Visibility index computations on grid terrains

Lambrechts, C.

Award date:
2015

Technische Universiteit **Eindhoven** University of Technology

/ Department of Mathematics and Computer Science / Algorithms

Visibility Index Computations on Grid Terrains

Master Thesis

Colin Lambrechts

Committee:
Prof. dr. Mark de Berg
Dr. George Fletcher
Prof. dr. ir. Jack van Wijk

Supervisor:
Prof. dr. Mark de Berg

Version 1.1 (Wednesday 9th September, 2015)

Eindhoven, Monday 28th September, 2015

Where innovation starts

[0] Image on the front cover is from http://www.eea.europa.eu/data-and-maps/figures/elevation-map-of-europe

Abstract

The visibility index of a cell c in a grid terrain T is defined as the percentage of cells that are visible from c. We consider the problem of computing the visibility index of all cells c in a grid terrain T of size $n \times n$. To this end we first study the 1-dimensional version of the problem. We propose an algorithm that efficiently computes the visibility index in $O(n \log^2 n)$ time for a given 1-dimensional terrain of size n. The algorithm is able to compute the visibility index of 500,000 cells within 150 seconds using real-life data sets, while the brute-force approach can only compute it for 5,000 cells within that time frame. Our proposed algorithm can be used to efficiently approximate the 2-dimensional version of the problem and we present our ideas on how to do that.

Contents

Chapter 1

Introduction

Determining the importance of points on a terrain is relevant for many applications. Examples are telecommunications, archaeological sites, water and fire guards towers and even in military logistics and building sites. Importance can be measured in various ways, but visibility usually plays a role. Prominence, as suggested by Arge et al. [1], is also a measure for importance. E.g. many archaeological sites have a good view of the surrounding area. So points with a high visibility are potentially interesting for archaeological research. Points that cannot be seen by many other points can also be interesting, especially for military purposes. We define the *visibility index* of a point p as the percentage of the terrain visible from p. The goal of our project is to compute the visibility index of all points on the terrain.

A terrain can have different representations. The most commonly used representations are *triangular irregular networks (TIN's)* and *digital elevation models (DEM)*. A TIN is mainly used to represent a terrain more realistically and is based on triangles. The triangles form the continuous surface of a terrain. The points that form the triangles can be placed arbitrarily. For non-triangle points the elevation is interpolated based on the triangle points of the intersecting triangle. A DEM, or *grid* terrain, represents a terrain as a regular grid of squares cells on the xy-plane in which each cell is assigned an elevation. The interpretation of a DEM has two approaches. The first approach is that all points within a cell have the same elevation. This means that the DEM forms a stepped surface. The downside of this is that neighboring cells cannot see each other, due to the edges of the steps. The second approach is to interpret the elevation as the z-coordinate of the center of the cell. Connecting the centers of neighboring cells to form the surface of the terrain. For non-grid points the elevation is interpolated based on the neighboring points. This approach is more realistic then the stepped approach and easier to reason about than a TIN. DEM's are commonly used in GIS applications, while TIN's are commonly used in visualization. The terrain model used throughout this document is a DEM.

Using a DEM as terrain model we can more precisely state the problem we will be studying. The terrain is represented as a grid of n^2 cells ($n \times n$) and for each cell the elevation is known. The *visibility index* of a cell is now defined as the percentage of other cells that are visible from that cell. Our goal is to compute the visibility index for all cells in the terrain. Determining the visibility index for each cell naively would take $O(n^5)$. We will try to come up with a way to compute or approximate the visibility indices of all points more efficiently.

We start by presenting possible solutions found in the literature in Chapter 2. As we will see, those solutions do not solve our problem or do not solve it efficiently. Therefore we propose a new algorithm. First we discuss the 1-dimensional version in Chapter 3 and then we present our ideas for the 2-dimensional version in Chapter 4. To verify the practical use of our algorithms we have conducted various experiments. The experimental setup can be found in Chapter 5 and their results are presented and discussed in Chapter 6. Chapter 7 presents our conclusions and Chapter 8 presents some open questions and possible future research.

Chapter 2

Literature and Related Work

There are several different problems that can be studied, when it comes to visibility computations on grid terrains. The most studied problem is to compute the viewshed of a single cell. These type of algorithms are discussed in Section 2.1. The data used in GIS applications can be very large and cannot be stored in main memory. Therefore I/O-efficient and parallel distributed algorithms increase the performance drastically. Some adaptations of the single-cell viewshed algorithms can be found in Section 2.2. Algorithms that address the observer placement problem and contain interesting approaches or optimizations are addressed in Section 2.3. The one algorithm that is most closely related to the problem we are interested in, namely total visibility index computations, can be found in Section 2.4.

Different types of problems

We can distinguish several types of computations on terrains.

- Computing information about a single cell in the terrain, like the viewshed of a single cell.
- Computing information about all cells in the terrain, like the total-viewshed.
- Optimization problems on terrain, like finding a minimum-sized set of observers to see the complete terrain.

In our study the main focus is visibility. The *viewshed* of a cell is the part of the terrain that is visible from that cell. The viewshed of a single cell can be the complete terrain and therefore it is possible that the output contains n^2 cells. The *total-viewshed* of a terrain consists of the viewshed of all cells in the terrain. The result can contain the complete terrain for each cell, which results in an output size of $\Omega(n^4)$ cells. Computing the total-viewshed is in many cases too much, often you don't need the actual viewshed of each cell. You only need a derived property of the viewshed, for example the size. The visibility index is such a derived property. The *visibility index* of a cell is the percentage of the terrain that is visible from that cell. The output would only have size $\Omega(n^2)$. The visibility index can be used to determine interesting points in the terrain. For those points the viewshed can be computed in a later stage.

Partial terrain computations consist of many different problems. Most of them have to do with optimization. Select a minimum/maximum number of cells to fulfill a certain property, for example to cover 95% of the terrain with the minimum number of observers. Selected cells used to observe a terrain are called *observers*. This specific problem is called the *observer placement problem* and is a well known problem in GIS applications. Placement of observers is relevant for many communication and observation problems. For instance, to find good locations for telecommunication antennas, fire guard towers and water towers.

An overview of many visibility problems is given by De Floriani and Magillo [6], who describe some basic notions concerning terrain models and visibility. They also define visibility structures and visibility queries. The central part of the paper contains a survey of algorithms proposed in the literature to compute visibility structures and to solve visibility queries. There are many different possible definitions for visibility and each one has its own benefits and its own drawbacks. Different data representations address different problems and require different approaches to handle them. The different approaches, definitions and comparison are useful as a starting point for visibility problems. This papers guides you through the different approaches.

Terrain models tend to be huge in size and therefore parallel visibility algorithms give interesting results. According to De Floriani and Magillo [6] the most promising approaches are based on domain partitioning, which can be implemented in a distributed environment. Reducing the size of DEM's can also be achieved by using multi-resolution terrain models. In many cases too many details in far areas result in redundant information. Multi-resolution terrain models can reduce the amount of redundancy in far areas without reducing the accuracy of the surface in the neighborhood of the viewpoint. Another problem is that visibility structures are extremely sensitive to data error, as well as conventions used both in DEM construction and in implementation. For this reason, a probabilistic approach to visibility is perhaps more suitable than the traditional, exact, classification of visibility. Below we discuss the papers most closely related to our work.

