Radboud University Nijmegen

Radboud University Nijmegen

Towards faster internet connections

Connecting cabinets to the fiber glass network

Author:
Niels Neumann

Coordinator:
Dr. F. Phillipson (TNO)
Dr. P. Hochs (RU)
Prof. dr. E. Koelink (RU)

May 25, 2016

Abstract

In this thesis we will consider a problem raised by a telecom company regarding the implementation of the fiber glass network. We have considered the problem of connecting cable dividers, also referred to as cabinets or active points, to the fiber glass network. This problem was split in three separate problems and for each an algorithm has been constructed. Instead of connecting directly to the fiber glass network, we use an extra network to minimize the costs. First the shortest connection between a cabinet and a network is determined. Now multiple networks can be used to route to the fiber glass network. In order to use the multiple networks, we determine where the two networks intersect, as only at those points a transition from one network to the other is possible. The last algorithm determines the least cost-path between the cabinet and the fiber glass network.

We expect the reader to be familiar with basic probability theory. Basic graph theory and complexity theory will be treated briefly in the theoretical background of this thesis.

Contents

Chapter 1

Introduction

As internet is taking a more prominent role in our lives, we want it to meet our ever growing demands, including the desire for fast connections. In the past ADSL (Asymmetric Digital Subscriber Line) had been used in most cases. Using ADSL data could be send via copper cables to the end users. Starting from a *central office* thick copper wire bundles started and kept branching until the end user was reached. In most cases there was a cabinet located between the central office and the end user serving as a cable divider. As of now, there are new types and generations of the ADSL technology, where VDSL (Very-high-bit-rate Digital Subscriber Line) is the technique telecom providers are most interested in. VDSL is an improved version of ADSL which is used for the implementation from cabinets onwards. The efficiency of the technique used mainly depends on the length of the copper cables between the cabinets and the end users. At one kilometer of copper cable, the speed of ADSL2+ (an improved version of ADSL) and VDSL will be equal [1]. For larger distances the speeds will be almost equal, see also Figure 1.2.

As most networks still consisted of copper cables, companies had to make a decision on the next step. This step could either be a full transition to a fiber glass network or an intermediate solution. Doing a full transition to fiber glass however, would be too expensive, both in actual cost and in the man hours it would take to upgrade the complete network. Therefore the choice for an intermediate solution was made. Instead of upgrading the complete network all at once, the network is upgraded gradually from a full copper network to (possibly) a full fiber glass network (also called Full Fibre to the Home or Full FttH). This way the problems arising with the Full FttH are resolved. The two intermediate steps most used are *Fibre to the Cabinet* (FttCab) and *Hybrid Fibre to the Home* (Hybrid FttH). With FttCab, the connection from the central office to the cabinets (cable dividers) is replaced by fiber, from where on the copper cables are used. In Hybrid FttH the fiber glass network will be further extended 'to the curb', meaning that the network consists mainly of fiber glass cables, only the last part (from the street on) will be copper. Therefore, Hybrid FttH is also referred to as *Fibre to the Curb*. The four possible situations we have are shown in Figure 1.1.

The use of FttCab and Hybrid FttH also gives rise to new techniques such as *G.Fast* and VDSL2. These techniques are ways to send the data over the cables. G.Fast was established in December 2014 and has the claimed potential to reach 1 Gb/s with a copper distance of at most 200 meter [2]. VDSL2 on the other hand can have a larger

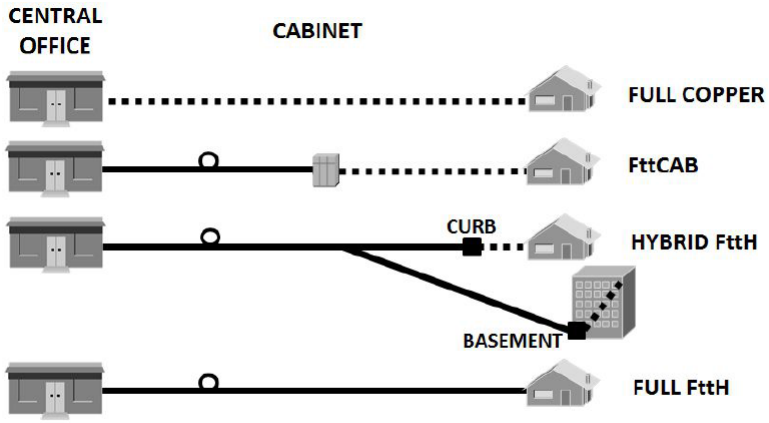

Figure 1.1: The four possible situations [1].

distance to the cabinets, while still giving acceptable results. However, at a distance of 1500 meters, the speed drops significantly and at 2500 meters, the speed equals that of ADSL2+ and VDSL. This is illustrated in Figure 1.2. Note that the distance is the length of the copper cables used, which in general is larger than the distance between the two points. As we see that larger distances result in slower connections, one should try to keep the length of the copper cables as small as possible.

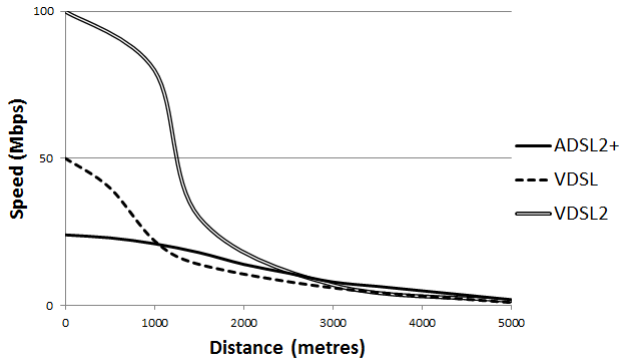

Figure 1.2: Typical DSL speed [3].

As the distances may not be too large, multiple cabinets may be needed to cover a given area. However, using multiple existing cabinets might not always be sufficient to cover the area. In that case, new cabinets have to be built and placed in the network. On the other hand, already existing cabinets might also turn out to be useless. In that case one might choose to turn that particular cabinet off, note that this does not imply that the cabinet cannot be turned back on again in the future. The location of the cabinets which need to be built is already known. Also, for both the new and the already existing cabinets it is known which are turned on and which are turned off. We are only interested in the cabinets which are turned on. In the remainder of this thesis,

when we refer to the cabinets, we will mean the cabinets which are turned on and we will not make a distinction between cabinets which need to be build and already existing ones. These cabinets are also referred to as active points.

Note that it is not always necessary to make the transition to Full FttH, even though this will give the best results. Suppose we have a situation for which Hybrid FttH, or even FttCab, suffices, then there is no need in further upgrading the network. As the maximal bandwidth is not reached in the situation of Hybrid FttH or FttCab, it will certainly not be reached in the situation of Full FttH. Upgrading will cost money, while not giving improved results. As mentioned before, using FttCab and Hybrid FttH, new techniques will also lead to improved results.

The problem we will be considering is connecting the cabinets to the fiber glass network. As direct connections to the fiber glass network might be too expensive, other networks can be used. Using already existing networks such as the street and cable patterns, connections to the fiber glass network can be established. This happens by first connecting to the street or cable (trench) pattern and then use that (already existing) network to connect with the fiber glass network. This way the problem splits in three smaller problems. First the active points have to be connected with a street or cable network. Of course this has to be done as efficient as possible. Note that the most optimal solution is not always the most practical solution to be implemented. This has to do with the other variables such as permits or the lack of possibilities to make certain connections. Therefore we would like an algorithm which finds the solution for multiple active points at once, but which also gives the results relatively quickly. For larger data sets, typically in the order of large cities, the algorithm has to finish within fifteen seconds, for smaller data sets we expect the algorithm to finish quicker. Once the cabinets are connected with the street or cable network, we still need to make a connection to the fiber glass network. For this we can use the already existing networks which saves costs. It is even possible to use both the street and the cable network. However, if both networks can be used, we should be able to make transitions from one network to the other. Such a transition is the easiest when a street and a trench cross each other. Graphically these intersection points are easily determined, however constructing an algorithm which finds these points is not straight forward. This is our second problem. Given two networks, find where they cross and create a single network with no crossings from it. The last problem comes down to finding the best connection between a cabinet and the fiber glass network. In practice this reduces to two points which have to be connected. Note that this combines both the first and the second problem. First we can connect with a network, after which we can route through the network to connect the first point with the second point. Note that the network can either be a single network, or a combination of two networks. In this last case, we first have to create a single network of it, which was the second problem sketched. The typical running time of the last algorithm should be in the order of seconds and for larger data sets not longer than a minute.

These three problems we will be considering are part of a problem raised by the Dutch telecom provider KPN. TNO has built a tool which helps solve the problem, however some approximations were made to solve the different problems easier. We will try to construct an algorithm which is exact, while also having a shorter running time. For the first algorithm, we aim for the algorithm to finish within fifteen seconds

for large data sets. For smaller data sets we expect the algorithm to finish quicker. This short time is necessary as it might be necessary to run the algorithm multiple times. Even more so if new active points are considered. The second algorithm does not need to be run multiple times. Given two networks (or shapes) we can construct the intersection points once, then create the resulting network and use this network for further calculations. This algorithm is allowed to take longer as the networks will stay the same over time. Only the active points might differ, but they do not influence the network. The running time of the second algorithm should be in the order of minutes at most. The last algorithm again needs to finish within a few seconds. As there are multiple active points and every active point has to be connected with the fiber glass network. Again, the optimal solution is not always the best solution to implement, hence the algorithm needs to finish quickly.

In Chapter 1 we will give some background information and we will introduce the problem. In Chapter 2 we will give some basic definitions on graph theory, while also considering some results on random graph theory, we will take a brief look at some complexity theory and data structures. Also some theorems will be proven which are needed in the next chapter 3. In that chapter, we will consider the algorithms constructed. However, we will first take a brief look at the data we will be working with. Apart from the algorithms constructed, we will also consider some heuristics of them and we will discuss their complexities. In Chapter 4 we will take a look at the results the algorithms give. For this we will look at the results for different data sets, compare the result of the algorithm with that of the heuristics and draw conclusions from there on. Also an advice will be given on whether to use the algorithm or an approximation of it. In this chapter we will also consider the results of the algorithms when applied to random graphs.

1.1 Results

We have constructed algorithms to solve the three problems. The first algorithm, which finds the optimal connections points for cabinets with a network, has been tested using different data sets. The largest data set with nearly 40.000 edges and 570 active points gave a result within ten seconds. We have also considered some approximations of this algorithm, however we found that in some cases the running time increased, while giving less accurate results. Even if the running time did decrease, the decrease was small, while the results were worse. For smaller data sets we found results within a second. Thus, as desired, the algorithm finishes within fifteen seconds, even for large data sets.

For the second problem we have constructed two algorithms to determine the intersection points between two networks. both algorithms have a single network with no self-intersections as output. The first algorithm is a smart version of the brute force-approach, which comes down to checking every possibility. The second algorithm is a modified version of the Bentley-Ottmann-algorithm (Section 3.3.2). We have also considered two approximations of the smart brute force-algorithm. Even though the running time did decrease in some cases, the results were less accurate. The decrease was only by a small amount, hence we suggest to not use the approximations. The modified Bentley-Ottmann-algorithm gave faster results than the smart brute force-algorithm, while still giving all intersection points. Hence, we suggest to use the modified Bentley-

Ottmann-algorithm. For small data sets the modified Bentley-Ottmann-algorithm finds results within seconds, for larger data sets, it takes up to a few minutes, as desired. For the largest practical data set (consisting of two cities) the modified Bentley-Ottmann-algorithm took only three minutes, while the smart brute force-algorithm took more than ten minutes. Hence, it meets our expectations.

The last algorithm constructed finds the least cost-path between two points. Typically, one of the points is a cabinet, while the other is a connection point to the fiber glass network. The cost is divided in two parts, one part of the cost is for digging from the points to the network, while the other cost comes from the routing through the network. In order to find the least cost-path we use the first algorithm and an implementation of Dijkstra's algorithm (see Section 3.4.1). The algorithm makes use of the fact that the least cost-path does not necessarily is the path which requires the least digging. No heuristics have been constructed, but there were some configurations which had to be set, this is done in Section 4.3.1. The goal was to create an algorithm which would find the least cost-path for two points within seconds, a minute at most. This goal was met. For smaller data sets the algorithm finds results within a few seconds, while for larger data sets the algorithm takes up to half a minute on average. For extremely large data sets such as that of the southwest of the Netherlands, the median running time of the algorithm still is only two minutes.

Chapter 2

Theoretical background

Before we start and explain the algorithms and the techniques used to solve the problems, we first give some definitions which help to understand the problem.

2.1 Basic graph theory

We first start with some basic definitions of graphs and graph theory.

Definition 2.1. An *undirected* graph G is an ordered triple $(V(G), E(G), \psi_G)$, where $V(G)$ is a nonempty set of *vertices*, $E(G)$ is a set of *edges*, $(E(G) \cap V(G) = \emptyset)$, and ψ_G is an *incidence function* which maps each edge to an *unordered* pair of vertices.

Definition 2.2. A directed graph is an undirected graph with the incidence function mapping each edge to an *ordered* pair of vertices.

Definition 2.3. A weighted graph G is an undirected graph together with a cost-function $\omega : E(G) \to \mathbb{R}$ assigning to each edge e a weight $\omega(e)$.

In practice, the weight of an edge can for example represent the length of an edge. Note that an unweighted graph can be thought of as a weighted graph with the cost-function ω mapping all edges to cost one (or another constant value). Unless stated otherwise all following graphs will be undirected unweighted graphs. Note however that all definitions can easily be extended to the definitions for directed or weighted graphs. Let us now give an example to clarify the above definition.

Example 2.4. *Let* $V(G) = \{v_1, v_2, v_3, v_4, v_5\}$, $E(G) = \{e_1, \ldots, e_{10}\}$ *and let* ψ_G *be given by*

$$\begin{aligned}
&\psi_G(e_1) = \{v_1, v_2\} && \psi_G(e_5) = \{v_1, v_3\} && \psi_G(e_9) = \{v_2, v_5\}\\
&\psi_G(e_2) = \{v_2, v_3\} && \psi_G(e_6) = \{v_1, v_3\} && \psi_G(e_{10}) = \{v_1, v_1\}\\
&\psi_G(e_3) = \{v_3, v_4\} && \psi_G(e_7) = \{v_4, v_5\}\\
&\psi_G(e_4) = \{v_4, v_1\} && \psi_G(e_8) = \{v_3, v_5\}
\end{aligned}$$

Instead of the above notation, we can also write $\psi_G(e_5) = v_1v_3$.

Graphs are called graphs, because they can be represented graphically. This graphical representation gives more insight in the properties of the graph. In practice, a graphical representation of a graph is used more than the abstract definition. Note that the graphical representation of a graph is not unique. However, in the following we will talk about *the* graphical representation, instead of *a* graphical representation.

Example 2.5. *The graphical representation of the graph in Example 2.4 is given by Figure 2.1.*

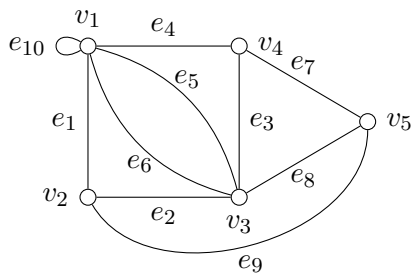

Figure 2.1: The graph of Example 2.4.

As the graphical representation of a graph is not unique, it can result in obscure or nasty representations. Therefore, there are some rules regarding the graphical representations of graphs. For instance, every edge is drawn such that it does not intersect itself. Also every pair of edges intersect at most once. Apart from these rules, also the notation can be a bit superfluous. Therefore, we will in general refer to an edge e_i as the connection of its endpoints. Thus in the above example we have $e_5 = v_1v_3$. We will also often write $G = (V, E)$ instead of $G = (V(G), E(G), \psi_G)$.

Let us now introduce some more definitions before considering some results.

Definition 2.6. An edge is said to be *incident* with its endpoints. Two edges incident with a common vertex are said to be *adjacent*. Two vertices are adjacent if they are connected by an edge.

In Example 2.5 both vertices v_4 and v_5 and edges e_7 and e_8 are adjacent.

Definition 2.7. An edge e_i is called a *loop* if it connects a vertex with itself. Two edges e_i and e_j are said to be *parallel* if they are both incident to the same two vertices.

A graph is called *simple* if it has no loops and each pair of vertices is connected by at most one edge.

In the following all graphs will be simple unless explicitly stated otherwise. Note that in the above example edge e_{10} is a loop and edges e_5 and e_6 are parallel edges. Each graph can be turned in a simple graph by deleting edges and/or vertices.

Definition 2.8. Given a graph $G = (V, E)$, $V' \subseteq V$ and $E' \subseteq E$. Then $G - V'$ is obtained from graph G by deleting the vertices in V' and deleting their incident edges. In a similar way $G - E'$ is obtained from the graph G by deletion of the edges in E'.

If $V' = \{v\}$ we denote $G - \{v\}$ by $G - v$. Similarly, if $E' = \{e\}$, then $G - e$ means $G - \{e\}$.

This extends naturally to the addition of vertices or edges.

Note that combinations of the above operations are also possible. One often would like to say something about one graph being a subset of another graph. For instance, when we delete an edge, we end up with a subgraph of the original one.

Definition 2.9. Let $G = (V, E)$ be a graph. A graph $H = (W, F)$ is said to be a *super graph* (or *supgraph*) of G if $V \subseteq W$ and $E \subseteq F$. G is then said to be a *subgraph* of H.

For the incidence function this implicitly means $\psi_G = \psi_H|_E$.

Example 2.10. *The graph in Example 2.5 becomes a simple graph if we delete edges e_5 and e_{10}. Similarly we can delete vertex v_1 to get a simple graph. These operations result in the simple graphs depicted in Figure 2.1.*

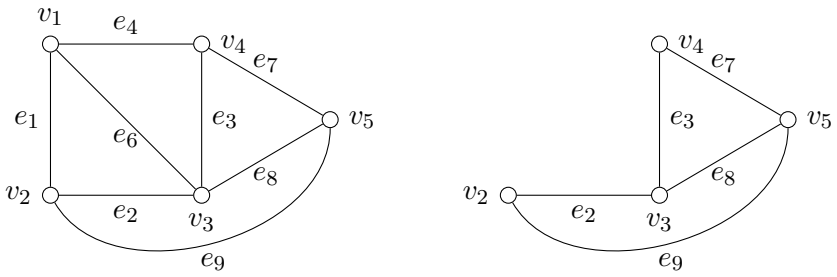

Figure 2.2: $G - \{e_5, e_{10}\}$ and $G - v_1$.

In the above example we see that $G - \{e_5, e_{10}\}$ is a subgraph of G and that G is a super graph of $G - v_1$.

A lot of the problems involving graphs focus on going from one point to another point as quick or as cheap as possible with respect to a weight function. In terms of the graphs this means going from one vertex to another vertex using the edges. If we can go from one vertex to another vertex, there exists a list of edges, all in E, which together connect the two vertices. Such a list is called a walk or a path.

Definition 2.11. Given a graph $G = (V, E)$. A *walk* in $G = (V, E)$ is a tuple $(v_1, \ldots, v_k) \in V^k$ such that $v_i v_{i+1} \in E$ for all $i = 1, \ldots, k-1$.

A *path* in the graph is a walk such that $v_i = v_j \Leftrightarrow i = j$. A *cycle* (or closed path) is a path such that $v_1 = v_k$.

With this definition we can define connectivity for graphs.

Definition 2.12. Given a graph $G = (V, E)$. A subset $V' \subseteq V$ is called a *connected component* of G if for all v'_i and v'_j in V' there is a path between v'_i and v'_j and for all $v'_i \in V'$ and $w_j \notin V'$ there is no path between v'_i and w_j.

A graph $G = (V, E)$ is called *connected* if and only if there is only one connected component. A graph is called *disconnected* if it is not connected.

We see that this definition is equivalent to stating that G is connected if and only if for every pair of vertices v_i and v_j, there exists a path between them. Note that the trivial graph, i.e., the graph with only one node and no edges is also connected. From this definition, we can also define *trees*.

Definition 2.13. A *tree* is a connected graph $G = (V, E)$, such that $|V| = |E| + 1$. Here $|V|$ and $|E|$ stands for the number of elements in V and E respectively.

A node is called a *leaf* if it is adjacent with only one other node.

Given a graph $G = (V, E)$, then T is said to be a spanning tree of G if T contains all vertices of G and T is a tree.

A rooted tree is a tree in which one vertex is chosen as root.

One can think of the root of a rooted tree as the node from which the tree starts. Note that deleting any edge in a tree results in a disconnected graph. Deleting a node will also result in a disconnected graph unless the node is a leaf. Let us now say something about the number of edges incident with each vertex.

Definition 2.14. Given the graph $G = (V, E)$ and given a vertex $v \in V$, then the degree $d_G(v)$ of a vertex v denotes the number of edges incident with v. Note that a loop counts as two edges incident with v.

The number of edges incident with both v and w is denoted by $d_G(v, w)$.

If possible we will omit the subscript G. This definition can be extended to directed graphs, using indegree and outdegree. The indegree $d^-(v)$ denotes the number of edges ending at v, while $d^+(v)$ denotes the number of edges starting in v. Using the definition of the degree of a node, we can construct the adjacency matrix of a graph. With this matrix we have all the information we need to reconstruct the graph.

Definition 2.15. The adjacency matrix A_G of a graph G is defined by

$$A_G(i, j) = d_G(v_i, v_j).$$

If G is simple, this definition reduces to

$$A_G(i, j) = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{else} \end{cases}$$

Thus $A_G(i, j)$ equals 1 if there is an edge between vertex v_i and vertex v_j and 0 if there is no such edge.

Note that the sum of each column equals two as each edge has two endpoints. Also note that this implies that the sum of the degrees equals two times the number of edges, as each edge is counted twice. Similar to the adjacency matrix is the incidence matrix.

Definition 2.16. An incidence matrix B of a simple graph $G = (V, E)$ is a $n \times m$-matrix, with n the number of vertices and m the number of edges, such that

$$B(i, j) = \begin{cases} 1 & \text{if } v_i \text{ and } e_j \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

In general the adjacency matrix is preferred over the incidence matrix. Let us now give an interesting proposition regarding the degrees in a graph.

Proposition 2.17. *Every finite graph has an even number of vertices with odd degree.*

Proof. Suppose not, then the sum of the degrees is odd which contradicts that the sum of degrees equals twice the number of edges. □

Corollary 2.18. *Every finite tree has at least two leaves.*

When we consider the graphical embedding of a graph, there are some interesting properties which give some rather interesting results.

Definition 2.19. A *planar* graph is a graph which has a graphical representation in the plane such that the edges only intersect at their endpoints.

In a planar graph the regions bounded by edges are called the *faces* of the graph.

Note that each planar graph has at least one face, being the exterior of the graph. Convince yourself that the graphs in Example 2.10 are planar graphs. Planar graphs satisfy a relation which became known as Euler's formula.

Theorem 2.20. *Let G be a planar graph, let n denote the number of vertices, m the number of edges, c the number of connected components and f the number of faces. Then*

$$n + f = m + c + 1. \tag{2.1}$$

Proof. Let us first prove this for the case that G is connected, by induction to the number of faces. If G has only one face and G is connected, we can have no cycles, therefore we see that G is a tree. Thus $n = m + 1$ and hence

$$n + f = m + 1 + 1 = m + c + 1.$$

Now suppose it holds for f faces and suppose G has $f + 1$ faces. Let e be an edge separating two different faces. Then $G - e$ has one face less, but also one edge less. By the induction hypothesis we know that for $G - e$ Euler's formula holds, therefore the formula also holds for G.

Now suppose that G is disconnected. By adding precisely $c - 1$ edges, we can turn G in a connected graph. For this graph the formula holds. Therefore by removing the $c - 1$ edges again, we get $c - 1$ extra connected components. Therefore the formula also holds for disconnected graphs. □

2.2 Random graphs

In general when we consider a graph everything about that graph is given. We know which vertices we have and we know, deterministically, which vertices are connected. However, in everyday life, it is not always known if there is a connection between two vertices. Take for example as vertex set a group of people and let there be an edge between two vertices if the two corresponding people are friends. Of course we can construct the precise form of the graph by asking every person who their friends are,

however this is a lengthy process. Apart from that one is in general more interested in the general form of the graph than the precise form of the graph. A more efficient approach to this problem would be to construct a random graph, where each edge can be added with a certain probability. The two famous Hungarian mathematicians Paul Erdös and Alfréd Rényi published an article in 1959 on random graphs [4]. In this article they discussed the behaviour as the probability to add edges changed. Many new models have been constructed, but the Erdös-Rényi model is still the most studied model. Other examples of random graphs (not necessarily via the Erdös-Rényi model) are the World Wide-Web [5], collaboration graphs [6],[7] and neural networks [8].

Erdös and Rényi considered the model $\widetilde{G}(n, M)$. In this model, with fixed n and M in $\mathbb{N}$, we choose with equal probability a simple graph with n nodes and M edges. Thus in the $\widetilde{G}(3, 2)$-model we have a graph with three nodes and two edges. Each of the three possibilities can be chosen with equal probability $1/3$. We will consider a closely related model $G(n, p)$ in which edges are added with certain probability. This model was first studied by Gilbert in [9]. We will then prove some theorems of the behaviour of this model as the edge probability grows.

Definition 2.21. The graph $G(n, p)$ is the simple graph with n (fixed) vertices and where each possible edge is added with probability p. Every graph with n nodes and m edges has equal probability

$$p^n(1-p)^{\binom{n}{2}-m}.$$

Note that for $G(n, 0.5)$ every graph has equal probability. The model $G(n, p)$ is in general favored as each edge is selected independently, while for $\widetilde{G}(n, M)$ this approach does not work. Let us first consider the expected number of edges of $G(n, p)$ and then say something about the way the two models are related. As each edge is chosen with probability p, the expected number of edges is $p\binom{n}{2}$. The law of large numbers tells us that as n grows, $G(n, p)$ will have $p\binom{n}{2}$ edges with increasing probability. Thus as $n \to \infty$, $G(n, p)$ will tend to $\widetilde{G}(n, M)$ for $M = p\binom{n}{2}$, that is to say we have

$$\lim_{n\to\infty} \mathbb{P}\left(\#\text{ edges in } G(n,p) = p\binom{n}{2}\right) = 1. \tag{2.2}$$

Theorem 2.22. *Let $M = p\binom{n}{2}$, then*

$$\lim_{n\to\infty} \mathbb{P}\left(\#\textit{ edges in } G(n,p) = p\binom{n}{2}\right) = 1,$$

in which case $G(n, p)$ behaves similar as $\widetilde{G}(n, M)$.

Proof. Each edge is added with probability p, hence the expected number of edges will be $M = p\binom{n}{2}$. By the law of large numbers, the probability that the graph has this number of edges goes to one (more on the law of large numbers can be found in [10], chapter ten).

Note that every graph with M edges is equally likely, as for each graph the probability of it emerging equals $p^M(1-p)^{\binom{n}{2}-M}$ and we have a total of $\binom{\binom{n}{2}}{M}$ possible graphs. Therefore, both models are similar. □

2.2.1 Behaviour as $n \to \infty$

We will now prove some results on random graph theory and the behaviour of the graphs as $n \to \infty$. For further reading on this topic we refer to [11], [12] and [13], also detailed proofs of the following theorems can be found there. In the following we will only consider the model $G(n, p)$. We will furthermore, for fixed n, consider the set of all graphs with n vertices.

When we consider normal graphs we can give the degree of each vertex, say which vertices are connected with each other, determine the number of cycles of a given length and so on. We can, by looking at the graph, determine all properties of the graph. For random graphs this is a bit harder. We cannot determine which vertices are connected as edges are added with a certain probability. However as the number of vertices grows, the graph will behave according to a pattern which is dependent on p. Let us, for different p, take a closer look at the different properties of the graph as n grows.

Even though we are considering random graphs we can say something about their behaviour as n tends to ∞. Let us first consider the degree distribution for $G(n, p)$. Each vertex has expected degree $\mathbb{E}(d(i)) = p(n-1) \approx pn$, however the probability that a vertex has degree k is given by

$$\mathbb{P}(d(v) = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k} \approx \binom{n}{k} p^k (1-p)^{n-k}, \tag{2.3}$$

where the factor $\binom{n}{k}$ comes from the number of ways to pick k vertices from n. We cannot deterministically say something about the number of times every degree will appear. However, we can say something about the expected number of times every degree will appear. As this is just the probability for that degree times the number of vertices. For fixed n we see that as p grows, the expected degrees of vertices increase. However, if we fix p and we let n grow, we also see changes happen. As it turns out p and n are intertwined, meaning that the growth of one of these two results in structural changes in the graph. As an example we can think of $p = 0$ and $p = 1$, one will lead to a completely isolated graph, while the other will result in a complete graph.

As it turns out, the degree of the points is not the only property for which the graph undergoes structural changes as p grows. Most properties of random graphs undergo changes as p (or n) grows. Most of these changes are abrupt, therefore probabilities at which these abrupt changes happen are often referred to as threshold values. An example is the threshold value $p = \frac{\ln n}{n}$ for which isolated vertices will vanish as we will prove in Lemma 2.31. The graph properties can be anything from connectivity to the existence of trees of a given length, the only restriction is that the property is monotone increasing.

Definition 2.23. A property P is called *monotone increasing*, or monotone, if when a graph G has the property P, then so does any super graph H of G.

A monotone increasing graph property is said to be *non-trivial* if not all graphs have the property.

Similarly we can define monotone decreasing graph properties, but with H being a subgraph of G. Note that the properties we want to say something about have to be monotone increasing, otherwise the statement that all super graphs have the same

property does not have to hold. Consider for instance the property for the graph being disconnected. We see that this graph property is certainly not monotone increasing (in fact, its is monotone decreasing). An example of a trivial graph property is the property of being a graph. All graphs have this property. In the following we will only consider non-trivial monotone increasing graph properties.

Note that the graphs G and H in the above definition are deterministic graphs. For random graphs this definition reduces to a statement about the probability that the graph has the monotone increasing property.

Lemma 2.24. *Let P be a monotone increasing property and let $0 \leq p \leq q \leq 1$, then the probability that $G(n,p)$ has property P is at most the probability that $G(n,q)$ has property P.*

Proof. First generate a graph on n vertices with edge probability p, so generate $G(n,p)$. Now generate $G(n, \frac{q-p}{1-p})$ independently of $G(n,p)$ and take the union of both, $H = G(n,p) \cup G(n, \frac{q-p}{1-p})$. Thus e is an edge of H if it is an edge of $G(n,p)$ or $G(n, \frac{q-p}{1-p})$. We now see that H has the same distribution as $G(n,q)$, as the probability for an edge to be in H is

$$p + (1-p)\frac{q-p}{1-p} = q.$$

As P is a monotone increasing property we see that if $G(n,p)$ has property P, then so does H. Hence, the probability that $G(n,p)$ has property P is at most the probability that H has property P. The lemma now follows from the fact that H and $G(n,q)$ have the same distribution. □

Note that these definitions require n to be fixed. Let us now define thresholds for monotone increasing graph properties.

Definition 2.25. $p : \mathbb{N} \to [0,1]$ is called a threshold function for a monotone increasing graph property P if

(i) For $p_1 : \mathbb{N} \to [0,1]$ and $p_1(n) \ll p(n)$ the probability that the graph $G(n, p_1(n))$ has property P goes to zero as $n \to \infty$;

(ii) For $p_2 : \mathbb{N} \to [0,1]$ and $p_2(n) \gg p(n)$ the probability that the graph $G(n, p_2(n))$ has property P goes to one as $n \to \infty$.

Here $p_1(n) \ll p(n)$ means "$p_1(n)$ *is much smaller than* $p(n)$". Similarly $p_2(n) \gg p(n)$ means "$p_2(n)$ *is much greater than* $p(n)$". Note that *much smaller/greater* means much smaller/greater in the limit of n to infinity. In Section 2.3 we will give formal definitions of these notations. An example of two functions $f(n), g(n)$ such that $f(n) \ll g(n)$ is $f(n) = 1/(n \ln n)$ and $g(n) = 1/n$.

Notice that this definitions allows for multiple thresholds for the same property. A function which only differs by a constant from the threshold $p(n)$ is also a threshold. Random graph theory focuses mainly on finding thresholds for various monotone increasing properties, some of which we will consider. Therefore the result of Bollobás and Thomason in [14] is of great importance to the field of random graph theory.

Theorem 2.26. *Every monotone increasing graph property has a threshold.*

For the proof of this theorem we refer to [14].

As we have seen above a constant times a threshold function is also threshold function. However, there are threshold functions which are more subtle than others. We call them *sharp thresholds*, more precisely

Definition 2.27. Let $p : \mathbb{N} \to [0, 1]$ be a function. We say that p is the *sharp threshold* for the monotone increasing property P and a phase transition occurs at $p(n)$ if there is $\varepsilon > 0$ such that

(i) If for all $n \in \mathbb{N} : cp(n) \leq 1 - \varepsilon,\ c < 1$, then the probability that the graph $G(n, cp(n))$ has property P goes to zero as $n \to \infty$;

(ii) If for all $n \in \mathbb{N} : cp(n) \geq 1 + \varepsilon,\ c > 1$, then the probability that the graph $G(n, cp(n))$ has property P goes to one as $n \to \infty$.

Before we consider some monotone increasing properties, we will give a few lemma's which will help proving further results.

Lemma 2.28. *Let X be a non-negative integrable random variable, and $a \in \mathbb{R}^{>0}$. Then*

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}(X)}{a} \tag{2.4}$$

This inequality was stated by the Russian mathematician Andrey Markov. It can be proven using the definition of $\mathbb{E}(X)$. Note that the this lemma implies the case where X is discrete. A direct consequence of this lemma is the following.

Lemma 2.29. *Let X be a non-negative integrable random variable, and $a \in \mathbb{R}^{>0}$. Then*

$$\mathbb{P}(|X - \mathbb{E}(X)| \geq a) \leq \frac{\mathbb{E}((X - \mathbb{E}(X))^2)}{a^2} = \frac{\mathrm{Var(X)}}{a^2} \tag{2.5}$$

Proof. Apply Lemma 2.28 with the substitution $X \to (X - \mathbb{E}(X))^2$ and $a \to a^2$. Then use that $(X - \mathbb{E}(X))^2 \geq a^2$ if and only if $|X - \mathbb{E}(X)| \geq a$. □

Even though Lemma 2.29 follows directly from Lemma 2.28, it has become well known due to its practical uses. One often refers to it as *Chebyshev's inequality*. Using these two lemma we can prove the next result.

