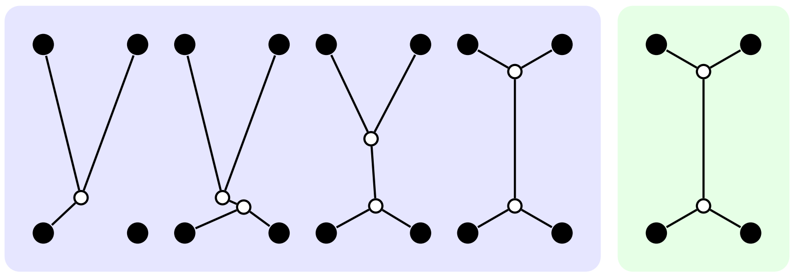


Figure 3.5: The left part of the figure shows the connecting and optimizing of the optimal topology for the given terminals (the first Steiner point is just placed arbitrarily). Notice how the Steiner points do not pull together, but instead apart, leading to the optimal tree. Thus this topology does not suffer the problem with the error function described in Figure 3.4.

$0.001 \cdot$ `SCALE` $\cdot$ `RANDM()`, where `RANDM()` is a random number between 0 and 1. `SCALE` is a number defined as shown in Listing 3.1.

```
SCALE = 0.0;
for (i = 1; i <= NUMSITES; i++) {
  for (j = 0; j < DIMENSION; j++) {
    q = XX[i][j] - XX[1][j];
    if (q < 0.0) q = -q;
    if (q > SCALE) SCALE = q;
    printf(" %g", XX[i][j]);
  }
  printf("\n");
}
printf("SCALE = %g\n", SCALE);
```

Listing 3.1: The implementation of `SCALE` in the original implementation. As can be seen it only compares the dimensions of the first terminal with the others, and not every terminal with every other terminal.

As can be seen the variable finds the largest difference in coordinates between the first terminal and all other terminals. Why this extra variable should be a part of the perturbation is however unclear. The name of the variable might be a hint, as it seems Smith wished to scale the perturbation in correspondence to the points. If this is the case however the code in Listing 3.1 seems to be faulty as it should then not only compare the first point to all other, but instead compare every point, to every other.

Thus it seems that the variable is not only not needed, but actually faulty. This of course depends on what Smith is trying to achieve with `SCALE`. But overall the variable seems unneeded, and thus the new implementation does not use `SCALE`.

tol

`tol` is the parameter given to the optimize function and is described as a small positive number. In the code whenever the function is called the variable has the value $0.0001 * \mathtt{r}/\text{NUMSITES}$ where $\mathtt{r}$ is the current error and and NUMSITES is the number of terminals, n. Having some small positive number in the optimize-function makes sense, as it is used to avoid divisions with zero (by adding it to the edge lengths, some of which may be zero). However the specific choice which is both dependent on the error and the number of terminals seems selected somewhat at random. A perhaps more obvious choice would be some *fixed* small number, e.g. 0.0001.

Loop Condition When Optimizing

In the main function when performing the optimization of a topology, the loop condition seems to suffer from the same arbitrariness as both `tol` and `SCALE`. Depending on whether we are optimizing a topology without all terminals, or a topology with all terminals and $n-2$ Steiner points the condition is either $\mathtt{r} > 0.005 \cdot \mathtt{q}$ or $\mathtt{r} > \mathtt{q} \cdot 0.0001$[14], $\mathtt{r}$ being the error and $\mathtt{q}$ the tree length. The problem here is that there seems to be no direct relation between the error and the length of the tree, and as such this loop condition seems somewhat arbitrary.

If-Clause When Outputting Trees

This is most probably a bug[15]. When a full topology with $n-2$ Steiner points is optimized its length is compared to the current upper bound, `STUB`. If it is better than the upper bound, this is updated to the newly found tree length. However the tree is only outputted as a record if its length is smaller than the previous upper bound times 0.99999. This means we risk situations where the upper bound is updated, but the tree never outputted. Most likely this extra check should have been removed. As we are dealing with floating point arithmetic doing the multiplication might make sense in the initial check, however the double check is faulty and may lead to wrong behavior. The faulty snippet and its possible corrections can be found in Listing 3.2.

Order of Building and Optimizing

When going over the implementation done by Smith a thing that initially stroke me as odd was the way in which Smith keeps track of which topology vectors has been tried, and how he chooses which to try next.

[14] The difference in the condition stems from wishing higher precision when optimizing a tree of a topology with all terminals.

[15] As I consider this a bug, it has also been fixed in the version of the original source code I have used for testing.

```
/* The faulty snippet */
if (q < STUB) {
  printf("\nnew record length %20.20g\n", q);
  for (i = 1; i <= k; i++) BESTVEC[i] = A[i];
  if (q < STUB*0.99999) output_tree();
  STUB = q;
}

/* First possible correction */
if (q < STUB) {
  printf("\nnew record length %20.20g\n", q);
  for (i = 1; i <= k; i++) BESTVEC[i] = A[i];
  output_tree();
  STUB = q;
}

/* Second possible correction */
if (q < STUB*0.99999) {
  printf("\nnew record length %20.20g\n", q);
  for (i = 1; i <= k; i++) BESTVEC[i] = A[i];
  output_tree();
  STUB = q;
}
```

Listing 3.2: The second conditional in the first code snippet may result in some SMT being set as upper bound, but outputted. This is fixed by both of the two next snippets.

In the following description, $*$ is used to represent all possible values, e.g. $A = [1, *]$ would be the same as writing $A = [1, 1], A = [1, 2], A = [1, 3], A = [1, 4]$. The implementation does as follows:

1. Start with the empty topology vector, meaning we have a topology with 3 terminals, 1 Steiner point and 3 edges.

2. Build and optimize all topology vectors of length 1, i.e. $A = [*]$.

3. When finished with optimizing a topology, store its length in a stack ordered such that the topology vector with the shortest tree length is on top.

4. Now select the top topology vector on the stack, and using this a prefix, build and optimize all topology vectors of length 2. So say we have chosen $A = [1]$, then build $A = [1, *]$. Again put the topology vectors on a stack in order of their increasing tree length.

5. Continue building the topology vector in this way, until we have a topology vector which connects all terminals.

6. At this point go back a level, and choose the next topology vector on top of the stack, i.e. the second best. So say e.g. we have 5 terminals in all and have just optimized $A = [1, *]$. Then if $A = [2]$ was the shortest tree after $A = [1]$ we would build and optimize $A = [2, *]$ next.

7. Continue building and optimizing the vectors in this way, putting them on, and popping them of the stack until the stack is empty.

Another way to look at the approach is to see the topology as a tree structure. What we then do is that we optimize every child of the node at which we are currently at, and select the one with the shortest tree. At this node we then again optimize all its children and select the shortest tree. When going backwards we then just select the second best, third best and so on.

This approach, in which we optimize all children of a topology and then select which vector to continue the depth-first search on, can be described as a best-first search. It may seem a bit elaborate in contrast to performing a simple depth-first search in which we e.g. always keep as far to the left as possible in the tree and optimize a topology of 6 terminals as

$$\begin{aligned}
A &= [1] \\
A &= [1,1] \\
A &= [1,1,1] \\
A &= [1,1,2] \\
&\vdots \\
A &= [1,2] \\
A &= [1,2,1] \\
&\vdots \\
A &= [2] \\
&\vdots
\end{aligned}$$

Best-first search however turns out to be much more efficient than simple depth-first. The new implementation did not use best-first initially but instead just depth-first. This caused a significant slow-down when compared to the original implementation[16]. Because of this, the new implementation now uses the best-first approach.

The reason that the best-first search is more effective lies in the fact that we often will be able to prune more topologies than with the depth-first search. When using the best-first we in general get better upper bounds much faster, and thus we become able to prune more topologies and do so faster. Imagine e.g. that we have $n = 6$. In simple depth-first we would simply start to plow through the topologies descending from $A = [1]$. However it may very well be that $A = [3]$ shows a lot more promise, and if we had optimized this we

[16] No formal tests were made as the original implementation was several magnitudes faster, and thus it was obvious that something needed to be changed.

could have pruned both $A = [1]$ and maybe also $A = [2]$. This is exactly what best-first tries to do by always going the route which currently shows the most promise.

4 Data Structures and Methods

This chapter will describe some more notable elements of the new implementation, with regard to data structures and methods not used by Smith [13]. Specifically it describes the new data structure introduced for holding and building topologies. It also describes the new iteration based on the simple iteration described in Section 3.3.1 and using the analytical method for solving the Fermat-Torricelli problem presented by Uteshev [19]. This also includes a description of said method, and a generalization to d-space, as the one presented by Uteshev [19] only is for 2D. Finally the chapter also describes a new way of sorting the terminals.

4.1 Building Topologies

In Smith's implementation of the algorithm when generating topologies every new topology is rebuilt completely from the bottom. E.g. say that we have just optimized the topology corresponding to the topology vector $A = [1, 2]$. We now wish to optimize the topology vector $A' = [1, 2, 3]$. In the original implementation A' would have to be built again from scratch instead of simply splitting and building on to A.

To avoid this extra work, the new implementation instead uses a different approach. The underlying data of a tree is in most aspects more or less identical to Smith's as it is a series of arrays to hold points and edges. However, these are coupled together in a struct (in Go, the programming language of choice, this is essentially a record on which one can add methods.). Instead of rebuilding this entire struct every time we split or restore the topology, it has methods to do exactly this. Thus, using the earlier example, from A we could either split an edge and get A', or any other length three topology vector having A as a prefix, or we could restore and go back from A to $A'' = [1]$[1]

The way this is implemented is a follows: All edges of the tree are kept in n list. An edge is represented as the tuple (a, b) meaning it starts at point p_a and

[1]From A'' we could then either split again to e.g. the next topology vector of length two, $[1, 3]$, or restore once again.

ends at point p_b. Say we have n terminals in all, and have currently connected the i first terminals. This means we have $i-2$ Steiner points continuously enumerated from $n+1$ up to $n+i-1$ and $2i-3$ edges enumerated continuously from 1.

Now we wish to split some edge $e_j = (c, d)$ where $1 \leqslant j \leqslant 2i-3$, so any of the existing edges, and connect the next terminal p_{i+1} to the existing topology. We do this by first changing the original edge to go from the first end point p_c to a new Steiner point p_{n+i}. Afterwards we append two new edges to the list of edges, $e_{2i-2} = (d, n+i)$ and $e_{2i-1} = (i+1, n+i)$ thus connecting p_d and p_{i+1} to Steiner point p_{n+i} as well. This process is straight forward and if the operation of accessing and appending to the list is done in $\mathcal{O}(1)$ time[2] then the entire operation will be done in $\mathcal{O}(1)$ time. The operation can be seen in Figure 4.1. In this way we can expand the topology one terminal at a time without rebuilding every time.

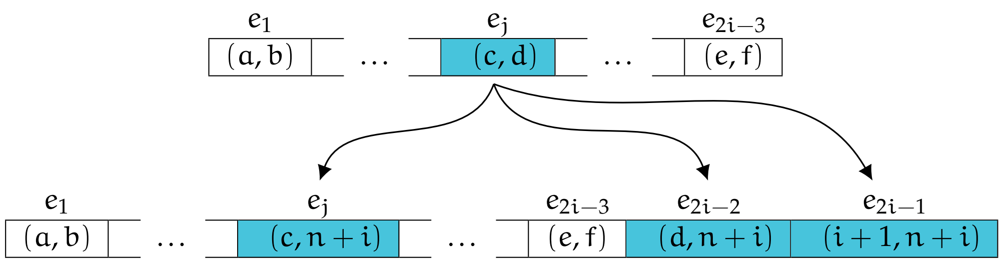


Figure 4.1: Splitting on edge e_j with i terminals currently connected, results in two new edges e_{2i-2} and e_{2i-1} and a change of the existing edge e_j. Finally we also get the new Steiner point $n+i$.

The reason for this way of expanding the topology however lies with the wish to also restore an edge. If we keep a list of all the edges we split on[3] we can always restore the last edge split in the following way. Select the last two edges, $(d, n+i), (i+1, n+i)$ in the list and the edge $e_j = (c, n+i)$. Now change edge e_j to (c, d) and remove the last two edges in the list. Thus we have $\mathcal{O}(1)$-time methods for splitting and restoring the topology, in a way which is consistent in the numbering with the one described in Section 3.2.1 and shown in Figure 3.1.

4.2 Analytical Solution to the Fermat-Torricelli Problem

An analytical solution for finding the Fermat-Torricelli point of three prespecified points in 2D is presented by Uteshev [19]—which is the same as finding

[2]which is the case if enough space is allocated for the list ahead of the appending.
[3]which we do, as this is the topology vector described in Section 3.2.1.

the Steiner point of those three points. The article presents a solution for two dimensions which will be described shortly. However as we are working in $\mathbb{R}^d$ a generalization of the solution to d-space is necessary and will also be presented. This generalization is as far as I know, not presented anywhere else in the literature.