2.1 Single Viewshed Computations

An algorithm to compute a single viewshed was proposed by Van Kreveld [20]. He shows that a single viewshed can be computed in $O(n^2 \log n)$ time on a grid of elevation data. This means that computing the total-viewshed can be done in $O(n^4 \log n)$. The algorithms is simple to implement and requires only little extra storage. This paper demonstrates the use of one of the most important geometric algorithmic design methods, namely, that of plane sweep. Many of the future papers on visibility computations refer to this paper.

The sweep algorithm computes the viewshed of a single point v by performing a radial sweep centered at v. The sweep line is a half-line that rotates around v and makes a full turn. There are three types of events. The first type occurs when the sweep line enters a cell, the second occurs when the sweep line goes through the center of a cell and the third type occurs when the sweep line leaves a cell. The status structure consists of a balanced binary search tree of cells intersecting the sweep half-line. The leaves represent the cells and store the gradient of the line from v to that cell. The internal nodes store the maximum of the gradients in their sub-tree.

There are several papers, e.g. Fishman et al. [9] and Ferreira et al. [8], which improve the performance of this algorithm. They use interesting techniques, but they are not so relevant for our problem.

2.2 I/O Efficiency and Parallel Distributed Single Viewshed Computations

I/O-efficient algorithms are important for huge data sets, which are common in the area of GIS applications. Especially for single cell viewshed computations the I/O's can be the bottleneck. Haverkort et al. [12] describes an I/O-efficient algorithm for computing the viewshed of a point on a grid terrain, which was latter improved by Fishman et al. [9]. Both algorithms are based on the algorithm of Van Kreveld [20]. Ferreira et al. [8] propose an I/O-efficient algorithm, built from scratch, that is faster than Haverkort et al. [12] and Fishman et al. [9]. The main idea of a radial sweep is also present in this algorithm.

A different approach for improving the performance is proposed by Ferreira et al. [7]. They propose a parallel algorithm based on the algorithm of Van Kreveld [20]. Kidner et al. [14] show that parallel applications can be very useful for GIS applications, due to the symmetrical nature of the computations. To achieve this, challenges have to be solved. For example the division of the work, also called load balancing, is difficult and introduces additional communication costs.

I/O-efficient single-viewshed algorithms can be used to compute the total-viewshed. An I/O-efficient algorithm is very useful in GIS applications, but the efficiency of the algorithm itself is also important. The total-viewshed problem is already hard enough to do efficiently within main memory. I/O-efficient algorithms are interesting and can play an important role, but are left for future research.

2.3 Observer Placement Problem

Lee [15] compares different algorithms and proposes some heuristics for visibility computations. Most of the heuristics can be used for the observer-placement problem. Lee also incorporates practical information, like costs, to determine where to situate observers. The different definitions of visibility and observer placement and their practical relevance are also addressed.

Speed-Accuracy Trade-offs

Franklin and Ray [10] propose various speed-accuracy trade-offs for viewshed algorithms. The main goal of Franklin and Ray is to place observers in an efficient and accurate manner. The different trade-offs are discussed by comparing different approaches. The fastest approach, which is an approximation, uses a ring that starts at the cell for which the viewshed is computed and expands each iteration. The visibility of the cells on the ring is computed by using the cells of the previous ring. The visibility line from the starting cell to a cell on the ring uses only two cells of the previous ring, so each processed cell uses only information of two other cells. Hence the complexity is $O(n^2)$, which would result in $O(n^4)$ if we would compute the total viewshed. The most accurate approach, on the other hand, computes the visibility for each cell within the ring with radius r in $O(r^3)$ time. For the total-viewshed the ring would have a radius of n, which result in $O(n^5)$. The authors improved the running time by computing the visibility only for the cells on the perimeter and use the line of sight for the cells within the ring. This reduces the running time to $O(r^2)$. This speed-up reduces accuracy, but it remains pretty accurate compared to the previous algorithm that runs in $O(r^3)$. This speed-up would result in $O(n^4)$ for total-viewshed computations. Approximating the visibility computations to gain a speed-up was the next algorithm the authors propose. The approximation uses more points close to the viewpoint and less points far away. It uses only a small number (16) of rays from the viewpoint to the perimeter. The distance to the perimeter doubles every 16 cells. This means that only $\log n$ cells are used to approximate a line of sight of n cells. The experiments of the paper show that the approximation has a good quality and that for the observer placement problem 32 rays per cell are almost as good as 128 rays per cell.

The algorithms can be used as basis for an approximation of the viewshed. This is done in the paper of Izrealevltz [13], who reuses the line of sight computations for points further away. The results and techniques are applicable and useful for fast and accurate viewshed approximations.

These papers seems quite relevant, however they are not. The papers are written towards an algorithm to select the best points for observers. This can also be noted by the title and the introduction of the paper. The experiments and their findings about the terrain properties are interesting. These findings can possibly be used to improve viewshed approximations (as done in Izrealevltz [13]).

Goodchild and Lee [11] proposes an algorithm to maximize the viewshed for a fixed number of observers or to find the minimum number of observers to make an area visible. They propose several different definitions and encounter challenges for visibility and coverage problems.

2.4 Total-Viewshed Computations

Tabik et al. [18], which is an improved version of Tabik et al. [19], propose an algorithm that computes the total viewshed of a terrain. The algorithm has a running time of $O(sn^2 \log n)$, where s denotes the number of sectors. A sector is a slice of the terrain in one single direction. The number of sectors is typically 360, which means that sectors are $1°$. The cells along the line of a sector together from a 1-dimensional representation of elevations. An example is given in Figure 2.1.

The main idea of the algorithm is that the sectors and viewsheds are similar for neighboring points and therefore can be reused to a great extent. The techniques they use to accomplish this are 1) reliable sampling points to represent the sub-areas of study (sectors), 2) using a compact and stable data structure and 3) increasing the re-use of the data structures and computations of neighboring points by adopting them from the previous step to the current step.

The algorithm works in two main phases. First it finds the end limits of the visibility region within a sector, where an end limit denotes a point at which the visibility ends, for example a peak. The next phase computes the start limits, a start limit represents the first point of a visible region. Within a sector different visible regions can be found and they alternate with non-visible regions. The intermediate results are stored in complex data structures. Also see Figure 2.1.

The results cannot be compared to other total-viewshed algorithms, as there are no other algorithms. Instead they compare it to single-viewshed computations (r.viewshed under GRASS and Viewshed under ArcMap). The performance can be compared by normalizing the time needed for a single cell and taking the average of multiple runs of the single cell viewshed algorithms. The results are compared by taking 30 sample cells and compare their viewsheds. The experiments are run on a Intel(R) Core(TM)2 Duo Processor E8500 on a terrain of $2,000 \times 2,000$ cells. The run time of the total-viewshed algorithms is 0.0032 seconds for a single cell compared to 18 seconds of r.viewshed and 10 seconds of Viewshed. The different algorithms provide similar values for about 69% of the total number of cells with a relative difference between $[-20\%, +20\%]$, which they consider more than acceptable due to the numerically instability of the viewshed problem. They state that differences in the order of 25% are acceptable for this kind of problem.

This paper is the first paper to address total-viewshed computations. The results look promising and practical, the only downside is that the algorithm itself is quite complex. The different parts, phases and the corresponding pseudo code are clearly explained, but they do not explain how it actually computes the total-viewshed.