Lemma 2.30. *Let (X_n) be a sequence of stochastic variables. Then we have*

a. If $\mathbb{E}(X_n) \to 0$ and $\mathrm{Var(X_n)} \to 0$, then $\mathbb{P}(X_n > 0) \to 0$.

b. If $\mathbb{E}(X_n) \to \infty$ and $\mathrm{Var(X_n)} \to 0$, then $\mathbb{P}(X_n = 0) \to 0$.

Proof. a. Let $a > 0$ and $\varepsilon > 0$ be arbitrary. Choose $N \in \mathbb{N}$ such that for all $n \geq N$: $\mathbb{E}(X_n) < \frac{a}{2}$ and $\mathrm{Var(X_n)} < \frac{\varepsilon a^2}{4}$. Note that this is possible as both $\mathbb{E}(X_n) \to 0$ and $\mathrm{Var(X_n)} \to 0$. Now apply Lemma 2.29 which implies

$$\mathbb{P}(X_n \geq a) \leq \mathbb{P}(|X_n - \mathbb{E}(X_n)| \geq \frac{a}{2}) \leq \frac{4\mathrm{Var(X_n)}}{a^2} < \varepsilon.$$

As this holds for all $a > 0$ and $\varepsilon > 0$ we see that $\mathbb{P}(X_n) > 0) \to 0$.

b. Let $\varepsilon > 0$ and $a > 0$ be arbitrary. Choose $N \in \mathbb{N}$ such that for all $n \geq N$: $\mathrm{Var(X_n)} < \varepsilon \mathrm{a}^2$ and $\mathbb{E}(X_n) > a$. Note that both is possible as in the limit the variance goes to zero, while the expectation value goes to infinity. Also note that

$$\mathbb{P}(|X_n - \mathbb{E}(X_n)| < a) + \mathbb{P}(|X_n - \mathbb{E}(X_n)| \geq a) = 1.$$

By the first part of this lemma we have

$$\mathbb{P}(|X_n - \mathbb{E}(X_n)| \geq a) \leq \frac{\mathrm{Var(X_n)}}{a^2} < \varepsilon,$$

hence we have

$$\mathbb{P}(|X_n - \mathbb{E}(X_n)| < a) = 1 - \mathbb{P}(|X_n - \mathbb{E}(X_n)| \geq a) > 1 - \varepsilon.$$

As a and ε were arbitrary we see that in the limit $\mathbb{P}(|X_n - \mathbb{E}(X_n)| < a) = 1$ for all $a > 0$. Hence, in the limit $\mathbb{P}(X_n \to \mathbb{E}(X_n)) = 1$. Therefore $\mathbb{P}(X_n \to \infty) = 1$, thus we get $\mathbb{P}(X_n = 0) \to 0$. □

Let us now prove some results on different monotone increasing graph properties. First we will consider the property of the graph not having isolated vertices, note that this is indeed a monotone increasing graph property for fixed n. A graph having isolated vertices means that there is no vertex with degree zero. As it turns out, for probabilities smaller than $p(n) = \frac{\ln n}{n}$ the graph will have isolated vertices as $n \to \infty$, while for higher probabilities the graph will not have isolated vertices with probability going to one. Hence $p(n) = \frac{\ln n}{n}$ is the sharp threshold for the graph to have isolated vertices.

Lemma 2.31. *The sharp threshold for a graph in $G(n, p)$ having no isolated vertices is $p(n) = \frac{\ln n}{n}$.*

Proof of Lemma 2.31. A vertex v is isolated if it has no connections with other vertices. This happens with probability $(1 - p(n))^{n-1}$. Let x be the number of isolated vertices in the graph, then we can say $x = \sum_{i=1}^{n} x_i$, where $x_i = 1$ if node i is isolated. Then

$$\begin{aligned}
\mathbb{E}(x) &= \mathbb{E}\left(\sum_{i=1}^{n} x_i\right) \\
&= \sum_{i=1}^{n} \mathbb{E}(x_i) \\
&= n\mathbb{E}(x_1) \\
&= n(1 - p(n))^{n-1}
\end{aligned}$$

This follows from the fact that a node is isolated with probability $(1 - p(n))^{n-1}$. We claimed that the sharp threshold was $p(n) = \frac{\ln n}{n}$, thus let us consider $p = cp(n) = \frac{c \ln n}{n}$.

Then we get

$$\begin{aligned}\lim_{n\to\infty}\mathbb{E}(x) &= \lim_{n\to\infty} n\,(1-p)^{n-1}\\ &= \lim_{n\to\infty} n\left(1-\frac{c\ln n}{n}\right)^{n-1}\\ &= \lim_{n\to\infty} n\left(1-\frac{c\ln n}{n}\right)^{n}\\ &= \lim_{n\to\infty} ne^{-c\ln n}\\ &= \lim_{n\to\infty} nn^{-c}\end{aligned}$$

If $c > 1$ this limit goes to zero, while for $c < 1$ this limit goes to infinity. If this limit goes to zero, we see that the expected number of isolated vertices goes to zero. However, for $c < 1$, this limit tending to infinity does not imply the existence of isolated vertices with probability going to one. Therefore, we are not yet finished.

First note that we have

$$\begin{aligned}\mathbb{E}(x^2) &= \mathbb{E}\left((x_1+\ldots+x_n)^2\right)\\ &= \sum_{i=1}^{n}\mathbb{E}\left(x_i^2\right)+\sum_{i\neq j}\mathbb{E}\left(x_ix_j\right)\end{aligned}$$

As $x_i \in \{0,1\}$ we see $x_i = x_i^2$. We also see that the term $\mathbb{E}\left(x_ix_j\right)$ is independent of i and j, hence all these terms are equal. This then gives

$$\begin{aligned}\mathbb{E}(x^2) &= \sum_{i=1}^{n}\mathbb{E}\left(x_i^2\right)+\sum_{i\neq j}\mathbb{E}\left(x_ix_j\right)\\ &= \sum_{i=1}^{n}\mathbb{E}\left(x_i\right)+\sum_{i\neq j}\mathbb{E}\left(x_1x_2\right)\\ &= \mathbb{E}(x)+n(n-1)\mathbb{E}\left(x_1x_2\right)\\ &= \mathbb{E}(x)+n(n-1)(1-p)^{2(n-1)-1}\end{aligned}$$

Note the extra -1 in the above expression which comes from the double counting of edge $1-2$. We will now use Lemma 2.30. We will take $a = 1$ and we will prove that $\mathrm{Var(x)} \to 0$ as $n \to \infty$, the lemma then states that the probability that x equals zero goes to zero, which implies that the probability that there are no isolated vertices goes to zero. Hence, the result for $c < 1$ will also be proven.

Note that proving that the right hand side equals zero is equal to proving that $\frac{\mathbb{E}(x^2)}{\mathbb{E}(x)^2}$

tends to one in the limit.

$$\begin{aligned}\frac{\mathbb{E}(x^2)}{\mathbb{E}(x)^2} &= \frac{n(1-p)^{n-1} + n(n-1)(1-p)^{2(n-1)-1}}{n^2(1-p)^{2(n-1)}} \\ &= \frac{1 + (n-1)(1-p)^{n-2}}{n(1-p)^{n-1}} \\ &= \frac{1}{n(1-p)^{n-1}} + \frac{(n-1)}{n(1-p)} \\ &= \frac{1}{n(1-p)^{n-1}} + \frac{1}{1-p} - \frac{1}{n(1-p)}\end{aligned}$$

Now recall that $p = cp(n) = \frac{c \ln n}{n}$, $c < 1$ and $\lim_{n\to\infty} \mathbb{E}(x) = \lim_{n\to\infty} n(1-p)^{n-1} = \infty$. Taking the limit then yields

$$\begin{aligned}\lim_{n\to\infty} \frac{\mathbb{E}(x^2)}{\mathbb{E}^2(x)} &= \lim_{n\to\infty} \left(\frac{1}{n(1-p)^{n-1}} + \frac{1}{1-p} - \frac{1}{n(1-p)} \right) \\ &= \lim_{n\to\infty} \left(\frac{1}{n\left(1 - \frac{c\ln n}{n}\right)^{n-1}} + \frac{1}{1 - \frac{c\ln n}{n}} - \frac{1}{n\left(1 - \frac{c\ln n}{n}\right)} \right) \\ &= 0 + \frac{1}{1-0} - 0 \\ &= 1.\end{aligned}$$

Thus by the lemma we now see that $\mathbb{P}(|x - \mathbb{E}(x)| \geq 1) \to 0$, hence $\mathbb{P}(X = 0) \to 0$. So for $c < 1$ the probability that the graph has no isolated vertices goes to zero.

Combining the above results we see that $p(n) = \frac{\ln n}{n}$ is a sharp threshold for graphs in $G(n, p)$ to have isolated vertices. □

In the above lemma we had a threshold of $p(n) = \frac{\ln n}{n}$, however this value of p is also the threshold at which the graph becomes connected with probability one as $n \to \infty$. Note that for fixed n, being connected is a monotone increasing graph property.

Theorem 2.32. *Let $G(n, p)$ be a random graph, then $p(n) = \frac{\ln n}{n}$ is the sharp threshold for a graph in $G(n, p)$ to be connected.*

Proof. Claiming that a graph in $G(n, p)$ is connected is the same as claiming that as $n \to \infty$, there are no isolated components on k vertices for all $k = 1, \ldots, \frac{n}{2}$. We will prove that this holds for $p = p(n) = \frac{c \ln n}{n}$ with $c > 1$. For $c < 1$ there are isolated vertices with probability going to one, hence the graph is disconnected with probability going to one.

Given a set of k vertices. The probability that these vertices form a connected component of size k, equals the probability that the k vertices are connected times the probability that the k vertices are not connected to other vertices. For this we will consider the spanning tree of the connected components, as extra edges will not change the situation. On k vertices there are at most k^{k-2} possible spanning trees. Therefore the probability that these k vertices form a connected component of size k is at most

$$k^{k-2} p^{k-1} (1-p)^{k(n-k)}.$$

Note that the *at most* follows as the probabilities for multiple spanning trees are not necessarily independent. The factor $(1-p)^{k(n-k)}$ takes into account the fact that the k vertices cannot be connected with vertices not in the connected component. Let x_k be the number of connected components of size exactly k. Then

$$\mathbb{E}(x_k) \leq \binom{n}{k} k^{k-2} p^{k-1} (1-p)^{k(n-k)}.$$

The extra binomial factor takes into account the number of ways we can pick our k vertices. We will now use that $p = cp(n) = \frac{c \ln n}{n}, \binom{n}{k} \leq \left(\frac{en}{k}\right)^k, (1-p) \leq e^{-p}$ and $x = e^{\ln x}$. Then we get

$$\begin{aligned}
\mathbb{E}(x_k) &\leq \binom{n}{k} k^{k-2} p^{k-1} (1-p)^{k(n-k)} \\
&\leq \left(\frac{en}{k}\right)^k \left(\frac{c \ln n}{n}\right)^{k-1} e^{-\frac{c \ln n}{n}(kn-k^2)} k^{k-2} \\
&= \exp\left(\ln\left(\left(\frac{en}{k}\right)^k\right) + \ln\left(\left(\frac{c \ln n}{n}\right)^{k-1}\right) + \ln\left(e^{-\frac{c \ln n}{n}(kn-k^2)}\right) + \ln\left(k^{k-2}\right)\right) \\
&\leq \exp\left(k + k \ln c + k \ln(\ln n) - 2 \ln k + \ln n - ck \ln n + ck^2 \frac{\ln n}{n}\right) \qquad (2.6)
\end{aligned}$$

Let us now consider

$$f(k) = k + k \ln c + k \ln(\ln n) - 2 \ln k + \ln n - ck \ln n + ck^2 \frac{\ln n}{n}. \qquad (2.7)$$

As exp is an increasing function, we see that Equation (2.6) is minimal if and only if Equation (2.7) is minimal. We now see that

$$f'(k) = 1 + \ln c + \ln(\ln n) - \frac{2}{k} - c \ln n + 2ck \frac{\ln n}{n}$$

and

$$f''(k) = \frac{2}{k^2} + 2c \frac{\ln n}{n}.$$

As $f''(k) > 0$ for all k and exp is convex we see that the maximum of $f(k)$ has to be attained at either $k = 2$ or at $k = \frac{n}{2}$, as these are the endpoints of the interval we are considering. For n large enough we have $f(2) > f(\frac{n}{2})$. If we now only consider the dominant terms for the maximum we get

$$f(k) = (1 - 2c) \ln n + \mathcal{O}(\ln(\ln n)),$$

which is independent of k. The definition of "$\mathcal{O}(\ln(\ln n))$" will be given in Section 2.3, however, what matters is that this part is insignificant in the limit compared with the other terms. Using this we get

$$\mathbb{E}(x_k) \leq \exp^{(1-2c) \ln n} = n^{1-2c}.$$

Now let us consider the expected number of cycles with length $k = 2, \ldots, \frac{n}{2}$, then we get

$$\mathbb{E}\left(\sum_{k=2}^{n/2} x_k\right) \leq \sum_{k=2}^{n/2} n^{1-2c} \leq \sum_{k=1}^{n} n^{1-2c} = n^{2-2c}.$$

Summarizing, for $p(n) = \frac{c \ln n}{n}$ we that for $c > 1$ we see that the right hand side tends to zero as $n \to \infty$. Hence, with probability going to one there will be no connected components of size $k = 2, \ldots, \frac{n}{2}$, hence graphs in $G(n, p(n))$ will be connected with probability going to one as $n \to \infty$. In Lemma 2.31 we had proven that for $c < 1$ there will be isolated vertices with probability going to one, hence a graph in $G(n, p(n))$ will be disconnected with probability going to one. Both combined gives that $p(n) = \frac{\ln n}{n}$ as a sharp threshold for graphs in $G(n, p)$ being connected. □

Between the sharp threshold for graphs in $G(n, p)$ being connected and the case where graphs in $G(n, p)$ are completely disconnected ($p = 0$), we have some intermediate stages. As $p(n)$ grows, connections will appear and we see that trees of arbitrary length begin to form. If $p(n)$ grows further, larger components begin to form. These components all have a size of the same order. At $p(n) = \frac{1}{n}$ we see that there will be one giant component, with several smaller components and isolated vertices. As $p(n)$ continues to grow, the graph will be connected with probability going to one, as $n \to \infty$, at $p(n) = \frac{\ln n}{n}$.

We have already considered the case $p(n) = \frac{\ln n}{n}$, however, let us also consider $p(n) = \frac{1}{n}$, the case of the *giant component*.

Definition 2.33. In random graph theory a *giant component* is a connected component containing a constant fraction of the nodes, i.e., there is a constant $\rho \in [0, 1]$ such that for all $n \in \mathbb{N}$ the giant component contains more than ρn vertices. The size of other components is small, typically proportional to $\ln n$ at most.

Example 2.34. *As an example of a giant component one can look at the social network of the world, where every person is represented by a node and a link exists if two people are friends. This network is presumably not connected, in fact, the behaviour of a single node (or a small set of nodes) can make the network disconnected. For instance, a single person with no living friends will result in a disconnected network. Another example can be a remote island whose inhabitants have never had contact with the outside world.*

Even though this network is not connected,we can consider the component we are in. All of our friends are in this component, all their friends are in this component and so on. People you have never met and might even be on the other side of the world are in this component. This component is likely to contain a large fraction of the world's population.

Even though a graph might not be connected, if there exists a giant component we can think of $G(n, p)$ as being almost connected. There is one component which contains most vertices, while the vertices which are not in the giant component are in relatively small connected components.

Theorem 2.35. *The sharp threshold for a graph in $G(n, p)$ to have a giant component is $p(n) = \frac{1}{n}$.*

The proof relies on branching processes and it requires a lot of definitions and prior knowledge before we can give a clear proof. See Theorem 2 in [4] for a proof of this theorem. More on the giant component can for instance be found in Chapter 6 of [11].

As stated, $p(n) = \frac{1}{n}$ is a sharp threshold for a random graph to have a giant component. In fact, in the subcritical regime ($p(n) = \frac{c}{n}$, $c < 1$) there are multiple smaller components whose size is relatively the same. In the supercritical regime (where $c > 1$) there is a giant component with a significant fraction of the nodes. At the sharp threshold abrupt changes will happen. The abrupt changes can have great influence. This is illustrated in the book *Guns, Germs, and Steel: The Fates of Human Societies* ([15]). Around the year 1500 there were two components (that of America and that consisting of Europa and Asia), due to colonization trips they merge into one giant component. This book describes the impact this had.

Let us now prove a different result for the threshold $p(n) = \frac{1}{n}$.

Theorem 2.36. *The threshold for the existence of cycles in a graph in $G(n, p)$ is $p(n) = \frac{1}{n}$.*

Proof. As there are no cycles of length 0, 1 or 2, let $k \geq 3$. Let x_k be the number of cycles in the graph of length k and let $x = x_3 + \ldots + x_n$. Consider a cycle of length k, we can pick the k vertices in $\binom{n}{k}$ ways and make the cycle in $(k-1) \cdot (k-2) \cdot \ldots \cdot 1$ ways. As we have counted each possible cycle twice, we have $(k-1)!/2$ possible cycles given a set of k vertices. The probability that all those edges are in the graph is given by p^k, which gives

$$\mathbb{E}(x_k) = \binom{n}{k} \frac{(k-1)!}{2} p^k$$

and

$$\mathbb{E}(x) = \sum_{k=3}^{n} \binom{n}{k} \frac{(k-1)!}{2} p^k.$$

Note that it does not matter whether or not other edges are added. Now consider again

$p = cp(n) = \frac{c}{n}$. For $c < 1$ we now find

$$\begin{aligned}
\mathbb{E}(x) &= \sum_{k=3}^{n} \binom{n}{k} \frac{(k-1)!}{2} p^k \\
&\leq \sum_{k=3}^{n} \frac{1}{2} \frac{n!(k-1)!}{k!(n-k)!} p^k \\
&\leq \sum_{k=3}^{n} \frac{n^k}{2k} p^k \\
&\leq \sum_{k=3}^{n} (np)^k \\
&= \sum_{k=3}^{n} c^k \\
&= \frac{c^3 - c^{n+1}}{1-c} \\
&= c^3 \frac{1 - c^{n-2}}{1-c} \\
&\leq c^3 \\
&< 1
\end{aligned}$$

Hence, for $c < 1$ in the limit we expect there to be no cycles of any given length. Let us now consider the case $c > 1$. Then we have

$$\begin{aligned}
\mathbb{E}(x) &= \sum_{k=3}^{n} \binom{n}{k} \frac{(k-1)!}{2} p^k \\
&= \frac{1}{2} \sum_{k=3}^{n} \frac{n(n-1)\dots(n-k+1)}{n^k} \frac{c^k}{k} \\
&\geq \frac{1}{2} \sum_{k=3}^{n} \frac{n(n-1)\dots(n-k+1)}{n^k} \frac{1}{k}.
\end{aligned}$$

Let us now only consider the first $\log n$ terms of the sum. As we know that $\frac{n}{n-i} = 1 + \frac{i}{n-i} \leq e^{\frac{i}{n-i}}$, we see that $\frac{n(n-1)\dots(n-k+1)}{n^k} \geq \frac{1}{2}$, hence

$$\begin{aligned}
\lim_{n\to\infty} \mathbb{E}(x) &\geq \lim_{n\to\infty} \frac{1}{2} \sum_{k=3}^{n} \frac{n(n-1)\dots(n-k+1)}{n^k} \frac{1}{k} \\
&\geq \lim_{n\to\infty} \frac{1}{4} \sum_{k=3}^{\log n} \frac{1}{k} \\
&\geq \lim_{n\to\infty} \log(\log n) \\
&= \infty
\end{aligned}$$

So for $c > 1$ we see that $\mathbb{E}(x) \to \infty$. Using Lemma 2.30 one can prove that the variance goes to zero which completes the proof, as in that case the probability that there are no cycles goes to zero.

Summarizing we see that the sharp threshold for the existence of cycles is $p = \frac{1}{n}$. □

2.3 Complexity

When we are comparing two algorithms with each other, we often want to classify one as more difficult or harder than the other. Doing this classification with respect to the running times of the algorithms will in general not tell us which algorithm is harder. Even if we would use the running time as a measure of complexity, we would get different results every time. To run the algorithm, a number of steps have to be taken. However, the time spent to do all steps might vary, depending on both the hardware you use as well as the moment in time you do the calculations. Moreover, the number of steps might be dependent on the input of the algorithms. Therefore, when comparing two algorithms, we want to use something that is independent of the hardware or other processes which might be running. Therefore we will consider the number of operations needed to run the algorithm as a measure of its complexity. In general it is not necessary to know the exact number of iterations necessary given an input. Instead one often looks at the limiting behaviour of the algorithms in terms of the input size. Most common is the use of the *Big O*-notation.

Definition 2.37. Given a function f, then $f(x) \in \mathcal{O}(g(x))$ as $x \to \infty$ if there exist a $C \geq 0$ and x_0 such that

$$|f(x)| \leq C\,|g(x)| \qquad \forall x \geq x_0.$$

Then $\mathcal{O}(g)$ is the set of functions which in the limit of $x \to \infty$ are bounded by g and we write

$$\mathcal{O}(g) = \{f \mid f(x) \in \mathcal{O}(g(x)) \text{ as } x \to \infty\}.$$

In practice we will often omit "*as* $x \to \infty$". We will also in general write $f = \mathcal{O}(g)$, while technically we should write $f(x) \in \mathcal{O}(g(x))$. The notation $f = \mathcal{O}(g)$ means that $f(x)$ is bounded from above by a constant times $g(x)$ for x large enough. In other words, g is an asymptotic upper bound of f. This upper bound may be very weak as we can see in the next example.

Example 2.38. *Given $f = x^2 + 4$, then we see that $f = \mathcal{O}(x^2)$. However, we also have $f = \mathcal{O}(x^{10})$.*

Related to the definition of $\mathcal{O}$ is that of the *little o*-notation.

Definition 2.39. Given a function f, then $f(x) \in o(g(x))$ as $x \to \infty$ if for every constant $C > 0$ there exists a constant x_0 such that

$$|f(x)| < C\,|g(x)| \qquad \forall x \geq x_0.$$

Then $o(g)$ is the set of function which in the limit of $x \to \infty$ are bounded by g and such that $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$. We write

$$o(g) = \{f \mid f(x) \in o(g(x)) \text{ as } x \to \infty\}.$$

Note that both the $\mathcal{O}$-notation and the o-notation give an upper bound. However, the o-notation gives a better upper bound.

Example 2.40. *For the function $f = x$ we have $x \in \mathcal{O}(x)$, $x \in \mathcal{O}(x^2)$ and $x \in o(x^2)$. However, we also have $x \notin o(x)$. In general we have $f \in \mathcal{O}(f)$, but $f \notin o(f)$.*

Where $\mathcal{O}(g)$ and $o(g)$ give asymptotic upper bounds of f, we can also give asymptotic lower bounds of f.

Definition 2.41. Given a function f, then $f(x) \in \Omega(g(x))$ as $x \to \infty$ if there exist a $C > 0$ and a x_0 such that

$$|f(x)| \geq C\,|g(x)| \qquad \forall x \geq x_0.$$

Now $\Omega(g)$ is the set of functions which in the limit of $x \to \infty$ have g as a lower bound. We write

$$\Omega(g) = \{f \mid f(x) \in \Omega(g(x)) \text{ as } x \to \infty\}.$$

Note that f having g as a lower bound in the limit implies that asymptotically f grows at least as fast as g. Again, we will often write $f = \Omega(g)$, when meaning $f(x) \in \Omega(g(x))$ as $x \to \infty$. As before we had the $\mathcal{O}$-notation and the o-notation. The counterpart of the Ω-notation is the ω-notation.

Definition 2.42. Given a function f, then $f(x) \in \omega(g(x))$ as $x \to \infty$ if for every constant $C > 0$ there exists a constant x_0 such that

$$|f(x)| > C\,|g(x)| \qquad \forall x \geq x_0.$$

Now $\omega(g)$ is the set of functions which in the limit of $x \to \infty$ have g as a lower bound and such that $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$. We write

$$\omega(g) = \{f \mid f(x) \in \omega(g(x)) \text{ as } x \to \infty\}.$$

Again we see that the ω-notation gives a stronger lower bound than Ω. Let us now define when a function is of higher or lower order than another function.

Definition 2.43. A function h is said to be of lower order than a function g if $h = \mathcal{O}(g)$. The function h is of strictly lower order than a function g if $h = o(g)$. A function h is said to be of higher order than a function g if $h = \Omega(g)$. Similarly, h is said to be of strictly higher order than g if $h = \omega(g)$.

Definition 2.44. A function h is said to be much smaller than f ($h \ll f$) if $h \in o(f)$. Similarly, f is much greater than h ($f \gg h$) if $f \in \omega(g)$.

The Ω- and $\mathcal{O}$-notation tell us something about the behaviour of the function near infinity, as we know that one function grows at least/at most as fast as another function. This allows us to define bounds on functions.

Definition 2.45. We say that $f = \mathcal{O}(g)$ is a tight upper-bound if $f \notin \mathcal{O}(h)$ for any h of strictly lower order than g. In a similar way we have $f = \Omega(g)$ is a tight lower-bound if $f \notin \Omega(h)$ for any h of strictly higher order than g.

In general, for a function f we want the bounds to be as tight as possible, as this gives us the most information. Given a function f, then $f(x) = \mathcal{O}(x^2)$ gives us more information about f than, say, $f = \mathcal{O}(x^x)$, while both statements may hold.

The definitions of $\mathcal{O}(g)$ and $\Omega(g)$ are not mutually exclusive. Some functions lie in both $\Omega(g)$ and $\mathcal{O}(g)$.

Definition 2.46. We say $f = \Theta(g)$ if g is both a tight upper-bound and a tight lower-bound of f.

We can reformulate this definition as

Lemma 2.47. *We have:*

$$f = \Theta(g) \Leftrightarrow f \in \Omega(g) \cap \mathcal{O}(g).$$

Proof. $\Rightarrow$: If $f = \Theta(g)$, then $f \in \mathcal{O}(g)$ and $f \in \Omega(g)$, hence $f \in \Omega(g) \cap \mathcal{O}(g)$.

$\Leftarrow$: If $f \in \Omega(g) \cap \mathcal{O}(g)$, then $f \in \Omega(g)$ and $f \in \mathcal{O}(g)$. Therefore there exist $k_1, k_2 \geq 0$ and x_1 and x_2 such that

$$|f(x)| \leq k_1|g(x)| \qquad \forall x \geq x_1$$
$$|f(x)| \geq k_2|g(x)| \qquad \forall x \geq x_2$$

Hence g is both an upper- and a lower-bound of f, hence the the bound must be tight, hence $f = \Theta(g)$. □

Contrary to $\Omega(g)$ and $\mathcal{O}(g)$, the sets $o(g)$ and $\omega(g)$ have an empty intersection, hence they are mutually exclusive.

The above definitions all treat the case where f and g are one-dimensional functions. However, these definitions extend naturally to those where f and g are multidimensional functions. The same analysis as above can also be used to classify algorithms as complex or not complex. This can be done by constructing a function which corresponds to the number of iterations of the algorithm.

2.3.1 Complexity of algorithms

In the above we mainly talked about the complexity of functions. We can use this to determine the complexity of algorithms as well. This time, we use the size of the input of the algorithm to determine its complexity. From there the same analysis can be done as above in order to determine if an algorithm is more complex than another. Note that it is possible that this analysis leads to multidimensional functions, for example functions depending on the sizes of the input and other factors. As already mentioned however, the definitions for one-dimensional functions extends naturally to multidimensional functions.

Example of multidimensional functions can be found in Sections 3.2.5, 3.3.8 and 3.4.6 where we determine the complexity of the algorithms constructed.

Notation 2.48. *We will refer to the input of an algorithm as an instance I.*

When we are considering the complexity of an algorithm, we are most often interested in the complexity in the worst case scenario. That is, we are interested in the maximum number of resources needed to run the algorithm. Most often the resource considered is the running time of the algorithm. However, it may also be the memory space needed to run the algorithm. Some algorithms might for instance be extremely fast, but require too much space for practical purposes.

Definition 2.49. An algorithm is said to have polynomial complexity if its running time is bounded by a polynomial function of the input. An algorithm is said to have exponential complexity if its running time is bounded by an exponential function of the input.

An algorithm is said to scale polynomially with its input size (or just polynomial) if it has polynomial complexity. Similarly, an algorithm is said to scale exponentially with its input size (or just exponential) if it has exponential complexity.

The above definition extends naturally to other complexities such as but not limited to logarithmic, linear, quadratic and factorial.

An algorithm having a worst case exponential complexity does not imply that the algorithm is not suitable for practical purposes. It only means that there are instances I for which the algorithm has exponential complexity. An example of such an algorithm is the simplex method in Linear Programming created by George Dantzig ([16]). The objective of the simplex method is to maximize a linear function given some constraints. For most instances this method works efficiently and over the years the algorithm was improved even further. However, in 1972 Minty and Klee ([17]) showed that for every version of the simplex method as constructed by Dantzig, there is a family of instances for which we have an exponential complexity. Hence, the algorithm has exponential worst case complexity, even though for most practical purposes the complexity is polynomial. Note that exponential complexity is not the worst possible complexity we can get, a factorial complexity, i.e., a complexity which scales with the factorial of the input size, is worse.

2.3.2 Data structures

For the second algorithm we have used an algorithm which solves part of the problem. Data structures are used to simplify calculations. The data structure used is a so called *Binary Search Tree*, or *BST* for short.

Definition 2.50. Given a rooted tree. Given a node v, the *children* of v are all neighbours v_i of v such that v is not a child of v_i. v is then said to be the *parent*.

Note that this definition implies that for every pair of connected nodes, one is the parent and one is the child.

Definition 2.51. A *Binary Search Tree* is a rooted tree, such that each node has a number (or key) associated to it, satisfying:

- Each node has a most two children, usually denoted by *left* and *right*;
- The key of each node is greater than all keys of its left sub-tree and smaller than all keys of its right sub-tree.

The last property is commonly referred to as the *Binary Search Tree Property*.

Let us introduce the definition of the depth of a node and the height of a tree.

Definition 2.52. The depth of a node is the distance between the node and the root. Formally, the depth of the root is zero and the depth of other nodes is 1 plus the depth of its parents.

The height of a tree is given by 1 plus the maximum depth among the nodes.

As we will see later on, the efficiency of Binary Search Trees and algorithms using them, relies heavily on the form of the Binary Search Tree. Let us first define what *leaves* are and then define some special Binary Search Trees.

Definition 2.53. A Binary search tree is said to be

- *proper* if every node has either zero or two children;
- *perfect* if every node has either zero or two children and all leaves are at the same depth;
- *balanced* if it has the height equals $\lceil \log_2 n \rceil$, with n the number of nodes.

Note that it is not possible to create a binary search tree of height smaller than $\lceil \log_2 n \rceil$ where n equals the number of nodes. This height is sometimes also referred to as the minimum height of the binary search tree. Let us now explain the binary search tree by explaining the possible actions. We can insert a new node in the tree, we can delete a node from the tree and we can search for a specific node in the tree.

Remark 2.54. *Note that the above definitions assume the all keys are different, however, the definitions extend naturally to the situation where keys may be equal. In that case the key of each node is greater than all the keys of its left sub-tree and smaller than or equal to all the keys of its right sub-tree.*

Insert a node

If we start with an empty Binary Search Tree, inserting a node is trivial. If we have a non-empty binary search tree, inserting a new node means we have to look at the key of the node. We compare the key of the node we want to insert with the key of the root node. If the first is greater than the second, we end up in the right sub-tree of the root, otherwise we end up in the left sub-tree.

The sub-tree is then treated as being the new tree and we repeat the process. Note that on average half of the nodes is in the left sub-tree and half is in the right sub-tree. Therefore, in each iteration, the number of nodes in the (sub-)tree halves, hence inserting a node takes on average $\mathcal{O}(\log n)$ time, where n is the number of nodes in the original Binary Search Tree. Let us illustrate the process of inserting a new node with a small example.

Example 2.55. *Suppose we have a binary search tree with value of the root node equal to tree (the left binary search tree in Figure 2.3). Suppose now that we want to insert a node with key-value 4. As 4 is greater than 3, we will consider the right sub-tree. The 'new' root node is the one with key-value 5. As 5 is greater than 4, we will consider the left sub-tree. As this sub-tree is empty, we can insert the node with key-value 4 as the left child of node 5.*

Search for a node

Suppose we know that a certain node in the Binary Search Tree has a given key and we want to locate that node. A first approach can be to compare the key of every node

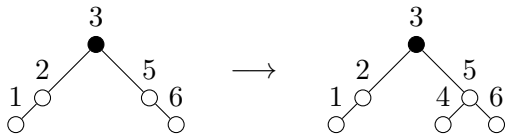

Figure 2.3: Insert a node with key-value 4 in the binary search tree

with the key we search for. However, this takes on average $\mathcal{O}(n)$ time. Instead we can use the the Binary Search Tree Property. Starting from the root node, we compare its key-value with the key we are looking for. If the two keys are equal, we have found the node, otherwise if the key we are looking for is larger than the key of the root node, we continue in the right sub-tree, else we continue in the left sub-tree.

This iteration either stops if a node is found with the corresponding key-value or if the sub-tree considered is empty. In the last case, the key-value does not appear in the Binary Search Tree. Again, the complexity of searching for a node is $\mathcal{O}(\log n)$ on average as in each iteration on average only half of the number of nodes remains to be checked. Let us illustrate this process by a small example.