The reason why this is interesting is that Smith [13] mentions that an iterative process which updates points one at a time would not be as quick to converge as the one he presents. There is however no experiments in the paper to support this. Thus this analytical method is interesting, as it is actually not computationally heavy to compute the location of a Steiner point, and as each optimization moves a Steiner point to the exact location at which it minimizes the length to its three neighbors at the time. One could therefore think that this might actually converge quite fast.

Using this analytical solution an optimization iteration similar to the one described in Section 3.3.1 has been implemented. The method roughly follows the one described, but also re-uses the main-loop of the method using Smith's iteration, with its conditions for pruning and stopping.

4.2.1 2D

The solution for two dimensions is presented by Uteshev [19] is for the generalized Fermat-Torricelli problem, which given three non-colinear points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $p_3 = (x_3, y_3)$ in the plane, finds the point $p_* = (x_*, y_*)$ which gives a solution to the optimization problem

$$\min_{(x,y)} F(x,y) \quad \text{for} \quad F(x,y) = \sum_{j=1}^{3} m_j \sqrt{(x - x_j)^2 + (y - y_j)^2}$$

Where the weight of the jth point, m_j, is a real positive number. The instance where $m_1 = m_2 = m_3 = 1$ is exactly the classical Fermat-Torricelli problem, which has a unique solution either coinciding with one of the points p_1, p_2, p_3 or with the Fermat-Torricelli or Steiner point of the triangle $p_1 p_2 p_3$. The instance with unequal weights is known as the generalized Fermat-Torricelli point.

As we only need the classical version of the problem, I have decided to rewrite the proof done by Uteshev [19] for the 2D case with the weights set to 1. In a lot of ways the proof is the same, though it does become a bit simpler and cleaner. The original proof for the generalized version can be found in Uteshev [19]. Thus what we wish to minimize is instead the following problem:

$$\min_{(x,y)} F(x,y) \quad \text{for} \quad F(x,y) = \sum_{j=1}^{3} \sqrt{(x - x_j)^2 + (y - y_j)^2}$$

Existence and uniqueness of the solution is guaranteed by Theorem 4.1

Theorem 4.1. *Denote the corner angles of the triangle* $\triangle p_1p_2p_3$ *by* $\alpha_1, \alpha_2, \alpha_3$. *Then if the conditions*

$$\begin{cases} 1 < 2 + 2\cos\alpha_1, \\ 1 < 2 + 2\cos\alpha_2, \\ 1 < 2 + 2\cos\alpha_3, \end{cases} \tag{4.1}$$

are fulfilled, corresponding to all angles being less than $120°$, *then there exists a unique solution* $p_* = (x_*, y_*) \in \mathbb{R}^2$ *for the classic Fermat-Torricelli problem lying inside the triangle* $\triangle p_1p_2p_3$. *This point is a stationary point for the function* $F(x,y)$, *i.e. a real solution of the system*

$$\begin{cases} \frac{d(F(x,y))}{dx} = \sum_{j=1}^{3} \frac{(x-x_j)}{\sqrt{(x-x_j)^2+(y-y_j)^2}} = 0, \\ \frac{d(F(x,y))}{dy} = \sum_{j=1}^{3} \frac{(y-y_j)}{\sqrt{(x-x_j)^2+(y-y_j)^2}} = 0 \end{cases} \tag{4.2}$$

If any of the conditions in Equation (4.1) is violated then $F(x,y)$ *attains its minimum value at the corresponding vertex of triangle.*

Calculating the coordinates of the point (x_*, y_*), if the conditions in Equation (4.1) are fulfilled, is done as follows

$$x_* = \frac{K_1K_2K_3}{2\sqrt{3}Sd}\left(\frac{x_1}{K_1} + \frac{x_2}{K_2} + \frac{x_3}{K_3}\right), \quad y_* = \frac{K_1K_2K_3}{2\sqrt{3}Sd}\left(\frac{y_1}{K_1} + \frac{y_2}{K_2} + \frac{y_3}{K_3}\right) \tag{4.3}$$

with

$$F(x_*, y_*) = \min_{(x,y)} F(x,y) = \sqrt{d} \tag{4.4}$$

Here

$$r_{jl} = \sqrt{(x_j - x_l)^2 + (y_j - y_l)^2} = |p_jp_l| \quad \{j,l\} \in \{1,2,3\} \tag{4.5}$$

$$S = |x_1y_2 + x_2y_3 + x_3y_1 - x_1y_3 - x_3y_2 - x_2y_1|$$

$$\begin{cases} K_1 = \frac{\sqrt{3}}{2}(r_{12}^2 + r_{13}^2 - r_{23}^2) + S \\ K_2 = \frac{\sqrt{3}}{2}(r_{12}^2 + r_{23}^2 - r_{13}^2) + S \\ K_3 = \frac{\sqrt{3}}{2}(r_{23}^2 + r_{13}^2 - r_{12}^2) + S \end{cases} \tag{4.6}$$

$$d = \frac{1}{\sqrt{3}}(K_1 + K_2 + K_3)$$

The proof of Equations (4.3) and (4.4) can be found in a generalized version in Uteshev [19], or in Appendix B.1 where it has been adapted to the unweighted version of the problem.

4.2.2 Generalization to d-Space

Generalizing the solution described above to higher dimensions is in general pretty straightforward. The biggest change is in the representation of S, and the fact that the proof requires a bit of sum manipulation. This generalized version of the proof is not known to have been presented in any paper before.

The generalization is as for the 2D variant here done for the classic Fermat-Torricelli problem. It could however just as easily be done for the weighted problem, by following the form of the proof in Uteshev [19] instead of the form in Section 4.2.1.

The first change is that we no longer try to minimize a function of just two parameters, but instead we intend to minimize a function of d parameters, where our points are contained in $\mathbb{R}^d$, i.e. we have d dimensions. As we normally represent this using vectors, we would not actually have d parameters, but instead one parameter being a d-vector. Thus we wish to find a point $p_* = (x_{(*,1)}, x_{(*,2)}, \dots, x_{(*,d)}) \in \mathbb{R}^d$ which minimizes the following function for the three points $p_1 = (x_{(1,1)}, x_{(1,2)}, \dots, x_{(1,d)}), p_2 = (x_{(2,1)}, x_{(2,2)}, \dots, x_{(2,d)}), p_3 = (x_{(3,1)}, x_{(3,2)}, \dots, x_{(3,d)}) \in \mathbb{R}^d$.

$$\min_{(x_1,x_2,\dots,x_d)} F(x_1,x_2,\dots,x_d) \quad \text{for} \quad F(x_1,x_2,\dots,x_d) = \sum_{j=1}^{3} \sqrt{\sum_{i=1}^{d} (x_i - x_{(j,i)})^2}$$

$$\Updownarrow$$

$$\min_{(p)} F(p) \quad \text{for} \quad F(p) = \sum_{j=1}^{3} |pp_j|, \quad p \in \mathbb{R}^d$$

The point p_* now no longer has to be a solution to the equation system in Equation (4.2), but to a similar system containing d equations, as follows

$$\left.\begin{array}{l} \sum_{j=1}^{3} \frac{(x_1 - x_{(j,1)})}{|pp_j|} = 0 \\ \sum_{j=1}^{3} \frac{(x_2 - x_{(j,2)})}{|pp_j|} = 0 \\ \vdots \\ \sum_{j=1}^{3} \frac{(x_d - x_{(j,d)})}{|pp_j|} = 0 \end{array}\right\} = \sum_{j=1}^{3} \frac{(p - p_j)}{|pp_j|} = \vec{0} \tag{4.7}$$

Then as before, if the conditions in Equation (4.1) are fulfilled[4] the solution to Equation (4.7) can be found as follows

$$p_* = (x_{(*,1)}, x_{(*,2)}, \dots, x_{(*,d)}) \tag{4.8}$$

with

$$x_{(*,j)} = \frac{K_1 K_2 K_3}{2\sqrt{3} S d} \left(\frac{x_{(1,j)}}{K_1} + \frac{x_{(2,j)}}{K_2} + \frac{x_{(3,j)}}{K_3} \right), \quad 0 < j \leqslant d \tag{4.9}$$

[4]If not the point is as before located at the corresponding vertex.

and

$$F(p_*) = \min_{(p)} F(p) = \sqrt{d}$$

Here

$$r_{jl} = \sqrt{\sum_{i=1}^{d} (x_{(j,i)} - x_{(l,i)})^2} = |p_j p_l| \quad \{j,l\} \in \{1,2,3\} \tag{4.10}$$
$$S = \frac{1}{2}\sqrt{(r_{12} + r_{23} + r_{13})(r_{23} + r_{13})(r_{12} + r_{13})(r_{12} + r_{23})}$$

And the rest of the expressions are the same as in the 2D version. As can be seen, all of the expressions, except for S (and the distance which now of course have more than two dimensions), retain the same representation as in the two dimensional version. S however looks quite different. Exactly how the new S is related to the old S is still unclear to me. Uteshev [19] describes that in 2D if one rewrites S it can be recognized as the doubled area of the triangle $\triangle p_1 p_2 p_3$. This geometrical understanding however seems to be missing from the higher dimension case. Indeed the idea for the representation of S stems from talks with my supervisor Pawel Winter, who pointed me to an earlier implementation of the general case, that he has done himself. Thus an intuition of the new S is missing, however as can be seen from the proof Appendix B.2, it does indeed work.

4.3 Sorting the Terminals

The idea of sorting the terminals before one starts to build and optimize topologies is not new, and has also been suggested by Smith [13]. The idea is to find some way of sorting the terminals, such that the initial RMTs becomes as large as possible, as fast as possible. Doing this allows us to early on see if we need to discard a topology vector as prefix, if its RMT is longer than the upper bound, and thus be able to prune more topologies than if we added terminals in an arbitrary order.

At least two different kinds of sorting has been implemented by Fonseca et al. [8] and Van Laarhoven and Anstreicher [20]. Both methods have shown promising results regarding the number of trees one needs to optimize in contrast to not sorting the terminals.

The first method, presented by Van Laarhoven and Anstreicher [20], sorts the terminals by their decreasing distance to the centroid of the set of terminals, i.e. the first terminal is the farthest away, and the last terminal is the closest. A potential issue with this way of sorting, is that while the terminals may be far from the centroid, they can potentially still be rather close to each other, thus not pruning much faster.

The second method, presented by Fonseca et al. [8], sorts the terminals such that the first three terminals p_1, p_2, p_3 maximize the sum of their pairwise distances, and $p_i, i = 4, 5, \ldots, n$ is the farthest away from $p_1, p_2, \ldots, p_{i-1}$. This method seems to have a lot of potential as it tries to ensure that every terminal added is as far away from the current RMT as possible. However one could easily think of positions for the terminals, such that the terminals $p_4, p_5, \ldots, p_n$ all lie close to each other, and within the first three terminals. This could mean that the distance added every step after the initial RMT will be spread out on the next few steps, instead of adding as much as possible as quickly as possible.

I propose another method that is relatively similar to the one proposed and implemented by Fonseca et al. [8], but a lot simpler and requiring fewer steps. The initial idea for the method was to find the longest path in a fully connected graph, where the nodes in the graph are the terminals. This problem is however NP-hard [21], and it seems a bit silly to solve a NP-hard problem to try and improve on a NP-hard problem. Thus the method proposed here sorts the terminals as follows: Select the initial terminal p_1 arbitrarily. From there select the next terminal $p_i, i = 2, 3, \ldots, n$ such that the distance to p_{i-1} is maximized.

The idea behind this approach, is that the method is extremely easy to implement and hopefully gives better runtimes, comparable to those of e.g. Fonseca et al. [8].

5 Implementation

This chapter will describe the architecture of the new implementation, i.e. how the new implementation has been structured.

The new implementation has been written in the language Go that is still relatively new but is considered stable at this point. Quoting the homepage: "Go is an open source programming language that makes it easy to build simple, reliable, and efficient software." [15]. Go is a general-purpose, strongly typed and garbage-collected language. The syntax bears resemblance to C[1]. Furthermore Go has explicit support for concurrent programming. The language furthermore sports extremely fast compiling of source code and tries to have performance comparable to C.

The reasoning behind choosing Go for the new implementation were firstly that I already had experience with the language making it easy to get to work on the new implementation without also having to learn a new programming language. Secondly the language has, in my opinion, a very clean syntax which makes it easily readable for anyone familiar with any C-syntax styled language. Finally the easy use of concurrency in Go[2] made it interesting as continued work with the new implementation could include a potential and significant speedup by making the branching of the algorithm concurrent. The same could be be true for the Gauss elimination[3].

In general the implementation tries to follow guidelines given in *Effective Go* [4]. These are guidelines as to how one codes effective Go. Here effective means both in terms of speed, memory usage, stability and readability.

Source code for the new implementation, fixes for the original implementation

[1]But with a lot less braces, and no pointer arithmetic.

[2]Starting a new goroutine, which is the languages type of lightweight threads, is as simple as writing: `go SomeFunction (args...)`.

[3]No concurrency has actually been implemented in the new implementation. However the possibilities of this is discussed in Section 7.1.

and experiments can be found at `https://github.com/jsfr/SteinerExact`.