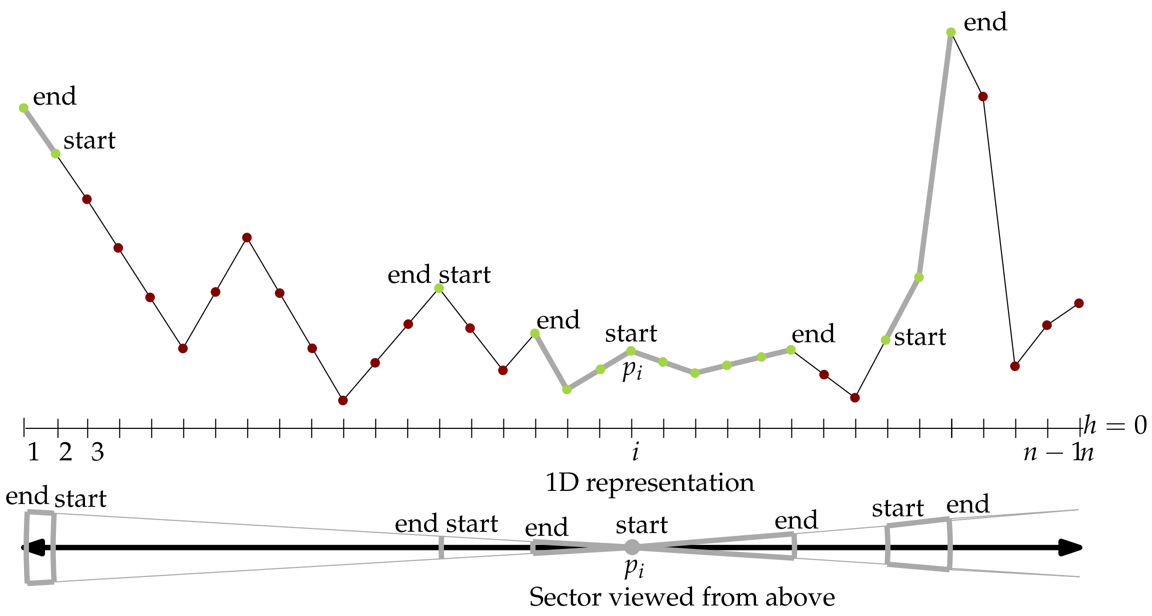


Figure 2.1: Illustrates a 1D representation of a sector, where the gray lines represent the visibility regions for p_i with *start* and *end* the respective start and end limits.

Chapter 3

The 1-Dimensional Problem

In this chapter we study the 1-dimensional version of the problem. Thus we are given an array $T[1..n]$ and we want to compute the visibility index of each grid cell $T[i]$. Our algorithm is based on the divide-and-conquer paradigm. A more detailed description can be found in Section 3.1. The pseudo code can be found in Algorithm 1. Section 3.2 describes the degenerate cases of 1DVISIBILITYINDEX and provide solutions for handling them. The analysis of the running time can be found in Section 3.3.

A small summary of the algorithm is as follows. In each recursive step the algorithm splits the input in two halves and recurses on those two halves. This means that in a recursive step for both halves the visibility index is known. Updating the visibility index, is done by computing for each cell the number of cells that are visible in the other half and adding them to the intermediate result. Computing the number of visible cells in the other half is done by determining the so-called critical rays, which are the lines below cells of the other half are not visible, and representing these rays in the dual plane. All dual elements from the left half are colored red, while those from the right half are colored blue. Then the actual number of visible cells in the other half is determined by counting the *Red-Blue intersections* in the dual plane. To efficiently update the critical rays, convex hulls are used.

3.1 Detailed Description

In this section a detailed description and analysis of the algorithm is presented. Before we start with the actual description (Section 3.1.2), notations and definitions that are used throughout the algorithm and the proofs, are presented in Section 3.1.1. The algorithm to compute the Red-Blue intersections is used as a subroutine in 1DVISIBILITYINDEX and therefore its detailed description can be found in Section 3.1.3.

3.1.1 Definitions

The input to our algorithm is an 1-dimensional grid terrain, stored in an array $T[1...n]$, where $T[i] = h_i$ denotes that (the center of) grid cell $T[i]$ has an elevation of h_i (z-coordinate) and the x-coordinate of the center of $T[i]$ is equal to i. Now define $p_i := (i, h_i)$ as the point on the terrain corresponding to (the center of) cell $T[i]$; see Figure 3.1 for an example. The *visibility index* of a cell is the number of other cells in the terrain that are visible from that cell. This requires a notion of visibility. Intuitively we say that points are visible from each other if they can see each other. A more formal notion is given in Definition 1.

Definition 1. *A point p_j is* visible *from p_i (p_i* sees *p_j) if all points p_k with $i < k < j$ lie strictly below the segment p_ip_j. A point p_j is also visible from p_i, if $i = j$; in other words a point is visible from itself.*

The line segment p_ip_j is part of the line through p_i and p_j. This line is called the *visibility line* of p_ip_j. We can also extend the line segment p_ip_j only on one side to create a *visibility ray*. The visibility ray of p_i starts in p_i and goes through p_j. The visibility ray of p_i can be used to determine if a point beyond p_j is visible from p_i with respect to p_j.

Definition 2. *The* visibility ray *from p_i to p_j, denoted by $\rho(p_i, p_j)$, is a ray that starts at p_i and passes through p_j.*

The *angle* of the visibility ray $\rho(p_i, p_j)$ is the smaller angle between the visibility ray of p_i and the ray starting at p_i and pointing vertically up ($\rho_{\text{vert}}(p_i)$), also see Figure 3.1. We also need the concept of *critical ray*, as defined next, to efficiently determine visibility between sets. For two indices l and r with $l \leq r$, define $P(l, r) := \{p_k : l \leq k \leq r\}$.

Definition 3. *Let $l \leq i \leq r$. The* left critical ray *$\rho_{left}(p_i, P(l, r))$ of p_i with respect to $P(l, r)$, is the visibility ray with the smallest angle that is, the steepest ray from p_i to some p_s with $l \leq s < i$. If $i = l$ then $\rho_{left}(p_i, P(l, r))$ is defined as the ray pointing vertically down. The* right critical ray *$\rho_{right}(p_i, P(l, r))$ of p_i is the visibility ray with the smallest angle from p_i to some p_t with $i < t \leq r$ and pointing vertically down for $i = r$.*

The critical ray can be used to determine visibility between two points, as the following lemma shows.

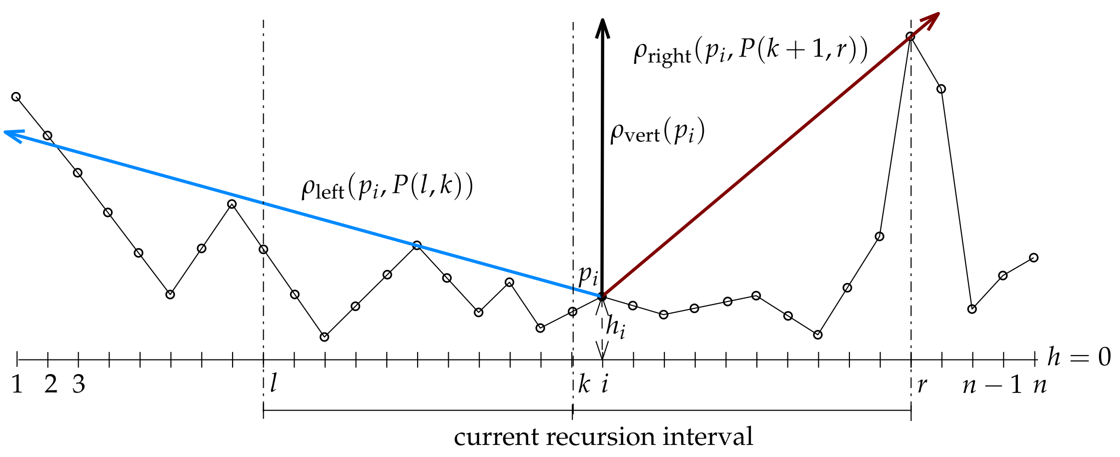
Lemma 1. *Two points $p_i \in P(l, k)$ and $p_j \in P(k + 1, r)$ are visible from each other if and only if p_i is above $\rho_{left}(p_j, P(k + 1, r))$ and p_j is above $\rho_{right}(p_i, P(l, k))$.*