Example 2.56. *Suppose we have the right binary search tree of Figure 2.3 and we want to know if there is a node with root value 4. Furthermore, if there is such a node we want to know its location. As 4 is greater than 3, we need to consider the right sub-tree rooted at the node with key-value 5. Now 4 is smaller than 5 hence we need to consider the left sub-tree rooted at the node with key-value 4. The value of this 'new' root node equals the value we are searching for. We now only need to determine if there are more nodes with the same value. For this we again will consider the right sub-tree, however this is empty, hence we are done.*

Note that the location of the node we were searching for is given by the path we have taken. This path was one step to the right and then one step to the left. Also note that two different locations will give two different paths and vice versa.

Deleting a node

Given a key-value, deleting the node with that key-value is similar to searching for a node. First we determine where the node is located and then we can delete it. However, now there are two possibilities. The first possibility is that the node had no children, in which case there is no problem and we can simply delete the node. If the node has children however, we cannot simply delete the node as this would give a *hole* in the BST. If the node has only one child, this can be taken care of by replacing the node by its child and then delete it. If the node has two children, we cannot replace the node by its child as it has two children. Instead, we will replace the node by the node whose key-value is the next greatest value (note that this node does exist). To determine the next greatest value, we take the right sub-tree and then go left until there is no left child anymore. That node has the next greatest value after the node we wish to delete. Note that this gives the next greatest value in the BST and that *holes* in the BST cannot occur anymore.

Again in every iteration, the number of nodes that need to be considered halves,

hence, we again have a time-complexity of $\mathcal{O}(\log n)$. Let us illustrate how to delete a node with a small example.

Example 2.57. *Suppose we would like to delete the root node. Locating the root node is easy. Afterwards we have to determine the node with the next greatest key-value. In this case that is the node with key-value 4. We then copy this node and replace the root node by this copy. The original root node is now deleted and we only have to remove one of the nodes with key-value 4.*

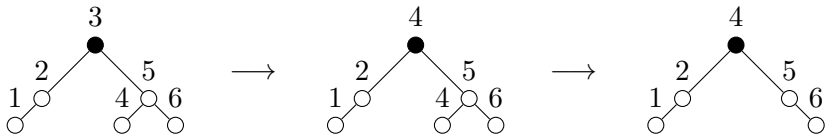

Figure 2.4: Delete a node with key-value 3 in the binary search tree

Drawbacks of Binary Search Trees

As we can see, on average the Binary Search Trees take $\mathcal{O}(\log n)$ time to insert, delete and search for a node. The complexity of many algorithms, including that constructed in Section 3.3.4, rely on the fact that on average these operations can be preformed fast. Note that storing the data in an unsorted list instead of a BST takes on average $\mathcal{O}(n)$ time to search for an entry. Hence, the Binary Search Tree is in general a better option. Using a sorted list instead of an unsorted one is also not an option. This follows as sorting the initial list already takes $\mathcal{O}(n \log n)$ time.

A major drawback of a Binary Search Tree is that the output, and hence the performance, is dependent on the input. Consider for instance the next example.

Example 2.58. *Suppose we want to construct a Binary Search Tree with six nodes, where the key-values are the ordered array* $[1, 2, 3, 4, 5, 6]$*. Inserting the nodes in this order will result in a root-node with key-value* 1*, an empty right sub-tree and a left sub-tree with key-value* 2 *and so on. In general, the Binary Search Tree would look like Figure 2.5.*

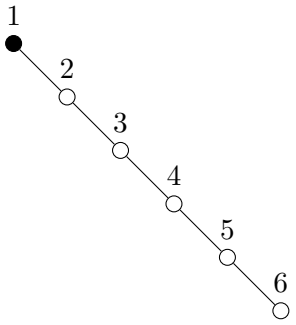

Figure 2.5: An unbalanced Binary Search Tree with height 6 and root node 1.

Such situations can happen with the ordinary Binary Search Trees. However, advanced structures, such as self-balancing Binary Search Trees, have been constructed which take care of such situation. Operations exist to swap two nodes in the Binary Search Tree in order to make it more balanced. These operations are similar to those used to delete nodes. Two balanced Binary Search trees of $[1, 2, 3, 4, 5, 6]$ are given by Figure 2.6.

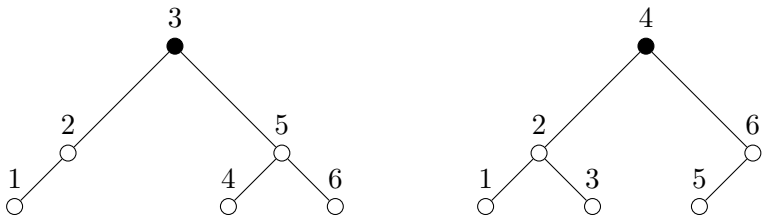

Figure 2.6: Two balanced Binary Search Tree with height 3 and root node 3 and 4 respectively.

Note that this figure also shows that Binary Search Trees are not unique. Using self-balancing Binary Search Trees, we can guarantee that inserting, deleting and search for a node all can be done in $\mathcal{O}(\log n)$ time, even in the worst case scenario.

More advanced data structures have been constructed starting from the binary search tree. An example is of course the self-balancing binary search tree, but also the Kd-tree (a multidimensional binary search tree) and the red-black tree (a binary search tree, where every node has an extra colour and there are restrictions on the colours of parents and children) find their origin in the binary search tree. Binary search trees can also be generalized such that a node may have more than two children. Setting restrictions on these generalizations will result in different data structures such as k-ary trees or B-trees. More on binary search trees and other data structures can be found in [18] and [19].

2.4 Miscellaneous proofs

In this section we will prove some theorems which the algorithms will use. The theorems apply to a wider range of line segment-problems.

In Section 3.2 we will compute the minimal distance between a line segment and a point. Let us consider a different, but similar case where we want to compute the distance between a line and a point. The line in our case will the line through two distinct points $x = (x_1, x_2)$ and $y = (y_1, y_2)$. The point has coordinates (z_1, z_2). See also Figure 2.7 for an illustration of the situation.

The distance between the point z and the line with the distinct points x and y is given by

$$d(xy, z) = \frac{|(y_1 - x_1)(x_2 - z_2) - (y_2 - x_2)(x_1 - z_1)|}{\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}} \tag{2.8}$$

Theorem 2.59. *Equation* (2.8) *gives the distance between the line and the point.*

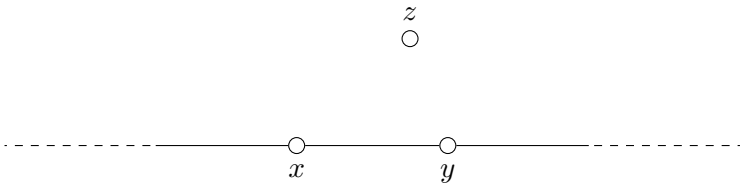

Figure 2.7: The situation where we want to compute the distance between a line and a point.

Proof. Given a line and two distinct points on the line $x = (x_1, x_2)$ and $y = (y_1, y_2)$. Let $z = (z_1, z_2)$ be the point for which we want to compute the distance to the line. We now want to calculate $d(xy, z)$.

Let us now calculate the area of the parallelogram spanned by the vectors $z - x$ and $y - x$. This is given by

$$\begin{aligned} A &= |(z - x) \times (y - x)| \\ &= |(z_1 - x_1)(y_2 - x_2) - (z_2 - x_2)(y_1 - x_1)| \end{aligned}$$

Note that this expression is the magnitude of the third component of the cross product, when x,y and z are considered as three dimensional vectors. Also note that A equals twice the area of the triangle given by the vectors $z - x$, $y - x$ and $z - y$. Recall that dividing the area of a triangle by its base, will give half its height. This then gives the distance from z to the line segment, which leaves us with

$$d(xy, z) = \frac{|(z_1 - x_1)(y_2 - x_2) - (z_2 - x_2)(y_1 - x_1)|}{\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}}. \qquad \square$$

In line of the above theorem, we can consider another problem. This time we want to connect two points, one is on a line, while the other is not on the line. See also Figure 2.8. The connection between the two points must be a least cost-connection. The cost of the connection can be determined by considering the cost per distance c_{line} to traverse the line xk and the cost per distance c_{dig} to connect z with the line xk. Note that the problem is trivial if $c_{dig} \leq c_{line}$, as the cheapest path is then also the shortest path.

There are many versions of this problem, for instance, given a river and two points on opposite sides of the river. Find the path between the two points which takes the least time. The two costs in this case typically are the time spend per distance walking over land and the time spend per distance crossing the river. To the public this problem became most known due to an article by T.J. Pennings ([20]) where he asks the question *Do dogs know calculus?*. This problem and generalizations of it were studied more thoroughly in [21] and [22].

Let us consider the situation as shown in Figure 2.8. The cost per distance c_{line} is related to the distance traversed over segment xk, while c_{line} is related to the distance traversed between z and the segment. Basic calculus then shows that the cost C related to this situation is

$$C = c_{line} \cdot (q - f) + c_{dig} \cdot \sqrt{f^2 + d^2}.$$

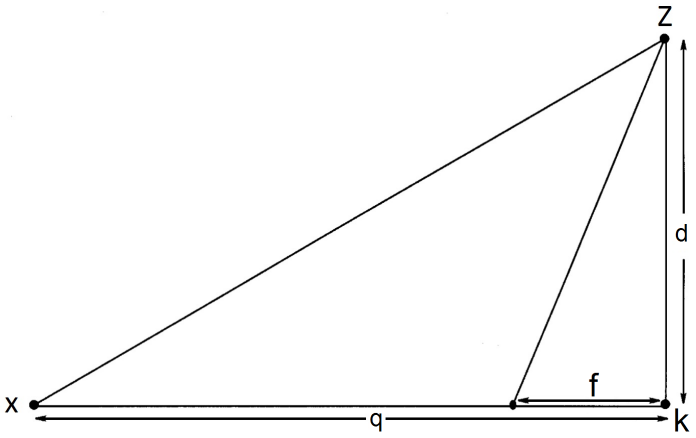

Figure 2.8: Find the least cost-path between z and x.

Theorem 2.60. *Given the above situation with costs $c_{dig} > c_{line} \geq 0$ and with $d > 0$. The minimal cost/least time path is obtained for $f = \frac{c_{line}d}{\sqrt{c_{dig}^2 - c_{line}^2}}$.*

Note that $C = C(f)$, i.e., the cost is a function of f and the other variables are kept constant. Also note that the situation where $d = 0$ is trivial and hence will not be considered.

Proof. The minimal cost-/least time-path is obtained if C is minimal and then we have

$$\frac{\mathrm{d}C}{\mathrm{d}f} = 0.$$

Computing the derivative, we find

$$\frac{\mathrm{d}C}{\mathrm{d}f} = -c_{line} + \frac{c_{dig}f}{\sqrt{f^2 + d^2}}.$$

If we then solve this, we get

$$\begin{aligned} -c_{line} + \frac{c_{dig}f}{\sqrt{f^2 + d^2}} &= 0 \\ \frac{c_{dig}^2 f^2}{f^2 + d^2} &= c_{line}^2 \\ c_{dig}^2 f^2 &= c_{line}^2 f^2 + c_{line}^2 d^2 \\ f_{min} &= \pm \frac{c_{line}d}{\sqrt{c_{dig}^2 - c_{line}^2}} \end{aligned}$$

Note that this works as $c_{dig} > c_{line} > 0$, hence $c_{dig}^2 > c_{line}^2 > 0$. Also note that the negative solution is of no interest to us, hence $f_{min} = \frac{c_{line}d}{\sqrt{c_{dig}^2 - c_{line}^2}}$. We now only need to

show that $C''(f_{min}) > 0$ as in that case we indeed have a minimum.

$$\begin{aligned}\frac{\mathrm{d}^2 C}{\mathrm{d}f^2} &= \frac{\mathrm{d}}{\mathrm{d}f}\left(-c_{line} + \frac{c_{dig} f}{\sqrt{f^2+d^2}}\right) \\ &= 0 + \frac{c_{dig}\sqrt{f^2+d^2} - c_{dig} f \frac{f}{\sqrt{f^2+d^2}}}{f^2+d^2} \\ &= \frac{c_{dig}(f^2+d^2) - c_{dig} f^2}{(f^2+d^2)^{3/2}} \\ &= c_{dig}\frac{d^2}{(f^2+d^2)^{3/2}}\end{aligned}$$

Now note that this last expression is always greater than zero as $d > 0$. Hence, the function is convex and hence the minimum is attained at $f_{min} = \frac{c_{line} d}{\sqrt{c_{dig}^2 - c_{line}^2}}$ as desired. □

Note that the result is independent of our choice of f over $q - f$ in the expression of C. Also note that the minimum only depends on the costs and the distance d between the point and the line.

Corollary 2.61. *If $c_{line} \to 0$, then $f \to 0$ and $C \to c_{line} q + c_{dig} d$. This means, if traversing over the line xk is cheap, use the line xk as much as possible.*

If $c_{dig} \gg c_{line}$, $f \approx \frac{c_{line} d}{c_{dig}} \to 0$ and $C \to q c_{line} + d c_{dig}$.

If $c_{dig} \approx c_{line}$, f becomes large and $C \approx \sqrt{c_{line}^2 q^2 + c_{dig}^2 d^2}$.

If $c_{dig} \leq c_{line}$ there is no solution, which is reasonable as in those cases we do not want to use the line xk, but we want to make a direct connection between the points z and x.

Note that in the above we suggestively chose the name c_{dig}. However, this variable can also represent the time per distance it costs to cross a river for instance. The suggestive name follows in the third algorithm where we can apply the above theorem and the corollary.

Let us now, instead of a line and a point, consider two line segments. In Section 3.3 we will use some properties of line segments which we will now formulate and prove. The first theorem we will prove tells us if two line segments intersect. Given four points A, B, C and D in the plane, let us now consider the triples (A, B, C), (A, B, D), (C, D, A) and (C, D, B). We will consider the line segments AB and CD and we want to determine if they intersect. We want this intersection to be proper, i.e., we want the segments to intersect other than in their endpoints.

Definition 2.62. Given two intersecting line segments. The intersection is said to be *proper* if the line segments share precisely one point and no three endpoints are collinear.

An intersection point which is not proper is said to be *improper*.

If the line segments properly intersect, we can divide the plane in two half-planes using the line AB. Then C and D will be in different half-planes. As a result, the orientation of the points (A, B, C) will be clockwise (counterclockwise), while the orientation

of (A, B, D) will be counterclockwise (clockwise). Note that a similar analysis holds for the other two triples. The orientation can be determined by checking if the points are traversed clockwise or counterclockwise in the plane.

Let us now think of the points as being three dimensional points with the third component being zero. We now start with the triple (A, B, C) we can determine the orientation of this triple using the sign of the cross product $(A - B) \times (B - C)$. Note that the usage of the cross product requires us to think about the points as being points in the three dimensional space. Also note that due to the fact that the third component is zero, in the resulting vector, only the third component is non-zero. As we have seen above, the orientation of (A, B, C) and (A, B, D) must have a different sign. However, the orientations of these two triples being different is not sufficient for the line segments to intersect. See for example Figure 2.9. In the left situation the orientation of (A, C, D) and (B, C, D) is different, while the segments do not intersect. Therefore we also need to have that the signs of the orientations corresponding to the other two triples are different.

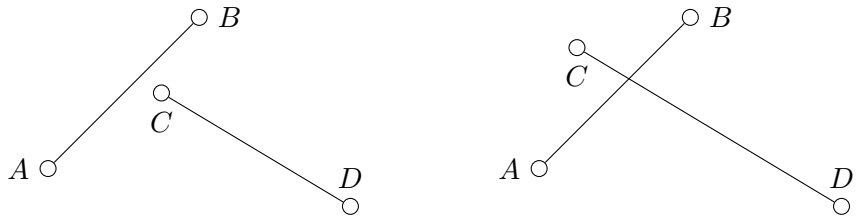

Figure 2.9: On the left two non-intersecting line segments. On the right two intersecting line segments.

Note that saying that the signs of the orientations corresponding to (A, B, C) and (A, B, D) must be different, is equal to saying that $((A - B) \times (B - C))_3 * ((A - B) \times (B - D))_3 < 0$. Note that we have used that only the third component is non-zero. Let us now give the theorem.

Theorem 2.63. *Given two line segments with endpoints A, B and C, D. The line segments intersect if and only if both*

$$((A - B) \times (B - C))_3 * ((A - B) \times (B - D))_3 < 0$$

$$((C - D) \times (D - A))_3 * ((C - D) \times (D - B))_3 < 0.$$

Here $((A - B) \times (B - C))_3$ is the third component of the cross product of the vectors $A - B$ and $B - C$.

The proof of this theorem is a combination of the observations made above. The orientation is then defined as the sign of the third component of the outer product.

Note that we have excluded the case where one of the two cross products is zero. This happens if the three of the four points are collinear, hence if we have an improper intersection.

We can now determine if two line segments intersect. However, if we know that they intersect, we would like to determine their intersection point. In order to do this, we

will consider two line segments, AB and CD, and we represent these line segments as $A + \lambda(B - A)$ and $C + \mu(D - C)$, where $\lambda, \mu \in [0, 1]$. Note that we still think of the points as being in the three dimensional space with the third component being zero. The intersection point between AB and CD is then found at $A + \lambda(B - A)$ with

$$\lambda = \frac{|(C - A) \times (D - C)|}{|(B - A) \times (D - C)|}, \tag{2.9}$$

where $|a \times b|$ is the magnitude of the cross product of vector a with vector b.

Theorem 2.64. *The intersection point of $A+\lambda(B-A)$ and $C+\mu(D-C)$ is $A+\lambda(B-A)$ with λ as in Equation (2.9).*

Proof. If the two line segments intersect, we have $A + \lambda(B - A) = C + \mu(D - C)$. Let us now take the cross product with $(D - C)$ on both sides, then we get

$$(A + \lambda(B - A)) \times (D - C) = C \times (D - C),$$

which we can rewrite as

$$\lambda(B - A) \times (D - C) = (C - A) \times (D - C).$$

Note that we have used that $(D - C) \times (D - C) = 0$. Then we also see that the length on both sides is equal

$$|\lambda(B - A) \times (D - C)| = |(C - A) \times (D - C)|.$$

If we now solve this for λ we get

$$\lambda = \frac{|(C - A) \times (D - C)|}{|(B - A) \times (D - C)|}.$$

Hence, the intersection points is $x + \lambda(y - x)$, with λ as in Equation (2.9). □

Corollary 2.65. *Taking a similar approach and solving for μ we see that the segments intersect at $C + \mu(D - C)$ with*

$$\mu = \frac{|(C - A) \times (B - A)|}{|(B - A) \times (D - C)|}.$$

Chapter 3

The algorithms

The general problem of connecting active points to the glass fiber network, can be subdivided in three smaller problems. The first is finding the shortest connection to a network, the second is constructing a single network with no self-intersections from two intersecting networks and the third problem is finding the least cost-connection between the cabinets and the fiber glass network. The strategy for connecting the cabinets is to first connect them with *a* network (e.g. a street or trench network), then route through the network and finally connect to the fiber glass network. The idea behind this is that it is in general cheaper to use already existing networks, such as street networks or trench patterns, than it is to construct new paths/trenches. Hence, we use other networks two connect the cabinets with the fiber glass network. Before we consider the algorithms however, we will first consider the data sets.

3.1 The data

Recall that the algorithms revolve around the active points and the networks. The active points are given by their coordinates. For the network, this is slightly more complicated, as we have line segments instead of points. Therefore it is hard to give all coordinates of a network. Instead we will think of the network as being a collection of connections (streets in the case of a street pattern). Each connection in their turn is a connected collection of line segments and these line segments will be denoted by their endpoints. This is both easier to denote, but also easier to work with. Of course taking the union of all connections will result in the network again. In most cases the network we will consider is either a street pattern or a trench or duct pattern. The connections then are either streets or trenches.

The coordinates of all connections and all active points are given in RD coordinates. The RD coordinate system (standing for *rijksdriehoekscoördinaten*) is a system used in the Netherlands. This system is a Cartesian system designed in such a way that the x- and the y-coordinates always have a different value. The x-values range from 0 to 300,000 meters, while the y-values range from 300,000 meters to 620,000 meters. The values are chosen in such a way that no confusion can arise whether the x- or the y-value is meant. The usage of RD coordinates allows us to treat the system as being on the 2D-plane. If we were to use the longitude and latitude of each point, we need more

complicated formulas to calculate distances between two such points. This will result in vague and obscure algorithms.

We use a cell array to encode the different connections we have, see also Figure 3.2. A cell array is an array where every entry can be a data structure on its own. In our case, every entry, hence every cell, contains a matrix and this matrix contains the information about the actual connections. Note that most of the time connections are not straight, but there is some curvature in it. We can fix this problem if we subdivide each connection in multiple smaller, but piecewise linear line segments. An example of this can be seen in Figure 3.1. The swirling street on the left is subdivided in smaller piecewise linear parts on the right, while still remaining its general form.

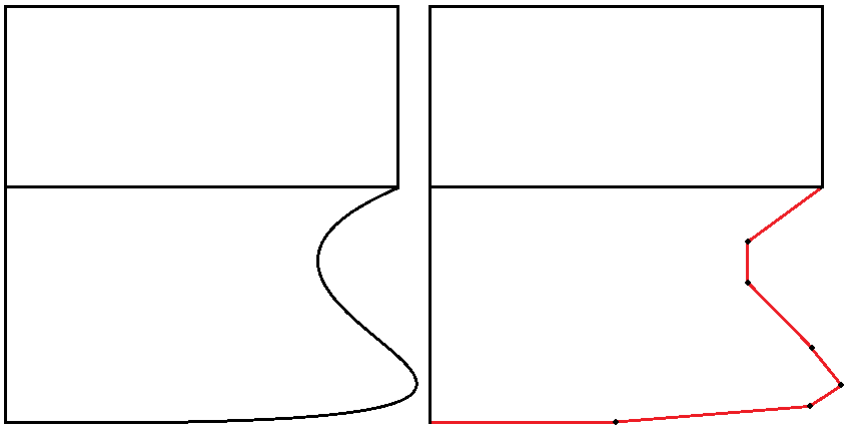

Figure 3.1: On the left the actual connection, on the right a linear line segment approximation.

As already mentioned, each entry in the cell array denotes a connection. This is done using a matrix which contains the information about the connection. Let us now take a closer look at the matrices in the cell array. We will do this by looking at the 25-th entry in the array of a given data set. The matrix can be found in Figure 3.2. Note that the matrix consist of six rows. Each adjacent pair of rows denotes a piecewise linear part, thus the first two rows denote the first line segment. Row two and three denote the second line segment, and so forth. Note that this approach does work as every line segment is determined if two points are given. The points are the rows of the matrix and the line segments are then formed by combining two rows. Also note that the endpoint of the first line segment is the starting point of the second line segment. This follows as together they make up the actual connection.

Thus summarizing we have seen that the data of the network is encoded using a cell array. Each entry of this array contains a matrix. The rows of this matrix denote points and hence two rows together form a piecewise linear part of the total connection. Thus when we talk about the connection we mean the collection of piecewise linear line segments, denoted using the matrix.

Note that the active points are not listed in this array. However, they are given by a matrix similar as those in Figure 3.2, but then with each row representing the coordinates of an active point.

Note that in the array shown the $2i-1$-th entry and the $2i$-th entry have the same

21	9x2 double
22	9x2 double
23	[1.9344e+05...
24	[1.9345e+05...
25	6x2 double
26	6x2 double
27	[1.9360e+05...
28	[1.9345e+05...
29	7x2 double

6x2 double

	1	2
1	1.934518860312130e+05	3.925909317299810e+05
2	1.934733050509616e+05	3.925896092664546e+05
3	1.934904411237013e+05	3.925884399589326e+05
4	1.935187705067981e+05	3.925841994247924e+05
5	1.935762094601049e+05	3.925710827518333e+05
6	1.935955794808388e+05	3.925699308435546e+05

Figure 3.2: On the left: a part of a cell array containing connections. On the right: the matrix corresponding to the 25-th entry of the array.

size. This follows from the way we set up our data. Namely, every connection is listed twice in the array, once from point A to point B and once from point B to point A, hence explaining this form. Also note that the cell array shown is a column-array. It is also possible to have a row-array, however this is just the transposed version of a column-array. Another possibility is for the cell array to have a square form. In this case we see that position (i, j) denotes the connection between node i and node j, while position (j, i) denotes the connections between node j and node i.

If we have a column/row-array, it is sufficient to only check for the even or odd entries. This follows as every connection appears twice. For the square-array this amounts to only checking for the upper- or lower-triangular part of the matrix. We chose to only consider the lower part in this case as MatLab is column-major order [23], meaning that the data is stored column-wise. We believe that this might positively influence the running time, however, the gains will most likely be insignificant.

For the algorithms we do not need an adjacency matrix, as we are only interested in the connections between the points. Do note however that the square-array also serves as an adjacency matrix. The empty entries serving as zeros, while the non-empty entries serving as ones.

We favor the column-arrays as they need the least amount of storage space. Consider the situation of n points and m entries in the array. The space needed to store the square-array is $\mathcal{O}(n^2)$, while the space needed for the column/row-array is $\mathcal{O}(m)$. This last is favoured as $m \ll n^2$. The gain in storage usage follows from the fact that for the square-array space has to be allocated for all positions in the array, independent on whether they are empty or not. In the case of the column/row-array, there are no empty entries.

3.1.1 Flaws in the data

We have analyzed the data and found that there were some flaws which might slow down the algorithm or give wrong answers. Therefore let us treat some of the flaws we have encountered.

If we have a squared array of connections, there were some positions with a non-empty (i, i)-position, meaning that we have a street going from one point to the exact

same point. This is only possible if the street is a loop. There have been cases where this was in fact a loop, however we have also found situations where this was not the case. We will not neglect these entries as this would lead to many checks, with only small benefits. Doing the calculations also for these points will most probably be more efficient. These entries will not affect the results the algorithm gives as they do not contribute to the algorithm.

Apart from the locations of the connections, also the locations of the active points are given in the array. Suppose we have N active points. In the case of a square-array, these active points are embedded in the last N rows and columns. However, they only occupy $2N$ entries, while other entries of these rows and columns are empty. These points appear in the array as a 2×2-matrix, where both rows are identical. Due to tests in the algorithms, these points will not negatively influence the results. We can neglect these points. If we do not have a square array, we might not know which entries correspond to the last N rows or columns. This is because they are not simply added at the end of the array, but they appear between the other entries. However, as already mentioned, these points do not negatively influence the algorithm. Meaning that we do not need to remove these points from the array. Hence, even though these points do increase the number of calculations we have to do, they do not negatively influence the results. It is also more beneficial to just do the calculations for these points, than to locate them and then skip these points.

We have also found some errors in the coordinates of the active points we wish to connect. These coordinates come in two lists, one for the x- and one for the y-coordinate. Let us consider these list for the situation we have the technique G.Fast for the Dutch city Den Haag. The lists then consists of 2264 points, but a large part of the points is equal to zero. These points can be neglected, which leaves us with only 1764 entries. Considering the last ten entries of both lists gives

81386	454256
81386	454256
81388	454172
81393	455410
81393	455410
81393	455441
81393	455441
81416	455595
81416	455595
78262	455199

As it turns out of those ten entries, four appear twice. Neglecting the double appearing entries, leaves us with six of the ten entries. Doing this for all the 1764 entries, will lead to only 1462 active points to consider. Thus 1462 active points we wish to connect with the network in that case. An attempt to fix this last flaw was made in [24]. An explanation for this problem can be found in the way the data is obtained. As we are considering G.Fast, we know that the active points can not be too far apart, as G.Fast is only fast on small scales. The coordinates however are partially determined using zip codes. The problem then arises if we have two active points with a house in between them and the house is connected with both points. If the active points have the same

zip code as the house, they will get the same coordinates. However, they will get the coordinates of the house, instead of the coordinates of one of the points.

In the second algorithm we have also considered data sets drawn from trench patterns. However, the coordinates corresponding to the connections of the trench pattern were all integers (e.g. $(193, 201; 393, 732)$). As a results it is more likely that multiple line segments start in one point. Another thing we saw in the data set was (partially) overlapping segments. We found the following three line segments, all appearing in different connections:

$$(194853, 391729) \text{ to } (194854, 391729)$$
$$(194852, 391729) \text{ to } (194854, 391729)$$
$$(194852, 391730) \text{ to } (194853, 391730).$$

As we can see, the first two line segments overlap, while the third line segments lies one meter higher than the first two. This last observation is possible in practice, however the first observation is unwanted. A possible explanation for this situation can be that multiple pipe lines lie on the same position, after which they branch towards the end users.

3.1.2 The data sets considered

We have considered multiple different data sets in our analysis. Some were small, others were rather large. The data sets either originate from a street pattern or from a trench and duct pattern. The setup for both data sets is different. This is something we have to take into account even though it does not affect the results of our algorithms. The data sets generated from street patterns are set up such that two connections can only intersect in their endpoints. As a result, more connections are needed, but these connections are, in general, relatively short. The fact that this restriction does not need to hold for the data sets generated from trench and duct patterns, is part of the second algorithm we will consider later on.

Recall that the second problem revolved around finding where two networks intersect. If a data set does not have the above restriction, it is possible for two connections from the same data set to intersect. As a result we will also have to consider possible self-intersections in the second algorithm.

We have considered multiple data sets. For the first algorithm the largest data set considered was the street pattern of the Dutch city Den Haag. It has nearly 40,000 connections and 570 active points. The smallest data set considered for this algorithm was that of the city Venray which had 3,365 connections and 55 active points. Apart from that we also generated some large random data sets.

For the second algorithm we have used multiple data sets. The smallest considered was again that of the city Venray. The corresponding street pattern had 2,212 connections and the trench pattern had 7,589 connections. Some larger data sets were also considered, including that of Amsterdam and its surroundings villages with a total of more than 250,000 connections in the street and trench pattern combined. Also the southwestern part of the Netherlands was considered, this data set had 250,000 streets and over 500,000 trenches.

For the third algorithm we considered data sets generated by the second algorithm. For instance, for Venray we have used a data set consisting of 23,682 connections, while the data set for the city Tilburg consisted of 96,851 connections. Note that these data sets contain far more connections than data sets considered in the second algorithm combined. This is of course due to the fact that two intersecting connections result in four non-intersecting connections. We will also consider some random graphs.

3.2 Algorithm 1

The first algorithm finds the shortest distances between a set of points and a graph. Note that this graph is given by a set of fixed points with edges connecting them. We have also considered some heuristics which speed up the process, at the expense of less accurate results.

The algorithm will, after termination, give the shortest distances between a set of points we wish to connect and the network. It will also tell us where these shortest distances are attained. For the input, the algorithm needs an array of connections between the nodes and the coordinates of the points we wish to connect. Recall that the array contained the connections, as each entry was a matrix containing the endpoints of the piecewise linear line segments, which together form the connection. In Algorithm 3.3 the pseudocode of the first algorithm is given.

Algorithm 3.3 Find the minimal distances between points and a network

Input: Array of connections, coordinates of points to connect
Output: For every point to connect, the coordinates of connection point with minimal distance and the corresponding distance

```
1: for all non-empty entries in the array do
2:     for all line segments of the connection do
3:         Check case and compute distance
4:         if an improvement has been found then
5:             Update the optima
6:         end
7:     end
8: end
```

Let us explain the algorithm in more detail. Before we can do any calculation, we need to check which entries in the array are of interest for us. Entries which are empty, or which appear double in the array may be neglected. Note that every connection can be found twice in the array. Once from point A to point B and once from point B to point A. Thus we only need to check one of these entries, let us refer to these entries as the entries of interest. If the array has a square form, the entries of interest are the entries in the upper- or lower-triangle of the array, as the array is symmetric. If the array has the form of a column or a row, taking the even (or the odd) entries takes care of the double entries. Note that this all holds given our setup.

After this we will loop through the entries of interest and do the calculations for those. As each entry is a matrix denoting the endpoints of the line segments, this amounts

to checking for each line segment what the distance is to the active points. When calculating these distances, we can distinguish three cases. Two of which are similar. After a distance is found for an active point, we check if for this active point the distance found is smaller than the distances found for other line segments considered before. If so, we update a matrix with the optimal distances and the indices corresponding to the line segment at which the shortest distance was attained. If the distance is larger, the algorithm will continue. After all entries of interest are considered, we calculate the exact positions where the minimum distances between the active points and the line segments (and hence the network) are attained. These will then be given as output.

Note that the exactness of the algorithm is restricted to the exactness of the computer used to run the algorithm. If the computer can only give results in three decimal places, the algorithm cannot give results in four or five decimal places. Note however, that most computers have a fairly high accuracy, therefore this will not be a problem in all practical cases.

3.2.1 The three cases

The main part of the first algorithm is the distinction between three different cases. These cases arise from the different possibilities of the position of the point relative to the line segment. With the position we mean the position when the point is projected on the extended line segment. We do not yet care for the distance between the line segment and the point. In Figure 3.4 we see the three possible cases, z is the active point to connect and x and y are the endpoints of the line segment we are considering. In the first case the projection of the active point is to the left of the line segment, in the second case to the right and in the third and last case the projection of the active point is on the line segment. From left to right the cases in the figure are case one, case two and case three.

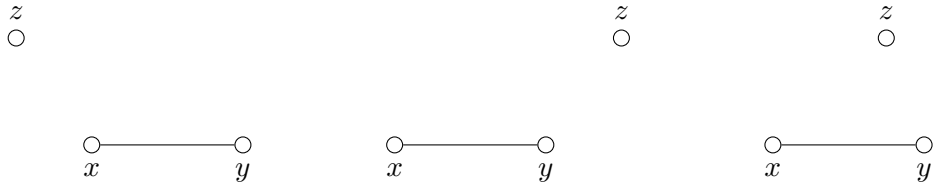

Figure 3.4: The three possible cases.

We use an inner product to determine for every pair of a point and a line segment in which case they belong. We are in Case 1 if $(z-x)\cdot(y-x) < 0$, we are in Case 2 if $(z-y)\cdot(x-y) < 0$ and we are in Case 3 if we are not in Case 1 or Case 2. Note that we are in the third case if $(z-x)\cdot(y-x) \geq 0$ and $(z-y)\cdot(x-y) \geq 0$. Also note that $(z-x)\cdot(y-x) < 0 \Rightarrow (z-y)\cdot(x-y) > 0$.