5.1 Overview

In general the code is structured into a package (or library) called `smt` and a main file (which is actually split into a main and configuration file). This is done to allow for easy reuse of the main amount of work, as it is a library.

The `smt` package consists of all the data structures and and their functions related to calculating SMTs. The main file contains the main function which initializes a configuration (using the configuration file) and then starts the main loop of the program with the correct parameters. The main loop generates topologies, similarly to Smith's main loop. Finally the configuration file consists of a data structure and function for reading the command-line flags.

The possible flags one can pass to the program are:

- `-sort` Sorts the terminals before initiating the main loop.
- `-1` 1-indexes all printed results on STDOUT. The program is 0-indexed as this simply makes more sense programming-wise. However one may wish a 1-indexed tree when printed for easier comparison with the original implementation and the theory.
- `-iteration=[method]` Selects the optimization method used. `[method]` can be either `simple` or `smith`. The default is `smith`.
- `-cpuprofile=[path]` Flag used to initiate a profiling of the program. If passed Go's internal profiling is enabled, and the trace is saved at `[path]`. This is only intended for debugging.
- `-help,-h` Shows this list of flags.

5.2 SMT Package

As Go is an imperative language we do not have classes, objects, instances and so on as in e.g. Java. We do however have structs similar to C, on which we can also define functions. Thus the `smt` package is structured such that most of the files of it is named similar to the struct and functions on it it holds. E.g. the file `smt/edge.go` this files contains the struct defining an edge, and the methods on this, such as calculating the length of the edge.

5.2.1 Structs

The most important structs for the new implementation are:

- **Point** Represents a d dimensional point. Implemented as a simple array of floats. For convenience points has a number of functions for subtracting points, calculating dot products, and finally the function for sorting terminals, which is simply implemented as a function that takes a list of points and sorts them as described in Section 4.3.

- **Edge** Represents an edge of a tree. The edge consists of a pointer to the tree and two integers which are the indicies of the end points of the edge in the point-array of the tree. The pointer is needed of for the indicies to make sense, and to easily calculate the length of the edge.

- **Tree** Represents a complete tree. The struct contains a list of all edges, a list of points, the dimension of the the points and the number of points which are terminals. Finally the struct also contains a list of adjacencies to the Steiner points, as this was considered the easiest way to hold these, instead of having to find them every time they are needed in the edges. The most important functions (besides the iterations) on a tree are those for splitting and restoring an edge, and the function for calculating the error of the tree.

- **Stack** The implementation of Smith's iteration utilizes quite a few stacks. These were in the original implementation just arrays on which Smith made sure to always index as if it was a stack. To ensure that the stacks are always used correctly, the new implementation instead implements a simple stack struct, which just consists of an array as the underlying structure, and then functions for pushing and popping elements of the stack.

5.2.2 Helpers

The `smt` package also contains a number of helpers. These are function for calculating the pertubed centroid of three points, calculating the Fermat-Torricelli point of three points, printing a trees representation to the console. Only the function for printing a tree to console can be accessed outside the package. The others are for internal use only.

5.2.3 Iterations

The package also contains a file, named `optimize.go` with an implementation of the two different iterations. The first one is the iteration described by Smith [13] (named `SmithsIteration`) and the second is the iteration based on the simple iteration described in Section 3.3.1 using the analytical solution for the Fermat-Torricelli problem presented by Uteshev [19] to place the Steiner points (named `SimpleIteration`).

The second iteration is relatively simple in its iteration. It iterates through all Steiner points, calculating their placement with regard to the other points' current placement.

The first iteration in general follows the same outline as done in the original implementation. In the same way it also utilizes a number of arrays and stacks which are global to the package (but inaccessible from the outside to avoid someone using the library from tampering with the data mid-iteration unintentionally). In general these global arrays is unwanted as it clutters the space of the other functions in the package. However initial runs of the iteration showed that the new implementation used a significant amount of time in the iteration on creating new local arrays every time it was being run. This was of course undesirable, as it both wasted time, and made comparison to the original code harder. Thus after some consideration, and for lack of a better solution, I decided to continue using global variables for the arrays and stacks of `SmithsIteration`.

5.3 Main and Configuration File

The new implementation contains a main file, and a configuration file.

The main file contains the main function being run when the program is executed on the command-line, and the main loop for enumerating and optimizing topologies. This is done using the structs and functions of the `smt` package, i.e. its public API. The main loop in general follow the same outline as the main loop of Smith's original implementation.

The main file furthermore uses the functions and structs of the configuration file for reading and handling the arguments given on the command-line. The implementation unlike Smith's implementation does not read a list of terminals on STDIN, but instead takes as argument a file-path to what it expects to be a JSON file containing a list, named *"points"*, consisting of equal length lists, corresponding to the terminals. The file uses standard JSON syntax [16].

5.4 Other Source Code and Files

Finally, at the repository for the implementation, `https://github.com/jsfr/SteinerExact`, one can also find code not directly related to the implementation, but instead related to the thesis as a whole. Apart from the implementation, the repository also contains all tree instances that has been used during the experiments, and all scripts for running and aggregating the experimental data. It also contains the runs on which Chapter 6 is based.

Furthermore the repository also contains version 3.1 of the GeoSteiner package, which can also be found at `http://www.geosteiner.com`, also used by the experiments and the implementation by Smith [13] with the bugfix for the if-condition described in Section 3.4.2 and added counting of trees and iterations.

6 Experiments

This chapter describes and discusses the experiments that have been performed in relation to the new implementation.

All experiments have been executed on computer with 8× Intel® Xeon® CPU E5-2630L at 2.00 GHz and with 16 GB RAM. Furthermore all tests were executed using version 1.4.2 of Go[1]. In the experiments the used methods are named as seen in Table 6.1.

Method	Description
`Simple`	New implementation using the analytical method. Implemented in Go.
`SimpleSort`	New implementation using the analytical method and with terminal sorting. Implemented in Go.
`SmithNew`	New implementation using Smith's iteration. Implemented in Go.
`SmithNewSort`	New implementation using Smith's iteration and with terminal sorting. Implemented in Go.
`SmithOld`	Original implementation by Smith [13], only slightly modified to fix the bug described in Section 3.4.2. Implemented in C.

Table 6.1: The table shows the used naming of the methods used during the experiments.

In general the experiments performed, relates to either correctness or performance of the implementation. The correctness has been measured by comparing a set of instances with the results found by the GeoSteiner program. The performance experiments is divided into experiments regarding the speed of the methods, the number of trees which are optimized, and the number of iterations being run.

[1]A newer version (1.5) which improves the performance of the language due to more efficient garbage collection was released after the experiments had been conducted.

6.1 Correctness

To test the correctness of the new implementation 150 random cube instances with $n = 10 \dots 12$ and $d = 2$ has been run. The cube instances are the same as used by Fonseca et al. [8], which can also be found at https://github.com/DIKU-Steiner/MPC15/tree/master/experiments/Correctness/Instances. The terminals of the instances are randomly distributed in a unit cube.

The experiments were run for all of the methods in Table 6.1, except `SmithOld`. The instances we also run using the official GeoSteiner implementation[2].

After all instances were completed, the results of the new implementation were compared with the results of the GeoSteiner program. This showed that the analytical solution, both with and without terminals sorting, in a few instances gave sub-optimal SMTs[3]. These and the difference to the optimal SMT found by GeoSteiner can be found in Table 6.2.

Instance	Method	Diff
cube_n12_d2_s27	`Simple`	+0.170%
cube_n12_d2_s49	`Simple`	+0.295%
cube_n10_d2_s42	`SimpleSort`	+0.015%
cube_n12_d2_s26	`SimpleSort`	+0.228%
cube_n12_d2_s43	`SimpleSort`	+0.292%
cube_n12_d2_s49	`SimpleSort`	+0.295%

Table 6.2: The table shows the instances in which some method gave sub-optimal results, and the difference in percent to the optimal SMTs found by the GeoSteiner program.

As can be seen `Simple` gave sub-optimal results in $2(1.3\%)$ instances, and `SimpleSort` in $4(2.7\%)$ instances. Neither `SmithNew` or `SmithNewSort` gave any errors.

When inspecting the SMTs generated by the simple method in the sub-optimal instances, it was clear, that the method in these cases indeed gave different topologies from the topologies found by the GeoSteiner program.

The reason for this, I believe, can be traced back to two things: the if-clauses in the implementation used when optimizing, and the possibility that the error function defined by Smith is faulty.

Consider the code snippet in Listing 6.1 or equivalently the flowchart in Figure 6.1, which is from the original implementation found in Smith [13]. What happens in the snippet is the following: before any optimization iterations

[2]The implementation can be found at downloaded at http://geosteiner.com

[3]By sub-optimal we mean that the length of the tree found is longer than the length of the optimal SMT. This may also mean that the topology is different.

```
ITER:
q = length();
r = error();
if ( q - r < STUB) {
  if (r > 0.005*q) {
    optimize(0.0001*r/NUMSITES);
    goto ITER;
  }
  /* Truncated code.
     Pushes topology and length to stack.
     Continue working with the topology and its descendants */
}
/* Continue to next topology vector of same length,
   if we have no been in the outer if-clause then all
   descendants of the current topology vector is
   effectively pruned as we have not stored it on the stack. */
```

Listing 6.1: The outer if-clause decides whether we prune a topology or not. This is applied before the topology is optimized to its RMT, which significantly speeds up the program. But whether this is allowed is unclear. The code snippet is from Smith's implementation, given in Smith [13]. The new implementation in Go uses the same two conditions.

has taken place, the tree length and error of the tree is calculated (afterwards it also occurs as it happens just after jumping to the label from the goto). A check (the first if-clause) whether the tree length minus the error is lower than the upper bound ($q - r < STUB$) is then performed. If the check yields false, the main loop continues without pushing the topology vector to the stack described in Section 3.4.2. This means that any descendant of that topology vector will not be optimized, i.e. they are pruned. If the check yields true, the program goes to the second if-clause which checks if the error is greater than $0.005 \cdot q$. If the second check yields true, the tree is optimized, and the program return to the `ITER` label. If it yields false, the program continues on, within the first if-clause, and eventually push the topology vector to the stack.

The structure just described means that we can both discard a topology before optimizing it, or after optimizing some number of iterations. This significantly speeds up the program, in contrast to optimizing the topology to its RMT and then deciding whether it must be pruned or not (by comparing its length to the upper bound). The new implementation therefore retained this structure.

However as described in Section 3.4.2 there seems to be no direct relation between the tree length and error of the tree. Also as described in Section 3.4.1 it is unclear whether the error function is even correct. Thus both of these if-clauses are a bit sketchy.

Consider the first if-clause. It assumes that subtracting the current error r from the current tree length q, will always yield something which is lower than or equal to the length of the RMT of the topology q^* (seen in Figure 6.2a). If this is not the case, then it could be possible to get a tree which if optimized would be shorter than the current upper bound, but where subtracting the current

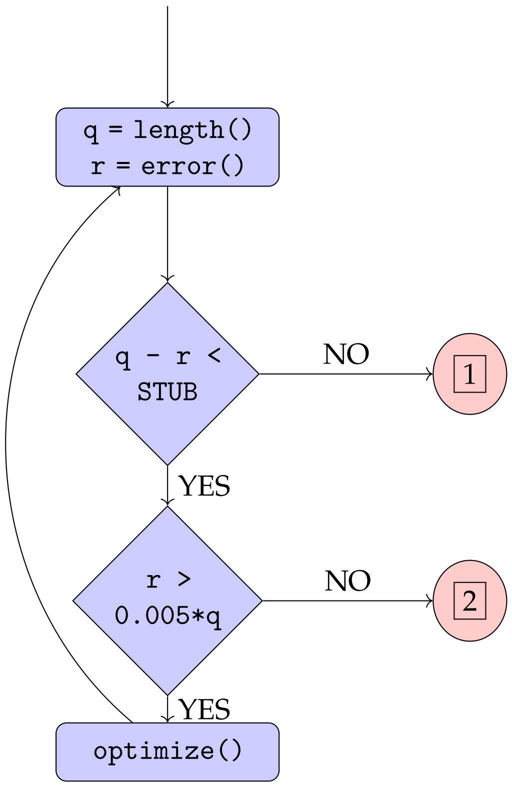
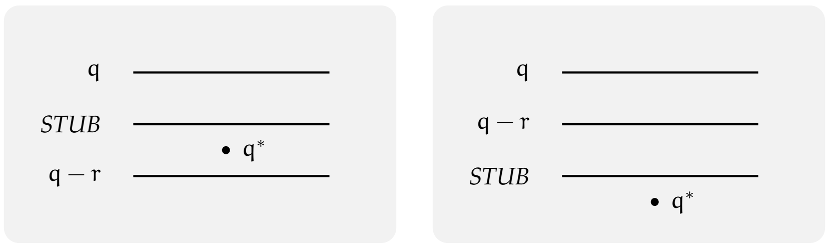


Figure 6.1: The flowchart shows the structure of the he code from Listing 6.1. $\boxed{1}$ = *Reject the topology and its descendant.* $\boxed{2}$ = *Error is sufficiently small. Push on stack for future expansion or fine tuning.*



(a) Example where subtracting the error from the tree length yields a length shorter than the current upper bound. In this case the program would continue to optimize on the topology, and there is no risk of prematurely pruning it.