Proof. Let $\rho_{\text{right}} := \rho_{\text{right}}(p_i, P(l, k))$ be the right critical ray of p_i and let $\rho_{\text{left}} :=$ $\rho_{\text{left}}(p_j, P(k + 1, r))$ be the left critical ray of p_j. Consider the line segment p_ip_j. Assume that p_i is above ρ_{left} and that p_j is above ρ_{right}. Then all points $p_i, p_{i+1}, ..., p_k$ are below the line segment, because ρ_{right} has the smallest angle. Symmetrically, all points $p_{k+1}, ..., p_j$ are below the line segment, due to ρ_{left}. Hence p_i and p_j are visible from each other.

Now assume that p_i and p_j are visible from each other. That means that all points p_s $(i < s < j)$ are below the line segment p_ip_j. All points that can possibly determine ρ_{right} and ρ_{left} are therefore also below the line segment. Hence p_i is above ρ_{left} and p_j is above ρ_{right}. □

3.1.2 Algorithm 1DVisibilityIndex

As already mentioned 1DVisibilityIndex is based on the divide-and-conquer paradigm. In each recursive step it computes the visibility index and the critical rays of each cell. A detailed description, accompanied by proofs, is presented below in a step-by-step fashion. The pseudo code can be found in Algorithm 1. In the description references to the pseudo code's line numbers are presented between brackets, for example (ln:1-5) indicates lines 1 to 5.



$\rho_{\text{right}}(p_i, P(k+1, r))$
$\rho_{\text{vert}}(p_i)$
$\rho_{\text{left}}(p_i, P(l, k))$
p_i
h_i
$h = 0$
1 2 3
l
k i
r
$n-1$ n
current recursion interval

Figure 3.1: Illustration of an 1-dimensional grid terrain and some important notions and definitions.

Algorithm 1 1DVisibilityIndex (T, l, r, VisIndex, CriticalRays)

Input: array T of n cells with elevations and two indices l and r.
Input: $VisIndex[1..n]$, where $VisIndex[i]$ denotes the visibility index of cell i before the call.
Input: $CriticalRays[1..n]$, where $CriticalRays[i].left$ and $CriticalRays[i].right$ denotes ρ_{left} and ρ_{right} of cell i before the call, respectively.
Output: $VisIndex[i] = \#$ visible cells in $P(l, r)$ for cell i for $l \leq i \leq r$.
Output: $CriticalRays[i].left = \rho_{\text{left}}(p_i, P(l, k))$ for $k + 1 \leq i \leq r$
Output: $CriticalRays[i].right = \rho_{\text{right}}(p_i, P(k + 1, r))$ for $l \leq i \leq k$.

```
1: if $l = r$ then
2:     Set $VisIndex[l] := 1$ and set $CriticalRays[l].left$ and $CriticalRays[l].right$ to be rays pointing downward. return
3: end if
4: $k \leftarrow \lfloor \frac{r-l}{2} \rfloor + l$
5: 1DVisibilityIndex $(T, l, k, VisIndex, CriticalRays)$
6: 1DVisibilityIndex $(T, k + 1, r, VisIndex, CriticalRays)$
7: $\mathcal{R} \leftarrow \{\delta_{\text{right}}(p_i, P(l, k)) : l \leq i \leq k\}$, $\mathcal{B} \leftarrow \{\delta_{\text{left}}(p_i, P(k + 1, r)) : k + 1 \leq i \leq r\}$
8: Count (for each half-line) the intersections between $\mathcal{R}$ and $\mathcal{B}$ and update $VisIndex$ using RedBlueIntersectionCount $(\mathcal{R}, \mathcal{B}, VisIndex)$.
9: Compute the upper convex hull for $P(l, k)$, $\mathcal{H}_{\text{left}}$, and for $P(k + 1, r)$, $\mathcal{H}_{\text{right}}$.
10: For each cell in $P(l, k)$ update $\rho_{\text{right}}$ to the tangent of $\mathcal{H}_{\text{right}}$, if the tangent has a smaller angle than the current $\rho_{\text{right}}$. Symmetrically for the each cell in $P(k + 1, r)$ update $\rho_{\text{left}}$ using $\mathcal{H}_{\text{left}}$.
```

Base Case

The base case (ln:1-3) is when the input contains a single cell. The visibility index is 1 and both the critical rays point vertically down.

Recursive Step

The recursive step (ln:4-10) consists of several steps. The first step recurses on the two halves of the input (ln:5-6). This means that for both the halves the visibility index and the critical rays are computed and therefore we only need to compute for each cell how many cells in the other half are visible. At the end of the recursive step we also need to update the critical rays to cover the entire input instead of only one half.

Visibility Index Computation

Computing the visibility index (ln:7-8) for each cell can be done by using duality. In the following we denote by o^* the dual of an object o. In particular the dual of the point $p : (p_x, p_y)$ is the line $p^* : y = p_x x - p_y$ and the dual of the line $l : y = ax + b$ is the point $l^* : (a, -b)$.

In the dual plane we can easily represent all visibility rays for a point p_i with respect to $P(l,r)$. All possible visibility lines of p_i go through p_i and therefore all dual points, that represent the visibility lines, lie on the dual line p_i^*. The relevant visibility lines are bounded by the critical rays of p_i. Any visibility line with an angle greater than the critical ray is not relevant, because the point that determines the critical ray is above that visibility line. Therefore we can define the *dual half-line* as part of the dual line that contains all relevant dual points of the visibility rays. The dual half-line forms a half line, because it is only bounded by the critical ray. The visibility rays are also bounded by the line vertical up through p_i, because you only look to the right (or to the left). The dual point of the vertical line is located at infinity. So the dual half-line is a half line that starts in the dual point of the critical ray and extends to infinity. An example is given in Figure 3.2. Given two dual rays, we can compute if the two points are visible from each other.