As there is no efficient way of checking whether a point is in case 1, 2 or 3, we will check for all points if they are in Case 1. For those which are not, we will check if they are in Case 2. The points which are also not in Case 2, must be in Case 3 as that is the only possibility left. Let us now treat each case in more detail.

Case 1

In Figure 3.5 we have the first case. Note that the projection of z is to the left of x if and only if

$$(z - x) \cdot (y - x) < 0.$$

For each line segment we will calculate this inner product. If this inner product happens to be larger than zero, the point is not in Case 1, hence no further calculations are done at the moment. However, if this inner product happens to be smaller than zero, we are in Case 1 and we can calculate the distance.

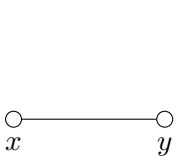

Figure 3.5: Case 1.

If z is projected on the left side of x, we see that the shortest distance between z and the line segment is equal to the distance between z and x. Hence, we can calculate the Euclidean distance between z and x. After the distance is calculated, we check if the distance found is smaller than the best distance up until then for that active point. If that is true, we update this distance and we save the indices corresponding to the line segment for further calculations.

Case 2

The second case is similar to the first case. However, this time we check if the projection of the active point is on the other side of the line segment, i.e., we check if the projection of the active point is to the right of y. In order to do this we check if

$$(z - y) \cdot (x - y) < 0$$

holds. If that is not the case, the active point has to be in Case 3. However, if the inner product is smaller than zero, we will not necessarily continue our calculations as there are multiple situations possible. Recall that each connection between two points was made up of piecewise linear parts. This was to take care of the curvature that the connection might have. We will only compute the distance if we are considering the last piecewise linear part of the connection. The distance is then again calculated as the Euclidean distance between z and y. If we are not in the last piecewise linear part of the connection, there are multiple situations possible as shown in Figure 3.6. In all those situations, we do not have to calculate the distance. Note that w is the endpoint of the next piecewise linear part of the connection.

Let us explain why we do not have to calculate the distance if we are in one of the above three scenarios. In the first scenario we see that the shortest distance is to the right endpoint y, but when projected on the piecewise linear part yw it is to the left of y. We see that in the next iteration of the loop, when the part yw is considered, the

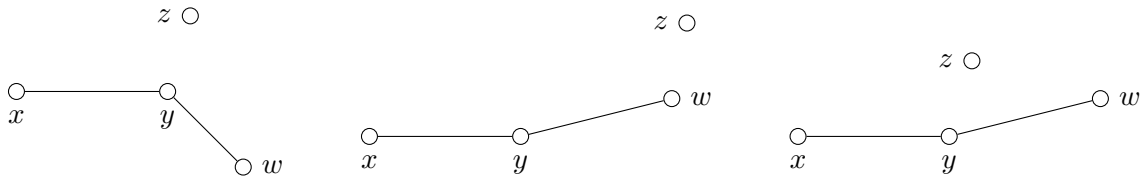

Figure 3.6: The three scenarios for case 2.

point z is in Case 1. In this case the distance is calculated. Thus we see that we do not have to calculate it this time. In the second scenario, we see that the active point z is to the right of y, but that it is also to the right of w with respect to the part yw. Hence, in the next iteration of the loop the active point z will be in Case 2. Thus the distance will be calculated if yw is the last piecewise linear part of the connection or we again have three scenarios. Thus once again we do not have to calculate the distance. The last scenario we can have is that z is to the right of y, but such that it is projected on the line segment yw. This time the point z will be in the Case 3 in the next iteration of the loop. Hence, the distance is then calculated.

Note that we cannot make this distinction in Case 1 as well, this is due to the first scenario. The distance between the line segment and the point has to be calculated. The three scenarios are only used to make sure that we do not calculate the same distance multiple times.

So, we will only calculate the distance between the active point and the right endpoint of the part, if it is the last piecewise linear part of the connection. Otherwise, the distance is calculated in another iteration of the loop. Again we check if the distance found for an active point is smaller than the distances found for the other line segments already considered. If that is the case, we update the best distance for that active point and we save the indices corresponding to the line segment. Otherwise we will continue with our algorithm.

Case 3

The active points that are not in Case 1 or Case 2, must be in Case 3. In Figure 3.7 we can see the third case. The active point z is projected onto the line segment xy. In terms of the inner products this means that both $(z - x) \cdot (y - x)$ and $(z - y) \cdot (x - y)$ are greater than or equal to zero.

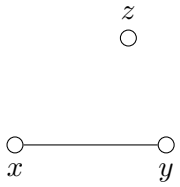

Figure 3.7: Case 3.

For the points in Case 3, we see that the projection of the point is on the line segment. We may then apply Theorem 2.59 from Section 2.4. However, in this case we do not have a line, but we have a line segment. The two points on the line, x and y, are now the endpoints of the line segment. Note that due to the fact that the projection of z is on the segment xy, we may extend the line segment xy to the line through the points x and y. Hence, we may apply the theorem. Note that x and y are assumed to be distinct. In the algorithm we will use an extra check to make sure this is the case. The distance is then given by

$$d(xy, z) = \frac{|(y_1 - x_1)(x_2 - z_2) - (y_2 - x_2)(x_1 - z_1)|}{\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}}$$

Note that we can rewrite this equation as

$$d(xy, z) = \frac{|(y - x) \cdot q + y_1 x_2 - y_2 x_1|}{\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}} \tag{3.1}$$

with $q = (-z_2, z_1)$. Again, for the indices which give an improved result, the distance and the corresponding indices are stored for later use.

3.2.2 Generating results

After we have found the shortest distances with the coordinates of the corresponding line segments, it is time to generate the results. For the points where the minimal distance was obtained in Case 3, we will also give the exact coordinates of the points where the minima was attained. For the other cases, the minima are attained at an endpoint, hence their locations are already known. To calculate the coordinates where the minima are attained in Case 3, we again think of the line segments as being spanned by their endpoints. So if the endpoints are x and y, we consider the line segment $x + \lambda(y - x)$, with $\lambda \in [0, 1]$. The minimum is attained at

$$\lambda = \frac{(z - x) \cdot (y - x)}{(y - x) \cdot (y - x)} \tag{3.2}$$

Theorem 3.1. *The minimum distance from a point z to the line segment xy is attained at $x + \lambda(y - x)$ with λ as in Equation (3.2).*

Proof. Let the line segment be given by $x + \lambda(y - x)$, $\lambda \in [0, 1]$, and let p be the point where the minimum is attained. Then we want

$$(z - p) \cdot (y - x) = 0,$$

as the shortest distance is attained when the two lines are perpendicular. Thus we want

$$(z - x - \lambda(y - x)) \cdot (y - x) = 0.$$

Solving this for λ gives

$$\lambda = \frac{(z_1 - x_1)(y_1 - x_1) + (z_2 - x_2)(y_2 - x_2)}{(y_1 - x_1)^2 + (y_2 - x_2)^2},$$

from which Equation (3.2) can be derived. □

Note that we cannot say $\lambda = \frac{z-x}{y-x}$, as this implies a division of vectors, which is not defined.

3.2.3 Speeding up the algorithm

We have used a few tricks in order to improve the running time of the algorithm. The first one is the distinction between the different cases as mentioned above. We have also used a more optimal way of finding the indices of the non-empty entries in the array. At first we checked for every entry if it was empty or not, while now we use a function which gives the corresponding indices after which we will only use the indices of interest. What has benefited the algorithm the most was the vectorization of some *for*-loops. With vectorization we mean doing calculations using a vector instead of doing the calculations for every entry separately. Let us illustrate this with a simple example.

Example 3.2. *Consider the vector* $x = (1, 2, 3, \ldots, 10,000,000)$ *and suppose we want to take the square of each entry. We can either do this one entry at a time using a* for*-loop, or we can vectorize the calculation and multiply the vector element-wise with itself.*

The first takes roughly 1.6 seconds, while the vectorized version takes only 0.06 seconds. This is a significant difference.

At first we calculated the inner products for each active point individually and then check for the possible cases. Vectorizing this loop resulted in a significant decrease in the running time. Now the inner products are calculated at once for all active points, while also the cases are checked for all active points at the same time.

We have considered further vectorizing the algorithm, but this turned out to not give an improvement. We tested the vectorization with an approximation algorithm, as this was the easiest to vectorize. We vectorized the *for*-loop which checked each entry of the list of the non-empty entries of the array. The distances for all these entries are now calculated all at once. The results for this vectorization, however, turned out to be worse than the non-vectorized algorithm. We will discuss this further in Section 4.1.1. The *for*-loop we vectorized was embedded in another *for*-loop. As it turned out that the running times increased, we will not try to further vectorize the algorithm as this will not yield better results. As the running times of this vectorized approximation algorithm did not improve, the running times of the algorithm will also not improve. This follows from the fact that this algorithm is more complex and therefore even harder to vectorize.

In order for the vectorized approximation algorithm to work, we have used a sorting function and a function which gave the unique values in a vector. These took relatively long, which explains the difference with the original algorithm. We cannot construct the vectorized version without these functions, as the algorithm uses multidimensional matrices with results and only a few of the entries are of interest to us. The indices of those few results, are found using these functions.

3.2.4 Heuristics

Apart from a better algorithm and better hardware, heuristics are also a way of improving the running time of the algorithm. This, however, comes at the cost of less accurate results. We have considered two heuristic methods and we have introduced an *acceptation distance*. The heuristics are a *node-approximation* and a *cut-approximation*. Let us discuss these heuristics in more detail. In Section 4.1.2 we will compare the results of

the algorithm with those of the heuristic methods. Let us now explain which heuristics we have and how they work.

Roughly speaking the node-approximation and the cut-approximation both check for only a part of the data set, while the acceptation distance accepts distances below a certain threshold. This will lead to (possibly) faster running times, but at the cost of accuracy.

One of the main advantages of the node-approximation and the cut-approximation is that we do not need any inner products anymore. Those took relatively long to calculate. As already mentioned in Section 3.1.1, there were some entries in the array representing the active points. These were represented with two rows of its coordinates. Those points will be neglected in the algorithm as the inner products are zero. This does not mean that they are also neglected in the approximations, since those do not use inner products. Therefore we need to do an extra check to make sure that these points will not negatively influence our results.

Node-approximation

The first heuristic is a node-approximation. Recall that our data was stored using an array of matrices. These matrices held the endpoints of line segments which took care of the curvature of connections. This means that each connection was subdivided in multiple smaller piecewise linear parts. With this heuristic, we will only check for the distances between the active points and the endpoints of each linear part. This implies that we do not have to check for the three different cases which were possible in the algorithm. This does simplify the algorithm a lot. As the cases are no longer possible, we also expect this heuristic to be quicker than the algorithm.

The results of the node-approximation will heavily depend on the type of network that is considered. As the node-approximation will only check the distance to endpoints of the piecewise linear parts of each connection, the results will depend on the number of parts each connection is subdivided in. In the case of a grid-like pattern, there are long straight connections, hence the results of the node-approximation will most likely be far from optimal. It is also possible to have short winding roads. In that case, each connection is subdivided in multiple smaller piecewise linear parts. Then the node-approximation will most likely give rather accurate results. More on the different types of patterns possible and properties of such patterns can for example be found in [25].

Cut-approximation

The second heuristic we have considered is a cut-approximation. Again, this heuristic checks the endpoints of the parts, but for longer parts, the algorithm also checks some extra intermediate points. Parts which are longer than a given threshold, which is required as input, are cut into multiple smaller pieces. This is done in such a way that the resulting extra pieces all have the same length. The approximation now also checks for the endpoints of the new smaller pieces. Note that the length of the piecewise linear parts is considered and not the length of the connection itself. We will refer to the threshold variable as the cut-value.

Note that this cut-approximation gives results which are at least as good as the node-approximation, as the same points, plus some more, are considered. We will

later on discuss the results of this approximation in comparison with the algorithm and the node-approximation. We will do this for different cut-values. We expect this cut-approximation to run slower than the node-approximation, as we have to consider more points. For high cut-values, so if we only cut very long edges, we expect the running time of the cut-approximation to tend to that of the node-approximation. However, they will not be equal as in the cut-approximation we also have to calculate the length of the parts of the line segments after which we have to compare those to the cut-value. For high cut-values we also expect the cut-approximation to be quicker than the algorithm. This is of course at the cost of accuracy. For smaller cut-values we expect the cut-approximation to run slower than the algorithm as each part is subdivided in multiple smaller pieces. Therefore multiple calculations have to be done.

Acceptation distance

Apart from the above two heuristics, we have also introduced an acceptation distance. The idea is that the solutions need not to be optimal, as long as they are close to optimal. The user can give an acceptation distance as input, the algorithm runs normally, but if for an active point a distance is found which is below the acceptation distance, it will not be considered in further calculations anymore. If the user does not give an acceptation distance as input, it is set to zero. We can think of the acceptation distance as being the threshold. If a distance is found which is below the threshold, we will not seek for better distances for that active point. However, active points which do not have a distance below the acceptation distance, will be checked until either the distance is below the acceptation distance or until all connections have been checked. Note that this implies that for acceptation distance of zero, all connections will be considered, but that this will give exact results.

Using an acceptation distance does not slow the algorithm down significantly. In fact, in most cases it will make the algorithm faster. This follows from the fact that after some iterations, it is possible that some active points need not to be considered anymore. We do a few extra computations, namely we check which points have a distance above the acceptation distance. The complexity of these extra computations, however, is $\mathcal{O}(1)$. Hence, the number of operations we can save using an acceptation distance outweigh the number of extra computations.

Note that the acceptation distance is only an addition to both the algorithm and the two approximations. The acceptation distance is not an algorithm on its own, but merely an addition to the already existing algorithm and the approximations. Therefore we will compare the algorithm and the two approximations for different acceptation distances.

3.2.5 Complexity of algorithm 1

Let us now take a closer look at the complexity of the algorithm and the two approximations. We will try to construct an upper bound for the running time, therefore we will consider the worst case scenario. That is, in the algorithm, every point is in Case 3, the length of the lists in the array is constant, but relatively long, and as acceptation distance we take zero. This last implies that for all active points, all line segments have to be considered.

Let n be the number of entries of interest in the array, i.e., in the case of a square-array, the number of non-empty entries in the lower triangular part of the array. If we have a column/row-array this means n equals half of the total number of entries in the array. Let m be the length of the matrix in the array and let k be the number of active points. We also define c as the number of pieces each piecewise linear part is cut in. Note that it is unlikely that *every* part is cut in c pieces, but as we are considering the worst case scenario this is allowed. Also note that c and the cut-value are inversely proportional.

The algorithm first searches for the non-empty entries in the array, after which the indices of interest are stored. Afterwards both in the algorithm and in the two approximations some space is allocated and some variables are set. Now the actual calculations can begin. We do this using a *for*-loop with which we consider the different entries in the array in which we are interested. We then check which active points still need to be considered, using the acceptation distance. Again we have a *for*-loop, this time over matrices in the non-empty entries of the array. These entries will be used to do the calculations regarding the distances.

For the algorithm this amounts to checking the three cases as discussed before. An inner product is used to determine for each active point in which case it is for the part that is considered. If a distance is found which is smaller than the optimal distance for that active point up until then, we save the distance as the new optimal distance. Also the indices corresponding to the part which is considered are saved, this is for later use.

The node-approximation immediately starts calculating the distances between the nodes and the points. This is possible as we do not have to check in which case we are. After the distances are calculated, the distances found are compared with optimal distances up until then after which we will update the matrix and the vector, with the indices corresponding to the optimal distances and the best distances up until then, if necessary.

The cut-approximation first calculates the length of the line segments and then determines in how many smaller parts this line segment has to be cut. Then, using a *for*-loop over the number of parts, we start doing the calculations regarding the distances. We again compare the distances with the best distances up until then, and update if necessary.

Now the exact coordinates of the points where the optimal distances were attained must be calculated. This comes down to a simple calculation which we have to do for every active point. It is also possible to print the results in a .txt-file, but this is in general not necessary.

Finding the indices of the non-zero entries takes $\mathcal{O}(n)$ time. Allocating space for matrices and vectors and setting variable takes $\mathcal{O}(1)$ time. We now have the first *for*-loop which takes $\mathcal{O}(n)$ time. Inside this *loop* we have another loop with complexity $\mathcal{O}(m)$. Inside this loop the calculations take place. Each calculation takes $\mathcal{O}(1)$ time, but $\mathcal{O}(k)$ of these calculations must be made. Therefore, in total these calculations take $\mathcal{O}(k)$ time. Finding the indices for which a calculation gave a result greater or smaller than zero can be done in $\mathcal{O}(k)$ time, updating the matrix and the vector can also be done in $\mathcal{O}(k)$ time. In the case of the cut-approximation we have to do some extra calculations regarding the cut-value and the length of the connections taking $\mathcal{O}(1)$ time, but we also have an extra loop regarding the number of parts each line segment is cut

in. This extra loop has complexity $\mathcal{O}(c)$. Generating the results takes $\mathcal{O}(k)$ time.

When we combine all the above we see that the algorithm has complexity $\mathcal{O}(\lambda nmk + \mu k)$, where $\lambda, \mu \in \mathbb{N}$ are integers greater than zero. However, recalling our knowledge about the $\mathcal{O}$-notation, this is equivalent to saying that the algorithm has complexity $\mathcal{O}(nmk)$. We see that the node-approximation has the same complexity. The cut-approximation however, has complexity $\mathcal{O}(nmkc)$. This follows from the cut-value used. Note that c is constant, hence technically we may neglect it in the expression. However, we chose to leave it as this enables us to better compare the two expression. For small c (hence for large cut-values) the complexity of the cut-approximation is the same as of the node-approximation and the first algorithm. However, if c increases , and hence if the cut-value decreases, the c plays a more prominent role in the complexity of the cut-approximation and the running time of the cut-approximation will most likely increase drastically.

Consider Table 3.8 for the complexities we have found above. Note that being in the same complexity class does not imply that the running times are equal. It only implies that the behaviour of the running times is the same if the input grows in size. We know that in the worst case scenario in the algorithm more calculations have to be made with respect to the node-approximation. However, as all these calculations only take $\mathcal{O}(k)$, we see that this does not change the complexity of the algorithm even though it does affect the running time.

	Algorithm 1	Node-approximation	Cut-approximation
Complexity	$\mathcal{O}(nmk)$	$\mathcal{O}(nmk)$	$\mathcal{O}(nmkc)$

Table 3.8: Complexities of the algorithms and heuristics for the first problem.

3.3 Algorithm 2

The second algorithm can be used in order to find where two networks intersect. Examples of such networks are a street network or a trench pattern. We need to determine the intersection points as we need to know where a transition from one network to the other network is possible. This then can be used to route through the network, which can help to establish least cost-connections between the active points and the fiber glass network. Apart from the intersections of the two shapes with each other, we are also interested in the intersections of a shape with itself. Consider the simple example of two pipe lines which, when projected on the plane, intersect. However, it is possible that the first pipe line lies higher than the second, hence, in the real situation they do not intersect. As it might be beneficial to go from one pipe to the other, we also want to report such intersections.

We will approach the problem in two different ways. The first being a *smart brute force*-version, the second being a modified version of the *Bentley-Ottmann*-algorithm.

3.3.1 The *smart brute force*-algorithm

The first algorithm of the second problem is a smart version of the *brute force*-algorithm. In a brute force-algorithm one would compare every possible pair of two line segments if

they intersect. This will certainly find all intersection points, however, the running time of this approach will also blow up as its complexity is $\mathcal{O}(m^2)$, where m is the number of line segments. Note that we consider the line segments and not the connections. Algorithm 3.9 shows the pseudo code of the smart brute force-algorithm.

Algorithm 3.9 Smart brute force-algorithm

Input: n connections, each consisting of multiple line segments, number of blocks c
Output: Array with non-intersecting connections

Initialization: Determine the minimum and maximum x- and y-value of every connection

```
1:  Divide the data set in c blocks
2:  for all blocks do
3:      for every pair of connections in a block do
4:          if bounding rectangles overlap then
5:              for every pair of line segments do
6:                  if there is an intersection then
7:                      Report the intersection
8:                  end
9:              end
10:         end
11:     end
12: end
13: Given the intersections found, create the array with no self-intersecting connections
```

One of the reasons we call it a *smart* brute force-algorithm is that we subdivide the data set in multiple smaller blocks. Each block consists of a number of connections which might intersect. However, if two connections do not appear in the same block, they can certainly not intersect. Apart from the number of blocks the data set needs to be subdivided in, the algorithm also needs the connections for which we want to determine the intersections as input. Recall that each connection consists of one or more line segments. The output produced by the algorithm is an array consisting of the connections which were given as input, however, every intersection point is now treated as an endpoint of a connection. Therefore, in the new array there will be no intersecting connections.

Let us now explain how we determine the different blocks. Note that most segments are not that long and therefore most pairs do not need to be considered. Therefore we divide the data set in multiple smaller data sets or blocks. In order to do so, we calculate the minimum and the maximum x-value among all connections. The difference between these two values is the width of our data set. This value is then divided by the number of blocks we want. This can either be given as input by the user, or a default-value of fifty is used. The data set is then subdivided in the given number of blocks such that the width of each block is equal (the width of a block is defined by the difference between the minimum and the maximum x-value among all line segments in that block). A connection belongs to a block if it has overlap with that block, meaning

that the minimum x-value of the connection must be smaller than the maximum x-value of the block and the maximum x-value of the connection must be greater than the minimum x-value of the block. Note that this allows for connections to be in multiple blocks. It is not possible to subdivide the data set in disjoint blocks, as this would imply that we known which line segments certainly do not intersect. However, for some cases (near the edge of the blocks) this can only be determined by explicit calculations. Hence, we should already know if segments intersect before we can subdivide them in disjoint blocks. Therefore, we allow connections and hence line segments to be in multiple blocks. Also note that the use of blocks still results in all intersection points. This approach saves time as connections which do not appear in the same block will certainly not intersect. Namely, as they do not appear in the same block, at least one of the connections does not have overlap with the blocks. Therefore the connections do not have overlapping x-values and hence they can never intersect.

For each block we will also use some tricks due to which we do not have to check all possible pairs of connections. Note that two line segments can only intersect if they have overlapping x-values and also overlapping y-values. In Figure 3.10 we see two situations. On the left we see two line segments which have overlapping y-values, but non-overlapping x-values, on the right we see two line segments which have both overlapping x- and overlapping y-values. This last pair of line segments is said to have overlapping bounding rectangles, meaning that the smallest bounding rectangles, each of which contains one line segment, overlap. We chose the orientation of these rectangles such that one side is parallel to the x-axis, while the other is parallel to the y-axis, however other orientations will work similarly.

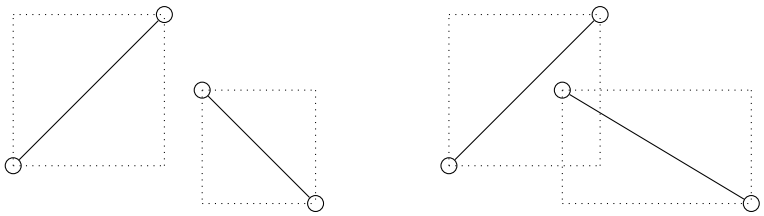

Figure 3.10: On the left two line segments with non-overlapping x-values. On the right two line segments with overlapping x- and y-values.

Note that the same figure also tells us that overlapping bounding rectangles does not imply intersecting line segments. In the algorithm itself we will not compare the bounding rectangles of individual line segments as this would lead to many extra calculations, which most likely take longer than applying Theorem 2.63 immediately would. Instead we will consider the bounding rectangles of the connections as a whole, see Figure 3.11 for a bounding rectangle of a connection. Note that this works as given two intersecting line segments, then their bounding rectangles overlap. However, the bounding rectangles of the line segments lie in the bounding rectangles of the connections as a whole. Hence, the bounding rectangles of the connections also overlap.

Note that in most practical cases using bounding rectangles for the connections will lead to faster algorithms than using bounding rectangles for every line segment individually. This for instance follows from the fact that we need to determine the

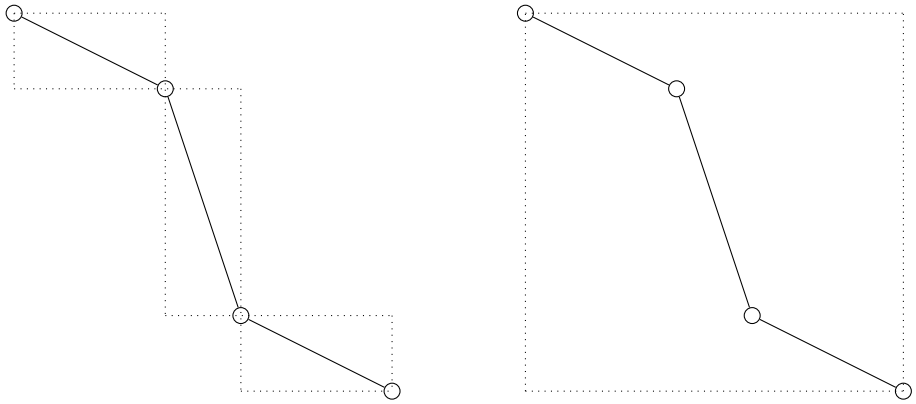

Figure 3.11: On the left a connection with the bounding rectangles for every line segment. On the right a connection with the bounding rectangle for the connection.

minima and maxima for all connections opposed to the minima and maxima for all line segments of every connection. Another point due to which this approach is faster is that most connections do not intersect with other intersections. Doing only one comparison for the connections is then faster than doing it for every pair of line segments of the connections. Note that this is given the fact that the bounding rectangles of the connections do not intersect.

If the bounding rectangles of the connections overlap, we will determine for every pair of line segments of the two connections if they intersect. Now Theorem 2.63 is used to determine if two line segments do intersect. If two line segments do intersect, we report the intersection together with the indices corresponding to the line segments.

After all line segments have been found, we will use the indices we saved to create the output. So, every connection is subdivided in multiple smaller connections according to the intersection points. Note that the smaller connections are still collections of line segments. Also note that we have to take care of intersections which are found twice due to the use of blocks. This can be taken care of by only considering one of the two intersections found as they are equal.

Note that we do some extra calculations such as subdividing the data set in multiple blocks and determining the bounding rectangles of connections. In cases where most of the line segments pairwise intersect, this will lead to more computations and therefore longer running times. However, in most cases we will be considering data sets where most line segments do not pairwise intersect. Using such tricks as mentioned above, will make sure that connections and line segments which certainly do not intersect, will not be checked in further calculations. Connections and as a result line segments which can intersect will be checked. Therefore, we conclude that all intersections are still found.

3.3.2 *Bentley-Ottmann*-algorithm

In 1976 Michael Ian Shamos and Dan Hoey published an article ([26]) in which they present an algorithm which determines if given a set of n line segments, any two intersect. They also applied their method to detect whether two simple polygons would

intersect. This algorithm was extended by Jon Bentley and Thomas Ottmann in 1979 ([27]) to an algorithm which could detect and give all the intersection points of a set of n line segments. The complexity of this algorithm is $\mathcal{O}((n+k)\log(n))$, where k is the number of intersections. Faster algorithms with complexity $\mathcal{O}(n\log(n)+k)$ have been constructed ([28], [29]), the Bentley-Ottmann-algorithm however remains the most popular choice. This is due to its simplicity and the low memory requirements. Other, faster, algorithms either use more space, or use a randomized algorithm, making it more difficult to understand and implement the algorithm. Note that the Bentley-Ottmann-algorithm is output-sensitive, meaning that the complexity and therefore the running times depend on the size of the output.

The Bentley-Ottmann-algorithm is a line sweep algorithm, meaning that a sweep line is used which sweeps through the plane looking for intersections. The sweep line detects events and an event is either a starting point or an endpoint of a line segment, or two lines intersecting. Even though this algorithm gives exact results within reasonable time, there are a few drawbacks. For instance, some special cases, such as two segments starting in the same point, are excluded. Apart from that some dynamic structures such as binary search trees are used which MatLab does not support. Let us first consider the pseudo-code of the algorithm, then discuss the algorithm in detail and finally discuss how the drawbacks are taken care of.

Algorithm 3.12 Bentley-Ottmann-algorithm

Input: n line segments, defined by their endpoints
Output: k intersections of line segments

Initialization: Sort the $2n$ endpoints according to their x-values and store the information in E, line L is empty

1: **while** $E \neq \emptyset$ **do**
2: $p = \min E$
3: **if** $p =$ start(s) **then**
4: Insert s in L
5: Check if s intersects with its neighbours in L. If yes, insert the intersection point in E
6: **else if** $p =$ end(s) **then**
7: Remove s from L
8: Check if the old neighbours of s intersect. If yes, insert the intersection point in E
9: **else if** p intersection point **then**
10: Report the intersection point with corresponding line segments
11: Transpose the order of the line segments in L
12: **end**
13: Delete p from E
14: **end**

The algorithm starts by sorting the $2n$ endpoints of the n line segments. The sorting happens according to the x-values of the endpoints. This data is then stored in a *priority*

queue E, also referred to as event list E, used to keep track of the possible future events. We now want to determine all intersecting pairs of line segments and the coordinates of the intersection point. Once the priority queue is set up, we use the sweep line L to determine all intersections. First we take the event with the minimum x-value in E and we check what sort of event it is. As an event can be an intersection of two line segments, or the beginning or ending of a line segment we have three possible types of events which are all fundamentally different. See also Figure 3.13 for the possible events we have.

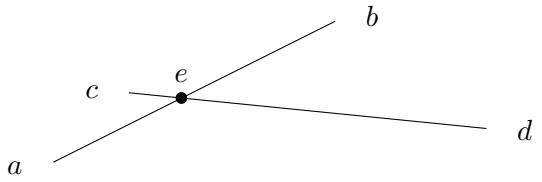

Figure 3.13: An example of the possible events: a, c are starting points, b, d are endpoints and e is an intersection point.

After the event is treated, it will be removed from the priority queue. We will then find the next minimum x-value and the corresponding event. We will do this until the priority queue is empty, at which point all intersections have been found. After that we generate output such that every intersection point is a node in the new network. Let us now take a closer look at the three types of events.

Event: startpoint

If the event happens to be the start of a line segment, we insert the line segment in the sweep line L. The line segments in L will be stored according to the y-coordinates. In Figure 3.14 we see that the sweep line is at the starting point a. The segment ab will then be added to L according to the y-value of a. We will then check for a possible intersection of the newly added line segment with its direct neighbours in L. If an intersection is found, it will be added to our event list E. Note that in the example below, we will not have to check for intersections as a is the first point added to L. If c is added to L we will check if ab and cd intersect.

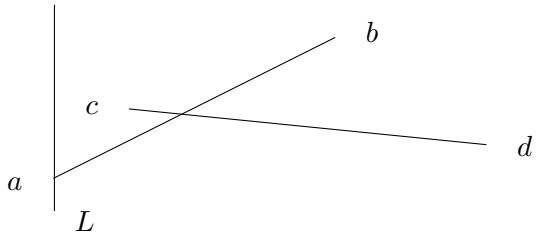

Figure 3.14: The sweep line L starts at the first event a.

Line segments for which the left endpoint is in L are said to be *active* segments.

Note that the order in which line segments are added to the sweep line depends on their x-value. However, the place where the line segments are inserted in the sweep line depends on the y-value of the line segments. As a consequence, when we insert a new segment in the sweep line, we have to be careful with the y-values of the other segments. The y-value a segment had when it was inserted, does not have to be equal to the y-value of the segment at the current position of the sweep line. This is of course due to the slope of the line segments. In order to take care of this, we do some extra calculations to determine the y-value of some segments at the current position of the sweep line. Note that we do not have to do this for every segment if we choose thee segments in a smart way.

Event: intersection

If the event is an intersection, it will be reported and deleted from E. However, we also need to flip the order of the corresponding lines in L, see also Figure 3.15. After we have swapped the order of the line segments in L which intersected, ab and cd in the example, we will check for new possible intersections with the new neighbours. That is, if the order before an intersection is $(\ldots, v_1, v_2, v_3, v_4, \ldots)$ and $(\ldots, v_1, v_3, v_2, v_4, \ldots)$ after the intersection, where every v_i is a line segment, we will check for a possible intersection between v_1 and v_3 and between v_2 and v_4. Note that these segments are the only segments which have to be checked. Possible intersections with other line segments have already been found or will be found in future calculations.

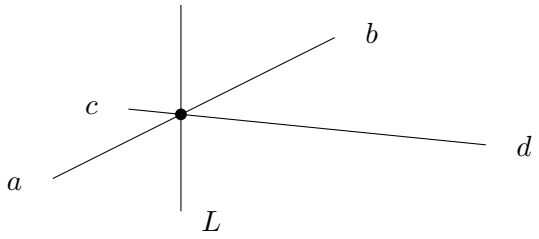

Figure 3.15: The sweep line L arrives at an intersection. Before the intersection we have $L = (cd, ab)$ after the intersection we have $L = (ab, cd)$.

Note that it is possible that we find an intersection from the past. In this case we do not have to add it as an event to the event queue. Consider Figure 3.16 for an example where an intersection from the past is found. Here line segments ab and cd become neighbours again in the sweep line after the intersection point between ef and ab is reported. As we have mentioned above, if two lines become neighbours in the sweep line, we have to check if they intersect. We know that ab and cd intersect as we have already reported it. As the intersection point between the two segments is to the left of the sweep line, we may neglect it.

Event: endpoint

The last type of event we have is an endpoint. This comes down to removing the corresponding line segment from L. Line segments for which the right endpoint is to the

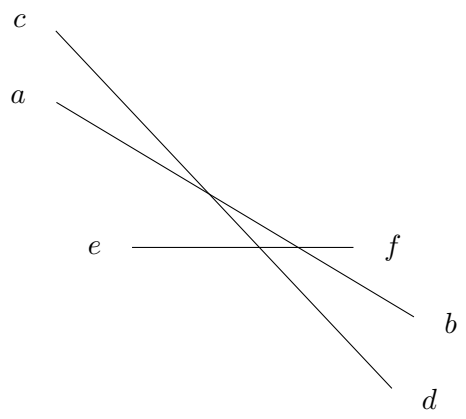

Figure 3.16: After the intersection between ef and ab is reported, segments ab and cd become neighbours in the sweep line again.

left of L are said to be *dead* line segments, see also Figure 3.17. Once a line segment is removed from the sweep line it is of no interest to us anymore. After it has been removed we have to check for possible new intersections between the two old direct neighbours of the removed line segment. If these new neighbours intersect and the intersection point is to the right of L, we will insert it in the event list E.