(b) Example where subtracting the error from the tree length yields a length greater than the current upper bound. If it is not always the case that $q - r$ *is shorter than or equal to the tree length of RMT* q^**, this could result in the topology being pruned being pruned prematurely.*

Figure 6.2: The possible situations when reaching the first if-clause.

error from the current tree length would not be shorter than the current upper bound (seen in Figure 6.2b). This could result in a topology being thrown away even if it should have actually been kept. Furthermore the possible situations after optimizing the topology is shown in Figure 6.3.

To test the impact of the first if-clause the instances were run again where the condition of the first if-clause was changed to $q - 10 \cdot r < STUB$. The observed result that some of the instances which before gave sub-optimal results were solved to optimality, while other new ones gave sub-optimal results. These can be seen in Table 6.3. As can be seen `Simple` and `SimpleSort` were still the only methods to give sub-optimal results.

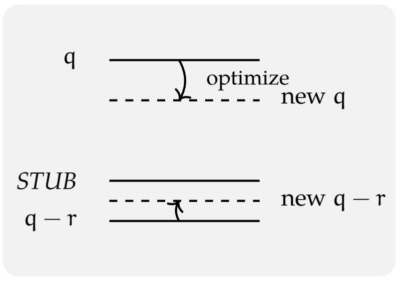
Instance	Method	Diff
cube_n11_d2_s9	`SimpleSort`	+0.262%
cube_n12_d2_s12	`Simple`	+1.405%
cube_n12_d2_s43	`SimpleSort`	+0.292%
cube_n12_d2_s9	`SimpleSort`	+0.214%

Table 6.3: Changing the condition of the first if-clause to $q - 10 \cdot r < STUB$ causes the number of sub-optimal results drop to 4. However some of these are were not sub-optimal before the change.

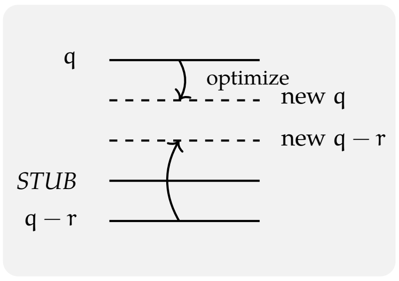
The instances which were not solved optimally before but were after the change, seem to indicate that this condition indeed is partly to blame for the sub-optimality. The reason that new instances suddenly gave sub-optimal results, I believe, is because these instances now explore topologies which before were discarded.

Now consider the second if-clause. This assumes two things, firstly that the size of the error is proportional to the tree length, and secondly that when the error for the entire tree is low that the tree entire tree then optimal. The first assumption does have some merit. If one looks at the error function, as it is defined in Equation (3.2) it is clear that if we have longer edges, they will contribute more than shorter edges. Whether there is a direct relation between the error and the tree length is however still unclear. The second assumption however faces some problems in the new implementation. This assumes that we will not hit a a situation in which most of the tree is optimal, but e.g. a single Steiner point contributes a large error and is at a sub-optimal placement. This is a problem, as a single Steiner point at a wrong position can propagate changes throughout the tree when it is optimized. It may therefore be, that the tree is not at all optimal, and that we risk stopping prematurely.

This may not have been a problem in the original implementation which rebuilt the topology every time it was changed, however the new implementation does not rebuild the entire topology, but reuses the already found placements



(a) Example where optimizing the tree results in a new shorter tree length, but where the error is still large enough that q − r *stays below the upper bound. In this case we would proceed to either optimize the topology, or put it on the stack, depending on the what the second if-clause yields.*



(b) Example where optimizing the tree results in a new shorter tree length, and the error becomes small that q − r *is longer than the upper bound. In this case we prune the topology and all its descendants. If* q − r *is always less than or equal to the length of the RMT of the topology this is correct. If this is however not the case this could result in premature pruning.*

Figure 6.3: The possible situations when optimizing a topology.

Instance	Method	Diff
cube_n12_d2_s27	`Simple`	+0.170%
cube_n12_d2_s49	`Simple`	+0.295%
cube_n10_d2_s42	`SimpleSort`	+0.015%
cube_n12_d2_s26	`SimpleSort`	+0.228%
cube_n12_d2_s38	`SimpleSort`	+0.071%
cube_n12_d2_s43	`SimpleSort`	+0.291%
cube_n12_d2_s49	`SimpleSort`	+0.295%

Table 6.4: Changing the condition of the second if-clause to $r > 0.0005 \cdot q$ *causes the number of sub-optimal results increase to* 7*.*

when splitting the topology. Thus the issue might be magnified by this new approach.

A question that emerges is why the issue with sub-optimality only occurs when using `Simple` and `SimpleSort` but not when using `SmithNew` and `SmithNewSort`. I.e. why does the analytical solution show this issue, but not Smith's iteration?

The reason for this, I believe, is that the analytical solution when it runs in general obtains much lower error-figures than Smith's iteration. I therefore believe that the issue exists for both iteration methods, but is magnified and therefore only seen when using the analytical solution.

The potential problem is summed up by Figure 6.4. We may have some topology which has been optimized to have a very low error, this is particularly likely using the analytical solution which attains very low error-figures. Upon splitting the topology and adding the next terminal and a new Steiner point, the error is raised. However because the error of the rest of the tree is very low, the error is not raised enough for the new $q - r$ to stay below the current upper bound, and as such the topology will be pruned even though optimizing it may result in the new Steiner point moving and propagating changes throughout the tree, which could have resulted in a new upper bound.

The problem could also escalate over the course of a few of these situations in a row, where one adds a terminal, but does not optimize as the error continues to stay below the threshold.

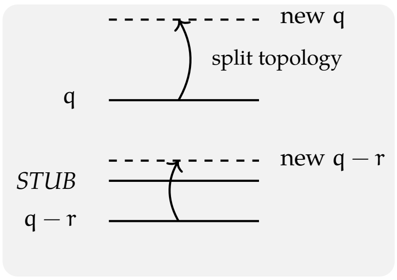


Figure 6.4: Splitting a topology raises the error of the tree. However if the error does not go up enough for the tree length minus the error to stay below the upper bound, the topology will be pruned. This situation again assumes that $q - r$ will also less than or equal to the length of the RMT for the topology. If this is not the case, it could be thought that this situation could also occur when the tree should no be pruned, but because of a low error on most of the tree it will be pruned prematurely.

To test the impact of the second if-clause the instances were run with the condition of the clause change to $r > 0.05 \cdot q$ and $r > 0.0005 \cdot q$. The sub-optimal results from these runs can be found in Tables 6.4 and 6.5. As can be seen, allowing a higher error gives fewer sub-optimal results, whereas setting the error even lower raises the number of sub-optimal instances produced by the analytical solution.

Instance	Method	Diff
cube_n12_d2_s27	`Simple`	+0.170%
cube_n12_d2_s43	`SimpleSort`	+0.292%

Table 6.5: Changing the condition of the second if-clause to $r > 0.05 \cdot q$ causes the number of sub-optimal results drop to 2.

These observations seem to align with the before described problems. Consider allowing a higher error in the second if-clause. This would cause the method to halt earlier in the optimization process, and thus the chance of $q - r$ being lower than the upper bound would probably be better, as we would less often hit a situation where most of the tree has very low error and only a single Steiner point has high error. However lowering the allowed error would cause the program to further optimize the trees, which would further magnify the problem where all but one Steiner point has very low error.

Further raising the error allowed in the second if-clause to $r > 0.5 \cdot q$ resulted in sub-optimal results being observed across all four of the tested methods. This is due to the fact the we at this point have raised the error so much that the optimal trees cannot be obtained in all situations. This also shows that we cannot simply raise the allowed error for the entire tree to ensure we will not get sub-optimal results, as it will have to walk a tightrope between sub-optimality due to to high errors, or sub-optimality due to the previous described issues.

To correct this problem I believe one would have to do two things. Firstly, before deciding whether to prune a topology and its descendants one would have to optimize the tree until it was the RMT, i.e. drop the condition $q - r < \mathit{STUB}$, and instead optimize until we have the RMT and then simply check $q < \mathit{STUB}$, or perhaps $q - \epsilon < \mathit{STUB}$ where ϵ would be some very small positive number. This would prevent pruning of topologies prematurely. Secondly, at least when using the analytical solution (*Simple* and *SimpleSort*) the second if-clause should not look at the error of the entire tree, but instead at the error of each individual Steiner point. This would prevent the situation where most of tree has an error of zero, and one point contributes most of the error. This approach might also be the correct one for *SmithNew* and *SmithNewSort* as they also, in contrast to *SmithOld*, do not re-position the already placed Steiner points when splitting the topology (due to the new method were we can split and restore a topology, in contrast to the original method where one had to rebuild the tree from the topology vector every time it was changed).

As the issue was first discovered during the experiments, the current implementation does not fix it. I however believe this could be done using the changes described above. It is however unclear how much this would affect the run-time of the program.

However I do not believe it invalidates the performance data completely, as the number of sub-optimal trees in the correctness test, was still very low, and the sub-optimality of these trees also was relatively low.

6.2 Performance

To test the performance of the implementation the test sets in Table 6.6 have been run for all of the methods described in Table 6.1. The *Carioca* set is from the 2014 DIMACS challenge, the *Iowa* set is from Fampa and Anstreicher [5], *Cube* and *Sausage* are both from Fonseca et al. [8]. The *Cube* set is a set of randomly generated instances which has the terminals of each instance distributed randomly in a unit cube. The *Sausage* set consists of d-sausages, which are linear concatenations of regular d-simplices. these have the lowest currently known Steiner ratio in dimensions 3 and up [14].

Set	Dimensions	Terminals	Set size	Point configuration
Carioca	$d = 3 \ldots 5$	$n = 11 \ldots 16$	90	Random in cube
Cube	$d = 2 \ldots 4$	$n = 10 \ldots 15$	360	Random in cube
Iowa	$d = 3 \ldots 5$	$n = 10$	30	Random in cube
Sausage	$d = 2 \ldots 5$	$n = 10 \ldots 15$	24	Simplex sequence

Table 6.6: The table shows the tests sets, their dimensions, number of terminals, number of instances and type of instances, which has been used to test the performance of the implementation.

The sets *Carioca* and *Cube* have been pruned. This has been done due to the sheer size of these sets. Even with these sets pruned, running all of the sets for all of the methods took a little more than a week in which a quad-core computer ran at all hours of the day. Furthermore all runs were set up such that they would halt if more than 12 hours passed, and it was then concluded infeasible to solve that specific instance using the given method. How many instances completed before the time limit can be seen in Table 6.7.

A few initial test runs showed that, especially `SimpleSort` but also occasionally `SmithNewSort`, could solve instances for $n = 17$ within the 12 hour limit. Time-wise it was however still infeasible to perform a proper test with these instance sizes. Thus the new implementation solves instances about the same size as Fonseca et al. [8].

The data collected from the runs have been used to compare the speed (i.e. run-time), the number of iterations, and the number of trees optimized. The results are presented and described in the two sections below.

	Method				
Set	`Simple`	`SimpleSort`	`SmithNew`	`SmithNewSort`	`SmithOld`
Carioca	81/90	89/90	73/90	87/90	76/90
Cube	✓	✓	352/360	✓	356/360
Iowa	✓	✓	✓	✓	✓
Sausage	22/24	23/24	18/24	22/24	21/24

Table 6.7: The table shows the number of successful instances, i.e. instances which finished within 12 hours, of the test sets which has been run. A checkmark indicates all instances of the set was solved.

6.2.1 Speed

To compare the speed of the new implementation with the implementation by Smith [13] box-plots of the run-times for *Carioca*, *Cube* and *Iowa*, have been plotted in Figures 6.5 to 6.9. The y-axis of the box-plots is logarithmic to make the plots manageable. Notice that Figures 6.5 to 6.8 have the number of terminals on the x-axis and are for fixed dimensions, whereas Figure 6.9 has the dimensions on the x-axis and is the for a fixed number of terminals. This is simply due to the fact that *Iowa* only contains $n = 10$ instances, whereas the other sets contain several instances of different dimensions and number of terminals. This also illustrates that the number of dimensions affects the run-time, but not nearly as much as the number of terminals does.

As can be seen in the figures the running time of `SmithNew` (implemented in Go) is in general a bit higher than that of `SmithOld` (implemented in C), whereas the new analytical solution `Simple` (implemented in Go) in general is lower. The reason for the higher running time of `SmithNew` is probably due to the fact that the implementation is done in Go instead of C as the original implementation. Unfortunately we are therefore not really able to say anything about the performance of the new data structure for topologies (described in Section 4.1) being better of worse than the original implementation. It is however interesting to see that the analytical solution seems to have better running times than not only `SmithNew`, but also `SmithOld`, as this suggests that an implementation of this method in a more efficient language, such as C, would yield even better running times[4]. Further this seems to debunk the claim by Smith [13] that a simple iteration would converge slower, at least speed-wise. Of course we still need to keep in mind the fact that `Simple` have resulted in some sub-optimal trees. It however seems unlikely, when looking at the percent of sub-optimal results that this should skew the results so much that this would be changed.