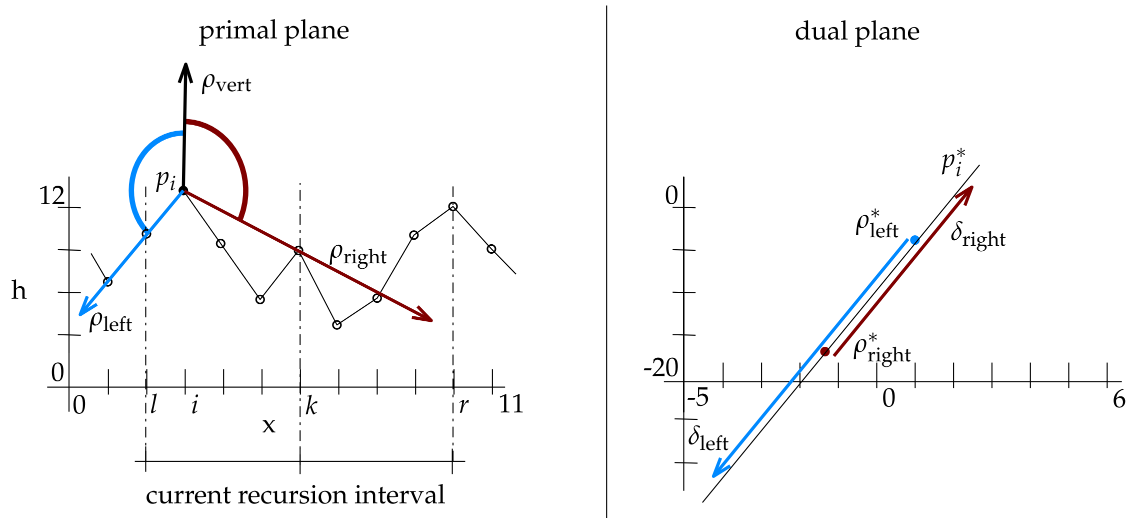
Definition 4. *Let $\delta_{left}(p_i, P(l,r))$ denote the* dual half-line *of the critical ray $\rho_{left}(p_i, P(l,r))$. Symmetrically, let $\delta_{right}(p_i, P(l,r))$ denote the* dual half-line *of the critical ray $\rho_{right}(p_i, P(l,r))$.*

Lemma 2. *Given two points $p_i \in P(l,k)$ and $p_j \in P(k+1,r)$ and the critical rays $\rho_{right}(p_i, P(l,k))$ and $\rho_{left}(p_j, P(k+1,r))$. Then p_i and p_j are visible form each other if and only if there is an intersection between their dual half-lines $\delta_{right}(p_i, P(l,k))$ and $\delta_{left}(p_j, P(k+1,r))$.*

Proof. Suppose p_i and p_j are visible from each other. Hence both points are above the critical ray of the other (Lemma 1). The visibility line through both points has a smaller angle than that of the critical rays, otherwise one of the points was below the critical ray of the other. This visibility line is represented by a point in the dual plane and it lies both on the line p_i^* and p_j^*. This point is the intersection between the two dual lines. As said before, the angle of the visibility line is smaller than both critical rays and therefore it is present in both dual half-lines. Hence there is an intersection between the two dual half-lines.

Let $\delta_{\text{left}} := \delta_{\text{left}}(p_j, P(k+1,r))$ be the dual half-line of p_j and $\rho_{\text{left}}(p_j, P(k+1,r))$ and let $\delta_{\text{right}} := \delta_{\text{right}}(p_i, P(l,k))$ be the dual half-line of p_i and $\rho_{\text{right}}(p_i, P(l,k))$, and suppose there is an intersection, the dual point q^*, between the two dual rays δ_{right} and δ_{left}. The dual point q^* corresponds to a visibility line q that goes through both p_i and p_j (intersection dual rays). The angle of q smaller than that of both critical rays. Therefore all points in between p_i and p_j are below q. Hence p_i and p_j are visible form each other. □

Lemma 1 (visibility with respect to critical rays) explicitly states that if a point p_j lies on the critical ray of p_i it is not visible from p_i (and vice versa). This means that the starting point of the dual half-line is not considered part of the dual half-line.



$p_i : (3, 13)$

ρ_{right} : right ray from p_i along $y = \frac{-4}{3}x + 17$

ρ_{left} : left ray from p_i along $y = x + 4$

$p_i^* : y = 3x - 13$

$\rho^*_{\text{right}} : (\frac{-4}{3}, -17)$

$\rho^*_{\text{left}} : (1, -4)$

Figure 3.2: An example with the elements in the primal and dual plane. The dual half-lines δ_{left} (blue) and δ_{right} (red) corresponds to the half-wedges corresponding to ρ_{left} (blue) and ρ_{right} (red), respectively, in the primal plane.

Counting all visible points in the right half ($P(k+1, r)$) of the recursive step for a given point p_i in the left half ($P(l, k)$), can be done by counting the intersections between $\delta_{\text{right}}(p_i, P(l, k))$ and the dual half-lines $\delta_{\text{left}}(p_j, P(k+1, r))$ of all p_j in the right half. For each half we create a set with all the dual half-lines of the points in that set; the right-oriented dual half-lines of the left half in the *red* set $\mathcal{R}$ and the left-oriented dual half-lines of the right half in the *blue* set $\mathcal{B}$. Finding all intersections between $\mathcal{R}$ and $\mathcal{B}$ is called the *Red-Blue intersection problem*. This problem has however an assumption, namely that the segments (in our case dual half-lines) within a set do not intersect.

Lemma 3. *Let p_i and p_j be two points in $P(l, k)$. Then the dual half-lines $\delta_{right}(p_i, P(l, k))$ and $\delta_{right}(p_j, P(l, k))$ do not intersect. Similarly, $\delta_{left}(p_i, P(l, k))$ and $\delta_{left}(p_j, P(l, k))$ do not intersect.*

Proof. Suppose $\delta_{\text{right}}(p_i, P(l, k))$ and $\delta_{\text{right}}(p_j, P(l, k))$ do intersect, which means that there is a visibility line between the p_i and p_j (in the primal plane). It also means that both $\rho_{\text{right}}(p_i, P(l, k))$ and $\rho_{\text{right}}(p_j, P(l, k))$ have an angle that is greater than or equal to that of the visibility line. An angle greater than the visibility line is not possible, due to the definition of the critical ray (Definition 3). Hence the angle must be equal to that of the critical ray and therefore the visibility line is the critical ray. This means that the intersection is at the starting point of the dual half-line. The starting point of a dual half-line is not considered part of the dual half-line and therefore l and k do not intersect. □

Red-Blue intersections can be counted efficiently with the algorithm of Palazzi & Snoeyink [17]. That algorithm computes the total number of intersections and that is not what is needed for

1DVISIBILITYINDEX . We need the number of intersections per ray, so we need to adopt it to our needs. The main idea and steps of the algorithm are maintained and therefore their proofs remain relevant. The modifications to the algorithm are explained in more detail in Section 3.1.3.

Critical Rays Update

Updating the critical rays (ln:9-10) is the next step in the recursive step. The critical ray of the complete input (from l to r), is the critical ray with the smallest angle of the critical ray of the right half ($k+1$ to r) and the critical ray of the left half (l to k). For one of the halves the critical ray is already known (from the recursion) and for the other half we need to compute it. This can be done efficiently by finding the tangent of the upper hull of the convex hull of that half. The lower hull is is not needed in 1DVISIBILITYINDEX and therefore we restrict ourselves to the upper hull.

Lemma 4. *The tangent at the upper hull of the right half ($P(k+1,r)$) is the critical ray $\rho_{right}(p_i, P(k+1,r))$ of p_i in the left half ($P(l,k)$). Symmetrically, the tangent at the upper hull of $P(l,k)$ is the critical ray $\rho_{left}(p_i, P(l,k))$ of p_i in the right half ($P(k+1,r)$).*

Proof. Suppose a point p_l, that is not on the upper hull, would determine the critical ray of p_i, then p_l is outside the upper hull. If p_l is inside the upper hull, the visibility line from p_i to p_l is below a point on the upper hull. So, p_l is outside the upper hull and therefore the convex hull is not a valid convex hull. Hence only points on the upper hull can be candidates for the critical ray.

Let p_l be the point on the upper hull, such that the tangent goes through p_l. Then the visibility line of p_i to the other points on the upper hull are below the tangent (property tangent) and therefore the tangent is the only visibility line that is not below any other point of the upper hull. Hence the tangent is $\rho_{\text{right}}(p_i, P(k+1,r))$. □

Correctness of 1DVISIBILITYINDEX

Correctness of 1DVISIBILITYINDEX follows from the following lemma.