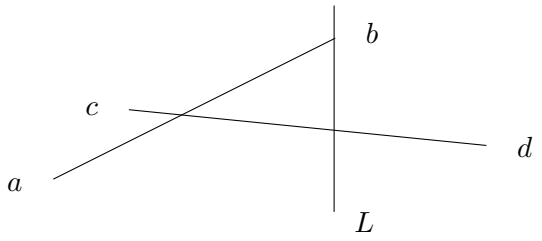

Figure 3.17: Line segment ab is removed from L.

3.3.3 Correctness of Bentley-Ottmann

Let us now prove that the Bentley-Ottmann-algorithm does in fact find all intersection points. Recall that there were some assumptions made on the data set when using the Bentley-Ottmann-algorithm. These are

1. No line is vertical;
2. No two line segments overlap;
3. An endpoint of a line segment does not lie on another line segment;
4. At most two line segments intersect in a point;
5. Every event has a different x-coordinate.

These assumptions are made to make sure that every event has different coordinates and that every intersection is a proper intersection. Note that the Bentley-Ottmann-algorithm relies on the fact that two line segments can only intersect if they are adjacent in the sweep line, i.e., two line segments can only intersect if they are direct neighbours. If that is not the case, there must be an event prior to the event where they intersect such that the two segments can become direct neighbours. Let us now prove that given these assumptions, the Bentley-Ottmann-algorithm gives all results. First we prove the following lemma.

Lemma 3.3. *During the* while*-loop of the Bentley-Ottmann-algorithm at any position of the sweep line all pairs of intersecting dead segments have been reported.*

Recall that a segment was said to be dead if its rightmost endpoint was to the left of the sweep line. In other words, the lemma states that at every position of the sweep line, all intersections to its left have been found.

Proof. Consider any position of the sweep line. Proving that all pairs of intersecting dead segments have been reported is the same as proving that each intersection between dead segments has appeared as the minimum element in the event list E. If this is true, then, in the third *if*-statement of the algorithm, the intersection is reported. Hence, all pairs of intersecting dead segments have been reported.

We will prove this by contradiction. Assume there is an intersection between two dead segments that did not appear as the minimum of the event list E. Let p be the leftmost intersection point with this property and let S and S' be the dead segments that intersect in p. Now let q be the rightmost event to the left of p. Note that q exists as S and S' are active after q. Also note that q has appeared as the minimum of the event list E. Let us now consider what happens after the event q is treated. As S and S' are both active after q, they are both stored in the sweep line L. There are two possible cases.

Case 1: After q the line segments S and S' are neighbours in L. As they are neighbours, their intersection point p is contained in E. Also, q was chosen such that p was the next event. Hence, immediately after q, the event p is the minimum in the event list E. Which is a contradiction.

Case 2: After q the line segments S and S' are not neighbours in L. Then there must be an event between q and p in which a line segment deleted from L or in which an intersection point is reported. This is a contradiction with our choice of q.

Hence p appeared as the minimum in the event list E and therefore the intersection was reported. So we see that all pairs of intersecting dead line segments have been reported. □

Theorem 3.4. *The Bentley-Ottmann-algorithm finds all intersection points.*

Proof. At the end of the algorithm the event list is empty, implying that the sweep line is to the right of all segments. Therefore, at the end of the algorithm all line segments are dead. Using Lemma 3.3 it follows that all intersection points are reported correctly. □

3.3.4 Applying Bentley-Ottmann to our situation using MatLab

As already mentioned there are a few assumptions made in order to be able to correctly use the Bentley-Ottmann-algorithm. However, let us first discuss why we chose the Bentley-Ottmann-algorithm over other algorithms which could also solve this problem. Algorithms have been constructed which, given two sets of line segments, find all intersection points between these two sets of segments. Typically one thinks of the sets as being a set with red line segments and a set with blue line segments. One is then interested in the intersections between red and blue line segments. Algorithms which solve this so called *Red-Blue Line Segment Intersection*-problem can for instance be found in [30] and [31]. However, using his approach will not give a complete answer. Namely, we have to take into account that networks can self-intersect, meaning that two line segments of the same network intersect other than in their endpoints. As already mentioned, these intersection points are also of interest as also at these points a transition is possible. Therefore, treating the problem as a *Red-Blue*-problem will either not give us complete answers, or forces us to do extra calculations to detect the self-intersections afterwards. This last option is not viable, as this would lead to two extra algorithms alike Bentley-Ottmann for both sets separately. Instead we could thinks of the two sets as being a single bigger set and the use a modified version of the Bentley-Ottmann-algorithm on this bigger set. The term modified refers to the fact that there are some problems which have to be dealt with. Using this modified version will give both the intersection between the segments from both sets and it will give the self-intersections. We can encode the segments in such a way that we know to which set they belonged. However, as we are only interested in the resulting network, it does not matter whether an intersection was an intersection between line segments from different sets or from the same set.

Let us now consider the assumptions on the data set that were made. Recall that these assumptions were

1. No line is vertical;
2. No two line segments overlap;
3. An endpoint of a line segment does not lie on another line segment;
4. At most two line segments intersect in a point;
5. Every event has a different x-coordinate.

These assumptions are made to make sure that every intersection point is found. Even though these assumptions simplify the algorithm, in most real world applications these assumptions do not hold. Hence, we need to modify the algorithm such that these assumptions are no longer necessary. There are multiple possibilities to solve these assumptions. One is to extend every endpoint by a small ε. That way every intersection is still found, while no extra intersections are found given that ε is small enough. A drawback of this method however is that in our situation extra intersections will be found. As some connections, and therefore some line segments, intersect in their endpoints, extending them slightly will give an extra intersection point which we do not

want. Therefore this approach will not be used. Note that as we are extending each line segment by a small ε we will still find all the intersections.

Another possibility we have is to generalize our definitions used in the algorithm. If we allow events to have the same x-coordinate, but we sort the events with the same x-coordinate according to the y-coordinate, we have solved the assumption 1 and 5. The approach for solving the second assumption is to consider lists of line segments corresponding to a certain event instead of considering individual line segments. This way, overlapping line segments are treated at the same time, which solves the problem. Assumption 3 can be taken care of by generalizing the definition of an intersection. Instead of considering only proper intersections, we will also allow intersections where three of the endpoints are collinear. The fourth assumption can be solved by generalizing the definition of an intersection point. Instead of two line segments which intersect, more line segments may intersect in a single point. When treating the event, we must be careful. Instead of swapping the position of two line segments, we must reverse the order of the multiple line segments we are considering. Note that for two line segments this is the same as swapping the position. So we have the following solutions for the assumptions sketched above.

1. Define the endpoint with the higher y-coordinate to be the starting point and the event with the lower y-coordinate to be the endpoint;

2. Consider overlapping line segments at the same time instead of one after the other;

3. The definition of an intersection is generalized;

4. Multiple line segments may intersect in a single point. At the intersection point, we reverse the order of the corresponding segments in the sweep line;

5. Events with the same x-coordinate are sorted according to the y-coordinate.

Note that we talk about line segments intersecting instead of connections intersecting. This follows as the Bentley-Ottmann-algorithm works for line segments only. Therefore we will treat each piecewise linear part of a connection as a separate line segment and then use a modified version of Bentley-Ottmann. We have excluded the case that two line segments can intersect in their endpoints, as these are just the nodes of the network. Note that excluding these points as being intersections prevents extreme cases where almost every node is said to be an intersection. Also note that the case of overlapping line segments is excluded by our assumptions on the data sets. Namely, the data sets are constructed such that no line segments overlap, hence this assumption is taken care of.

Note that due to the fact that we found solutions for the assumptions, we cannot prove that this version works. This follows as multiple events can have the same x-coordinate, in some cases events even overlap. The proof of Lemma 3.3 and hence Theorem 3.4 relied on the fact that this was not the case. However, in Section 4.2, where we compare the results, we will see that all intersections will be found.

The last thing we have to take care of is the data structures used in the Bentley-Ottmann-algorithm. Binary search trees (Section 2.3.2) are used for both the event list E and the sweep line L, however, binary search trees are not supported by MatLab.

MatLab works extremely well when working with matrices. However, when we would like to express data in a dynamic way, for instance using a binary search tree or a linked list, things get a little more complicated. There are no built-in commands to take care of this, however some packages have been created which recreate the structure of binary search trees using other structures, for instance [32]. We have tried to implement this, however, as it turned out, the running time of the algorithm would drastically increase and the modified Bentley-Ottmann-algorithm would in this case take longer than the smart brute force-algorithm we have discussed before. Therefore we will not use such packages. Note that other programming languages such as C and C++ do support these dynamic structures, however, the algorithm constructed will be used in an already existing MatLab program. Therefore, we will program the algorithm using MatLab. In order to deal with the data structures problem, we have recreated the structures using arrays.

Note that we now have two algorithms. The Bentley-Ottmann-algorithm as constructed in 1979 and a modified version of it constructed by us. In the following when we refer to our version, we will refer to the modified Bentley-Ottmann-algorithm. The original algorithm will be addressed as the Bentley-Ottmann-algorithm, as one could expect.

Event list

Recall that the event list contained all events which have not happened yet. This includes both the starting points and endpoints of the line segments and the, at a certain moment known, intersection points between line segments. As the intersection points are not known beforehand, the event list must be a dynamic structure which can add entries and dynamically sort them according to their x-values.

Instead of implementing this list, we recreated it using an array. As input every entry of the array contained the start- and endpoints of the line segments, an index saying if it is an endpoint or a starting point, the slope of the line segment and an index to link the line segments with its original position in the array. This last index is used to create the output of the algorithm. This array is then sorted according to the x-coordinates of the line segments. In the case that those were equal, we sort according to the y-values.

The intersections are listed in another array, as otherwise we have to insert the intersection points in the array with intersections and this is very time-consuming. Once an intersection has been found, both the intersection point and the coordinates of the corresponding line segments are added to the array. The algorithm determines the next event according to the x-values of the events. For the array containing the intersections this means finding the minimum x-value among the intersection points. Note that we can also sort this array every time a new intersection is added, however that would take at least as long as taking the minimum x-value over the arrays. For the array containing the start- and endpoints this comes down to determine the x-coordinate of the next entry. An index is used to determine which events have already been processed and which events have not. Once an intersection is treated, it is deleted from the array. Hence we can take the minimum over the entries in the array. We use an index for the other array as this was most efficient time-wise. Due to the fact that the array was sorted, we can process the events in order of appearance.

Sweep line

The second data structure the algorithm uses is a sweep line, hence the name sweep line algorithm. Theoretically a self balancing binary search tree would be the best to use for the sweep line. However, it turned out implementing this structure resulted in running times which were too large, most likely this was a result of the package used for the implementation. Therefore, also for the sweep line we have used an array.

Again the start- and endpoints of the line segments are denoted in the array, together with the corresponding indices. At an intersection, which implies swapping the positions, we swap the rows of the array. We also like to add new line segments to the sweep line. However, to be able to do that, we need to compare the y-value of the line segments. If we would do this naively and calculate the y-value for every line segment, we would do too many calculations. Therefore, we chose to construct a function which only checks for the middle value of the sweep line and then, according to the result, continues the calculation with half of the previous array. This way, in every iteration the size of the sweep line halves, hence we do not have to do the calculations for every entry. Note that this technique is also used in the Binary Search Trees resulting in their complexity $\mathcal{O}(\log n)$ to insert or delete an event or to locate it.

3.3.5 Generating results

Once the intersection points have been found, we still need to generate the output of the algorithm. Recall that we applied the algorithm to determine the intersections between two sets. In order to do that, we treated the data sets as being one large data set and we computed the intersections accordingly. The output of the algorithm is then an array consisting of the two original networks, however, such that there is no self-intersection. Hence, every intersection point we have found using the algorithm, must be a node in the new network.

If two connections intersect, we want to make four non-intersecting smaller connections of it, the intersection point being an endpoint of all four smaller connections, see also Figure 3.18. Note that on the left side e serves as an intersection point, while on the right side e is an endpoint of the four line segments. Recall that intersections were recorded using the intersection point, but also the coordinates of the corresponding line segments and the indices of the line segments. Recall that we denoted the intersections using an array. In this array, both the intersection point and the indices corresponding to one of the connections was denoted. Hence, every intersection generates two entries in the array with intersections, one for every connection.

We now first check if an intersection is reported multiple times. Double entries are then neglected in further calculations. After that, we sort the array according to the indices in the array. We then use the indices to determine if a connection has an intersection point and if so, we split the list of line segments in two separate lists. One of the lists is moved to another entry of the array as in the output we regard this as a new connection. The last entry of one list and the first of the other list are the coordinates of the intersection point. This is due to the fact that this will be regarded as a new node in the network.

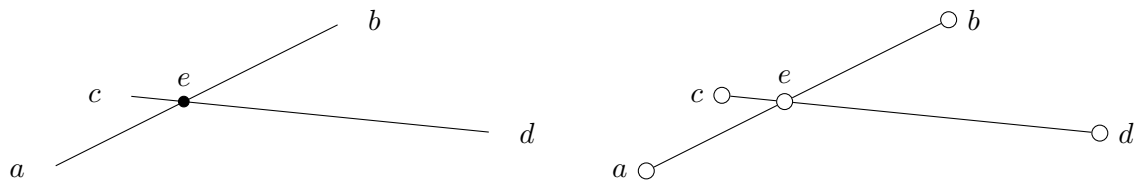

Figure 3.18: On the left two intersecting line segments, on the right the resulting four non-intersecting line segments.

3.3.6 Speeding up the algorithm

As already mentioned, we used some minor tricks to increase the speed of both the modified Bentley-Ottmann-algorithm and the smart brute force-algorithm. For the modified Bentley-Ottmann-algorithm this was recording the indices and coordinates of the line segments in the arrays as well. That way we did not have to use a *find*-function in MatLab anymore to determine which entries had to be considered in certain calculation. Instead we could use the indices saved in the array. The *find*-functionality takes relatively long, therefore, if we do not have to use it, we will save time.

For the smart brute force-algorithm we have used some tricks which increase the speed of the algorithm. The first one was subdividing the data set in multiple smaller data sets (or blocks). The second trick used is using an extra array which denotes the minimum and maximum x- and y-values of each connection. Recall that we used the bounding rectangles to determine if an intersection was possible. If in every iteration we have to determine these minima and maxima of the two connections, we would do extra, unnecessary work and the algorithm would slow down. Instead we calculate the minimum and maximum x- and y-value once for every connection and save them in an array. Determining if two rectangles overlap, then reduces to comparing predetermined entries in the array.

As we sort the connections according to the x-values of their starting points, we can use the minimum and maximum x-values of the connections to determine if some part of the data set need not be be checked anymore. For this we define the width of a connection to be the maximum x-value minus the minimum x-value. The maximum width is then determined by taking the maximum over the widths. We now make use of the fact that two line segments will never intersect if the x-values of their starting points are more than the maximum width apart. Now suppose that the starting points of connection i and connection j are more than the maximum width apart, then in the next iteration (for $i + 1$) we do not have to consider the connections 1 through j. This follows as the array was sorted. Therefore, all connections $1, \ldots, j$ will certainly not intersect with line segment $i + 1$. Apart from starting our search at a later point, we can also prematurely stop the search. The idea is the same as above. Once the two endpoints are too far apart, we will not have to check the points anymore.

3.3.7 Heuristics

Just as we did for the first algorithm, we have also constructed some heuristics for the second algorithm. As the modified Bentley-Ottmann-algorithm will most likely be faster than the smart brute force-algorithm the heuristics will be applied to the smart brute force-algorithm. Note that, due to the setup of the data in the modified Bentley-Ottmann-algorithm, heuristics will also be harder to construct and implement in this case.

Two heuristics have been constructed, the first is a *connection*-approximation and the second is a *slope*-approximation. Both approximations do an extra check (besides the overlapping rectangles) which have to hold before we will actually search for an intersection.

Note that the heuristics will give (possibly) faster results, but this comes at the cost of accuracy, meaning that not all intersections need to be found. Let us now treat the heuristics in detail.

Connection-approximation

The connection-approximation (or con-approximation for short) only checks some connections if there is an intersection. For this approximation we will think of each connection as being a single line segment between the two endpoints of the connection. This line segment will be called the general line segment corresponding to the connection. We will then only look for an intersection between two connections if the general line segments intersect.

Note that using this approach does not give all intersections. An intersection between the general line segments does not imply an intersection between the connections. Vice versa, an intersection between two connections does not imply an intersection between the general line segments. See also Figure 3.19 for examples of such situations.

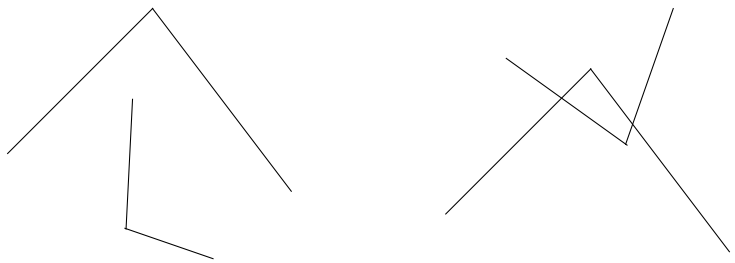

Figure 3.19: On the left two non-intersecting connections with intersecting general line segments. On the right two intersecting connections with non-intersecting general line segments.

Note that this approximation will most likely give results faster than the smart brute force-algorithm. This is due to the fact that only for some pairs of connections we have to check if there is an intersection.

Slope-approximation

While in the previous approximation we checked if the two general line segments corresponding to two connections intersect, we will now look at the slopes of the two general line segments. Two line segments with a near equal slope, are very unlikely to intersect, as in this case they are almost parallel. Two line segments with a very different slope, however, are more likely to intersect. Note that we cannot take the absolute difference between the two slopes as this might give wrong results. Consider for instance a vertical line and a line with a slope of, say, 10,000, they differ by more than any constant, while in practice they will most likely not intersect. Therefore, we will compute the angle the two general line segments make and proceed if this angle is larger than a given constant. We compute the angle using the slopes of the line segments, hence the name slope-approximation. The angle will be computed using the arctangent. This gives values in the interval $(-\pi/2, \pi/2)$, hence we need to give the angle by which the general line segments have to differ in radians.

Now, pairs of connections will only be checked for intersections if the angle between the two general line segments differ by more than the given constant. This constant can be given by the user or a default-value of $\pi/4$ is used. Note that $\pi/4$ corresponds to an angle of $45°$ between the two line segments.

Note that this will most likely not give all the intersections, as not all intersecting line segments have to intersect at an angle larger than that constant.

3.3.8 Complexity of algorithm 2

Let us now determine the complexity of the two algorithms and the heuristics. For simplicity we will refer to the modified Bentley-Ottmann-algorithm as the mBO-algorithm, the smart brute force-algorithm will be referred to as SBF-algorithm and we will refer to the heuristics as the con-approximation and the slope-approximation. We may also abbreviate or omit the suffixes *-algorithm* and *-approximation*, given that it will not lead to confusion. Note that both heuristics are algorithms on their own, however, to stress that they are approximations of the SBF-algorithm, we will refer to them as approximations.

Again we let n be the number of entries in the two arrays together and m the length of the lists in each entry of the array. Note that $m-1$ equals the number of line segments per connections and that it can change per entry, however, as we are considering the complexity we may fix m as the typical or average length of the entries of the array. The number of intersections will be denoted by k. Also for the smart brute force-algorithm and its heuristics let c be the number of blocks we divide the data set in.

Let us first consider the SBF-algorithm. First the data set is divided in multiple blocks and on each block a smart brute force-algorithm is used. As we know that two connections can only intersect if they are in the same block, we do see that we find all intersections. It is however possible that some intersections our found multiple times. However, this does not negatively influence our results. On average each block will contain $\frac{n}{c}$ line segments. Note that in practice the first and last blocks will have less connections, while the middle blocks have more. Also note that some connections might appear in multiple blocks. For each block this then leads to $\frac{1}{2}\frac{n}{c}(\frac{n}{c}-1)$ possible pairs of connections. For each block we then determine if the two connections have overlapping

rectangles and if they do, we determine if their line segments intersect. This last results in $\mathcal{O}(m^2)$ pairs of line segments. For each pair we have to do calculations taking $\mathcal{O}(1)$ time. After the intersection points have been found, we still need to generate the output. This comes down to creating new connections such that the intersection points are noted as endpoints. Creating the new connections takes $\mathcal{O}(1)$ time and we have to do this $\mathcal{O}(k)$ times. This results in a complexity of $\mathcal{O}(\frac{n}{c}(\frac{n}{c}-1)m^2+k)$ on each block. As we have c blocks we see that in total we get a complexity of

$$\mathcal{O}\left(\frac{n^2}{c}m^2+k\right).$$

Note that c is constant, hence we may neglect it in the consideration. Similarly, k is smaller than n^2m^2, hence this may also be neglected. For the sake of comparison we leave both k and c in the above expression.

Let us now take a look at the complexity of the heuristics of the SBF-algorithm. Note that for both approximations we again subdivide the data set in c blocks. This leads to an extra factor $\frac{1}{c^2}$ with the n^2, however, it also gives a factor c, as we consider c blocks. Resulting in total in an extra factor $\frac{1}{c}$. For every pair of connections, the slope-approximation has to do calculations to determine if the connections can intersect, this takes $\mathcal{O}(n^2)$ time. For pairs which might intersect given the bounding rectangles, we need to do extra calculations corresponding to the slopes, this takes $\mathcal{O}(1)$ time however. Given an arbitrary connection, we expect only a fraction, say $\mu \in [0,1]$, of the connections to have a slope which differs by more than the constant the user set with that of the arbitrary connection. For those pairs we need to do the extra calculations taking $\mathcal{O}(m^2)$ time. For the other pairs we do not need to do the calculations. Afterwards, after the intersections have been found, we need to generate the output. Note that even though not all intersections are found, a portion of them is, say a fraction of $\lambda \in [0,1]$ of the intersections is found. This gives a complexity of $\mathcal{O}(\mu\frac{n^2}{c}m^2+\lambda k) = \mathcal{O}(\frac{n^2}{c}m^2+k)$. Again, c and k can be left out of the expression, but for the sake of comparison we will leave it.

For the con-approximation we have a similar situation. This time we check if the general line segments corresponding to two connections intersect. If these general line segments intersect, we will check if the individual line segments will intersect. This again leads to $n(n-1)/2$ pairs of connections we have to the bounding rectangles for and also if the general line segments intersect. Only for a part of them, say $\mu \in [0,1]$, we have to check the individual line segments resulting in an extra $\mathcal{O}(m^2)$ calculations. Afterwards we again have to generate the output, again only a fraction $\lambda \in [0,1]$ of the output is found. This leads to $\mathcal{O}(\mu\frac{n^2}{c}m^2+\lambda k) = \mathcal{O}(\frac{n^2}{c}m^2+k)$. Note the extra factor $\frac{1}{c}$ resulting from the usage of blocks.

Hence, we see that the SBF-algorithm and the slope- and con-approximation have the same complexity. Notice however, just as for the first algorithm, that this does not imply that the running times are the same. It only means that under growing input, the running time of the algorithm will scale similarly.

Let us now consider the modified Bentley-Ottmann-algorithm and its complexity. The mBO-algorithm works on line segments and not on connections as a whole. Therefore we have to modify the input in such a way that we are left with only the line segments. Both can be done using a *for*-loop over each connection and the piecewise

linear line segments of each connection. This results in a term $\mathcal{O}(nm)$ for the complexity. After that, we have to order our data according to the x-coordinates. This can be done in $\mathcal{O}(nm \log(nm))$. Note that we use nm here opposed to just n as we have to consider every line segment and not every connection. After that the *while*-loop of the algorithm starts. This amounts to processing all events and do calculations according to the type of event. The starting points of line segments need to be inserted in the sweep line. The time this takes scales according to the logarithm of the number of events in the sweep line, which is bounded by $\mathcal{O}(\log(nm))$. Also some calculations have to be done to see if we have a new intersection, which takes $\mathcal{O}(1)$ time. If we have to delete a line segment from the sweep line, we have to locate it in the sweep line, then delete it and then check for possible new intersections. The time this takes is also bounded by $\mathcal{O}(\log(nm))$. For intersection points, we have to swap the corresponding line segments in the sweep line and we have to report the intersection. This complexity is bounded by $\mathcal{O}(\log(nm))$. Given that we have k intersection points, this leads to $\mathcal{O}(k \log(nm))$. After all events have been processed, we have to generate the output. The time this takes scales according to k. This leads to a complexity of

$$\mathcal{O}\left((nm + k) \log\left(nm\right)\right)$$

for the modified Bentley-Ottmann-algorithm. Note that we used the fact that some complexities are bounded by $\mathcal{O}(\log(nm))$. This is due to the fact that the complexity scales according to the logarithm of the size of the sweep line. However, the size of the sweep line changes with every event. Therefore we use nm as the size of the sweep line is bounded by nm, hence the complexity for these parts is bounded by $\mathcal{O}(\log(nm))$.

So, to conclude, we find the complexities as shown in Table 3.20. Note that the mBO-algorithm is the best in terms of the complexity compared to the others. Also note that taking $c \gg 1$ will not be beneficial as in this case most connections will be considered multiple times. Note that this does not necessarily follow directly from the complexities, but imagine the situation where we divide the data set in n blocks. Then almost all connections will appear in multiples blocks. As already mentioned, having the same complexity does not imply that the running times are equal.

	mBO-alg	SBF-alg	Con-approx	Slope-approx
Complexity	$\mathcal{O}((nm + k) \log(nm))$	$\mathcal{O}(\frac{n^2}{c} m^2 + k)$	$\mathcal{O}(\frac{n^2}{c} m^2 + k)$	$\mathcal{O}(\frac{n^2}{c} m^2 + k)$

Table 3.20: Complexities of the algorithms and heuristics for the second problem.

Recall that the modified Bentley-Ottmann-algorithm was output-sensitive. This implies that for large values of k the complexity of the modified Bentley-Ottmann-algorithm might blow up. In fact, if $k = \Theta(n^2 m^2)$ the complexity of the modified Bentley-Ottmann-algorithm will be $\mathcal{O}(n^2 m^2 \log(nm))$, which is even worse than the complexity of the smart brute force-algorithm. Hence, in those cases the smart brute force-algorithm is preferred in terms of complexities. See Figure 3.21 for an example of a configuration of line segments for which $n = 6$ line segments result in $k = 15 = \Theta(n^2)$ intersections. Note that given our setup such situations are highly unlikely.

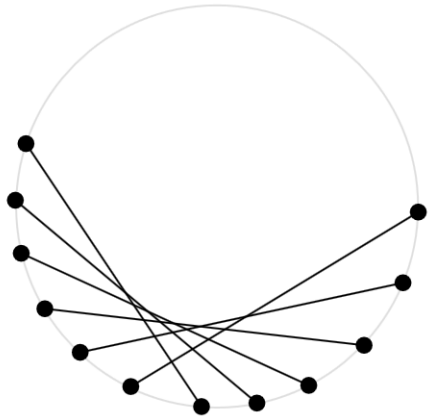

Figure 3.21: Six line segments with $15 = \frac{6(6-1)}{2}$ intersections.

3.4 Algorithm 3

The third problem revolves around connecting the cabinets with the fiber glass network. Mathematically this means that given a graph and two points, not necessarily on the graph, the two points have to be connected in the cheapest way possible. The cost of a path is defined by two cost parameters. The first is the cost per distance to route through the network, i.e., the cost per distance to go from one point in the network to another point in the network. The second cost per distance is the cost associated to connecting the points with the network. Note that the two costs are typically not equal. Routing through the network uses already existing networks, pipes and trenches, while connecting the cabinet to the network requires us to dig and create new trenches. Therefore, in practice the cost per distance to connect the points with the network will be much larger than the cost per distance to route through the network.

Also note that due to the two costs, the shortest connection point is not necessarily the cheapest connection. This is also shown in Section 2.4. In general however, the cost of digging will be much larger than the cost of routing through the network. Hence, we may assume that shortest connection point is also the point for which the connection has minimum cost. Moreover, the shortest connection to the network is in general favored over the least cost-connection as this is the easiest to realize, even though the possibly higher costs.

The third algorithm uses a modified version of the first algorithm to determine points in the network close to the points to connect and then uses Dijkstra's algorithm to find the least cost-path between the two points. Let us first discuss Dijkstra's algorithm and then explain how the third algorithm works.

3.4.1 Dijkstra's algorithm

In 1959 an article by Edsger Dijkstra was published ([33]) in which an algorithm is explained to find the shortest path between vertices in a weighted graph. A restriction on Dijkstra's algorithm is that all edges must have non-negative weight. A relaxed version of Dijkstra's algorithm is the Bellman-Ford algorithm ([34],[35]) which allows for

negative weights and only requires the non-existence of negative cycles, i.e., no cycles exist for which the sum of the weights is negative. We chose to use the algorithm by Dijkstra as there will be no negative costs involved in our problem, so there is no use in allowing them to exist. This will only give more complicated algorithms

During the algorithm we will distinguish two sets, one set X consisting of a partial spanning tree and one set Y with nodes not yet in the tree. The tree is said to be minimum spanning as it has minimum weight. Note that the two sets are disjoint. Given two vertices v, w the cost of edge $v - w$ is given by $c(v, w)$. Let us now give the pseudo-code of Dijkstra's algorithm. The algorithm will give two outputs with which a minimum spanning tree can be constructed, i.e., with which we can construct a spanning tree such that the sum of the weights is minimal.

Algorithm 3.22 Dijkstra's algorithm

Input: Connected graph $G = (V, E)$ with weights c given as weight matrix, source node s

Output: d a vector with distances from the nodes to s, p a vector with for every node its predecessor

Initialization: Initialize the distance $d(v)$ to every node v at infinity and the distance to the source node $d(s)$ at zero. Set $X = \emptyset, Y = V$.

1: **while** $Y \neq \emptyset$ **do**
2: Choose the vertex $u \in Y$ such that $d(u) = \min_{w \in Y} d(w)$
3: $Y = Y - \{u\}$, $X = X + \{u\}$
4: **for all** neighbours $v \in Y$ of u **do**
5: $d(v) = \min(d(v), d(u) + c(v, u))$
6: **end**
7: Update the predecessor of u in p
8: **end**

Note that a vector with predecessors is enough to construct a spanning tree, as we have a root node s and for every node we have the predecessor. The vector d then gives the weight between the root node and every other node. Also note that the algorithm finishes as the input graph is connected. Therefore, there is a minimum cost path between every pair of nodes. The weight matrix is a matrix with on position (i, j) the weight of the connection between nodes i and j and zero otherwise. Note that in the above $d(v)$ is a tentative distance and is the sum of the weights of the edges in the path between v and w. Finally note that the algorithm finds a minimum spanning tree. We can modify the algorithm by replacing "$Y \neq \emptyset$" by "$t \notin X$" in the *while*-loop to only find the least cost-path between s and t. The algorithm can also be modified to work for disconnected sets, the *while*-loop should then be replaced by ***while*** *not all neighbours of points in X are in X*. So, while the points in X still have neighbours which have not yet been visited, we continue our search. Note that in this case not all points can be reached, hence, the distance to those points equals infinity.

Let us now prove that the algorithm indeed gives a minimum spanning tree. In order to do this let $d(v)$ be the distance found by the algorithm and let $\delta(v)$ be the minimum

path-distance between s and v. Also let $X \subseteq V$ be as in the algorithm. We will now prove that the algorithm works by induction to the size of X.

Lemma 3.5. *For all $v \in X$ $d(v) = \delta(v)$.*

Proof. During the algorithm the size of X only grows, hence, the only time when $|X| = 1$ is when $X = \{s\}$. By construction $d(s) = 0 = \delta(s)$. Hence, for $|X| = 1$ the lemma holds.

Now suppose it holds for $|X| = n$, we now want to prove that it holds for $|X| = n+1$. Let u be the last vertex added to X and set $X' = X - \{u\}$. Note that we only need to prove that $d(u) = \delta(u)$ as for the other vertices, the situation did not change. So for the other vertices v we already have $d(v) = \delta(v)$.

By definition we already have $\delta(u) \leq d(u)$, so let us now prove $d(u) \leq \delta(u)$. Suppose P is the shortest path between s and u with weight $w(P) = \delta(u)$. The path first uses some vertices in X' and then leaves X' in order to get to u (as $u \notin X'$). Let xy be the first edge in P that leaves X'. Let $w(P_x)$ be the weight of the partial path between s and x. Then

$$w(P_x) + c(x, y) \leq w(P).$$

By the induction hypothesis we know that $d(x)$ is the minimum weight of a path between s and x. Hence, $d(x) \leq w(P_x)$ and

$$d(x) + c(x, y) \leq w(P).$$

We know that the node y is chosen by the algorithm, hence, $d(y) \leq d(x) + c(x, y)$. However, the algorithm chose u instead of y, therefore we have $d(u) \leq d(y)$. If we now use the above inequalities we get

$$d(u) \leq d(y) \leq d(x) + c(x, y) \leq w(P_x) + c(x, y) \leq w(P) = \delta(u),$$

which completes the proof. Hence, $d(u) = \delta(u)$. □

Note that we could also have started with X being empty for which the statement trivially holds and proceed from there.

Theorem 3.6. *Dijkstra's algorithm finds the minimum spanning tree.*

Proof. Apply the above lemma with $X = V$. □

Corollary 3.7. *If Dijkstra's algorithm gives $(v_1, \ldots, v_i, \ldots, v_n)$ as shortest path between v_1 and v_n, then $(v_1, \ldots, v_i)$ is a shortest path between v_1 and v_i.*

This follows directly from Dijkstra's algorithm.

3.4.2 Using Dijkstra to solve our problem

Recall that the third problem focuses on connecting a cabinet with the fiber glass network. This comes down to connecting two points with each other, using a network. The first point is the cabinet while the second point is the connection point with the fiber glass network. As there are costs associated to making the actual connection, the best option mathematically is not always the best option financially, i.e., the shortest path

is not always the least cost-path. Note that the network used to connect the two points does not necessarily need to be the fiber glass network. It is also possible to use a trench network or a street network for instance.