As can furthermore be seen in the figures, the run-time of both methods drop significantly when using terminal sorting (`SmithNewSort` and `SimpleSort`). In almost all cases, no matter the number of terminals or dimensions, is the third quartile of for the methods with terminal sorting below the first quartile for the respective method without terminal sorting. Furthermore in most cases is the third quartile of `SimpleSort` below the first quartile of `SmithOld`, and the run-times for `SmithNewSort` are better than or comparable to that of `SmithOld`.

The reason that the *Carioca* and *Iowa* for $d = 5$ and *Cube* for $d = 2$ have not been plotted is, both that they show the same situation, but also that $d = 2$ is so small that they go below the graph, and would require a change of y-axis,

[4]Thus in retrospect the new implementation should maybe have been done in C, especially as the concurrency of Go. A potential scheme for concurrency is described and discussed in Section 7.1.

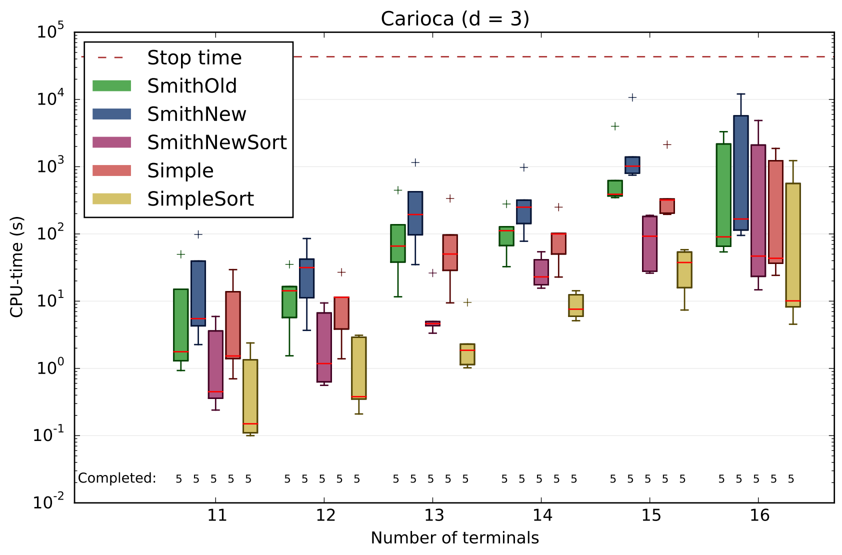


Figure 6.5: Box-plot showing the run-times for the Carioca set with $d = 3$.

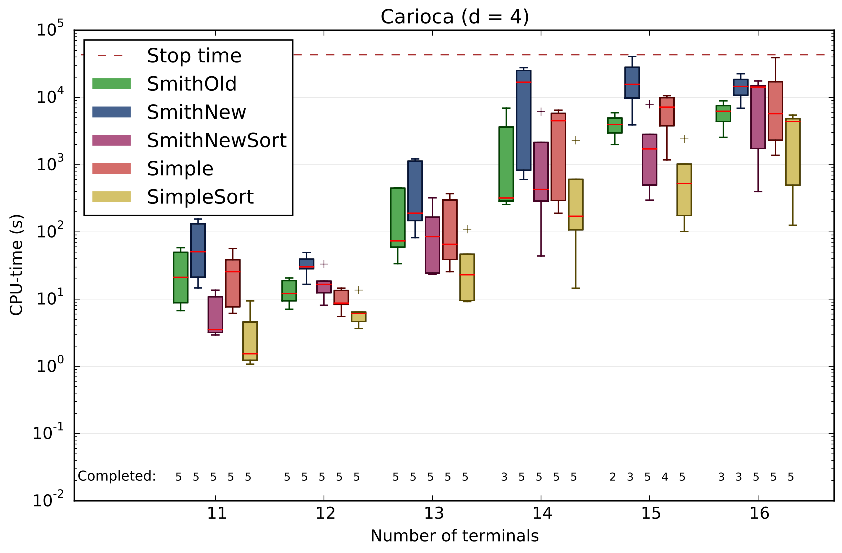


Figure 6.6: Box-plot showing the run-times for the Carioca set with $d = 4$.

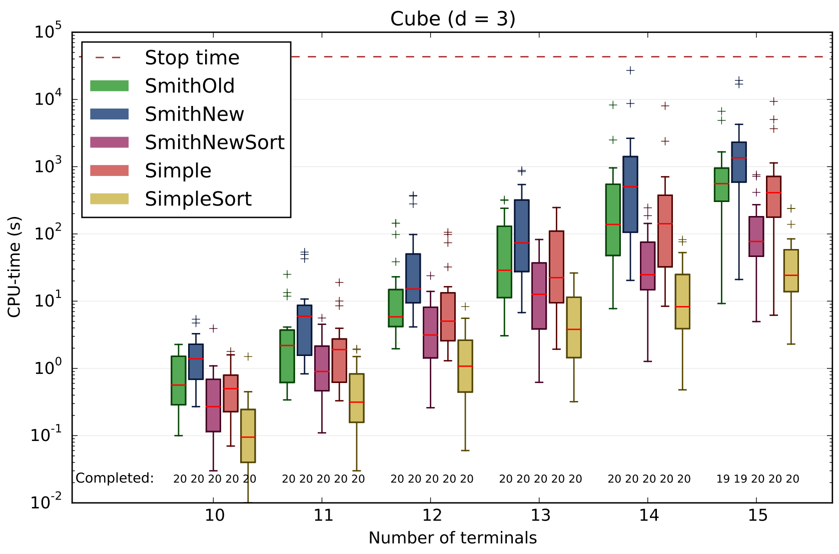


Figure 6.7: Box-plot showing the run-times for the Cube set with $d = 3$.

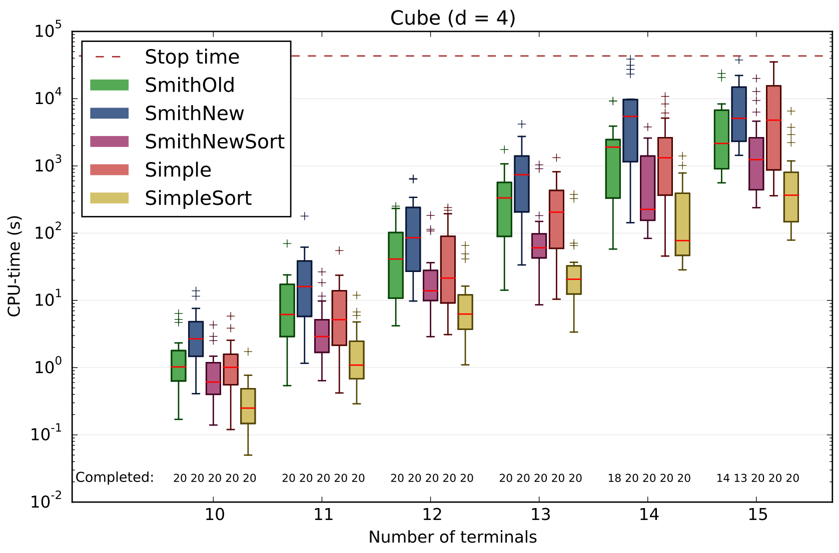


Figure 6.8: Box-plot showing the run-times for the Cube set with $d = 4$.

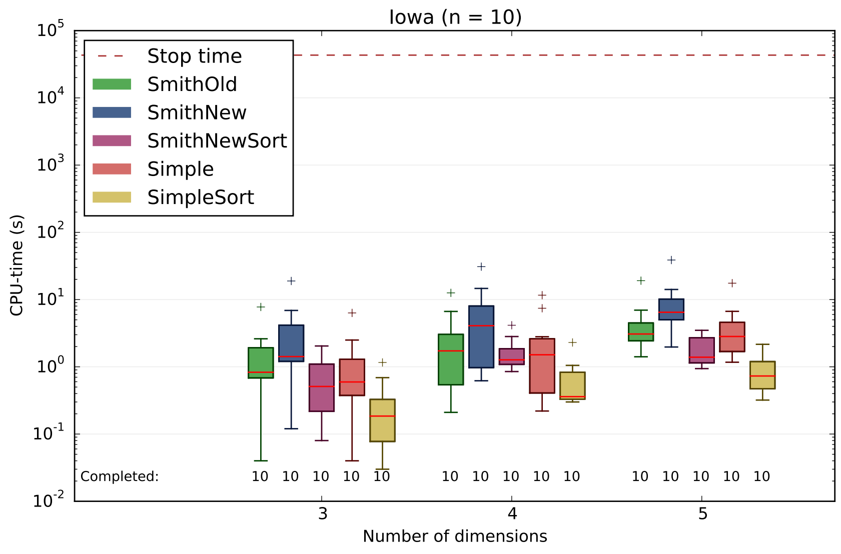


Figure 6.9: Box-plot showing the run-times for the Iowa set with $n = 10$.

and $d = 5$ has a bit too few solved instances for $n = 15$ for it to be quite as reliable.

6.2.2 Trees and Iterations

Apart from the actual run-time of the implementation it is also worth measuring the number of optimization iterations one runs when solving an instance, and the number of trees optimized.

The number of optimization iterations is counted as the number of times the optimization function is called when solving an instance, i.e. every time perform an optimization (no matter the number of terminals in the topology) we increment the number of optimization iterations with one.

The number of trees optimized is counted as the sum of trees which is optimized at least once. As described earlier we may have some trees which are never optimized at all. We may also have trees which are optimized more than once (this is the most likely, as a single tree normally requires several optimization iterations). Every time we optimize a tree which we have not optimized before (no matter the number of terminals in the topology) we increment the number of optimized trees by one.

Table 6.8 shows the number of trees optimized, as a ratio of the trees optimized by `SmithOld`[5], and Table 6.9 shows the number of iterations as a ratio of the

[5]So if e.g. `SmithOld` has optimized 100 trees and `Simple` has optimized 56 trees, the ratio for `Simple` would be $56/100 = 0.56$.

		Method				
n	d	`Simple`	`SimpleSort`	`SmithNew`	`SmithNewSort`	`SmithOld`
10	2	1.04	0.13	1.42	0.17	1.00
	3	1.21	0.25	1.70	0.37	1.00
	4	1.17	0.60	1.61	0.89	1.00
	5	1.54	0.85	2.25	1.34	1.00
11	2	1.05	0.10	1.62	0.13	1.00
	3	1.23	0.19	1.90	0.29	1.00
	4	1.47	0.36	1.97	0.56	1.00
	5	1.54	0.77	2.57	1.22	1.00
12	2	1.17	0.08	1.90	0.11	1.00
	3	1.42	0.16	2.34	0.25	1.00
	4	1.73	0.26	2.83	0.52	1.00
	5	1.53	0.59	2.24	1.11	1.00
13	2	1.11	0.05	2.22	0.09	1.00
	3	1.43	0.12	2.80	0.20	1.00
	4	0.74	0.25	3.19	0.43	1.00
	5	1.66	0.45		1.14	1.00
14	2	0.97	0.05	2.63	0.08	1.00
	3	1.61	0.10	3.44	0.19	1.00
	4	1.52	0.16		0.39	1.00
	5					
15	2	1.32	0.04	3.12	0.06	1.00
	3					
	4					
	5					

Table 6.8: The table shows the ratio of trees optimized in relation to `OldSmith`. The number of trees is measured such that if a topology vector has been optimized at least once, then the number of optimized trees is optimized by one. Topologies that are pruned before any optimization has taken place is not counted, and any topology can at max be counted once. The data shown are from the Sausage set. An empty field means that either the instance for that method or the instance for `SmithOld` could not be solved within the time limit.

iterations used by `SmithOld`. Both of these tables are based on the data collected for the data set `Sausage`. This set is used as there is one instance for each combination of n and d making it easy to calculate the ratios.

As can be seen by the first table `Simple` and `SmithNew`, optimizes more trees than `SmithOld`. Especially `SmithNew` in some instances optimizes up to three times as many trees. This is of course not very desirable and the exact reason for this is unfortunately unknown.

There seems to be several possibilities for the observed behavior. The first possibility is that the counting of the trees has a bug in either the new or original implementation, or that it does not count them in the same way. After further inspection of the source code, this however does not seem to be the case. The second possibility is that there is a bug somewhere in the new implementation, causing it to not prune as many trees as the original implementation. This seems a more likely reason, but inspection of the source code again failed to reveal such a problem. This could however be tied together with the if-clauses described in Section 6.1 and the variable described in Section 3.4.2. The original implementation uses the variable *SCALE* in the calculation of the initial placement for new Steiner points. If this causes the Steiner points to have better initial placements it could be thought that the if-clauses of the main loop were able to prune earlier.

A third possibility, and probably the must likely, is that the way in which topologies are split in the new implementation is the cause of the extra iterations. As described before, when splitting a topology in the new implementation we re-use the existing coordinates for the Steiner points already in the topology. When we do this a large portion of the tree is at a low error. This in turn can cause Smith's iteration to converge very slowly. This could therefore be the cause for the extra iterations for `SmithNew`, but should not affect `Simple`. For `Simple` a likely explanation is, that the iteration may indeed just require more iterations to converge as is postulated by Smith [13], but because each iteration is much lighter computationally they are done quicker and thus we still see better run-times.