Lemma 5. 1DVISIBILITYINDEX *correctly computes the visibility index and the critical rays for a 1-dimensional grid terrain $T[1..n]$.*

Proof. Correctness of 1DVISIBILITYINDEX is proven by induction.

The base case is when $r = l$, so we have only one cell c. 1DVISIBILITYINDEX correctly computes the visibility index and the critical rays, by definition.

The induction step is for $l < r$. We first compute $VisIndex[l..k]$, $VisIndex[k+1..r]$, $CriticalRays[l..k]$ and $CriticalRays[k+1..r]$. We call the first half (l to k) the left half and the second half ($k+1$ to r) the right half. By the induction hypothesis we know that within the halves $VisIndex$ and $CiriticalRays$ are computed correctly. What remains to compute is for each cell in the left part the number of cells that are visible in the right part and vice versa. For each cell i in the left part we compute the dual half-line $\delta_{\text{right}}(p_i, P(l,k))$ that represents the right oriented visibility rays of that cell and add it to $\mathcal{R}$. Symmetrically we do the same for the cells j in the right half and add the corresponding dual half-lines $\delta_{\text{left}}(p_j, P(k+1,r))$ to $\mathcal{B}$. Due to Lemma 2, the number of intersections found between the dual half-lines in $\mathcal{B}$ and a dual half-line in $\mathcal{R}$, correspond to the number of visible cells of the right part for a certain cell in the left part. Symmetrically the number of intersections with dual half-lines in $\mathcal{R}$ for a certain dual half-line in $\mathcal{B}$, correspond to the number of visible cells in the left part for a cell in the right part. Hence we have computed the total number of visible cells in the other part for each cell. We have already proven that we can update the critical ray for a cell in the left part using the convex hull from the right part and vice versa in Lemma 4. Hence the algorithm correctly computes the visibility index and the critical rays. $\square$

3.1.3 Algorithm RedBlueIntersectionCount

The algorithm proposed by Palazzi & Snoeyink [17] computes the total number of intersections between red and blue line segments. There are several modifications required to adapt the algorithm to our needs. The main reasons for these modifications are the use of dual half-lines instead of line segments and the need to count the number of intersections per dual half-line instead of the total number of intersections. To clarify the algorithm and our modifications we present a high-level description in pseudo code (Algorithm 2), followed by a step-by-step description in text. References to different lines in the algorithm are done using the following notation: (ln:1-5), which refer to lines 1 to 5. A detailed description of these steps can be found in Palazzi and Snoeyink [17]. Before we start describing the algorithm, we first introduce some definitions.

Definitions

The main data structure used throughout the algorithm is a hereditary segment tree. A segment tree is a data structure used to store intervals in 1D and can be used to report all intervals containing a point. However, it can also store segments in 2D if an appropriate associated structure is used. In a hereditary segment tree the x-coordinates of the endpoints of the segments subdivide the x-axis in intervals. Then we construct a balanced binary search tree whose leaves are in 1-1 correspondence with these intervals, ordered form left to right. Each internal node U of this tree is associated with an interval I_u of the x-axis, consisting of the union of the intervals of its descendant leaves. We can think of each such interval as a vertical slab S_u whose intersection with the x-axis is I_u. A more detailed explanation can be found in the book by De Berg et al. [4]. Palazzi and Snoeyink deviate from this common definition and use midpoints instead of endpoints to define the slabs. They allow several segments to be in the same leaf, provided that they are of the same color. By allowing multiple segments in a leaf, they are able to reduce the number of slabs, though in worst case there is still only one endpoint per leaf. A segment is called *short* if it starts or ends in a slab and is called *long* if it completely cuts through a slab and not through the parent's slab. The crucial property used by Palazzi and Snoeyink is that it suffices to count for each node v the number of long-long and long-short intersections; short-short intersections need not to be considered.

The associated structures, which are stored in the nodes of the segment tree, depend on the problem being solved. The associated structure used by Palazzi and Snoeyink are four linked-lists per slab σ, a linked list with red long segments $\mathcal{LL}_{\text{red}}(\sigma)$, a linked list of blue long segments $\mathcal{LL}_{\text{blue}}(\sigma)$, a linked list of red long segments and blue short segments $\mathcal{SL}_{\text{red}}(\sigma)$ and a linked list of blue long segments and red short segments $\mathcal{SL}_{\text{blue}}(\sigma)$. The elements in these lists are in order from high to low. To compute such an ordering Palazzi and Snoeyink define the *aboveness* relation. The aboveness relation is a relation on sets in the plane and is defined as follows: $A \succ B$ if there are points $(x, y_a) \in A$ and $(x, y_b) \in B$ with $y_a > y_b$. The aboveness relation is used to compute the order from high to low beforehand and, once computed, is used in each level of the segment tree to fill the associated structures.

Algorithm 2 RedBlueIntersectionCount($\mathcal{R}, \mathcal{B}, VisIndex$)

```
Input: Set $\mathcal{R}$ of $r$ red half-lines and a set $\mathcal{B}$ of $b$ dual half-lines, where the red half-lines do not
    intersect each other and the blue half-lines do not intersect each other.
Input: $VisIndex[1..n]$, where $VisIndex[i]$ denotes the visibility index of cell $i$. For each dual half-
    line we know the corresponding index $i$.
Output: $VisIndex[i]$ is increased with the number of intersections between the dual half-line cor-
    responding to $i$ and dual half-lines of the opposite color, for all $i$ that correspond to a dual
    half-line in $\mathcal{R} \cup \mathcal{B}$.
 1: Sort $\mathcal{R}$ lexicographically, first on $x$-value (left to right) of the starting points of the half-lines
    and second on slope (high to low).
 2: Idem for $\mathcal{B}$ (only slope order is from low to high).
 3: Sort the red dual half-lines and the blue points according to the aboveness relation ($\mathcal{AR}_{\text{red}}$).
 4: Sort the blue dual half-lines and the red points according to the aboveness relation ($\mathcal{AR}_{\text{blue}}$).
 5: Build lowest level of the segment tree.
 6: while $\#slabs > 1$ do
 7:     For each slab $\sigma$ create $\mathcal{LL}_{\text{red}}(\sigma)$, $\mathcal{LL}_{\text{blue}}(\sigma)$, $\mathcal{SL}_{\text{red}}(\sigma)$ and $\mathcal{SL}_{\text{blue}}(\sigma)$.
 8:     Count the red long - blue short intersections using a sweep-line approach.
 9:     Count the blue long - red short intersections using a sweep-line approach.
10:     Count the red long - blue long intersections using a ordered walk.
11:     Count the blue long - red long intersections using a ordered walk.
12:     Merge slabs, update midpoints and update the start and end slabs of all segments.
13: end while
```

Pre-processing

Before the actual intersections can be counted efficiently, some pre-processing has to be done (ln:1-5). An important step is to compute the aboveness relation between red half lines and blue starting points. The result of this computation is saved in the linked-list $\mathcal{AR}_{\text{red}}$. Symmetrically, the aboveness relation between blue half lines and red starting points is computed and stored in $\mathcal{AR}_{\text{blue}}$. This is done by building two sweep trees and extracting the order with an in-order traversal. One sweep tree is built for red half lines and blue starting points and one is built for blue half lines and red starting points. The sweep trees are built by performing a sweep from left to right. During the sweep, the sweep tree denotes the aboveness order at the vertical sweep line. In the sweep tree the following binary-tree property is satisfied, the right child is below the parent and the left child is above or on the parent. Important to note here is that red half lines, which are the duals of right-oriented critical rays, start at their starting point and end at infinity, while blue half lines, which are duals of left-oriented critical rays, start at minus infinity and end at their starting point.