The location of the cabinets and the location of the connection points with the fiber glass network are fixed. However, we are free to choose the connection point of the cabinet with a network, which then will be used to route to the fiber glass network. We will make use of this fact in order to determine the best connection point. In Figure 3.23 we see the cabinet (denoted by 'cab'), the connection point to the fiber glass network (denoted by 'fib') and a (part of a) network (denoted by the blue lines). The red lines indicate possible connection points for the cabinet with the network. The solid red line cab-a indicates the shortest connection between the cabinet and the network. The dotted red lines indicate other possible connections starting from the cabinet.

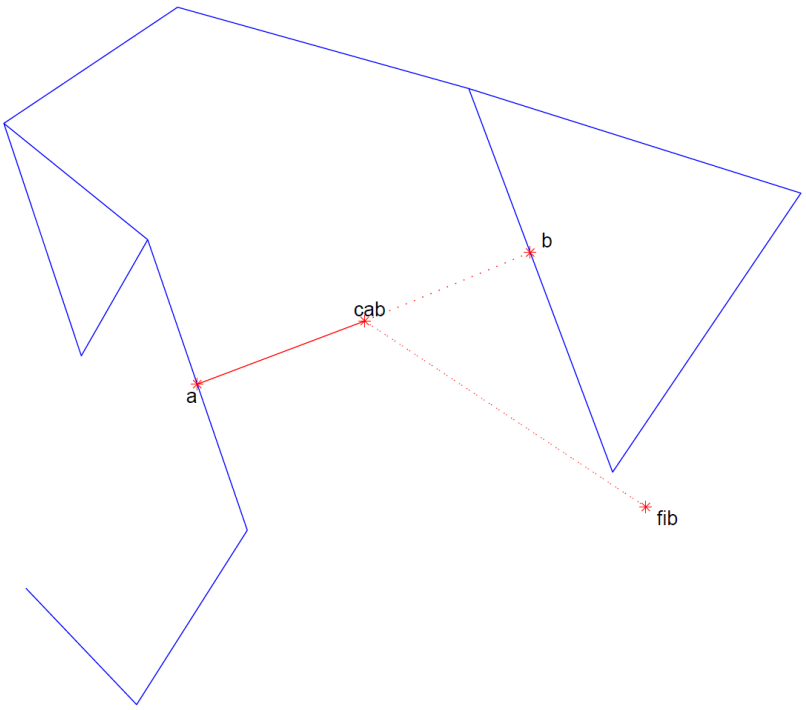

Figure 3.23: A network (blue), with the shortest connection to the cabinet (solid red) and two other possible connections (dotted red).

The distance between the cabinet and point a is 1.6070 units. However, the distance from the cabinet to point b equals 1.6076 units, which is only slightly larger. As we can see in the figure, it takes longer to route from point a to the fiber point, than it takes when the routing starts in point b. Therefore, when considering the costs, it might be more beneficial to connect to point b for the least total cost instead of point a. Note that for both options, we still need to connect the last part with the fiber glass network. We have also included a direct connection between the cabinet and the fiber glass network. This might be beneficial for some costs, in particular, if the cost of digging is smaller than the cost of routing through the network, the direct connection will always be the

connection of lowest cost. Note that in the above we have neglected the case that the cheapest connection between a point and a line segment is in general not the shortest, as proven in Theorem 2.60. In the following we neglect this and we will treat the case as if the shortest connection is also the cheapest. In practice such cases will also be neglected and a choice for the shortest connection will always be made.

The first problem we face is that Dijkstra's algorithm assumes we have a weight matrix. This matrix serves as both an adjacency matrix and as a matrix containing the weights of the edges. Position (i, j) denotes the length of the connection between i and j. Note that (i, j) is zero if there is no connection between the two vertices. In the second algorithm we created an array of the network, but we still not have a weight matrix. Therefore we have written a script which can generate both an adjacency matrix and a weight matrix. The running time of this script will not be included in the running time of the third algorithm as this is a feature of the network itself and it can be computed once outside of the algorithm and then used in further calculations. As between most nodes there is no connection, most entries in the weight matrix are zero and hence only a few entries are of interest to us. Therefore, we will save the matrix as a sparse matrix, that is, the entries in the matrix are saved using the corresponding indices. The zero-entries will not be saved. Note that this saves space if the number of non-zero entries is much smaller than the total number of entries in the matrix.

Even though sparse matrices use less space to save the matrices, they are not the most useful structures when running the algorithm. Therefore, we will modify a MatLab function by David Gleich ([36]) and use it to construct three vectors which carry all the information stored in the sparse weight matrix.

Another problem we face is that the data set need not to be connected in general. This can be due to the fact that the data set was generated by giving bounding x- and y-coordinates. However, in some cases the data set leads to disconnected connections. This is a restriction we have to deal with. Note that in practice there will not be isolated connections. In our case the data set does not give all connections due to which it seems like the network is disconnected. Consider Figure 3.24 for an example of a disconnected network.

Note that in some cases it is clear with which connections the isolated parts should be connected. However, in some situations this is not so clear. Note that detecting isolated connections requires the algorithm to run at least once. Apart from that, there is only a small fraction of the total connections which is isolated. Therefore, we will neglect such cases and run the algorithm. If the shortest connection is attained at an isolated component, we can use that to further route to the network or we can find another connection point which might give a smaller total cost. Note that using this isolated connection might require extra digging. See Section 3.4.3 and 4.3.1 for a more thorough analysis of such situations.

Given the weight matrix corresponding to the network, we can now determine what the best option to connect to the network is. We first apply a version of the first algorithm to determine a number of shortest connection points for both the cabinet and the connection point with the fiber glass network (this number can be given as input). For these connection points we need to determine the endpoints of the connection they lie on. The two points to connect are added to the weight matrix with appropriate distances to other points, this is dependent on the endpoints of the connections the

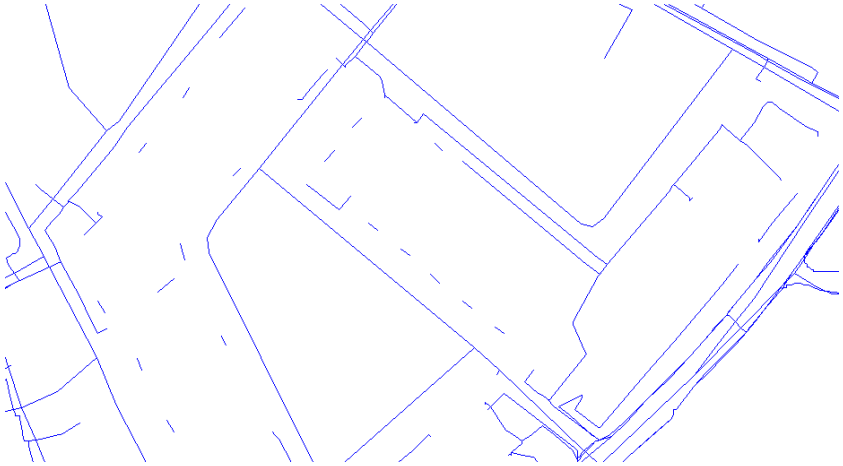

Figure 3.24: An example of a disconnected network. This part is from the data set of Venray.

points lie on. Dijkstra is then applied and we will get a least cost-path. Note that this least cost-path consists of three parts, two parts for the digging from the point to the network and one part for routing through the network. Together they give the total least cost.

Algorithm 3.25 Find the least-cost path between two points using a network

Input: A network, the points to connect and the costs of digging and routing, x the number of best connection points we want.

Output: The least cost-path between the two points

1: Compute cost of direct digging between the two points
2: Apply the first algorithm to find the x best connection points for both points with the network
3: Determine the endpoints of the corresponding connections
4: Determine the costs corresponding to the endpoints
5: Deal with possibly isolated connections
6: Apply Dijkstra's algorithm
7: Compute the path with the least cost

In Algorithm 3.25 we can find the pseudo-code of the third algorithm. Using the first algorithm we can find the x shortest connection points to the network. In order for these results to make sense, we demand that the connection points are attained on different line segments (not connections). Hence, every line segment can only give one 'shortest connection point'. Otherwise we can pick $x - 1$ points on the line segment arbitrary close to the shortest connection point and we will, in total, have x shortest connection points all attained at a single line segment. Demanding that the x points are all attained on different line segments takes care of this problem.

Note that Line 5 of the pseudo-code reads *Deal with possibly isolated connections*. It is possible that a connection point is connected with an isolated segment. These

situations have to be dealt with. In Section 3.4.3 we will talk about the restrictions of the algorithm and about ways to solve such situations. Note that if both points do not connect with isolated connections, this part will be skipped in the algorithm.

Note that it is possible to redefine the costs. At this moment the costs per meter is fixed. However, it might be possible that routing through the network costs more if the routing takes longer. This can rather easily be implemented at the end of the algorithm.

3.4.3 Restrictions of the algorithm

Initially the algorithm would find the least cost-path between the two points, however, under the assumption that we dig only two times, namely, once from the cabinet to the network and once from the connection point to the fiber glass network to the network. There are situations possible for which this is not the case and for which we want to create new paths, i.e., for which we want to dig more than two times.

Consider as an example Figure 3.26 drawn from the Dutch city Venray. For simplicity we have only considered the situation where we want to connect the cabinet (cab in the figure) with an intermediate point (point 6 in the figure). In the actual algorithm, the complete route is taken into account. As costs we have taken 25 per unit to dig and 3 per unit to route, given by c_{dig} and c_{route} respectively. Point 1 yields the shortest connection point and point 3 gives the connection point which gives the least cost-path. The total cost of connecting with point 3 and routing to point 6 is

$$11.76 \cdot c_{dig} + 20.33 \cdot c_{route}.$$

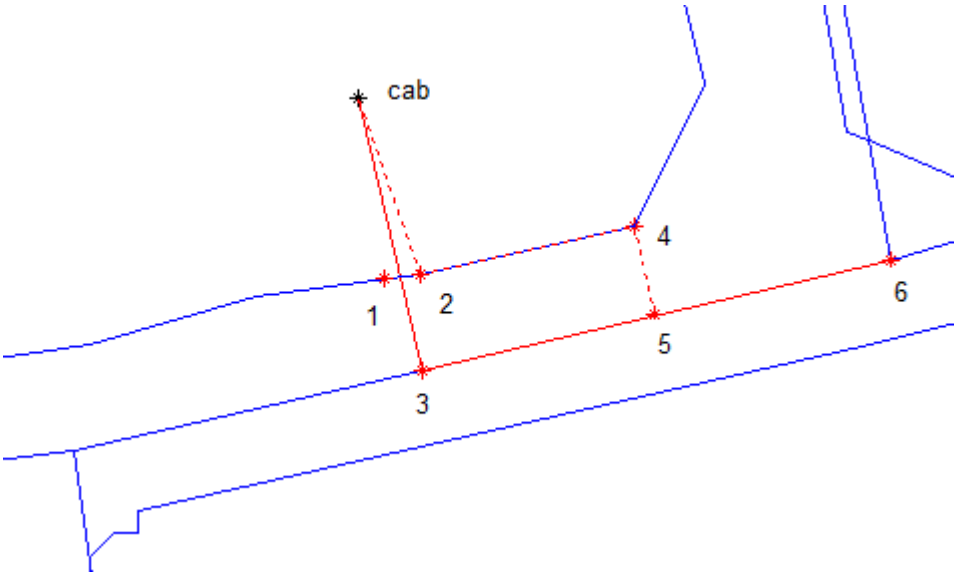

Figure 3.26: In solid red the least cost-path found by the algorithm: cab $\rightarrow 3 \rightarrow 6$. The dotted red line shows a path of lower cost: cab $\rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$.

However, this is not the least cost-path. Namely, a path which has a lower cost consists of digging to point 2, routing to point 4, digging to point 5 and routing to point

6. In total this will give a cost of

$$7.84 \cdot c_{dig} + 9.22 \cdot c_{route} + 3.87 \cdot c_{dig} + 10.24 \cdot c_{route} = 11.71 \cdot c_{dig} + 19.46 \cdot c_{route}.$$

Note that this situation has a lower cost for all possible non-negative costs c_{route} and c_{dig}. Hence, instead of digging to point 3 and routing to the network, it is better to dig to point 2, route to point 4 and then create a new trench between point 4 and point 5. Note that this is not necessarily the least cost-path as this is dependent on the actual costs (consider $c_{route} = 0$, then point 1 gives the least cost-path).

Associated with this is the fact that the original algorithm could not handle disconnected networks very well as Dijkstra will give a distance of infinity between two nodes in two different connected components. In practice disconnected networks will not happen. The fact that the data says the network is disconnected is a drawback of the data set. Either a connection is denoted which does not exist or not all connections are denoted in the data set. Solving this problem, results extra digging. Instead of the standard *dig-route-dig*-connection, we can also dig to the isolated part, route to another point in that part and then dig to the network. This then results in a *dig-route-dig-route-dig*-connection.

Summarizing, there are two problems which needs further consideration. The first is that in some cases it is cheaper to create new connections between two points instead of routing through the network. The second problem is that the cabinet or the connection point to the fiber glass network can be connected with an isolated segment of the network.

Solving the first problem is very hard. However, if we want to determine if digging is a better option than routing, we can determine for every connection its length and the distance between the two endpoints. If the associated cost of digging is smaller than the cost of routing, it is better to create a new connection between the endpoints instead of routing. However, this analysis should also be extended to groups of two or more connections and consider if digging in that case is better or not. This can then further be extended to the endpoints of the piecewise linear parts instead of the endpoints of the connection as a whole and finally we can even look at intermediate points on the line segments and connections. As we can see this problem becomes extremely complex extremely fast. Therefore we will not consider this problem. We will only consider paths in which we dig only at the beginning and at the end of the path.

An example of this problem was given in Figure 3.26. We saw that the path cab $\rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ was favored for some costs. This path was found using the graphical representation of the network. However, writing an algorithm which could solve it would be hard and would lead to complex algorithms.

The second problem can be solved in a few ways, namely:

1. Demand a connected network;
2. Before the actual algorithm, check for connectivity. Possibly connect the isolated parts to form a connected network;
3. Neglect isolated parts;
4. Connect isolated parts only if they might be needed in the least cost-path.

Note that the first option would solve this problem immediately, but also that this option is not practical. Data sets need not to be connected. Both as we generate the data set using minimum and maximum x- and y-values and because in some data sets isolated segments exist. This last problem can only be solved by removing isolated segments. An algorithm was constructed which does remove isolated segments. In Section 4.3.1 the results of this version are compared with those of the algorithm which do not remove isolated connections.

The second option is not practical as time will be spend on connecting isolated segments which might not even be needed for the least cost-path. Furthermore, for some isolated connections it is fairly straightforward to which segment they have to be connected, however, there are also isolated connections for which it is very hard to see with which connections we have to connect it. Let alone, to let the algorithm determine with which connection we have to connect it.

The third and the fourth option are rather similar. In both cases we allow isolated connections to exist. However, for the third option, we will neglect them if they show up as a shortest connection point. As we considered multiple 'shortest connection points' this will still give a path between the two points. The fourth option allows for connection with isolated connections, however, in these cases we will do extra digging to connect the isolated parts with the network. Of course this results in extra costs. Note however that in most cases this will be cheaper than it is to dig to the network at once without considering the isolated connections. In the last case, the algorithm will give a notification if extra digging is required for the least cost-path. In Section 4.3.1 we will compare the different possibilities we have.

3.4.4 Generating results

The locations of the shortest connection points are generated in a similar way as we did in the first algorithm, see also Section 3.2.2 and Theorem 3.1. The output of Dijkstra's algorithm consists of a vector with the costs from the starting point to every other node in the network and it gives a vector with the predecessor of each node. That is, if we have a rooted spanning tree, Dijkstra gives for every node to which it is connected the cost of the path and for every node, it gives the predecessor in that path. Note that the spanning tree is rooted at the starting point, hence the starting point has no predecessor.

Note that Dijkstra's algorithm works on the endpoints of the connections. However, most connections from the cabinet or fiber glass network to the actual network happen in the interior of a connection. Therefore, when the connection point is found, we have to route to one of the endpoints of the connection. Using these endpoints we can apply Dijkstra's algorithm. When we have found the least cost-path however, this is a path between the endpoints of the connections, while we were looking for the least cost-path between the two points. Cost-wise this can be taken care of by adding the already computed costs to connect the point with the connection and then route to the endpoint. In order to denote the connection point correctly, we have added an extra vector which took care of that. In the end this will give a vector which denotes the first point, the connection point, the least cost-path between the endpoints, the connection point and the second point. Note that the first and second point are the points we wish to connect with each other. In the case that we have to create a new connection, this

will also be denoted and we will give a notification to the user.

The least cost-path will be given as output. Furthermore we will give the least total cost and the coordinates of the endpoints of the connections we use together with the connection points.

3.4.5 Speeding up the algorithm

Their are a few possibilities to speed up the algorithm. The first is of course the application of the heuristics of the first algorithm. However, in Section 4.1.2 it is shown that there either is no significant decrease in the running time or the accuracy does drop significantly.

Dijkstra's algorithm itself cannot be improved significantly in terms of its running time and complexity. The version of Dijkstra's algorithm we use, uses heap data structures due to which it is faster than the algorithm as original constructed by Dijkstra. We can modify the input of Dijkstra's algorithm such that the running time decreases. This follows as the complexity of Dijkstra's algorithm is dependent on the number of nodes given as input (see also Section 3.4.6). As Dijkstra's algorithm finds the shortest path in a network, it will not consider nodes of degree one, i.e., it will not consider leafs, unless one of the points is connected with a leaf. Therefore we can prune the network and neglect the nodes of degree one with which the two points we are considering do not connect.

In the algorithm this comes down to determine which nodes have degree one, determine where the shortest connection points are for the cabinet and the fiber glass network, find the endpoints corresponding to the connection points, neglect them in our list of nodes with degree one and remove the resulting nodes from the weight matrix.

We can furthermore reduce the number of times we have to run Dijkstra's algorithm. Note that for both points we will find at most $2x$ endpoints of line segments of connections. Therefore, there are at most $4x$ possibilities. However, instead of running Dijkstra's algorithm $4x$ times, it is sufficient to run it only once if we add the two points to the network with some extra connections. Dijkstra's algorithm finds the least cost-path between two points. So if we are given two endpoints of two line segments, determine the least cost-path between these points and then add the costs for digging and routing to those endpoints, we get a total cost of that path. However, if we add the two points we are considering to the network, we can run Dijkstra on one of those points. For each point we add extra connections between that point and the at most $2x$ endpoints. The costs we give to these extra connections are the costs it would take to dig and route from the point to the endpoint. Note that this will indeed give the least cost-path. Also note that in this case we only have to run Dijkstra's algorithm once.

The number of 'shortest distance' (x in the previous sections) can also be used to speed up the algorithm. As the shortest connection does not necessarily lead to the least cost-path (see Figure 3.23 for instance), we may need to consider multiple connection points. Taking $x = 1$, forces the algorithm to take the shortest distance between the point and the line and continue from there. Using larger values of x will possibly lead to path of lesser cost. If x is taken very large, the running time is increased drastically. This follows as in this case, most connection points of x will not be of interest. In the analysis of the algorithms, we will consider different possibilities for x.

Another possibility to reduce the running time is restricting the total network to a network which is of interest to us. Similar as pruning the network and neglecting leafs, we will only consider as smaller part of the network. This part is determined by the two points we are considering and connections which lie close to those points. This reduced network is then given as input to the first algorithm which determines the points in the network closest to the points we want to connect. Note that this might be an improvement as less connections have to be considered. Also note that this requires extra computations. The restriction of the network to a smaller network relies on the bounding rectangles. This will make sure that all connections within a certain distance to the points considered, will be taken into account.

As sketched there are a number of possibilities to reduce the running time of the algorithm. In Section 4.3.1 we will compare both the running times and the results of the different possibilities and we will make a decision on which we will use.

3.4.6 Complexity of algorithm 3

For the complexity of the algorithm we can look at the complexity of the individual parts. Let n be the number of connections, m the typical number of line segments per connection and let x the number of shortest connection points we will consider. In terms of the first algorithm, x behaves similar as the number of active points k in Section 3.2.5. As complexity for this part we therefore find $\mathcal{O}(nmx)$. The points found all need to be processed and the corresponding connection endpoints have to be determined. For these points the length of the connection between the connection point and its endpoints has to be determined. This is done using the line segments which make up the connection. As we have x connection points, hence at most $2x$ endpoints and a maximum of m line segments, this gives a term $\mathcal{O}(mx)$ to the complexity. Note that in the worst case scenario, all these line segments are isolated, hence we need to apply the first algorithm again, as this only gives an extra term in the $\mathcal{O}$-notation, we may neglect it. Afterwards we have to apply Dijkstra's algorithm. In the worst case secanario we have to do this $2x$ times, that is, every connection point lies on a connection which has two endpoints, resulting in $2x$ possible endpoints in total. Note that in most cases not all $2x$ endpoints will be different.

Dijkstra's algorithm has to be run only once as already mentioned. The algorithm as originally constructed has a complexity of $\mathcal{O}(n^2)$. This followed as for every vertex we considered every other vertex not yet in the set and computed the distance to those points. In the case of a dense graph (a graph with almost all possible connections appearing) the complexity of Dijkstra will in fact be $\mathcal{O}(n^2)$. However if we use a so called heap structure instead of an unordered list of vertices, we can decrease the complexity. A heap is a data structure similar to a Binary Search Tree, however, this time the key of a node is always smaller than or equal to the key of its parent. Again we have n vertices, however, now the time it takes to consider all other vertices is $\mathcal{O}(\log n)$ (due to the usage of heaps). This gives a total time complexity of

$$\mathcal{O}(n \log n).$$

Note that the implementation we have used, uses the version of the heap structure, due to which this complexity applies to our case. Also note that in the case of a dense graph,

the usage of heap structures will result in a higher complexity and therefore possibly a higher running time. Finally notice that there is no m in this situation, as only the length of the connection as a whole is of interest and not the length of the piecewise linear parts.

Keeping track of the least total cost and finding the shortest path between the two points can all be done in $\mathcal{O}(n)$ time. In total this gives a complexity of

$$\mathcal{O}(nmx + nm + n\log n + n) = \mathcal{O}(nmx + nm + n\log n).$$

Note that in general $m \ll n$, however for smaller networks the two can be similar.

Chapter 4

Results

Now that we have the algorithms, it is time to compare their results both in terms of the running time and in terms of the accuracy. One has to be careful when comparing the running times of the algorithms, as this depends on more than the algorithm alone. When we run the algorithm on two computers such that the first computer can do twice as many computations per time unit as the second one, then it is highly likely that the algorithm will finish sooner on the first computer. Apart from the computer itself, i.e., the hardware, also the processes that are running on the computer can influence the running times. Consider a computer with no processes running on the background and a computer on which we run multiple programs at the same time. Then the algorithm will most likely finish earlier on the computer with no additional processes running.

Let us now compare the results for the different algorithms separately. We are given a few data sets with which we can run and test the algorithms. Some of these data sets will be used in order to get the results.

Note that in the following, the running times shown are median running times over a number of runs. This follows as the running times of the algorithm depend on the hardware you are working with and on other processes running on the computer. Even if, on the same computer, an algorithm is run twice directly after each other the running times do not need to be equal. Using the median will nullify the effects from bad runs of the algorithm. The reason we take the median instead of the average can be illustrated with an example. Suppose we have an algorithm and let it run ten times. Of those ten times, nine times the running time is around one second. However, one of the runs gave a running time of eleven seconds. When taking the average, we end up with a running time of two seconds, while we saw that one second gives a more realistic image of the running time of the algorithm. The running time of eleven seconds could be explained by other processes also running at the same time.

4.1 Algorithm 1

Let us compare the results of the first algorithm, the node-approximation and the cut-approximation. In the following when we write cut-100, we mean the cut-approximation where all parts of a connection longer than 100 meters are cut in smaller pieces until the longest part is smaller than 100 meter. In the comparison we will consider the

cut-approximation for multiple cut-values. One of which is rather small, one of which can be viewed as normal and one relatively large cut-value. This last one is to check whether the running time of the algorithm will tend to that of the node-approximation if the cut-value becomes very large. But first we will compare the results of the node-approximation with its vectorized version.

In Section 4.1.3 we will generate some random data sets and let the algorithm run. We will consider the running times for random data sets. Afterwards we will compare a randomly generated graph with an actual data set and discuss their results in terms of both the distance and the running time.

4.1.1 Further vectorization or not?

In this section we compare the results of the node-approximation and a further vectorized version of the node-approximation. We have chosen to consider the node-approximation as this was an algorithm we actually use, but it had the least subtleties in the vectorization. Recall that vectorizing means replace a calculation done by a *for*-loop with a calculation using a vector (or more generally a multi-dimensional array). Both the algorithm and the cut-approximation had some technicalities due to which it was harder to vectorize them further.

In the following we refer to the vectorized version as the node3D-approximation, since this version uses three dimensional arrays. We have compared the results and the running times of both algorithms for different data sets. As expected the results of both algorithms were equal. Therefore, we will only consider the running times in the following. We will consider a few data sets and draw conclusions from thereon. Note that we will only consider the running times of the algorithm itself, that is, we do not write data to a file.

In Table 4.1 we see the running times of the algorithm for the first data set. We have considered the Dutch city Venray. With only 3,365 connections and 55 active points, we can regard this as a rather small data set.

Acceptation distance	Algorithm	
	Node	Node3D
0	0.2856	0.2951
5	0.2835	0.2948

Table 4.1: Venray: Median running times of a hundred runs in seconds.

The table shows the median running times of a hundred runs of the algorithm. As we can see the results found with the node-approximation and with its vectorized version are similar, though the node-approximation is slightly faster. We also see that the difference between the running times for the different acceptation distances is negligible. This might follow from the fact that only a few active point have a distance smaller than five meters to the network.

The second data set we have considered is that of the city Zwolle. Zwolle has almost fifteen thousand streets and 110 active points, hence we can think of this set as being of normal size. In Table 4.2 we again see the median running times of a hundred runs of the

algorithm. Note that the node-approximation is significantly faster than its vectorized version, this holds for both acceptation distances.

Acceptation distance	Algorithm	
	Node	Node3D
0	1.8296	2.2016
5	1.8011	2.1853

Table 4.2: Zwolle: Median running times of a hundred runs in seconds.

The last, and also the largest, data set we will consider is that of the Dutch city Den Haag (The Hague), with almost 39,000 streets and 570 active points. In Table 4.3 we again see the median running times of a hundred runs of the node- and node3D-approximation. We see that the percentage wise difference is smaller in this case than in the previous. However, the vectorized version is still slower than the non-vectorized version. The difference of the running times for the different acceptation distances is again rather small.

Acceptation distance	Algorithm	
	Node	Node3D
0	5.8225	6.4787
5	5.7350	6.4787

Table 4.3: Den Haag: Median running times of a hundred runs in seconds.

As we can see, the node-approximation is either as good as its vectorized version or it is faster. This can be explained by the way the node3D-approximation works. As all endpoints of the parts of a connection are considered at the same time, also all calculations are done at the same time and we end up with a three dimensional array with distances. As for each active point we need to know which endpoint gave the shortest distance, we sort the array according to the distances and take the smallest per active point. These distances are than compared with the optimal distances up until then. We also us a function which finds unique entries in a vector. Using this function we know which distances belong to which active point and to which part of the connection. Both functions are combined to find the corresponding indices. However, both functions also take relatively long. The sorting for instance takes $\mathcal{O}(k \log k)$, with k the number of active points. Recalling that this functions replaced a *for*-loop with complexity $\mathcal{O}(k)$, we see that the complexity of the vectorized version becomes $\mathcal{O}(nmk \log k)$. For small data sets with only a small number of active points, this will not lead to big differences. However, if larger data sets are considered, the node3D-approximation will run slower than the node-approximation.

As we already mentioned, the distances found using the node-approximation and the vectorized version were equal. Therefore we can base the choice of the version of the approximation completely on the running time. We have seen that for some data sets (those with a small number of active points and a small number of streets) the results are near equal. However, if the number of active points grows, so does the difference of the running times of both versions. The same holds if more streets are considered as we

need to sort more vectors which takes longer. Taking these two points into account, we advise to use the node-approximation as this is faster than the node3D-approximation.

Note that as the node3D-approximation was slower than the node-approximation, as a consequence both the algorithm and the cut-approximation will not have a faster vectorized version. This follows as the complexity and the number of cases we should take into account is the smallest for the node-approximation. If vectorization does not lead to improved results for the node-approximation, it will also not lead to improvements in the case of the algorithm and the cut-approximation. The multiple cases which are possible make the algorithm harder to vectorize. The cut-approximation has an extra *for*-loop due to which it is even harder to vectorize. Therefore, we advise to not further vectorize the algorithm and its approximations as this will most likely not lead to improved results in terms of the running time.

4.1.2 Algorithm versus heuristics

Let us now take a closer look at the differences between the algorithm and its approximations. We will compare the algorithm with the node-approximation and with the cut-approximation for three different cut-values. One rather small value, one *normal* value and a relatively large value. The normal cut-value will be in the order of the cut-value used in the already existing tool built by TNO. We expect the running time of the cut-approximation for the large cut-value to tend to that of the node-approximation. We will also have to compare the distances found by algorithm with those found by the approximations as they need not to be the same. Again we let the algorithm and its approximations run a hundred times, both for an acceptation distance of zero and an acceptation distance of five, and take the median of the running times afterwards. Those will then be compared to each other. As cut-values we take 5, 15 and 100 meters. The output will be compared to those of the algorithm with acceptation distance zero as this gives the optimal distances. The same data sets as in the previous comparison will be used this time. Note that the algorithm and the node-approximation will only give the same result for an active point if the shortest distance is attained at an endpoint of a piecewise linear part. The cut-approximation will then give the same result as well.

Note that higher cut-values can give better distances than smaller cut-values. For higher cut-values most line segments will be cut in fewer, but longer parts. However, more parts does not imply better results. Suppose we have a line segment going from $(0, 0)$ to $(100, 0)$ and the optimum is attained at $(0, 51)$. Then the cut-50 approximation will lead to two line pieces and fairly accurate results. The cut-51 approximation however will give us three line pieces and less accurate results. However, in general a smaller cut-value will lead to more accurate results.

First we will consider the city Venray, which has 3,365 streets and 55 active points. The results for both acceptation distances were the same, therefore we have shown the all results in Table 4.4. It turned out that for two active points the results were equal for the algorithm and all approximations. We see that the results of the cut-approximation tend to that of the node-approximation as the cut-value increases. The maximal error of 35,200 % for the node-approximation comes from a corresponding distance of 0.03 meter for the algorithm and 11.91 meter for the node-approximation, which explains the high percentage wise error. As expected the error in terms of the distance is relatively small

when considering small cut-values. Note that the errors shown below are the difference in the distances found by the approximations compared to the distances found with the algorithm for the active points.

	Node	Cut-100	Cut-15	Cut-5
Max. % error (×100%)	352.46	352.46	14.70	68.52
Avg. % error (×100%)	10.51	9.92	1.04	1.44
Max. error (m)	72.43	27.94	5.71	2.31
Avg. error (m)	13.95	10.49	1.32	0.30

Table 4.4: Venray: Results for both acceptation distances.

Let us now compare the running times of the algorithms and its approximations. In Table 4.5 we see the results found. Even though the results of the cut-5-approximation are acceptable, its running times is far greater than that of the algorithm itself. Therefore there is no use in using the cut-5-approximation. We see that the difference in running times for the two different acceptation distances is very small. That is expected as the results were equal as seen above. Only the node-approximation and the cut-100-approximation give quicker results than the algorithm, however they give less accurate results. In this case we suggest to use the algorithm with acceptation distance zero.

Acceptation	Algorithm				
distance	Alg1	Node	Cut-100	Cut-15	Cut-5
0	0.5883	0.2856	0.4797	1.7540	4.7767
5	0.5737	0.2834	0.4788	1.7295	4.6947

Table 4.5: Venray: Median running times of a hundred runs in seconds.

Let us now consider the city Zwolle again for acceptation distances zero and five. In Tables 4.6 and 4.7 the results are shown. For only nine active points the algorithm gave different results for the two acceptation distances. For two points the approximations gave the same results as the algorithm. We see that again the cut-5-approximation gives accurate, close to optimal, results, while the node-approximation and the cut-100-approximation give far from optimal results. An average error in the distance of 10 meters is found for the node-approximation, which is a lot compared to the cut-approximations.

	Node	Cut-100	Cut-15	Cut-5
Max. % error (×100%)	131.52	77.65	25.89	4.96
Avg. % error (×100%)	5.97	4.37	1.02	0.26
Max. error (m)	102.17	38.95	5.79	1.86
Avg. error (m)	10.03	7.29	1.50	0.29

Table 4.6: Zwolle: Results for acceptation distance zero.

When comparing the running times of the algorithm with those of the approximations, we get Table 4.8. As expected the running times for acceptation distance five are smaller than that for acceptation distance zero. This is what we expected as some

	Node	Cut-100	Cut-15	Cut-5
Max. % error (×100%)	131.52	77.65	25.89	4.96
Avg. % error (×100%)	5.97	4.37	1.03	0.30
Max. error (m)	102.17	38.95	5.79	2.60
Avg. error (m)	10.03	7.30	1.51	0.37

Table 4.7: Zwolle: Results for acceptation distance five.

points need not to be taken into account after some iterations. However, the difference is small, especially for the node-approximation and the cut-100-approximation. For the other cases the difference is larger, but not significant.

Acceptation distance	Algorithm				
	Alg1	Node	Cut-100	Cut-15	Cut-5
0	3.94	1.82	2.65	6.85	17.60
5	3.68	1.80	2.63	6.73	17.04

Table 4.8: Zwolle: Median running times of a hundred runs in seconds.