As observed with the speed of the program, the number of trees optimized in general also drops drastically when sorting the terminals (`SimpleSort` and `SmithNewSort`). In all cases the ratio becomes smaller than the unsorted counter-part, and in most cases it also becomes a lot smaller than `SmithOld`.

Looking at Table 6.9 we see more or less the same situation, as in Table 6.8. A column worth noting however, is the one for `Simple`. As can be seen in the first table, the ratio for the number of optimized trees in general is slightly above 1. However the ratio for iterations is in general below 0.5. This means that `Simple` is a lot quicker at reaching a low error and thus stopping to optimize the trees. This could mean that the analytical solution converges a lot faster than Smith's iteration. However again we need to remember the results from Section 6.1,

		Method				
n	d	`Simple`	`SimpleSort`	`SmithNew`	`SmithNewSort`	`SmithOld`
10	2	0.15	0.03	1.10	0.14	1.00
	3	0.23	0.04	1.47	0.28	1.00
	4	0.23	0.08	1.49	0.53	1.00
	5	0.29	0.18	1.37	1.10	1.00
11	2	0.14	0.02	1.13	0.08	1.00
	3	0.20	0.03	1.57	0.18	1.00
	4	0.24	0.05	1.57	0.32	1.00
	5	0.24	0.08	1.56	0.58	1.00
12	2	0.14	0.01	1.18	0.06	1.00
	3	0.19	0.02	1.69	0.16	1.00
	4	0.24	0.03	1.67	0.29	1.00
	5	0.22	0.05	1.81	0.40	1.00
13	2	0.12	0.01	1.23	0.04	1.00
	3	0.17	0.01	1.83	0.12	1.00
	4	0.09	0.03	1.98	0.21	1.00
	5	3.18	0.73		8.76	1.00
14	2	0.09	0.01	1.32	0.03	1.00
	3	0.17	0.01	2.03	0.10	1.00
	4	0.63	0.06		0.71	1.00
	5					
15	2	0.12	0.00	1.42	0.02	1.00
	3					
	4					
	5					

Table 6.9: The table shows the ratio of iterations in relation to `SmithOld` for the Sausage set. The structure is as in Table 6.8. However an iteration means every time we perform an optimization, i.e. a tree can contribute many times to this if we run the iteration for multiple times on the tree (which we most likely do).

which means that we need to be a bit cautious with such a conclusion. But considering that the ratio is consistently lower for the number of iterations[6], and thus is seems unlikely that the result is solely due to sub-optimality, as described earlier.

A thing worth noticing is that the higher run-times for `SmithNew` (and partly `SmithNewSort`) observed in the previous section make a lot more sense when observing the ratios for trees and iterations. Thus it is very possible, that if one could bring down these numbers the new implementation would have faster run-times than `SmithOld` all around.

[6]Except for a single outlier

7 Discussion

This chapter discusses some of the possible extensions and directions one could do with the work of the thesis. Apart from the more obvious possible improvements, such as applying the lower bounds from Fampa and Anstreicher [5] or prune using the geometric properties Van Laarhoven and Anstreicher [20], this chapter lists some possible improvements and extensions for the new implementation.

7.1 Concurrency

Both the original and new implementation of the algorithm is implemented in a sequential fashion. There also seems to be no literature concerned with making the algorithm by Smith concurrent.

Looking at Figure 7.1 however may give an incitement to further investigate the possibilities of making the algorithm concurrent. The figure shows a CPU-time profile of the execution of the new implementation on a single random instance (using Smith's iteration for optimization). As can be seen, more than 50% of the elapsed time is spent in the function with the iteration (this is the time spent in the actual function, not waiting on other function calls). Above that we see that if we count waiting for function call that more than 80% of the elapsed time was spent in the optimize function. Here waiting on other function calls count towards the 80%. Making especially the main loop concurrent and then running it concurrently on e.g. 4–8 cores (which is a very common amount of cores in e.g. modern CPUs in of any home desktop computer) could provide a very significant speedup.

We would then have to decide on some way of splitting up the problem instances. One possible way of doing it would be to start one process for each half of the children for the current topology vector. So say we currently have the topology vector $[1, 2]$, we would then split this into two new processes, one which will look at $[1, 2, 1]$ to $[1, 2, 4]$ and one which looks at $[1, 2, 5]$ to $[1, 2, 7]$. As long as we split in this fashion (no matter how many we decide to split into) these can all be executed independently of each other, but would of course benefit from synchronizing upper bounds. Thus we would make the upper

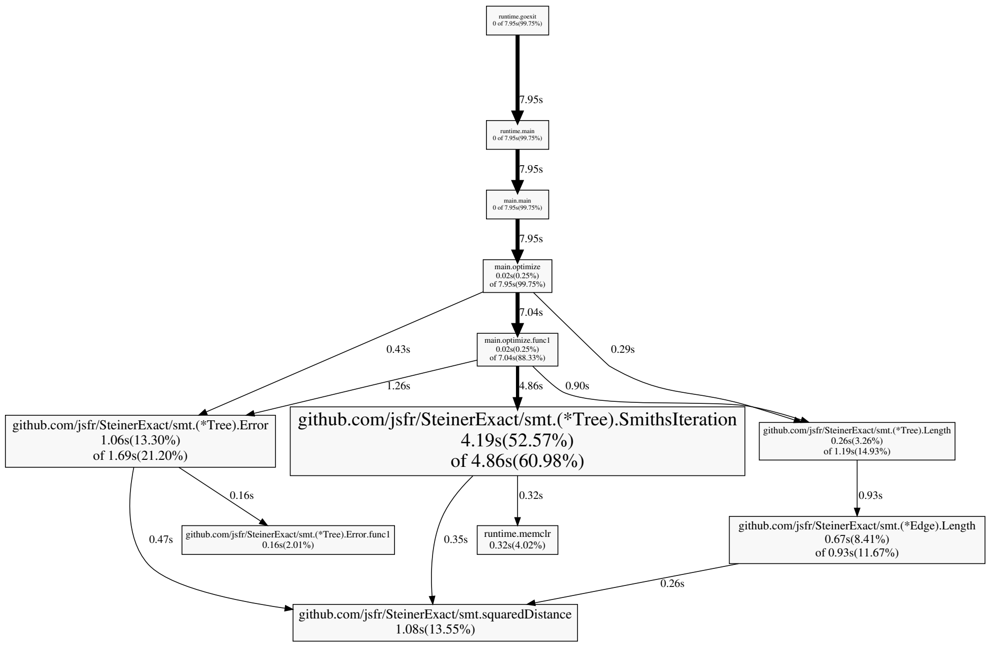


Figure 7.1: CPU-time profile of the execution of the new implementation on a single random instance. The figure does not show the complete profile, as some of the smaller insignificant contributions have been remove for clarity. The initial entry-point of the program is the top box, showing the first function that is called in the program. Arrows indicate that the function in a box have called the function which the arrow points to. The times in the box shows the amount of seconds actually spent in the function (not waiting for other functions called within the function) of the amount of seconds spent in the function waiting for function calls as well.

bound with a read-write lock which as mutual read (i.e. many can read the upper bound at the same time) and exclusive write (i.e. only one can write a new upper bound at a time). It is clear that the max number of splits we can make, is the same number of edges we can split on every time we extend the topology. Thus the max number the for the first split is 3, foreach of these 3 we could split into 5, and then for each of these into 7, and so forth.

Implementing a scheme as the one described above would allow us to both generate and optimize multiple topology vectors in parallel which could provide a speedup dependent on the number of cores available. Also, this approach would also allow us to find upper bounds in parallel, which could mean that topologies which we before were not able to prune, will now be pruned. Thus the shared upper bound between processes could result in a further speedup.

The new implementation is as mentioned written in Go, which is a language that makes implementing concurrency very straightforward. Thus extending the new implementation to use concurrency could be done by refactoring a slightly changed version of the main loop to a new function, and then spawn new goroutines. These should then e.g. have a topology vector and some some boundaries for which part of the descendants it should generate. Due to Go's way of making arrays, we would also need to expand the `Tree` struct with a new function to clone the tree. This would result in a slight slowdown, but more importantly it would require the more memory equivalent again approximately to the number of running processes.

7.2 Propagating Changes in the Simple Iteration

Currently the implementation of the simple iteration (using the analytical solution) iterates through the Steiner points in a naive way when optimizing them. It simply iterates through all Steiner points in the order they have been added, and optimize them.

However consider adding the next terminal and a new Steiner point to a topology. When adding the new Steiner point, we replace one edge with two, and add one new edge. Thus all of the tree except for that one replaced edge, looks exactly the same.

The order we could now choose to optimize the Steiner points in would be to optimize the new Steiner point and let the change propagate from there. We therefore next optimize any of the new Steiner points neighbors that are Steiner points. If a Steiner point moves when it is optimized, we optimize any of its neighbors that are Steiner points (and which has not yet been optimized). In this way we propagate the optimization out through the tree from the new Steiner point.

The reason this approach might be interesting is that we potentially avoid optimizing whole sub-trees of the FST, as if a Steiner point does not move, nothing connected to the other side of it will move (assuming that part of the tree was optimized during the previous topology).

Furthermore we also have that any Steiner point overlapping a terminal splits the tree into two smaller sub-FSTs and no change in one of the trees will propagate to the other as long as the overlapping Steiner point does not move away from the terminal[1].

A further extension might be to not optimize out from the new Steiner point, but to instead optimize out from the Steiner point which currently has the largest error, under the assumption that the Steiner point with the largest error will move the most and therefore cause its neighbors to move the most, thus propagating the biggest change and leading to faster convergence.

[1]This property on sub-FSTs also holds true for Smith's iteration, and could therefore also be investigated with this.

7.3 Initial Placement of Steiner Points

Currently both the original and new implementation place the Steiner points at the perturbed centroid of their neighbors. However calculating the centroid is about as computationally heavy as it is to calculate the Fermat-Torricelli point using the analytical solution. Thus the implementation would probably benefit from exchanging the perturbed centroid with the perturbed Fermat-Torricelli point as a initial placement. The reasoning behind this is that all Steiner points will be closer to their final optimal coordinates and thus we would need fewer iterations in total.

	Centroid		*Fermat-Torricelli point*		*Ratio*	
n	Iterations	Trees	Iterations	Trees	Iterations	Trees
10	73132.7	4231.94	58443.5	1645.08	0.80	0.39
11	307128	18429	238776	6944.14	0.78	0.38
12	1492050	87869.2	1122520	32192.1	0.75	0.37

Table 7.1: The table shows the average number of trees optimized and number of optimization iterations when placing the Steiner points at the perturbed centroid or the perturbed Fermat-Torricelli point initially. The data is for the cube instances used in Section 6.1. Smith's iteration without terminal sorting was used. The last two columns show the ratio between the numbers for easier comparison. There are 50 *instances of for each n.*

To test the potential impact of this change a quick test was performed were the use of the perturbed centroid was exchanged with the Fermat-Torricelli point, perturbed exactly the same amount as the centroid was. The cube instances from Section 6.1 were then run before and after the change, and the average for the number of trees optimized and iterations for $n = 10, 11, 12$ were calculated. The results can be found in Table 7.1. Only Smith's iteration (without terminal sorting) is used on the data to be sure that the data is not affected by the sub-optimality seen in Section 6.1 for the simple iteration[2]. Furthermore were the correctness test also performed after the change to ensure that no new sub-optimal result occurred. As can be seen, at least for this data set, the program ran $\approx 20\%$ fewer iterations, and optimized $\approx 60\%$ fewer trees. It therefore seems worthwhile to make this very simple change, but more extensive performance tests would probably have to be made.

[2]Furthermore the correctness test were performed again after the change just for good measure. This showed no new sub-optimal results, apart from those in the simple iteration already known.

8 Conclusion

The thesis has presented the ESTP and preliminaries for understanding it. The thesis has also presented methods for finding SMTs in two dimensions and have described why these do not generalize to higher dimensions. Known methods for finding SMTs in higher dimensions have been presented and apart from the branch algorithm by Fonseca et al. [8] all of these have been seen to be modifications or extension to the method proposed by Smith [13].

The thesis has presented the algorithm for finding SMTs in euclidean d-space proposed by Smith [13]. Apart from presenting it, the thesis has also identified and discussed elements of the article and implementation given in [13] which were questionable. The most distinctive of which were the choice of error function for which no real argument has been given and the if-clauses when pruning which were seen to possibly cause sub-optimality in the new implementation.

The thesis has presented improvements to the Smith's algorithm and implementation. The first were to use a new method and data structure for building topologies to avoid rebuilding the tree every time the topology is extended. The second were a new method for ordering the terminals before before building the topologies. This method greedily selected the terminals furthest apart. Finally the thesis have presented a new simple iteration as an alternative to the iteration given by Smith [13] for optimizing the tree of a full Steiner topology. The presentation of this simple method included the presentation and generalization to higher dimensions of the analytical solution to the Fermat-Torricelli problem for 3 points presented in 2D by Uteshev [19]. This iteration was proposed as Smith [13] had noted one such method would converge slowly but gave no data or proof to back up the claim. As the method was computationally very light it seemed likely that it would perform similarly or better than the iteration proposed by Smith [13].