Building the Segment Tree

Once the aboveness relation is know the segment tree can be built. The segment tree is not built explicitly, because each level is constructed based on the children in the level below. The lowest level of the tree only contains the leafs, which corresponds to the slabs only containing half lines of the same color, as described above. We store all slabs, of the current level, in a list $\mathcal{L}$, which is ordered from left to right. Advancing to a higher level in the tree is quite simple, just remove the shared slab boundary in between. The merged slabs correspond the slab of the parent, which is the union of the slabs corresponding to its children. Updating the start and end slab of each of the half lines is quite trivial, as we can divide them by two. E.g. start slab of an element e was 4 and now becomes 2, as the first two slabs are merged and the second two slabs are merged. Figure 3.3 illustrates the merging process using 6 slabs.

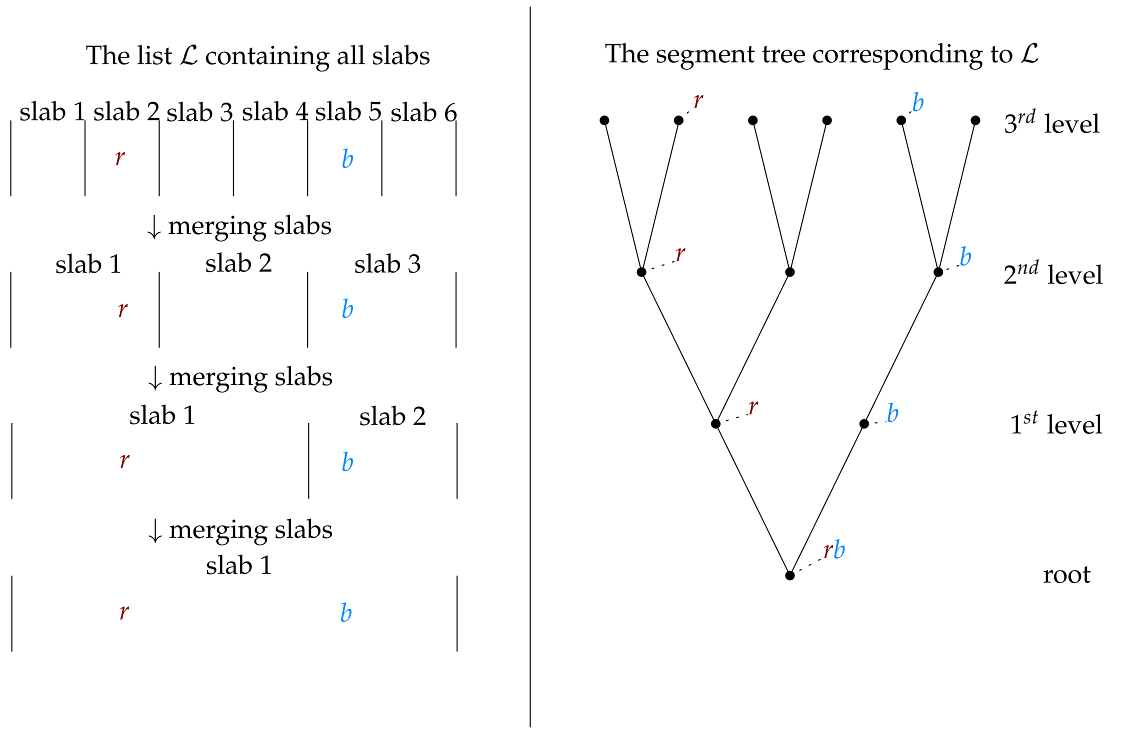

The list $\mathcal{L}$ containing all slabs
slab 1 slab 2 slab 3 slab 4 slab 5 slab 6
r
b
↓ merging slabs
slab 1 slab 2 slab 3
slab 1 slab 2
slab 1
The segment tree corresponding to $\mathcal{L}$
3^{rd} level
2^{nd} level
1^{st} level
root
rb

Figure 3.3: Illustration of the merging process of $\mathcal{L}$ using 6 slabs on the left. The resulting segment tree is shown on the right. The *red* starting point r and the *blue* starting point b are presented as reference for the *start* and *end* slab computation.

Traversing the Segment Tree

Counting the intersections is done per level in the segment tree (ln:6-13). The algorithm starts at the lowest level in the tree and keeps on moving up till only one slab remains.

At each level of the tree, the algorithm first needs to create the associated structures of the slabs (ln:7). The creation of these list is done, using the aboveness order of the corresponding color c. Using this order makes sure that the elements inserted in the lists are also in this order.

The lists with short and long segments for a given slab σ (of color c), $\mathcal{SL}_c(\sigma)$, initially does not contain the clipping of the short segments to the slab walls. For long segments this clipping is not relevant, as we know that they cut through the entire slab and the order already guarantees us the aboveness order. The clipping points of the short segments need to be inserted into $\mathcal{SL}_c(\sigma)$, as the ranks of the start and clipping points determine if a short segment intersect a long segment or not. We insert the clipping points of short segments in order according to slope, as half lines of the same color do not intersect. Left-oriented half-lines are ordered from low slope to high slope and right-oriented half-lines are ordered from high slope to low slope.

Long-Short Intersection Sweep

Within each slab $\sigma \in \mathcal{L}$ we can perform a sweep from high to low to find all long-short intersections (ln:8-9). This sweep is performed for red long segments and blue short segments and for blue long segments and red short segments. The sweep itself remains the same, so we describe it below for red long segments and blue short segments.

The intersection count is based on the ranks of the half lines, where the *rank* of a half line s in a slab v is the number of oppositely long segments above s in v. Consider the blue short segment b, with $start$ as its start point and end the point clipped to the slab wall. The number of intersections between b and red long segments is the difference in rank between $start$ and end. Figure 3.4 presents an example of a slab σ with some red long segments and some blue short segments. The ranks of the endpoints of the blue short segments are also shown.

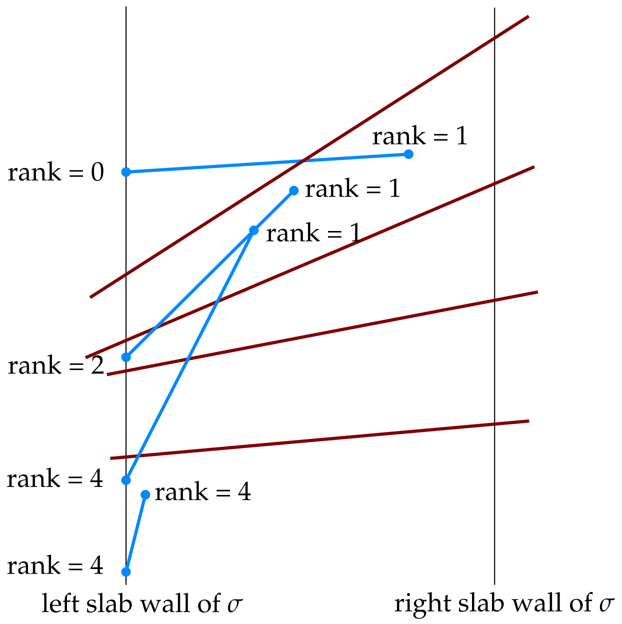


Figure 3.4: Example of a slab σ with 4 long red segments and 4 short blue segments. The ranks of the endpoints of the blue short segments are also presented.