As we saw in the two previous cases the algorithm with acceptation distance zero is most favored above the other option. Let us now conclude by considering Den Haag. For only 22 out of 570 active points the algorithm gave different results for acceptation distances zero and five. As we can see in Table 4.9 and 4.10 the results for both acceptation distances are near equal again. We do see that there are some percentage wise errors which are very large. This is possible if the algorithm gives a fairly small result, but the node-approximation gives a result of a few meters. Percentage wise this leads to large results. We see that the maximal error in meters for the cut-5-approximation becomes larger for an acceptation distance of five, while the other errors roughly stay the same. For the percentage wise errors, we see that acceptation distance five gave different results for the cut-5-approximation. This follows as the distances were accepted sooner, while an improvement would have been found if the calculations for that point continued. The average errors do differ a bit for both acceptation distances, the percentage wise average errors are smaller for acceptation distance zero, than for acceptation distance five, as expected.

	Node	Cut-100	Cut-15	Cut-5
Max. % error (×100%)	432.00	409.42	178.56	130.60
Avg. % error (×100%)	11.55	7.94	2.46	1.10
Max. error (m)	108.35	38.13	6.60	2.23
Avg. error (m)	15.57	9.63	1.47	0.36

Table 4.9: Den Haag: Results for acceptation distance zero.

Let us finalize the analysis with the comparison of the different running times of the algorithm and the approximations for this data set. The results can be found in Table 4.11. Again the results shown are the median running times of a hundred runs.

As expected the running times are greater than those of the previous data sets considered. This has to do with the fact that this data set was larger. We do see

	Node	Cut-100	Cut-15	Cut-5
Max. % error (×100%)	432.00	409.42	434.07	434.07
Avg. % error (×100%)	11.55	7.95	3.12	1.85
Max. error (m)	108.35	38.13	6.60	4.06
Avg. error (m)	15.57	9.64	1.51	0.45

Table 4.10: Den Haag: Results for acceptation distance five.

Acceptation distance	Algorithm				
	Alg1	Node	Cut-100	Cut-15	Cut-5
0	9.8327	5.8225	7.8027	25.4626	65.6841
5	8.7423	5.7350	7.5712	23.0907	59.7484

Table 4.11: Den Haag: Median running times of a hundred runs in seconds.

the same things happening as before though. The cut-5- and cut-15-approximation give running times longer than that of the algorithm, while the running time of the cut-100-approximation tend to that of the node-approximation. Both the cut-100- and the node-approximation are faster than the algorithm, however, as seen above, they do give results far from optimal. The difference between the two acceptation distances is larger than before. Given the running times we would suggest to use the algorithm with acceptation distance zero. It is also possible to use an acceptation distance of five as this still gives pretty accurate results, while being faster. However, this mainly depends on the demands set by the user.

As for all three data sets the algorithm with acceptation distance zero gave the most accurate results in reasonable time, we suggest to use the algorithm with acceptation distance zero in all cases. We must make a small remark however, whether the algorithm or an approximation is used and which acceptation distance is chosen depends on the user. If speed is the main issue, while accuracy matters less, we suggest the node-approximation with a larger acceptation distance. However, if accuracy is most important, which it will be in most cases, we suggest to use the algorithm with acceptation distance zero. Note that using the cut-approximation is not favored as either the running time is longer than the algorithm, or it is faster than the algorithm, but slower and as accurate as the node-approximation. Thus using the cut-approximation will not yield better results than the algorithm (or the node-approximation), both in terms of the running times and in terms of the actual distances found. This also has to do with the complexity of the cut-approximation. In Section 3.2.5 we already saw that the number of parts each edge is cut in, is of influence on the complexity. As that number is inversely proportional to the cut-value, we see that smaller cut-values lead to more computations and hence lead to longer running times.

An explanation for the small differences between the different acceptation distances in the results can be explained by the way the algorithm is set up. We have constructed the algorithm, and consequently also the approximations, in such a way that only after the calculations for a connection have finished, there will be checked if there are distances below the acceptation distance. This means that if in the first part of the connection a distance is found which is below the acceptation distance, then the other parts of

the algorithm are still considered. We now see that this can only go wrong if there are multiple connections which have a distance to the active point smaller than the acceptation distance. This typically can happen at points where connections intersect. The main difference however is that in further calculations the point is not further considered. Hence, the running time decreases.

We have compared the algorithm with its approximations, let us now compare the algorithm we have constructed with the already existing tool built by TNO. The program uses a cut-10-approximation similar to our cut-approximation. As we have seen above both the cut-5- and the cut-15-approximation took longer than the algorithm and gave less accurate results, therefore we conclude that also the cut-10-approximation will take longer than the algorithm and will give less accurate results. Therefore we suggest to use the algorithm in the program instead of the currently used cut-10-approximation.

4.1.3 The results for random graphs

Instead of the result found for the Dutch cities shown in the previous section, we can also construct random graphs and analyze the results for these data sets. Instead of creating arbitrary random graphs, it is also possible to derive random graphs from real data sets. Both will be done in the following.

First we will consider the behaviour of the algorithm as the size of the input grows. We hope to be able to say something about the scaling of the running times of the algorithm. We will do this using random graphs, i.e., we will randomly generate a data set and use that as input for the algorithm. We use random data sets as this way it is easier to let the size of the data set grow. After this, we will consider random data sets derived from real ones. That is, we will use the endpoints of the real data set, but the connections will be random.

As seen in Section 3.1.2 the data sets derived from street patterns were setup in a certain way. It was for instance not possible for two connections to intersect other than in their endpoints as this would lead to an extra node in the network. If random connections are considered there will most likely be intersecting connections. Another difference is that for the real data sets each connections was subdivided in smaller piecewise linear parts. For the random data sets there will be no such subdivision of connections. Another difference is that for real data sets two points close to each other are more likely to be connected than two points further apart, for random graphs this does not need to hold.

As we set up the random data sets such that each connection is a straight line, not subdivided in smaller parts, this will have some consequences for the behaviour of the algorithm and the approximations. Checking the different parts each connection is divided in reduces to checking only one part, namely the connection itself. Note however that the node-approximation only checks for the endpoints of the line segments. This implies that the results will most likely differ a lot from that of the algorithm. Another thing to note is that the running time will be slightly lower compared to those of real data sets of the same size as the connections are not subdivided in multiple smaller parts.

Behaviour of the algorithm given scaling input

We want to say something about the behaviour of the running times of the algorithm if the input changes in size. As mentioned in Section 3.2.5 the complexity of the algorithm is $\mathcal{O}(nmk)$ with n, m and k defined as before. This implies that the running time of the algorithm will increase with a factor if the input grows with a (possibly different) factor. So considering more connections or more active points will result in longer running times. In order to say something about the behaviour of the running times we will first consider four random, but special cases. These data sets are generated using a random generator in MatLab.

- 25,000 streets, 500 active points;
- 250,000 streets, 500 active points;
- 25,000 streets, 5,000 active points;
- 250,000 streets, 5,000 active points.

As we see that there is a scaling factor of ten in the above cases, we expect the running time of the algorithm to also increase according to a factor. For the analysis we will use the algorithm together with three different acceptation distances, namely 0, 5 and 10. Afterwards we will test the limits of our algorithm as we will increase the number of streets and the number of active points even further.

The analysis will be done both for the algorithm and the node-approximation. Even though both have the same complexity, they may scale differently. We will not consider the cut-approximation in the following. For small cut-values the running times were worse than that of the algorithm, while for large cut-values, they were larger than the running times of the node-approximation. Apart from that, the length of the connections can be higher for the random data sets than for the real data sets. Resulting in even worse results for the cut-approximation.

For the four possible cases mentioned above we found running times as shown in Table 4.12. The first thing we notice is that the running times for the node-approximation are near equal for all acceptation distances. This might seem strange at first, but recall that the node-approximation only checks for the distance between the active point and the endpoints of the line segments. As the line segments are not subdivided in smaller parts, there are only two points for every line segment to check. The probability that the distance is then smaller than five or ten meters is small. Therefore even for higher acceptation distances, the node-approximation has to check every line segment for almost all active points. We do see that for acceptation distance zero the running time for the algorithm is longer than for the node-approximation. This is as expected, as we have seen this for real data sets as well.

We see that there is some scaling factor between the different data sets. For acceptation distance zero this is approximately five and ten for increasing the number of active points and the number of streets by a factor ten, respectively. For the node-approximation these scaling factors are approximately seven and eight. Note that there are also some scaling factors for the other acceptation distances, but these are harder to draw conclusions from as no line segment is subdivided in smaller piecewise linear parts, which could heavily influence the results, particularly for the node-approximation.

Data set		Alg1			Node		
# streets	# active points	Acceptation distance			Acceptation distance		
		0	5	10	0	5	10
25,000	500	5.10	5.01	4.89	3.09	3.07	3.06
25,000	5,000	25.57	7.54	6.12	22.75	22.68	22.62
250,000	500	47.17	45.61	39.46	25.31	25.30	25.30
250,000	5,000	198.43	147.83	119.71	185.47	184.54	183.92

Table 4.12: Running times for random data sets in seconds.

Let us now increase the number of streets and the number of active points even further and see what happens with the running times of the algorithm. Again we will consider the algorithm and the node-approximation. We will consider acceptation distances zero and five, as we had done for the real data sets.

In Table 4.13 we find the results for the larger data sets. With *Alg1-0* the running times for the algorithm for acceptation distance zero are meant, the meaning of the others follow accordingly.

# streets	# active points	Alg1-0	Alg1-5	Node-0	Node-5
25,000	50,000	236	225	169	168
250,000	50,000	2,064	1,461	1,649	1,634
2,500,000	500	346	276	299	293
2,500,000	5,000	2,421	1,098	2,210	2,031
2,500,000	50,000	23,021	11,069	22,517	22,061

Table 4.13: Running times for large random data sets in seconds.

Again we see that the node-approximation has a shorter running time than the algorithm for acceptation distance zero. However, as discussed before, in the current setup the node-approximation gives a somewhat twisted result opposed to the real data sets. This also explains the differences when acceptation distance five is used. For the algorithm we see that this does lead to decreased running times. However, the node-approximation gives roughly the same running times for both acceptation distances. For some data sets this even leads to longer running times for the node-approximation than for the algorithm if larger acceptation distances are used. Note that this is partially a results of our setup. This time we find scaling factor around ten for the algorithm. Both for an increase of the streets by a factor ten and for an increase of the active points by the same factor. For the node-approximation we again find scaling factors around seven and eight, respectively.

Note that the running times for the largest data set are extremely large. With a running time of nearly six and a half hours this is not a calculation one would like to do several times in a row. However, the scaling of the running times still shows a linear scaling with respect to the number of streets and the number of active points. Thus, if the input size would be increased even further, the running time would not blow up quadratically or worse, but the running time would increase linearly instead.

One way to take care of the large running times for the largest data sets is to split the very large set in multiple smaller data sets. Consider for instance a data set of 2,500,000

streets and 5,000 active points. It will be more efficient to split the data set in ten smaller sets of (approximately) 250,000 streets and 500 active points and do the calculations for each set separately. This will lead to a total running time of approximately 500 seconds, while doing the calculations for the complete set takes almost 2,500 seconds.

Following this we see that using such large data sets will not be of practical use. In addition it is highly unlikely that we encounter such data sets. The largest data set we have considered was that of Den Haag, which 'only' had around 40,000 streets and 500 active points.

Semi random data sets compared to real data sets

To conclude the analysis of the first algorithm we will compare semi random data sets with real data sets. In order for the comparison to make sense, we construct a random data set based on a real one. Hence the name semi random data sets. We set up the semi random data set using the endpoints of the connections of the real data sets. The connections however will be made randomly. The number of connections will be the same for the random data set and the real data set. Also the coordinates of the active points will be the real coordinates. Those will not be randomly generated.

We will again consider acceptation distances zero and five and we will compare both the running times and the distances found. Again only the algorithm and the node-approximation will be considered as the running times of the cut-approximation are longer than that of the algorithm and the node-approximation for the various cut-values.

We first considered the city Den Haag. Let us first compare the results in terms of the distances. In Table 4.14 we have shown the distances found for the various active points and compared those with the distances found by the algorithm for acceptation distance zero. With *DH-5* we mean the results for Den Haag with acceptation distance five compared to those of acceptation distance zero and with *Rand-0* and *Rand-5* we mean the results compared to those of the random data set for acceptation distance zero and five respectively.

	DH-5	Rand-0	Rand-5
# smaller distances	0	557	402
# larger distances	45	13	167
# equal distances	525	0	1
Average distance	7.76	0.20	2.54

Table 4.14: Den Haag: Distances found for the different data sets compared to that found by the algorithm for acceptation distance zero. The results are in meters.

Note that we have not compared the results for the node-approximation in the above. We have skipped these results as the distances found this way are always longer than those found with the algorithm. As we only check for the nodes, the results differ even more due to our setup. We see that for DH-5 there is no active point which has a smaller distance than found with the algorithm. This is of course as expected. For the random graph this is different. Most of the active points have a smaller distance to the random graph than they have to the real data set, which explains the difference. While for the real data set the active points have an average distance of 7.63 meter to the network,

for the random data set this is 0.20 meter. Note that for acceptation distance five there is an active point with equal distance as for the algorithm. This distance was attained at a node, thus not at the interior of the line segment.

The difference between the average distance between the active points and the network for both data sets might be explained by the length of the connections. Note that for the real data set, the shortest distance is probably attained at a line segment between two nodes which are close to the active point. This has of course to do with our setup. For the random data set, even if two points are far apart, their connecting line might be close to an active point. Therefore, let us consider the typical length of the connections, both for the real data set and for the semi random data set.

In Table 4.15 we have shown the minimum, maximum and average difference of the x- and y-coordinate of the two endpoints of the connections. This is also done for the distance between the two endpoints. This is done for both the real data set and the random data set.

	Den Haag			Random		
	Δx	Δy	Distance	Δx	Δy	Distance
Maximum	5,058.67	4,863.57	7,017.44	19,352.02	11,769.59	21,017.02
Minimum	0.14	0.09	6.80	0.05	0.06	13.11
Average	67.88	67.09	100.94	4,585.02	3,303.59	6,201.02

Table 4.15: Den Haag: Length of the connections and differences in x- and y-coordinates in meters.

Note that the table does not say anything about individual connections, but it says something about the average connections. We do see that for the real data set of Den Haag the average street length is around a hundred meters. For the randomly generated data set this is more than six kilometers. The same behaviour is observed for the minimal distance and the maximal distance of the connections and the differences between the x- and y-coordinates. Note that the fact that the minimum distance between two points for the random data set is larger than the minimum distance for the real data set supports our claim. This also explains why there are that many active points with a smaller distance to the randomly generated data set than to the real data set. Let us finalize this observation for Den Haag by considering the running times. In Table 4.16 we can see the running times for the different possibilities.

	Alg1-0	Alg1-5	Node-0	Node-5
Den Haag	9.34	8.30	5.60	5.50
Random	7.48	3.42	5.02	4.98

Table 4.16: Den Haag: Median running times of a hundred runs in seconds.

The results we see are in accordance with those shown in the previous tables. As the length of the streets is longer for the random data set, active points are likely to be closer to the random network than to the real data set. Which explains the differences between the two data sets for higher acceptation distances. The difference between the results for acceptation distance zero can be explained by the fact that the random data set was constructed in such a way that all connections were only straight lines. This opposed

to the real data set where some connections were subdivided in smaller piecewise linear parts. This also supports the fact that for the node-approximation, there is only a small difference for both acceptation distances.

Let us finish with a similar observation, but this time for the data set of Venray and a randomly generated data set on the endpoints of connections of Venray. In Table 4.17 we show the results of the comparison of the results found. We have compared the distances found for the active points using the algorithm for both data sets. The results are compared with those for acceptation distance zero for the real data set. Note that the table shows a similar behaviour as for Den Haag.

	Venray-5	Rand-0	Rand-5
# smaller distances	0	50	42
# larger distances	1	5	13
# equal distances	54	0	0
Average distance	8.81	1.39	2.85

Table 4.17: Venray: Distances found for the different data sets compared to that found by the algorithm for acceptation distance zero. The results are in meters.

Again we have not included the observation for the node-approximation as all distances found using the node-approximation will be larger than those found with the algorithm. We see that of the 55 active points, 54 have equal distance to the network for acceptation distance zero and five. Again we observe that for most active points the distance to the random network is shorter than to the real network. Also the average distance the active points have to the network is different for the real data set and for the random data set. For acceptation distance zero we found an average distance of 8.79 meter for the real data set. For the random data set this is 1.39 meter. Again this will most likely have to do with the length of the connections. However, let us first illustrate the situation. In Figure 4.18 and Figure 4.19 we see the networks we are considering. In the first figure we clearly see a street pattern corresponding to Venray, while in the second no such pattern can be observed. We do see that especially the middle of Figure 4.19 is dense. Towards the edge of the data set we see that less connections appear compared to the real data set. Note that both observations support the fact that shorter distances are found for most active points, as most active points lie around the center of the data set. For active points near the edges of the data set, it is more likely to encounter larger distances in the semi random data set than in the real data set.

Let us now take a look at the typical length of the connections of the two networks. We do this in the same way as above. We consider the difference in x- and y-coordinate of the endpoints of each line segment, and we compute the distance between endpoints of the connections. The results for Venray and its randomized data set can be found in Table 4.20. We see that the typical distances for the real data set are smaller than those of the random data set. This has of course to do with the fact that in the real data set two points close to each other are more likely to be connected than two points further apart. For the random data set this is of course not the case. Two points close to each other are equally likely to be connected as two points further apart.

Let us finalize this section with a comparison of the running times for both data sets. Again this will be done for both the algorithm and the node-approximation and for both

Figure 4.18: Venray.

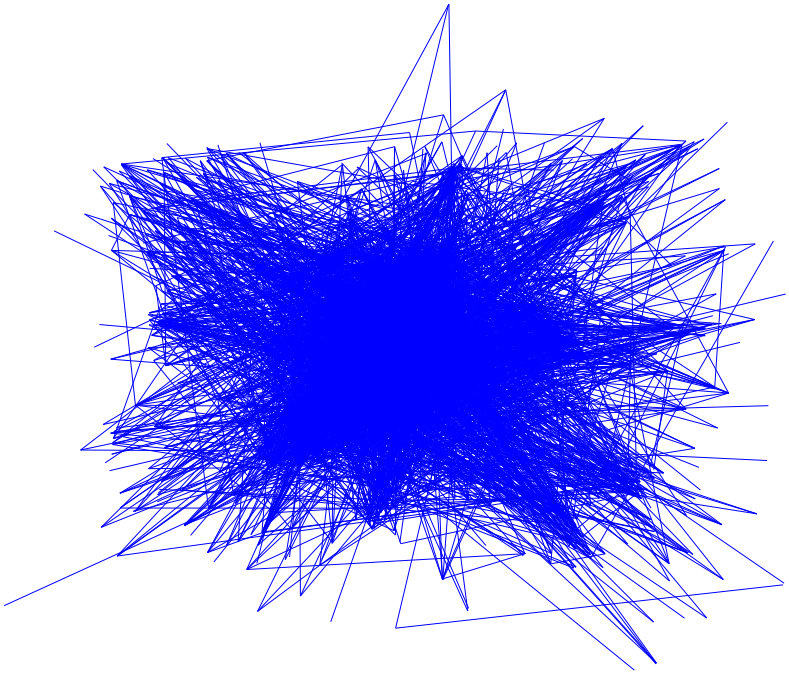

Figure 4.19: Venray, randomly generated.

	Venray			Random		
	Δx	Δy	Distance	Δx	Δy	Distance
Maximum	4,070.23	2,135.71	4,401.99	10,103.61	6,428.11	10,635.94
Minimum	0.08	0.05	6.88	3.25	0.92	53,29
Average	92.88	86.17	141.86	2,056.39	1,328.08	2,656.90

Table 4.20: Venray: Length of the connections and differences in x- and y-coordinates in meters.

an acceptation distance of zero and an acceptation distance of five. The running times can be found in Table 4.21.

Data set	Alg1-0	Alg1-5	Node-0	Node-5
Venray	0.59	0.58	0.29	0.29
Random	0.43	0.37	0.23	0.22

Table 4.21: Venray: Median running times of a hundred runs in seconds.

We see that the results are in accordance with the previous observations. The running times for Venray are close to each other for all acceptation distances. This has to do with the fact that only a few active points do not need to be considered after some iterations due to the acceptation distance. However, this means that most points are considered in every iteration, therefore only a small decrease in running time is observed. As already mentioned, the semi random data set was constructed in such a way that every connection is a straight line. This opposed to the real data set where a connection can be subdivided in smaller piecewise linear parts. This explains the results and the differences found between the running times. Also the fact that the average distance of connections of the random data set is larger than that of the real data set lead to minor differences in the running times.

To conclude, we see that randomly generated data sets give distances between the network and the active points which are in general smaller than those of the real data sets. For larger acceptation distance this results in a decrease in running time. We also see that the fact that the connections which were made randomly resulted in longer connection for the semi random data sets compared with the real data sets. We saw this in Table 4.15 and 4.20. Figure 4.18 and 4.19 support this observation. Thus the fact that the real data sets have in general shorter connections results in longer running times than for random data sets.

To conclude, our algorithm does also work for random data sets as we expected. However, the real data sets we have considered have some properties which random data sets do not need to have. The main property being that no two edges can intersect other than at their endpoints. This then results in shorter connections than for the random case. For acceptation distance zero, the difference between the running times of both data sets is small. The difference follows most likely from the fact that no connection was subdivided in smaller linear parts for the random case. For higher acceptation distances, the running times for the random data set decrease significantly, while for the real data set a smaller decrease is observed. This has to do with the fact that most active points have a smaller distance to the random network than to the real

network. This again follows from the restrictions we have for the real network.

4.2 Algorithm 2

Let us now compare both algorithms and the heuristics with each other. This will be done both in terms of the results found and in terms of the running times. We will only denote the number of intersections found and the resulting number of streets. Recall that two intersecting line segments result in four non-intersecting line segments. We again have considered different data sets and we will draw conclusions from there on. First we will consider if it is efficient to subdivide the data sets in multiple smaller data sets (blocks) in the smart brute force-algorithm. After that we will compare the modified Bentley-Ottmann-algorithm with the smart brute force-algorithm and its heuristics. We will conclude this section with an analysis for random graphs.

4.2.1 Subdivide the data set in smaller blocks or not?

Let us first look if it is beneficial to subdivide the data set in smaller blocks. We have done this in the smart brute force-algorithm, let us now determine if is is beneficial and if so, what number of blocks is recommended. To do this we will subdivide the data set in 5, 20, 100 and 500 blocks. Note that not subdividing the data set in multiple blocks, leads to the usage of 1 block. For each number of blocks, we can do our analysis. In all runs we saw that the results found were equal for the different number of blocks. Therefore, we can draw a conclusion solely based on the running times. Note that in the modified Bentley-Ottmann-algorithm, we did not subdivide the data set in smaller blocks. We will also consider the running times for this algorithm in order to support our claim of not subdivide the data sets in the modified Bentley-Ottmann-algorithm.

The first situation we will consider is that of the Dutch city Venray. We will consider a data set derived from a street network with 2,212 connections and a data set derived from a trench network with 7,589 connections. The running times for the smart brute force-algorithm and the modified Bentley-Ottmann-algorithm are shown in Table 4.22. While for the first problem we took the median over a hundred runs, we now take the median over twenty runs. This is due to the time the algorithms take.

	1 block	5 blocks	20 blocks	100 blocks	500 blocks
Smart brute force	63.37	34.28	29.87	30.19	71.81
Mod. Bentley-Ottmann	12.39	14.32	17.13	31.97	67.97

Table 4.22: Venray: Median running times of twenty runs in seconds.

For the smart brute force-algorithm we see that the use of blocks does decrease the running time if only a few blocks are used. However, if we subdivide the data set in a large number of blocks (e.g. 500) we see that the running time starts increases. In fact, the running time when using 500 blocks is longer than the running time when the data set is not subdivided in smaller blocks. This is most likely due to the fact that for this number of blocks, many connections appear in multiple blocks. Hence, they have to be checked multiple times. Moreover, the extra calculations done in order to construct the blocks take longer in this case. So we see that for this data set the usage

of blocks is preferred if we subdivide the data set in 20 to 100 blocks. For the modified Bentley-Ottmann-algorithm, we see that the running time is lowest when the data set is not subdivided into multiple smaller blocks. A possible explanation can be the fact that the modified Bentley-Ottmann-algorithm does not check every pair for an intersection, but only the line segments which are neighbours in the sweep line. Hence, doing the extra calculations to subdivide the data set in blocks will only increase the running time.

Let us now consider the next data set, namely that of the Dutch city Tilburg. The street network consisted of 8,523 connections, while the trench network consisted of 39,594 connections. The results we found for this data set can be found in Table 4.23. We again used the same number of blocks and again we took the median running times of twenty runs.

	1 block	5 blocks	20 blocks	100 blocks	500 blocks
Smart brute force	1,232.26	303.32	248.46	157.27	242.43
Mod. Bentley-Ottmann	46.21	58.80	88.44	158.27	292.98

Table 4.23: Tilburg: Median running times of twenty runs in seconds.

Again we see that the usage of blocks is preferred over using no additional blocks for the smart brute force-algorithm. However, we do see that in this situation subdividing the data set in a hundred blocks is more beneficial than subdividing it in twenty blocks. This is different from what we found for Venray, however, the running times for twenty blocks and hundred blocks were very similar for Venray. We again observe that the running times for the modified Bentley-Ottmann-algorithm are the lowest when the data set is not subdivided in smaller blocks. We also observe that, as was the case for Venray, if the data set is subdivided in many blocks, the running time of the modified Bentley-Ottmann-algorithm will surpass that of the smart brute force-algorithm.

Let us finalize this analysis by considering the Dutch city Almere and its surroundings. The area we are considering consists of 12,060 streets and 27,835 trenches. In Table 4.24 the results found are shown.

	1 block	5 blocks	20 blocks	100 blocks	500 blocks
Smart brute force	1,572.87	553.79	223.00	135.79	224.25
Mod. Bentley-Ottmann	43.16	100.37	121.16	122.24	279.92

Table 4.24: Almere: Median running times of twenty runs in seconds.

Again we see that subdividing the data set in smaller blocks is beneficial for the running time in the case of the smart brute force algorithm. We also see that in these cases the best number of blocks to subdivide the data set in, lies around a hundred blocks. In all cases considered, more blocks resulted in more connections appearing in multiple blocks and extra calculations to subdivide the data set in these multiple blocks. If an extra block leads to only a small decrease in the number of pairs we have to check, it might be not beneficial with respect to the extra calculations we have to do.

Note that these cases do not prove that a hundred block is the optimum number of blocks to subdivide the data set in. In fact, it will be very hard to determine the optimal number of blocks to subdivide a given data set in. This number can differ from one data set to the other. This again has to do with the gains of using more blocks compared to

the extra calculations needed to divide the data set in these blocks. Moreover, even if two data sets have the same number of connections, this does not imply that the optimum number of blocks is equal for both. This has to do with the way the connections are clustered within the data set. If all connections are spread uniformly over the interval, subdividing the data set in blocks will be much more beneficial opposed to when almost all connections are clustered in a small region.

To conclude, we see that for all three data sets, subdividing the data sets in multiple blocks is beneficial to not subdividing the data set. Hence, supporting our claim of doing so for the heuristics as well. The number of blocks to divide the data set in, lies around a hundred. Therefore, in the following we will subdivide the data sets in 75 blocks and in 125 blocks and then do our analysis. Note that only for the smart brute force-algorithm and its heuristics we will subdivide the data set in blocks. For the modified Bentley-Ottmann-algorithm we saw that such subdivision had a negative influence on the running time.

4.2.2 Modified Bentley-Ottmann versus smart brute force

We will compare the modified Bentley-Ottmann-algorithm with the smart brute force-algorithm and its corresponding heuristics. For the heuristics we will also look at the number of intersections found as they not necessarily find all intersections. Again we have considered different data sets and for every region considered, we have taken both the street pattern and the trench pattern. This is due to the fact that we need two data sets as input. Also note that for the block-approximation we can subdivide the data set in multiple smaller blocks, however this number is not fixed. Note that in the previous section we found that subdivide in a hundred blocks was optimal for the data sets considered. We will choose one value which is slightly smaller and one value slightly larger than hundred. That way we can better observe the behaviour of the running times around this value. Now let the block-number be the number of blocks the data sets are subdivided in, as block-numbers, we will use 75 and 125.

In Section 3.3.8 we introduced abbreviations for the algorithms and the heuristics. As already mentioned, we use mBO-alg, SBF-alg, con-alg and slope-alg as abbreviations for the different algorithm and approximations. Note that for the slope-approximation we considered the slopes of the two general line segments. We would only check if the corresponding two connections intersect if the corresponding slopes differ by more than a constant. In the following we will consider two constants, namely $\pi/3$ and $\pi/6$, corresponding to an angle of $60°$ and an angle of $30°$. We will denote this as slope-60-alg and slope-30-alg respectively.

Let us first consider the city Venray again. Again we have two data sets, a street network with 2,212 connections and a trench network with 7,589 connections. Let us first consider the number of intersections found by the modified Bentley-Ottmann-algorithm, the smart brute force-algorithm and its approximations. Note that we will not consider multiple block-numbers as the results for the different block-numbers are equal. In Table 4.25 we find the number of intersections found for Venray.

We see that the approximation-algorithms find around half of the intersections. We do see that the slope-30-approximation finds more intersections than the slope-60 does. However, only a few more intersections (ten percent of the total intersections) are found.

	mBO-alg	SBF-alg	con-alg	slope-30-alg	slope-60-alg
# intersections found	7,849	7,849	4,140	4,487	3,709
Percentage of total	100%	100%	52.74%	57.17%	47.25%

Table 4.25: Venray: Number of intersections found.

Let us now consider the running times. In Table 4.26 we see the running times found. For completeness we also gave the running times of the smart brute force-algorithm and its approximations if only a single block is used and the running times of the modified Bentley-Ottmann-algorithm when multiple blocks are used.

Block-number	mBO-alg	SBF-alg	con-alg	slope-30-alg	slope-60-alg
1	12.66	59.15	45.96	53.77	50.14
75	25.58	28.83	9.07	15.21	11.34
125	31.59	46.67	11.17	16.66	11.87

Table 4.26: Venray: Median running times of twenty runs in seconds.

We do see that the running time of the heuristics with block-number 75 or 125 is shorter than that of the modified Bentley-Ottmann-algorithm with block-number 1. However, as we saw in Table 4.25 the results of the approximations are far from optimal. With only a slightly longer running time, the modified Bentley-Ottmann-algorithm gives far better results. We also see that the smart brute force-algorithms takes longer than the modified Bentley-Ottmann-algorithm. This is of course as expected. For this data set the modified Bentley-Ottmann-algorithm is preferred.

Let us now again consider the city Tilburg. Recall that we had 8,523 connections in the street network and 39,594 connections in the trench network. The number of connections found for this data set can be found in Table 4.27, while the running times can be found in Table 4.28.

	mBO-alg	SBF-alg	con-alg	slope-30-alg	slope-60-alg
# intersections found	28,086	28,086	17,425	19,149	16,735
Percentage of total	100%	100%	62.04%	68.18%	59.58%

Table 4.27: Tilburg: Number of intersections found.

Block-number	mBO-alg	SBF-alg	con-alg	slope-30-alg	slope-60-alg
1	43.82	998.10	927.88	930.08	941.34
75	169.54	124.70	61.90	89.80	78.83
125	181.41	121.97	50.29	80.68	68.01

Table 4.28: Tilburg: Median running times of twenty runs in seconds.

Again we see that the modified Bentley-Ottmann-algorithm finds all intersections. The approximations do find more intersections than they did for the previous data set, however, still only sixty to seventy percent of the intersections are found. We also see that the modified Bentley-Ottmann-algorithm is the fastest among the algorithms

tested. While for the previous data set, the approximations of the smart brute force-algorithm were faster than the modified Bentley-Ottmann-algorithm, for this data set this is not the case. This also follows from the complexity as already discussed in Section 3.3.8. Again we see that subdividing the data set in smaller blocks has a negative influence on the running times of the modified Bentley-Ottmann-algorithm. The running times of the smart brute force-algorithm and its approximations for only 1 block, are longer than the other running times. This is of course as we expected. This data set supports our observation made for the first data set, namely that the modified Bentley-Ottmann-algorithm is preferred.

Let us finalize this analysis by considering the Dutch city Nijmegen. The area we are considering consist of 15,058 connections in the street network, while it has 97,732 connections in the trench network. In Table 4.29 we again find the number of intersections found for the different algorithms and approximations. In Table 4.30 we find the corresponding running times for different block-numbers.

	mBO-alg	SBF-alg	con-alg	slope-30-alg	slope-60-alg
# intersections found	69,814	69,814	36,723	42,795	35,414
Percentage of total	100%	100%	52.60%	61.30%	50.73%

Table 4.29: Nijmegen: Number of intersections found.

Block-number	mBO-alg	SBF-alg	con-alg	slope-30-alg	slope-60-alg
1	167.02	3,902.45	3,651.77	3,712.74	3,681.67
75	308.72	767.69	323.13	468.96	400.72
125	354.88	772.91	254.65	432.49	366.00

Table 4.30: Nijmegen: Median running times of twenty runs in seconds.

Again we observe that the modified Bentley-Ottmann-algorithm with block-number 1 has the shortest running time. We also see that the running times for the modified Bentley-Ottmann-algorithm are larger than those found for the previous data sets, however, this follows from the fact that we are considering a larger data set than before. Note however, that the scaling is differently for the modified Bentley-Ottmann-algorithm and for the smart brute force-algorithm. For the first we find a scaling factor around four, with respect to the data set of Tilburg, while for the smart brute force approach we found a scaling factor of more than six. This scaling factor is obtained given the fact that the data set is subdivided in multiple smaller blocks. Note that the running times for the smart brute force-algorithm for block-number one once again prove that subdividing the data set in smaller blocks is preferred in these cases. Note that the running times are over an hour. Once again we see that the heuristics do have a shorter running time, but they also find less intersection than the smart brute force-algorithm and the modified Bentley-Ottmann-algorithm.

We saw for all three data sets that, among the exact algorithms, the modified Bentley-Ottmann-algorithm was the best. It has the shortest running times while still being exact. Not subdividing the data set in multiple blocks is the best for the modified Bentley-Ottmann-algorithm. However, even if the data set is subdivided in multiple blocks, the modified Bentley-Ottmann-algorithm still has acceptable running times.