Furthermore the thesis has accounted for the structure of the new implementation which has been structured such that most of the work lies in a self-contained library for others to continue to work on or with.

The thesis has presented the experimental work performed to test the efficiency and correctness of the new implementation (i.e. both of the iterations, with and without sorting of terminals). The correctness tests revealed that sub-optimality would occur in some instances when using the new simple iteration. This sub-optimality was discussed and further investigated. Eventually the sub-optimality was ascribed mostly to the use of current error-function on the entire tree, and the if-clauses used when optimizing and pruning topologies. The thesis afterwards presented the tests done to test the efficiency of the new implementation when compared with the original implementation. The tests in general showed promising results for the new simple iteration which on most occasions was much faster than the original implementation. The new implementation of Smith's iteration in general performed slightly worse than the original. The cause for this is still unknown, but several qualified suggestions were made. These were the choice of programming language and more importantly a bug/change which causes the new implementation to optimize on many more trees than the original implementation. This higher number of optimizations was ascribed to the fact that the new implementation reuses the coordinates from the previous optimized tree, causing it to move very slowly. The new method for ordering terminals also showed good performance. It brought the speed of the simple iteration even further below the original implementation and the speed of the new implementation of Smith's iteration at the same level/slightly below the original implementation.

Furthermore the performance tests seemed to debunk the claim by Smith [13] that a simple iteration would converge much slower than the one he proposes.

Finally the thesis has a discussion of possible extensions and improvements for the new implementation. The first was the possibility of making the implementation concurrent, for which a possible implementation scheme was outlined. The second was more efficient way of propagating changes out through the tree when optimizing with the simple iteration. It was described how this could speed up convergence by allowing us to only optimize on the part of the tree which is changes. The third and final improvement was to exchange the initial placement of Steiner points in the implementation with the perturbed Fermat-Torricelli point, instead of the perturbed centroid. A test which showed much improved numbers for iterations and optimized trees was performed to further support the argument of making this change.

A Proof of Convergence

Before discussing the convergence we define the length L as is normally the case, calculated as the sum of all edge lengths. In this case we use the regular Euclidean ($\mathcal{L}_2$) norm. One could however think to possibly use other norms, e.g. the rectilinear ($\mathcal{L}_1$) norm. Thus

$$L(S) = \sum_{j,l:(j,l)\in\mathcal{T}} |p_l - p_j| \tag{A.1}$$

Note that Smith [13] defines L as a function of the set of Steiner points S instead of the tree T. Here we follow this convention, as we need the function when we argue about the iteration, in which only the Steiner points move and the terminals remain fixed. Also remember that $\mathcal{T} \equiv E(T)$.

Proof. The proof of convergence is as follows: After the first iteration all Steiner points lie within the convex hull of the terminals. By the Bolzano-Weierstrass theorem any infinite sequence of points inside a compact region (such as a convex hull) has some infinite subsequence approaching a limit point. We therefore wish to show that the only limit point which can exist, represents the RMT for the underlying topology.

Firstly after each (ith) iteration, according to Smith [13], $S^{(i+1)}$ exactly minimizes the following quadratic form $Q^{(i)}(S)$

$$Q^{(i)}(S) = \sum_{j,l:(j,l)\in\mathcal{T},l\in S,j<l} \frac{|p_l - p_j|^2}{|p_l^{(i)} - p_j^{(i)}|}$$

Note that the last two conditions on the sum are redundant and—as far as I can tell—are only given by Smith to emphasize the way in which trees are split and edges are numbered. It is easily seen that $Q(S)$ is related to $L(S)$ in the following way

$$
\begin{aligned}
Q^{(i)}(S^{(i)}) &= \sum_{(j,l)\in\mathcal{T}} \frac{|p_l^{(i)} - p_j^{(i)}|^2}{|p_l^{(i)} - p_j^{(i)}|} \\
&= \sum_{(j,l)\in\mathcal{T}} |p_l^{(i)} - p_j^{(i)}| \\
&= L(S^{(i)})
\end{aligned} \tag{A.2}
$$

Smith describes that, using Gilbert and Pollak [9]'s mechanical model one can think of Q as the potential energy of a system of ideal springs on the tree edges, where the force constant of each spring is proportional to the reciprocal of its original length before each iteration. The ith iteration causes all springs to relax, minimizing $Q^{(i)}$. After several talks with my supervisor, it is however still a unclear to me that $S^{(i+1)}$ really minimize $Q^{(i)}$.

However as the proof of convergence hinges on this fact[1] we can only proceed if this is the case. Thus I here assume that this is correct. We then do as follows

$$
\begin{aligned}
L(S^{(i)}) &\overset{(A.2)}{=} Q^{(i)}(S^{(i)}) \\
&\geqslant Q^{(i)}(S^{(i+1)}) \\
&= \sum_{j,l:(j,l)\in\mathcal{T}} \frac{|p_l^{(i+1)} - p_j^{(i+1)}|^2}{|p_l^{(i)} - p_j^{(i)}|}
\end{aligned}
$$

using that we can add and subtract the same thing without changing the equation, we do

$$
= \sum_{j,l:(j,l)\in\mathcal{T}} \frac{\left(|p_l^{(i+1)} - p_j^{(i+1)}| + |p_l^{(i)} - p_j^{(i)}| - |p_l^{(i)} - p_j^{(i)}|\right)^2}{|p_l^{(i)} - p_j^{(i)}|} \tag{A.3}
$$

We then write out and reorder the numerator of the fraction in Equation (A.3), here name *num*. To further simplify the equation, as it does otherwise get quite hairy, we define $a = |p_l^{(i+1)} - p_j^{(i+1)}|$ and $b = |p_l^{(i)} - p_j^{(i)}|$. Thus

$$
\begin{aligned}
\textit{num} &= \left(|p_l^{(i+1)} - p_j^{(i+1)}| + |p_l^{(i)} - p_j^{(i)}| - |p_l^{(i)} - p_j^{(i)}|\right)^2 \\
&= (a + b - b)^2 \\
&= (a^2 + ab - ab) + (ba + b^2 - b^2) + (-ba - b^2 + b^2) \\
&= (a^2 + b^2 - ab - ab) + (b^2 - b^2 - b^2) + (ab + ba) \\
&= 2ab - b^2 + (a - b)^2
\end{aligned}
$$

[1]As we need to utilize that $Q^{(i)}(S^{(i)}) \geqslant Q^{(i)}(S^{(i+1)})$

then dividing the numerator again with the denominator from Equation (A.3) *denom* $= |p_l^{(i)} - p_j^{(i)}| = b$ we get

$$\frac{num}{denom} = \frac{2ab - b^2 + (a-b)^2}{b} = 2a - b + \frac{(a-b)^2}{b} \tag{A.4}$$

Finally we can go back to Equation (A.3) and insert the fraction we have in Equation (A.4). Thus

$$\begin{aligned}
(\text{A.3}) = & \sum_{j,l:(j,l)\in\mathcal{T}} \left[2a - b + \frac{(a-b)^2}{b} \right] \\
= & \, 2 \sum_{j,l:(j,l)\in\mathcal{T}} |p_l^{(i+1)} - p_j^{(i+1)}| - \sum_{j,l:(j,l)\in\mathcal{T}} |p_l^{(i)} - p_j^{(i)}| \\
& + \sum_{j,l:(j,l)\in\mathcal{T}} \frac{(|p_l^{(i+1)} - p_j^{(i+1)}| - |p_l^{(i)} - p_j^{(i)}|)^2}{|p_l^{(i)} - p_j^{(i)}|} \\
\overset{(\text{A.1})}{=} & \, 2L(S^{(i+1)}) - L(S^{(i)}) \\
& + \sum_{j,l:(j,l)\in\mathcal{T}} \frac{(|p_l^{(i+1)} - p_j^{(i+1)}| - |p_l^{(i)} - p_j^{(i)}|)^2}{|p_l^{(i)} - p_j^{(i)}|} \Leftrightarrow \\
2L(S^{(i)}) \geqslant & \, 2L(S^{(i+1)}) \\
& + \sum_{j,l:(j,l)\in\mathcal{T}} \frac{(|p_l^{(i+1)} - p_j^{(i+1)}| - |p_l^{(i)} - p_j^{(i)}|)^2}{|p_l^{(i)} - p_j^{(i)}|}
\end{aligned}$$

As the last sum is obviously non-negative[2], we can remove it without the validity of the equation changing, and thus

$$L(S^{(i)}) \geqslant L(S^{(i+1)})$$

We now know that performing an iteration, will always either decrease the length of the tree, or it will remain the same. The only way that it can remain the same, i.e. that we can have equality $L(S^{(i)}) = L(S^{(i+1)})$ is if we are at a fixed point of the iteration. There are only two types of fixed points—the optimum[3] and certain places where one or more edges have length zero. As earlier described the iteration as we have written it is not strictly defined, due to a division with zero. However according to Smith the singularity is removable[4] [22] and thus we remove it.

[2]The numerator is squared and thus non-negative, and the denominator is a length and thus non-negative.

[3]Which is the RMT for this topology.

[4]How this is the case, is unclear and unexplained by Smith. But as I understand it, the singularities are removable by his perturbation.

The next part of the proof again hinges on a postulate by Smith which is not clearly true to me—that the non-optimum fixed points are unstable in any direction.

Firstly Smith claims that the non-optimum fixed points are unstable in any L-decreasing direction, and thus a small perturbation of the Steiner points S in any such direction will cause the iteration to continue. This makes sense as we know that L can never *increase* when we run the iteration. Thus after decreasing L by a small perturbation, either we will be past the fixed point and the iteration will continue to decrease, or we hit another fixed point[5]. Smith then refers to Gilbert and Pollak [9] who have pointed out that optimizing a pre-specified Steiner topology is a "strictly convex" optimization problem, i.e. the only local optimum is global. This means that there will always be a L-decreasing direction if we are not yet at the optimum.

This currently means we need to find a L-decreasing perturbation for the iteration to continue its convergence. Smith however further argues that any small step in the opposite direction of a L-decreasing one would also cause the iteration to resume decreasing L. The argument for this is based on the physical spring model of Steiner trees, introduced by Gilbert and Pollak [9]. I have unfortunately not been able to follow the argument Smith [13] gives, and after much discussion with my supervisor Pawel Winter it is still unclear to both of us whether this argument actually holds.

If we assume that the argument holds, then the iteration, when it is at a non-optimum fixed point, is unstable in every direction. Thus we have a 0-measure[6] set of initial iterates which end up at the non-optimal fixed points. Finally we can the conclude that any Bolzano-Weierstrass subsequence limit tree must be a fixed point of the iteration. This is done by continuity of L, the iteration function and since the only points where the iteration does not decrease L are fixed points. However as all non-optimum fixed points are in the 0-measure set, we have ruled those out and thus the only fixed point is the optimum. □

[5]at which point we then again will have to use a small perturbation.

[6]as we can disregard all these points because of their instability.