Note that for some data sets the approximations have shorter running times than the modified Bentley-Ottmann-algorithm, however, a significant number of the intersections is not found. The smart brute force-algorithm finds all intersections, however, it does take longer than its approximations. Note that the connection-approximation has a shorter running time than the slope-approximation. This might have to do with the way we calculate the angle between the general line segments. In order to do this, we compute for both general line segments the arctangent of the slope and then we take the difference between them. This gives the desired results. However, computing an arctangent of a number takes in general relatively long.

We saw that subdividing the data set in multiple blocks was beneficial for the running times of the smart brute force-algorithm and its heuristics. However, in most cases there was only a small difference between the different block-numbers. Furthermore, in some cases, using 125 blocks instead of 75 resulted in lower running times, while in others in resulted in shorter running times. From this we can conclude that using the blocks will indeed result in shorter running times. It is however hard to draw any conclusions on the number of blocks we should subdivide each data set in. A possible explanation might have to do with the number of connections. If the number of connections we have is large, using an extra block will in general influence the running time a lot positively. However, if every block has a relatively low number of connections, using an extra block will in general not lead to a significant decrease of the running time, it might even lead to an increase. Note that this heavily depends on the topology of the network. In the case that most connections are clustered, subdividing the data set in blocks will most likely result in a single (or a few) blocks with the clustered connections. In those cases, using extra blocks will most likely not lead to significant decreases in the running time.

4.2.3 The results for increasingly large data sets

In this section we will further increase the number of connections we are considering and we will look at the running times of the algorithms. Afterwards we will give an advice on which algorithm to use. Note that the data sets we will consider are far greater than those of any practical use. We use real data sets, however, we will take the street and trench network of larger areas than we normally would. One could think of data sets from multiple cities instead of one, or a data set of a larger city including smaller villages in its surroundings.

In the following we will only consider the modified Bentley-Ottmann-algorithm and the smart brute force-algorithm. The approximations of the smart brute force-algorithm will not be considered as not all intersections are found, while the running times are longer than those of the modified Bentley-Ottmann-algorithm. The smart brute force-algorithm, despite the long running times, will be considered in order to make a comparison with the running times of the modified Bentley-Ottmann-algorithm. We will, apart from the block-numbers 1, 75 and 125, also consider a larger block-number of 175. This way we hope to see the effect of increasing the number of blocks on the running time. In all data sets considered, the modified Bentley-Ottmann-algorithm found the same intersections as the smart brute force-algorithm. Hence, in the following we will only consider the running times.

First we will consider the two cities Tilburg and Breda and the villages in between

them. The area we considered is thirty kilometers wide and nine kilometers long, giving a total area of approximately 270 square kilometers. This results in 24,422 connections in the data set from the street network and 129,012 connections in the trench data set. The running times can be found in Table 4.31. As we expect the running times to be larger than for the previous data sets considered, we will take the median of only a few runs (typically around five) and draw conclusions from there on. This is allowed as we are less interested in the precise running times and more interested in the differences between both algorithms.

Number of blocks	1	75	125	175
modified Bentley-Ottmann	183.36	373.85	452.05	520.80
Smart brute force	3,904.37	762.74	649.33	687.19

Table 4.31: Tilburg and Breda: Median running times of five runs in seconds.

We see that the running times for the modified Bentley-Ottmann-algorithm are slightly larger than those found for the city Nijmegen. For the smart brute force-algorithm the running times are slightly smaller than they were for Nijmegen. Note that there still is a significant difference between the running times of the modified Bentley-Ottmann-algorithm and the running times of the smart brute force-algorithm. Even if multiple blocks are used. The results found are somewhat suprising as they are similar to those of Nijmegen, while more connections are considered. This may have to do with some clustering of connections. Using the blocks and the smart tricks as mentioned before, the running times of the smart brute force-algorithm will not blow up significantly.

Let us now consider the capital of the Netherlands, Amsterdam, and some smaller villages surrounding it. In total this gives 215,769 connections in the trench network and 50,703 connections in the street network. The running times can be found in Table 4.32

Number of blocks	1	75	125	175
modified Bentley-Ottmann	720.80	902.35	1,080.55	1,108.88
Smart brute force	17,255.60	1,756.41	1,423.15	1,473.68

Table 4.32: Amsterdam and its surroundings: Median running times of five runs in seconds.

Again we see that the running times for the modified Bentley-Ottmann-algorithm are the shortest when the data set is not subdivided in smaller blocks. The running times of the smart brute force-algorithm are longer when no blocks are used. Note that the running times of the smart brute force-algorithm are still longer, even when multiple blocks are used. When the data set is not subdivided in multiple smaller blocks, the smart brute force-algorithm may take as long as four hours. Such running times are not suitable for practical purposes. Even when the data set is subdivided in smaller blocks, the algorithm takes more than twenty minutes. This too is to long for practical purposes.

We will conclude this chapter with a data set derived from the southwest part of the Netherlands, including the provinces Zeeland and Noord-Brabant. In total we will

consider 253,113 streets and 524,588 trenches. In practice data sets of this portions will not be considered as multiple cities are included in this set and this will take longer than running the algorithm for the individual cities separately. The running times found are shown in Table 4.33.

Number of blocks	1	75	125	175
modified Bentley-Ottmann	4,624.87	7,273.83	7,925.26	10,035.91
Smart brute force	36,682.43	9,124.52	6,464.52	6,181.74

Table 4.33: The southwest of the Netherlands: Median running times of three runs in seconds.

Note that the running times of the modified Bentley-Ottmann-algorithm are once again the shortest. Hence, giving more prove that the modified Bentley-Ottmann-algorithm without subdividing the data set in smaller blocks, is the best choice to determine the intersections of two networks. We also see more prove that subdividing the data set in smaller blocks is beneficial for the running times of the smart brute force-algorithm. Note that for this data set the running times of the smart brute force-algorithm for 125 blocks are shorter than that of the modified Bentley-Ottmann-algorithm when blocks are used. However, since the running time is still greater than that of the modified Bentley-Ottmann-algorithm with no additional blocks, it is still better to use the modified Bentley-Ottmann-Algorithm.

Summarizing, we saw for all data sets that the modified Bentley-Ottmann-algorithm gave the best running times. These times were the smallest when the data set was not subdivided in multiple smaller blocks. For the smart brute force-algorithm however we found that the running times decreased when the data set was subdivided in multiple smaller blocks. This follows as we have to pairwise check every pair of connections. Subdividing the data set in blocks then results in less pairs which have to be considered. Using too much blocks will result in an increased running time due to the extra computations that have to be done. As both algorithms find all intersections, we can conclude that the modified Bentley-Ottmann-algorithm is the best algorithm to use. When the data set is not subdivided in smaller blocks, this will give results the fastest.

Recall that our goal was to create an algorithm which would give results within minutes at most. This goal was met. For every network, we only have to run the algorithm once to obtain a general network where every intersection in the old network becomes an endpoint in the new network. We found that both the smart brute force-algorithm and the modified Bentley-Ottmann-algorithm found all intersections. The latter however, doing so in less time. Therefore we suggest to use the modified Bentley-Ottmann-algorithm. We strongly advice against the use of the heuristics constructed as the running times are at best similar to those of the modified Bentley-Ottmann-algorithm, while only a fraction of the intersection is found.

If the size of a data set increases, we expect the running time of the modified Bentley-Ottmann-algorithm to increase less than the running time of the smart brute force-algorithm. Note that the usage of blocks was beneficial for the running times of the smart brute force-approach. Note that it is faster to run the algorithm multiple times for a smaller algorithm than once for a larger algorithm. For instance, running the algorithm for Venray (10,000 connections) 26 times is faster than running determining

the intersections for the data set of Amsterdam (260,000 connections). This may seem counterintuitive. However, the difference might be explained that in these cases no connections are checked multiple times and no extra computations are needed.

As mentioned before, we cannot prove that the modified Bentley-Ottmann-algorithm works. However, in the analysis above we saw that for all data sets considered the modified Bentley-Ottmann-algorithm does find the same intersections as the smart brute force-algorithm does. Hence, we can conclude that the modified Bentley-Ottmann-algorithm as constructed by us does find all intersections.

4.3 Algorithm 3

Let us now consider the results for the third algorithm. First we will determine which approach as sketched in Section 3.4.3 is the best. Afterwards we will consider the running times of the algorithm and draw conclusions from there on. We will finalize the section with some random graphs and the results found by the algorithm for them.

In the following x will denote the number of 'shortest distances' we have to calculate for both points. The costs will be denoted by c_{dig} and c_{route} for digging and routing respectively. Note that if x is taken too small, it is possible (due to the isolated connections) that no solution is found.

The points we want to connect can be chosen at random within the boundaries of the given data set. The costs are taken as 25 euro per meter for digging and 3 euro per meter for routing through the network. These values are are similar to the actual costs involved with such problems.

4.3.1 Configurations of the algorithm

As we have already mentioned, there are several configurations for the algorithms we have to consider. The first configuration we will look at revolves around connecting the points with the network. The second configuration revolves around restricting the network to a smaller network. This smaller network can then be used to find connection points and can be used as input for Dijkstra's algorithm. Apart from restricting the network, we can also chose to include leafs in the analysis or to remove them. This last case can be preferable as Dijkstra's algorithm depends on the number of nodes given as input, which includes the leafs of the network. As input we can also give the number of 'shortest points' we have to look for for both the cabinet and the connection point to the fiber glass network. In the previous section this number was referred to as x. We will first consider the first configuration and then the second one. Both will be done for x-values 5, 10 and 15. The points we give as input to the algorithm will be randomly generated. For the first option we will consider the following configurations:

- Deleting isolated connections from the network;
- Neglecting isolated connections from the network;
- Allow extra digging from isolated segments to the rest of the network.

For the second option, we will consider:

- Not restricting the network and considering all endpoints;
- Not restricting the network and considering all endpoints but the leafs;
- Restricting the network and considering all endpoints within the restriction;
- Restricting the network and considering all endpoints but the leafs within the restriction.

The restriction of the network in the last two cases, happens via bounding rectangles. Connections will only be considered if they are within a certain distance of the points. This distance is an input variable, hence we will consider multiple variables for that.

In the following, if we determine the best configuration for the first option, we will fix the configuration for the second option. Similarly, if the second option is considered, the first configuration will be fixed. Note that the two options and the configurations associated to them are independent of each other. Therefore we are allowed to do this. In all cases we will consider x-values of 5, 10 and 15.

The first configuration: Isolated connections

As already mentioned, demanding a connected network will most likely not be possible. Therefore, we have three options for the isolated connections. We can neglect them and only connect with connections of the connected network. Instead of neglecting them, we can also delete the isolated connections. Finally we can also allow disconnected connections to exist. We may even connect with isolated connections, if we then dig to the connected network. This last approach might be preferable in terms of the cost.

Let us again first look at the city Venray. The general network of Venray has 23,628 connections. The running times can be found in Table 4.34.

	$x = 5$	$x = 10$	$x = 15$
Delete isolated connections	6.84	6.94	6.95
Neglect isolated connections	3.36	3.38	3.40
Extra digging from isolated connections	3.42	3.47	3.57

Table 4.34: Venray: Median running times for fifty pairs in seconds.

Note that the running times for neglecting the isolated connections or using them and create new connections are very similar. The running times for the version where isolated connections are deleted is larger than the other two. Apart from the running times we also have to consider the results the algorithms found. As we saw before, heuristics which are very fast, but produce far from optimal results, are not favored. This is a similar situation. If an algorithm finds a path fast, but the associated cost is higher than for other algorithms, other algorithms might be preferred. In Table 4.35 we have shown the average costs for the pairs of points considered. Note that we take the mean and not the median of the costs. This is because this time minor differences are important. Taking the median of the results will give the same median cost for each algorithm.

As we can see, the least average cost is obtained when we may connect with isolated connections and allow extra digging. The least cost happens for $x = 15$. However, when

	$x = 5$	$x = 10$	$x = 15$
Delete isolated connections	6,811.89	6,208.23	6,208.10
Neglect isolated connections	6,811.89	6,208.23	6,208.10
Extra digging from isolated connections	6,238.00	6,199.94	6,199.81

Table 4.35: Venray: Average costs for the fifty pairs of points considered in euros.

considering the individual line segments, we see that for only one pair of points, $x = 15$ gave a smaller cost than $x = 10$. When we allowed extra digging, we have found 5 cheaper paths for $x = 10$ compared to the paths found using $x = 5$. When we neglect or delete isolated connections there are 6 respectively 4 paths of smaller cost. In the worst case this led to a difference of 700 euros in the cost, however, in other cases the difference was only a few euros. For this data set we advice against the use of the algorithms which neglect or delete isolated connections. Neglecting isolated connections leads to similar running times, but worse costs. Deleting the isolated connections will lead to similar costs, but the running times are far greater than those of the other two algorithms.

Let us now consider the general data set for the Dutch city Tilburg. This network has in total 96,851 connections. Again we considered the three possibilities for the algorithm and we have used x-values 5, 10 and 15. The running times can be found in Table 4.36 and the average costs can be found in Table 4.37.

	$x = 5$	$x = 10$	$x = 15$
Delete isolated connections	22.46	22.30	22.40
Neglect isolated connections	6.93	6.95	7.23
Extra digging from isolated connections	6.90	6.93	6.93

Table 4.36: Tilburg: Median running times for fifty pairs in seconds.

	$x = 5$	$x = 10$	$x = 15$
Delete isolated connections	14,997.71	14,988.93	14,983.76
Neglect isolated connections	14,997.71	14,988.93	14,983.76
Extra digging from isolated connections	14,997.71	14,988.93	14,983.76

Table 4.37: Tilburg: Average costs for the fifty pairs of points considered in euros.

As we can see in the above tables, the average costs are equal for the different versions of the algorithm. However, the running times do differ. When deleting the isolated connections, we see that the running time is much larger. This was also observed in the other algorithms. Note that the running times for neglecting isolated connections and allowing extra digging are similar. This follows as there was no least cost-path which required extra digging.

Summarizing, we advice against to not delete isolated connections. In this case, the running time is too large to give a relevant algorithm. Furthermore, neglecting the isolated connections will give the same results. We advice to use the version which allows for extra digging. This is a more general algorithm and in practice this will be more useful. This also has to do with the fact that the data set is incomplete, which results in the isolated connections. When a choice has to be made for the x-value, we

see that the results for $x = 5$ are similar to the results for $x = 15$. The running times for $x = 5$ are smaller, as expected, while the costs found are in some cases slightly larger. The requirements set by the user mainly determine which values will be used, but we advice to use $x = 10$. This way we have the accuracy of larger x-values, while having the shorter running times of the smaller x-values.

The second configuration: Restricting the network

In the previous part we discussed the options for the first configuration. We will now determine what is the best configuration for the restriction of the network. The options as sketched above will be considered. We expect the versions which restrict the network to be faster than the other algorithms. This follows as less connections have to be considered. Note that removing the leafs might lead to decreased running times as less points has to be considered in Dijkstra's algorithm. However, we have to do more computations and therefore it might also lead to longer running times. In the following we hope to determine what the best option is.

As mentioned, restricting the network happens via bounding rectangles. The bounding rectangles we will consider are enlarged bounding rectangles of the two points we want to connect. We already know what a bounding rectangle of a connection is. A bounding rectangle of the two points is determined in a similar way. All endpoints within this bounding rectangle will then be considered. Note that there are cases in which this might not give the best result. Therefore we can enlarge this bounding rectangle by a constant on each side which might be beneficial for the accuracy. In the following analysis we will consider values of this input variable of 0, 500, 1500 and 5000. In the following we will refer to this value as the restriction-value or r-value and we will say an r-value of 500 or $r = 500$. Note that we added the value $r = 0$ so that we can see if it is necessary at all to enlarge the bounding rectangle. We will refer to the complete network with $r = \infty$, note that this is in line with the meaning of the r-value. In the following *All endpoints* will mean that all endpoints are considered within a certain region. Similarly, *No leafs* will mean that all endpoints are considered within a region, but the leafs are removed from this network. See Figure 4.38 for an example where a restriction of $r = 0$ would not give the least cost-path.

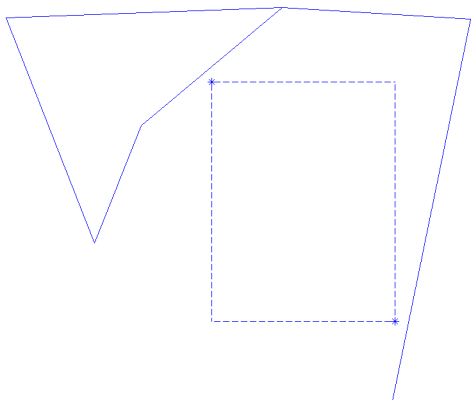

Figure 4.38: An example where $r = 0$ would fail.

Again we will consider the data set for Venray first. Table 4.39 we can find the running times for the different versions of the algorithm.

	$r = \infty$	$r = 5000$	$r = 1500$	$r = 500$	$r = 0$
All endpoints	3.47	4.03	3.47	2.04	1.35
No leafs	3.34	3.74	3.14	2.06	1.40

Table 4.39: Venray: Median running times for fifty pairs in seconds.

The distance found using these algorithms can be found in Table 4.40 below.

	$r = \infty$	$r = 5000$	$r = 1500$	$r = 500$	$r = 0$
All endpoints	6,199.94	6,199.94	6,199.94	6,199.94	11,276.56
No leafs	6,199.94	6,199.94	6,199.94	6,199.94	11,276.56

Table 4.40: Venray: Average costs for the fifty pairs of points considered in euros.

Note the large averages for $r = 0$. This follows as for nine of the fifty pairs of points we have considered, Dijkstra's algorithm could not find a least cost-path due to the restriction on the endpoints. Therefore, the 'best' option given by the algorithm would be to directly dig between the two points. Note that the results shown are averages, in some cases the difference is 'only' 5,000 euros. In other cases however the difference is more than sixty thousand euros.

Let us now consider the city Utrecht which has 243,282 connections in total. The running times can be found in Table 4.41 while the average distances can be found in Table 4.42. We see similar results as we saw for Venray, namely, the average costs are equal when the leafs are pruned or when the leafs are also considered, hence supporting our claim that we may neglect them. For the running times we see that the version where the leafs are neglected, has a slightly shorter running time.

We also see that this time the running times for $r = 5000$ are in fact smaller than those for $r = \infty$. For Venray this was not the case. This probably has to do with the fact that this data set was larger. Hence, using $r = 5000$ did in fact restrict the network. As we only considered a small region for Venray, using $r = 5000$ resulted in a situation where still all connections were considered. Hence, resulted in a situation where all connections were considered, but where more computations where needed.

	$r = \infty$	$r = 5000$	$r = 1500$	$r = 500$	$r = 0$
All endpoints	41.77	39.62	21.79	14.78	12.54
No leafs	41.47	36.18	20.95	14.69	11.86

Table 4.41: Utrecht: Median running times for fifty pairs in seconds.

	$r = \infty$	$r = 5000$	$r = 1500$	$r = 500$	$r = 0$
All endpoints	17,583.43	17,583.43	17,583.43	17,617.14	37,919.16
No leafs	17,583.43	17,583.43	17,583.43	17,617.14	37,919.16

Table 4.42: Utrecht: Average costs for the fifty pairs of points considered in euros.

We also see that for Utrecht, an r-value of 500 will not always give the correct results. In fact, for three pairs of points, $r = 500$ did give a higher cost than $r = \infty$ did. The average difference was 500 euros. For $r = 0$ we got for 29 pairs of points incorrect results. On average this resulted in an extra cost of 35,000 euro. Note that this follows mainly as in fourteen cases, using $r = 0$ resulted in direct digging as shortest connection.

To conclude, we saw that it is best to neglect the leafs in the algorithm as this had no effect on the costs, while the running times did decrease. We advice to restrict the network as we same that $r = \infty$ results in longer running times than smaller r-values. Especially for larger data sets this was the case. However, we advice against using the r-values 0 and 5000. Meaning that if we restrict the network too much or no enough, either the results will be far from optimal, or the running times will be too long.

4.3.2 Running times of the algorithm

Now that we have determined the configurations for the algorithm, we can consider the running times. In the following we will consider the version of the algorithm which allows for extra digging if there are isolated components. The running times are not that much larger compared to the situation where such connections were neglected, while the results found are better. In the previous section we also saw that x-values of 10 are preferred in most cases. They give rather acceptable results, while the running time is not that large. In the following we will use this x-value for all data sets. For the restriction of the network we will neglect the leafs. This follows as the running times decreased, while the results stayed the same. For restriction of the network we saw that the algorithm found the for $r = 1500$ the algorithm finds exact results, while for $r = 500$ this was not always the case. Therefore, Let us consider r-values of 500, 1000 and 1500.

Again we will randomly generate the points and we will then consider the running times of the algorithm and compare them.

First we will consider the Dutch city Zwolle. The data set for Zwolle consists of 73,066 connections. The median running times can be found in Table 4.43, while the average costs can be found in Table 4.44. Note that these are running times and costs for fifty pairs of randomly generated points.

	$r = 500$	$r = 1000$	$r = 1500$
Running times	3.92	3.96	4.12

Table 4.43: Zwolle: Median running times for fifty pairs in seconds.

	$r = 500$	$r = 1000$	$r = 1500$
Costs	8,843.27	8,843.27	8,843.27

Table 4.44: Zwolle: Average costs for the fifty pairs of points considered in euros.

Note that the average costs in all three cases are equal. Hence, $r = 500$ suffices in all three cases. Therefore, in this situation, we can choose the r-value for which the running time was the shortest which was $r = 500$. Note however that the differences in running times for the different values of r are small.

Let us now consider a larger city, namely that of Breda. With 230,999 connections, the region considered has more than three times as much connections as the region considered for Zwolle. Again we randomly generated fifty pairs of points we need to connect. This gave running times as shown in Table 4.45 and costs as shown in Table 4.46.

	$r = 500$	$r = 1000$	$r = 1500$
Running times	11.07	11.44	11.93

Table 4.45: Breda: Median running times for fifty pairs in seconds.

	$r = 500$	$r = 1000$	$r = 1500$
Costs	13,215.19	13,211.27	13,211.27

Table 4.46: Breda: Average costs for the fifty pairs of points considered in euros.

Note that the costs for $r = 500$ is slightly higher than for the other two r-values. This follows as for one pair of points, $r = 500$ found a larger cost than for $r = 1000$ and $r = 1500$. In this case we suggest to use $r = 1000$ as this gave the same results as $r = 1500$ while being slightly faster.

As the last data set we will now consider the city Rotterdam, which has nearly 344,059 connections. The running times and the costs can be found in Table 4.47 and Table 4.48 respectively.

	$r = 500$	$r = 1000$	$r = 1500$
Running times	14.27	14.48	15.21

Table 4.47: Rotterdam: Median running times for fifty pairs in seconds.

	$r = 500$	$r = 1000$	$r = 1500$
Costs	16,843.60	16,798.85	16,784.36

Table 4.48: Rotterdam: Average costs for the fifty pairs of points considered in euros.

We again observe that there is a difference in the costs for the different values of r, hence we cannot draw a conclusion solely based on the running times. The difference in running times for $r = 500$ and $r = 1500$ is less than a second, hence we can use $r = 1500$ in all cases. The costs will be smaller on average, while the running times are only a fraction larger. We saw that there were only three points for which $r = 500$ gave different results than for $r = 1500$, one may therefore argue that in most cases $r = 500$ is sufficient. For $r = 1000$ there was only one point for which the costs were higher than for $r = 1500$.

Surprising was that of the three pairs for which $r = 500$ gave different results than $r = 1500$, for one pair the cost was lower for $r = 500$ than for $r = 1500$. This can happen as we restrict the network using the r-values. For $r = 500$ the network was restricted in such a way that some of the shortest connection points in the case of $r = 1500$ were not found. The points found for $r = 500$ required at least as much digging as the points for

$r = 1500$ did, however, if the routing required is much less, this might be more optimal. Note that in general restricting the network will lead to higher costs.

In the next section we will consider the running times for increasingly large data sets and finally we will give an advice on which r-value one should use.

4.3.3 Running times for increasingly large data sets

In the previous section we considered running times for smaller data sets. The largest being that of Rotterdam. In the following we will consider larger data sets. That is, we will consider data sets of two cities, of a large city with smaller villages surrounding it and we will consider a data set spanning multiple cities and even provinces of the Netherlands.

First we will consider the running times for the case of Breda and Tilburg. The data set containing these two cities has 486,097 connections in it. The running times can be found in Table 4.49 and the costs can be found in Table 4.50.

	$r = 500$	$r = 1000$	$r = 1500$
Running times	17.14	17.37	17.68

Table 4.49: Breda and Tilburg: Median running times for fifty pairs in seconds.

	$r = 500$	$r = 1000$	$r = 1500$
Costs	61,659.15	39,146.98	39,100.36

Table 4.50: Breda and Tilburg: Average costs for the fifty pairs of points considered in euros.

We see that the average costs for $r = 1500$ are the smallest. Given that the running times for the three values of r are similar, we suggest to use $r = 1500$ for this data set. However, there are some interesting things we should note. For $r = 500$ there are 14 pairs which gave a higher cost. Moreover, for three of those pairs, it was not possible to find a path a the direct digging option was given by the algorithm. For $r = 1000$ the algorithm did find a path in all cases, however, for 4 pairs the costs differ from that for $r = 1500$. There was one pair of points for which both $r = 500$ and $r = 1000$ gave smaller costs than $r = 1500$ did.

Let us continue and consider the capital of the Netherlands, Amsterdam, together with some smaller villages surrounding it. This data set consists of 630,137 connections. The running times for the fifty pairs can be found in Table 4.51, while the average costs can be found in Table 4.52.

	$r = 500$	$r = 1000$	$r = 1500$
Running times	28.60	29.28	30.05

Table 4.51: Amsterdam: Median running times for fifty pairs in seconds.

We saw that the difference between the running times for different r-values is larger than before. However, the running times are in general also longer than before. For six pairs $r = 500$ did not give the same results as $r = 1500$ did. For both $r = 500$ and

	$r = 500$	$r = 1000$	$r = 1500$
Costs	47,549.64	47,459.10	42,321.89

Table 4.52: Amsterdam: Average costs for the fifty pairs of points considered in euros.

$r = 1000$ there was one pair for which no path was found, hence a direct digging option was given. For only two points did $r = 1000$ give higher costs than $r = 1500$.

Let us finalize our analysis by considering the southwestern part of the Netherlands. This includes the provinces of Zeeland and Noord-Brabant. In total this data set has 2,776,827 connections. In Table 4.53 and Table 4.54 we find the running times and the average costs of the algorithm for this data set.

	$r = 500$	$r = 1000$	$r = 1500$
Running times	124.04	123.73	123.94

Table 4.53: The southwest of the Netherlands: Median running times for fifty pairs in seconds.

	$r = 500$	$r = 1000$	$r = 1500$
Costs	614,935.38	381,448.91	277,362.69

Table 4.54: The southwest of the Netherlands: Average costs for the fifty pairs of points considered in euros.

Again we observe that the running times are similar for the different r-values. We do see that for $r = 500$ the running times are larger than for the other two r-values. As the difference is relatively small, we may assume this difference is due to outliers in the running times. We also see that the costs for $r = 1500$ are much smaller than for the other two r-values. This again has to do with the fact that for some points the algorithm does not find a path which requires routing through the network if r-values of 500 and 1000 are used. In fact, for 26 pairs, $r = 500$ gave higher costs than $r = 1500$. For $r = 1000$ there were 18 pairs of points which gave a higher cost than $r = 1500$. In both cases there were no pairs which gave a smaller cost than found for $r = 1500$. For $r = 500$ there were twelve pairs of points were the algorithm gave the option 'Direct digging' as the least cost-path. For $r = 1000$ there were three such pairs and for $r = 1500$ there was no pair for which the least cost-path would be to directly dig between the two points. For this data set we see that $r = 1500$ is the most useful r-value.

Summarizing, we saw that the running times are fairly similar for the different r-values. Of course, larger r-values imply larger running times, however the difference are rather small. In fact, the difference becomes insignificant if larger data sets are considered. Furthermore, it turned out that in some cases for smaller r-values (such as $r = 500$) no path is found which includes routing through the network. Even in the case of $r = 1000$ this sometimes was the case. The algorithm works even if larger networks are used such as that of Amsterdam or even if the entire southwest of the Netherlands is considered. For small data sets the algorithm finds results within a few seconds, for larger data sets up to a minute. This is the case if multiple cities are considered, or larger cities with smaller villages surrounding it. In the case of the southwest of the

Netherlands, the running times found are around two minutes. Given the size of the data set, this is acceptable.

One must note that the running times of the algorithm depend on the distance between the points which are considered and on the rest of the network between the two points. Due to the setup of the algorithm, the running times will be smaller if the points are closer to each other. On the other hand, if the points are further apart, the running times will increase. This is however as expected. However, even if two points are far apart, this does not immediately imply longer running times. If the network only has a few connections overlapping with the bounding rectangle spanned by the two points, then the running times will be lower compared to a situation where there are many connections in the bounding rectangle. In the case of the southwest of the Netherlands, the median running time found for $r = 1500$ was 124 seconds. However, the largest running time found was 462.13 seconds for two points which were only 45 kilometers apart. The smallest running time for this data set was 84 seconds, which happened for two points being only six kilometers apart.

We have seen that the goal set in the beginning was met, which was to construct an algorithm which would give a least cost-path within a minute. For extremely large networks this may take a little longer. We advice to use $r = 1500$. The running times might be slightly larger than those for smaller r-values, however the algorithm will find least cost-paths in practically all cases. Only in special cases the value $r = 500$ gives a lower cost (see Rotterdam), however this outweigh the drawbacks of smaller r-values. To save more time one may choose to use x-values smaller than ten, i.e., choose to connect each point with less than ten values in the network. We advice against this however as already seen in Section 4.3.1.

The algorithm also works for isolated connections, however this is under the restriction that once we dig to an isolated connection, in the next step we have to connect with the network. That is, we may not again connect with an isolated connection. There are special cases which the algorithm cannot yet solve. We have already discussed some of those in previous sections. The main issue is that in some cases it is beneficial to dig instead of route. However, in this setting the algorithm cannot coop with this fact well.

Bibliography

[1] F. Phillipson. *Efficient Algorithms for Infrastructure Networks: Planning Issues and Economic Impact*. PhD thesis, VU Amsterdam, 2014.

[2] Celtic-4GBB. `http://www.4gbb.eu/`, 2015.

[3] F. Phillipson. Fast roll-out of fibre to the cabinet: Practical approach for activation of cabinets. In *Networks and Optical Communications - (NOC), 2014 19th European Conference on*, pages 102–107, June 2014.

[4] P. Erdös and A. Rényi. On random graphs i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[5] R. Albert, H. Jeong, and A.L. Barabasi. The diameter of the world wide web. *Nature*, 401:130–131, 1999.

[6] M.E.J. Newman. Scientific collaboration networks. i. network construction and fundamental results. *Phys. Rev. E*, 64:016131, Jun 2001.

[7] M.E.J. Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Phys. Rev. E*, 64:016132, Jun 2001.

[8] M.E.J. Newman. Random graphs as models of networks. *eprint arXiv:cond-mat/0202208*, feb 2002.

[9] E.N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, pages 1141–1144, 1959.

[10] William Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1*. John Wiley & Sons, inc., 3rd edition, 1968.

[11] B. Bollobás. *Random graphs*. Cambridge University Press, second edition, 2001.

[12] A. Frieze and M. Karoński. *Introduction to Random Graphs*. Cambridge University Press, 2015.

[13] J. Hopcroft and R. Kannan. Computer science theory for the information age, 2012.

[14] B. Bollobás and A.G. Thomason. Threshold functions. *Combinatorica*, 7(1):35–38, 1987.

[15] J.M. Diamond. *Guns, Germs, and Steel: The Fates of Human Societies*. National bestseller / [W. W. Norton & Company]. W.W. Norton & Company, 1999.

[16] G.B. Dantzig. *A History of Scientific Computing*, chapter Origins of the Simplex Method, pages 141–151. ACM, New York, NY, USA, 1990.

[17] V. Klee and G.J. Minty. How good is the simplex algorithm? *Inequalities*, III:159–175, 1972.

[18] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[19] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[20] T.J. Pennings. Do dogs know calculus? *College Mathematics Journal*, 34(3):178–182, 2003.

[21] P. Mandrekar and T. Joseph. When to cut corners and when not to. *The Mathematical Intelligencer*, 36(3):70–74, 2014.

[22] T. Miick. Minimizing travel time through multiple media with various borders. Master's thesis, Western Kentucky University, 2013.

[23] Mathworks. MatLab Data. `http://nl.mathworks.com/help/matlab/matlab_external/matlab-data.html`, 2014.

[24] M. de Koning. Fibre to the curb planning. Master's thesis, VU Amsterdam, 2015.

[25] A. Cardillo, S. Scellato, V. Latora, and S. Porta. Structural properties of planar graphs of urban street patterns. *Phys. Rev. E*, 73:066107, Jun 2006.

[26] M.I. Shamos and D. Hoey. Geometric intersection problems. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:208–215, 1976.

[27] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[28] K. Mulmuley. A fast planar partition algorithm. i. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:580–589, 1988.

[29] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, jan 1992.

[30] A. Mantler and J. Snoeyink. Intersecting red and blue line segments in optimal time and precision. In *Discrete and Computational Geometry, Japanese Conference, JCDCG 2000, Tokyo, Japan, November, 22-25, 2000, Revised Papers*, pages 244–251, 2000.

[31] L. Arge, T. Mølhave, and N. Zeh. Cache-oblivious red-blue line segment intersection. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, pages 88–99, 2008.

[32] B. Moore. Data structures. `http://www.mathworks.com/matlabcentral/fileexchange/45123-data-structures`, 2014.

[33] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[34] L.R. Ford. Network flow theory. *RAND Corporation*, 1956.

[35] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[36] D. Gleich. gaimc: Graph algorithms in matlab. `http://nl.mathworks.com/matlabcentral/fileexchange/24134-gaimc---graph-algorithms-in-matlab-code`, 2009.