B Proof of the Analytical Method for the Fermat-Torricelli Problem

B.1 2D

Proof. To prove Equation (4.3) the first step is to prove the two following equations

$$K_1K_2 + K_1K_3 + K_2K_3 = 2\sqrt{3}Sd \tag{B.1}$$
$$r_{23}^2K_1 + r_{13}^2K_2 + r_{12}^2K_3 = 2Sd \tag{B.2}$$

Both can proven by simply substituting the right hand sides with the definition of each symbol, i.e. through direct computation. Using Equation (B.1) we can rewrite Equation (4.3) as

$$\begin{cases} x_* = \frac{1}{\frac{1}{K_1}+\frac{1}{K_2}+\frac{1}{K_3}}\left(\frac{x_1}{K_1}+\frac{x_2}{K_2}+\frac{x_3}{K_3}\right) \\ y_* = \frac{1}{\frac{1}{K_1}+\frac{1}{K_2}+\frac{1}{K_3}}\left(\frac{y_1}{K_1}+\frac{y_2}{K_2}+\frac{y_3}{K_3}\right) \end{cases} \tag{B.3}$$

Which will be useful in later calculations. The second part is to prove that

$$\sqrt{(x_*-x_j)^2+(y_*-y_j)^2} = \frac{K_j}{\sqrt{3d}}, \quad j \in \{1,2,3\} \tag{B.4}$$

As the calculations are similar for all j, only the one for $j = 1$ is shown here. Thus

$$\begin{aligned}
&(x_*-x_1)^2+(y_*-y_1)^2 \\
&\overset{(4.3)}{=}\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\left(\frac{x_1}{K_1}+\frac{x_2}{K_2}+\frac{x_3}{K_3}\right)-x_1\right)^2+\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\left(\frac{y_1}{K_1}+\frac{y_2}{K_2}+\frac{y_3}{K_3}\right)-y_1\right)^2 \\
&=\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\left(\frac{x_1}{K_1}+\frac{x_2}{K_2}+\frac{x_3}{K_3}-\frac{x_12\sqrt{3}Sd}{K_1K_2K_3}\right)^2+\left(\frac{y_1}{K_1}+\frac{y_2}{K_2}+\frac{y_3}{K_3}-\frac{y_12\sqrt{3}Sd}{K_1K_2K_3}\right)^2\right]
\end{aligned}$$

$$
\begin{aligned}
&\overset{(B.1)}{=}\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\left(\frac{x_2}{K_2}+\frac{x_3}{K_3}-\frac{x_1}{K_2}-\frac{x_1}{K_3}\right)^2+\left(\frac{y_2}{K_2}+\frac{y_3}{K_3}-\frac{y_1}{K_2}-\frac{y_1}{K_3}\right)^2\right]\\
&=\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\frac{(x_2-x_1)^2+(y_2-y_1)^2}{K_2^2}+\frac{(x_3-x_1)^2+(y_3-y_1)^2}{K_3^2}\right.\\
&\qquad\left.+2\frac{(x_2-x_1)(x_3-x_1)+(y_2-y_1)(y_3-y_1)}{K_2K_3}\right]\\
&\overset{(4.5)}{=}\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\frac{r_{12}^2}{K_2^2}+\frac{r_{13}^2}{K_3^2}+\frac{r_{12}^2+r_{13}^2-r_{23}^2}{K_2K_3}\right] \qquad (B.5)\\
&=\frac{K_1^2}{(2\sqrt{3}Sd)^2}\left[r_{12}^2K_3^2+r_{13}^2K_2^2+(r_{12}^2+r_{13}^2-r_{23}^2)K_2K_3\right]\\
&=\frac{K_1^2}{(2\sqrt{3}Sd)^2}\left[(r_{12}^2K_3+r_{13}^2K_2)(K_2+K_3)-r_{23}^2K_2K_3\right]\\
&\overset{(B.2)}{=}\frac{K_1^2}{(2\sqrt{3}Sd)^2}\left[(2Sd-r_{23}^2K_1)(K_2+K_3)-r_{23}^2K_2K_3\right]\\
&=\frac{K_1^2}{(2\sqrt{3}Sd)^2}\left[2Sd(K_2+K_3)-r_{23}^2(K_1K_2+K_1K_3+K_2K_3)\right]\\
&\overset{(B.1)}{=}\frac{K_1^2}{(2\sqrt{3}Sd)^2}\left[2Sd(K_2+K_3)-2\sqrt{3}r_{23}^2Sd\right]\\
&=\frac{2SdK_1^2}{(2\sqrt{3}Sd)^2}\left[K_2+K_3-\sqrt{3}r_{23}^2\right]\\
&\overset{(4.6)}{=}\frac{K_1^2}{6Sd}[2S]\\
&=\frac{K_1^2}{3d}
\end{aligned}
$$

Thus

$$\sqrt{(x_*-x_1)^2+(y_*-y_1)^2}=\sqrt{\frac{K_1^2}{3d}}=\frac{K_1}{\sqrt{3d}}$$

To finish the proof one would also have to show that K_1, K_2, K_3 are nonnegative. The proof of this can be found in [19, p. 5-6].

We can now prove Equation (4.3), by substituting it into Equation (4.2). As the equations are similar for x and y, only x is shown here

$$
\begin{aligned}
0&=\frac{(x_*-x_1)}{\sqrt{(x_*-x_1)^2+(y_*-y_1)^2}}+\frac{(x_*-x_2)}{\sqrt{(x_*-x_2)^2+(y_*-y_2)^2}}+\frac{(x_*-x_3)}{\sqrt{(x_*-x_3)^2+(y_*-y_3)^2}}\\
&\overset{(B.4)}{=}\frac{(x_*-x_1)}{\frac{K_1}{\sqrt{3d}}}+\frac{(x_*-x_2)}{\frac{K_2}{\sqrt{3d}}}+\frac{(x_*-x_3)}{\frac{K_3}{\sqrt{3d}}}
\end{aligned}
$$

$$
\begin{aligned}
&= \sqrt{3d}\left[\frac{x_* - x_1}{K_1} + \frac{x_* - x_2}{K_2} + \frac{x_* - x_3}{K_3}\right] \\
&= \sqrt{3d}\left[x_*\left(\frac{1}{K_1} + \frac{1}{K_2} + \frac{1}{K_3}\right) - \left(\frac{x_1}{K_1} + \frac{x_2}{K_2} + \frac{x_3}{K_3}\right)\right] \\
&\overset{(B.3)}{=} \sqrt{3d}\left[\frac{1}{\frac{1}{K_1} + \frac{1}{K_2} + \frac{1}{K_3}}\left(\frac{x_1}{K_1} + \frac{x_2}{K_2} + \frac{x_3}{K_3}\right)\left(\frac{1}{K_1} + \frac{1}{K_2} + \frac{1}{K_3}\right) - \left(\frac{x_1}{K_1} + \frac{x_2}{K_2} + \frac{x_3}{K_3}\right)\right] \\
&= \sqrt{3d}\left[\frac{1}{\frac{1}{K_1} + \frac{1}{K_2} + \frac{1}{K_3}}\left(\frac{1}{K_1} + \frac{1}{K_2} + \frac{1}{K_3}\right) - 1\right] \\
&= \sqrt{3d}\,[1 - 1] = 0
\end{aligned}
$$

As can be seen Equation (4.3) is a solution to Equation (4.2), and thus it minimizes $F(x, y)$. □

B.2 d-Space

Proof. As before to prove Equation (4.9) (and thus Equation (4.8)) the first step once again is to prove Equation (B.1) and Equation (B.2). These can as before be established through direct computation, i.e. by simply substituting the expressions for their definitions, until both sides only contain distances, after which the two sides can be seen to be equal. Using Equation (B.1) we can rewrite Equation (4.9) as

$$
x_{(*,j)} = \frac{1}{\frac{1}{K_1} + \frac{1}{K_2} + \frac{1}{K_3}}\left(\frac{x_{(1,j)}}{K_1} + \frac{x_{(2,j)}}{K_2} + \frac{x_{(3,j)}}{K_3}\right), \quad 0 < j \leqslant d \tag{B.6}
$$

similarly to Equation (B.3) in the 2D version. Secondly we wish to prove that

$$
\sqrt{\sum_{i=1}^{d} (x_{(*,i)} - x_{j,i})^2} = \frac{K_j}{\sqrt{3d}}, \quad j \in \{1, 2, 3\} \tag{B.7}
$$

As the calculations are similar for all j, only the one for $j = 1$ is shown here. Thus

$$
\begin{aligned}
&\sum_{i=1}^{d} (x_{(*,i)} - x_{(1,i)})^2 \\
&\qquad = \sum_{i=1}^{d}\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\left(\frac{x_{(1,i)}}{K_1} + \frac{x_{(2,i)}}{K_2} + \frac{x_{(3,i)}}{K_3}\right) - x_{(1,i)}\right)^2 \\
&\qquad = \left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\sum_{i=1}^{d}\left(\frac{x_{(1,i)}}{K_1} + \frac{x_{(2,i)}}{K_2} + \frac{x_{(3,i)}}{K_3} - \frac{x_{(1,i)}2\sqrt{3}Sd}{K_1K_2K_3}\right)^2\right]
\end{aligned}
$$

$$
\begin{aligned}
&\overset{(B.1)}{=}\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\sum_{i=1}^{d}\left(\frac{x_{(2,i)}}{K_2}+\frac{x_{(3,i)}}{K_3}-\frac{x_{(1,i)}}{K_2}-\frac{x_{(1,i)}}{K_3}\right)^2\right]\\
&=\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\sum_{i=1}^{d}\left(\frac{(x_{(2,i)}-x_{(1,i)})^2}{K_2^2}+\frac{(x_{(3,i)}-x_{(1,i)})^2}{K_3^2}\right.\right.\\
&\qquad\left.\left.+2\frac{(x_{(2,i)}-x_{(1,i)})(x_{(3,i)}-x_{(1,i)})}{K_2K_3}\right)\right]\\
&=\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\frac{\sum_{i=1}^{d}(x_{(2,i)}-x_{(1,i)})^2}{K_2^2}+\frac{\sum_{i=1}^{d}(x_{(3,i)}-x_{(1,i)})^2}{K_3^2}\right.\\
&\left.+\frac{\sum_{i=1}^{d}(x_{(2,i)}-x_{(1,i)})^2+\sum_{i=1}^{d}(x_{(3,i)}-x_{(1,i)})^2-\sum_{i=0}^{d}(x_{(2,i)}-x_{(3,i)})^2}{K_2K_3}\right]\\
&\overset{(4.10)}{=}\left(\frac{K_1K_2K_3}{2\sqrt{3}Sd}\right)^2\left[\frac{r_{12}^2}{K_2^2}+\frac{r_{13}^2}{K_3^2}+\frac{r_{12}^2+r_{13}^2-r_{23}^2}{K_2K_3}\right]
\end{aligned}
$$

As this point we have exactly the same as in Equation (B.5) and thus we can conclude

$$=\frac{K_1^2}{3d}$$

From here we can show that Equation (4.9) is a solution to Equation (4.7) in the exactly the same way as we proved that Equation (4.3) was a solution to Equation (4.2), by substituting Equation (4.9) into Equation (4.7) and using Equation (B.6) and Equation (B.7). □

Bibliography

[1] Marcus Brazil and Martin Zachariasen. *Optimal Interconnection Trees in the Plane*. Springer, 2015.

[2] Marcus Brazil et al. “A novel approach to phylogenetic trees: d-Dimensional geometric Steiner trees”. In: *Networks* 53.2 (2009), pp. 104–111.

[3] Marcus Brazil et al. “On the history of the Euclidean Steiner tree problem”. In: *Archive for history of exact sciences* 68.3 (2014), pp. 327–354.

[4] *Effective Go*. URL: http://golang.org/doc/effective_go.html (visited on 07/24/2015).

[5] Marcia Fampa and Kurt M. Anstreicher. “An improved algorithm for computing Steiner minimal trees in Euclidean d-space”. In: *Discrete Optimization* 5.2 (2008), pp. 530–540.

[6] Pierre Fermat. “Oeuvres, vol. 1”. In: *Gauthier-Villars, Paris* 1896 (1891).

[7] *Fermat-Torricelli problem*. URL: http://www.encyclopediaofmath.org/index.php/Fermat-Torricelli_problem (visited on 07/24/2015).

[8] Rasmus Fonseca et al. “Faster Exact Algorithms for Computing Steiner Trees in Higher Dimensional Euclidean Spaces”. In: *11th DIMACS Implementation Challenge on Steiner Tree Problems* (2014).

[9] E. N. Gilbert and H. O. Pollak. “Steiner minimal trees”. In: *SIAM Journal on Applied Mathematics* 16.1 (1968), pp. 1–29.

[10] F. K. Hwang. “A linear time algorithm for full Steiner trees”. In: *Operations Research Letters* 4.5 (1986), pp. 235–237.

[11] F. K. Hwang and J. F. Weng. “Hexagonal coordinate systems and Steiner minimal trees”. In: *Discrete mathematics* 62.1 (1986), pp. 49–57.

[12] Z. A. Melzak. “On the problem of Steiner”. In: *Canad. Math. Bull* 4.2 (1961), pp. 143–148.

[13] Warren D. Smith. “How to find Steiner minimal trees in euclidean d-space”. In: *Algorithmica* 7.1-6 (1992), pp. 137–177.

[14] Warren D Smith and J MacGregor Smith. “On the Steiner ratio in 3-space”. In: *Journal of Combinatorial Theory, Series A* 69.2 (1995), pp. 301–332.

[15] *The Go Programming Langugage*. URL: http://golang.org (visited on 07/24/2015).

[16] *The JSON Data Interchange Format*. Tech. rep. Standard ECMA-404 1st Edition / October 2013. ECMA, Oct. 2013. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[17] Evangelista Torricelli. “De maximis et minimis”. In: *Opere di Evangelista Torricelli* (1919).

[18] *Triangle Inequality*. URL: https://en.wikipedia.org/wiki/Triangle_inequality (visited on 07/15/2015).

[19] Alexei Y. Uteshev. “Analytical solution for the generalized Fermat–Torricelli problem”. In: *The American Mathematical Monthly* 121.4 (2014), pp. 318–331.

[20] J. W. Van Laarhoven and K. M. Anstreicher. “Geometric conditions for Euclidean Steiner trees in $\mathfrak{R}^d$”. In: *Computational Geometry* 46.5 (2013), pp. 520–531.

[21] Eric W. Weisstein. *Longest Path Problem*. URL: http://mathworld.wolfram.com/LongestPathProblem.html (visited on 08/14/2015).

[22] Eric W. Weisstein. *Removable Singularity*. URL: http://mathworld.wolfram.com/RemovableSingularity.html (visited on 08/06/2015).

[23] Pawel Winter. “An algorithm for the Steiner problem in the Euclidean plane”. In: *Networks* 15.3 (1985), pp. 323–345.