$\mathcal{SL}_{\text{red}}(\sigma)$ is ordered according to the aboveness relation, which means that the first element is the highest one and the last element is the lowest one. Therefore a simple traversal though $\mathcal{SL}_{\text{red}}(\sigma)$ yields the sweep order. During the sweep we can maintain two relevant counts. The first count is the number of red long segments we have encountered so far, call it $longscount$. The second count is the number of blue short segments for which we have encountered its start point, but not its end point, call it $shortscount$.

When we encounter an element e during the sweep, there are three cases.

- The first case is that e is a red long segment. We increment $longscount$ and add $shortscount$ to the visibility index of e. $shortscount$ denotes the number of short segments for which we have encountered their start points, but not yet their end points. We already encountered their start points, so we know that their rank must be lower than the current $longscount$ and because their end points are not encountered yet, those ranks must be higher or equal to the current $longscount$. Hence e intersects with $shortscount$ number of blue short segments.

- The second case is that e is a start point of a blue short segment. We increment the $shortscount$ and set the rank of e to $longscount$. The number of red long segments above e is the number of red long segments already encountered in the sweep.

- The third and last case is that e is an end point of a blue short segment. We decrement $shortscount$, as for this short segment we have encountered both its start and its end point. Then we add to the visibility index of e, the difference between the rank of e and $longscount$. As mentioned before, the difference in rank is the number of red long segments intersected by e.

Long-Long Intersection Sweep

For each slab $\sigma \in \mathcal{L}$ we count the long-long intersections using $\mathcal{LL}_{\text{red}}(\sigma)$ and $\mathcal{LL}_{\text{blue}}(\sigma)$ (ln:10-11). Again the order of the list is according to the aboveness relation, which means that we can traverse both lists and compute the ranks of the clipping points of the red long segments. The ranks are computed for the clipping point on the left wall and the clipping point on the right wall. The difference between these two ranks is the number of blue long segments intersecting the red long segment. A symmetrical sweep is performed to compute the ranks of the clipping points of the blue long segments. Figure 3.5 presents an example of red and blue long segments within a slab σ, including the ranks for both set of long segments.

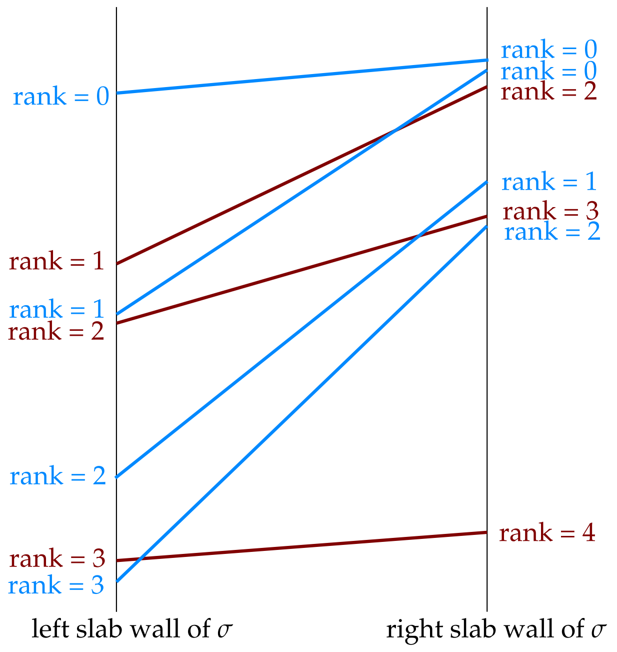


Figure 3.5: Example of a slab σ with 3 long red segments and 4 long blue segments. The ranks of the endpoints of the long segments are also presented.

3.2 Degenerate Cases

This section describes degenerate cases that can occur in the proofs described above. The solutions to handle these cases are also presented.

3.2.1 Dual Half-line Starting Point

The starting point of the dual half-line is not considered part of the dual half-line. This is, as mentioned before, needed to make sure visibility remains well defined. When a point p_j is visible from p_i, according to Lemma 1, p_j is above the critical ray of p_i and p_i is above the critical ray of p_j. If either of the two points lies on the critical ray of the other, it means that the points are not visible from each other. The critical ray corresponds to the starting point of the dual half-line. If an intersection would occur at the starting point, the intersection cannot be counted as visible and therefore it is not included in the dual half-line itself. Considering the starting point not part of the dual half-line also handles the case were a dual half-line starts on an oppositely colored dual half-line, because this is not considered an intersection.

3.2.2 Starting Points on a Vertical Line

It can occur that a red and a blue starting point are on the same vertical line. When this happens we conceptually perturb the blue staring point to the left. In other words, we place the blue staring point before the red starting point in the horizontal ordering. Blue half lines are left-oriented and red half lines are right-oriented, which means that the starting point of a blue half line is the right-most point of the half line and for red half lines the starting point is the left-most point. The two half lines can never intersect, as the starting point is not considered part of the half line and therefore perturbing it to the right can create an invalid intersection.

3.2.3 Intersection on the Slab Boundary

The intersection between two dual half-lines can occur at the slab boundary s. To make sure that the intersection is only counted once, in other words only in one of the two slabs sharing s, we conceptually move blue half lines below red half lines. In one of the two slabs, call it a, the blue half line ($blue$) is above the red half line (red) and in the other slab, call it b, $blue$ is below red. If this is not the case, the intersection would not occur as either $blue$ is completely above red or red is completely above $blue$ in $a \cup b$. Moving $blue$ below red increments the rank of $blue$'s clipping point at s. The rank of the clipping point of $blue$ in a at the other slab boundary is lower than that of red, as red is below $blue$. This means that the difference in ranks between the two clipping point is incremented by one, which corresponds to the counting of the intersection. The rank of $blue$'s clipping point at the other boundary in b is higher than that of the clipping point of red, as red is above $blue$. The difference between the ranks of the $blue$ clipping points is not incremented with one, as the ranks of both clipping points is higher than those of red. This means there is no intersection counted. A symmetrically argument holds for red's clipping points at the other boundaries. In summary, this means that the intersection is counted in slab a and not in slab b and hence is only counted once.

3.3 Running Time

The running time analysis is split up in different paragraphs, each paragraph analyzes a part of the algorithm and the last paragraph analyzes the complete running time. The line numbers refer to Algorithm 1.

Building a Convex Hull

Building a convex hull takes $O(m \log m)$ time [5], where m denotes the number of points for which the convex hull needs to be constructed. This is because it needs to sort the points lexicographically. In our case, the points are already sorted (cells are in order) and therefore the running time can be reduced to $O(m)$.

Duality Computations and Red-Blue Intersections

The duality computation consists of a conversion from the primal plain to the dual plane and the intersection counting. The conversion takes $O(1)$ per cell and therefore takes $O(m)$ in total, where m denotes the number of points of both sets. The red-blue intersection counting uses a modified version of the algorithm from Palazzi & Snoeyink [17], which runs in $O(m \log m)$. The modifications do not change the running-time analysis of Palazzi & Snoeyink [17] and therefore their analysis remains valid.

Critical Rays

Updating the critical rays takes $O(\log h)$ per ray (binary search for the tangent), where h denotes the number of points in the upper hull.

The Recursion

A single call on n cells excluding the time for the recursive calls takes $O(n \log n)$ time. The first computation in the step is the duality computation at line 7 and 8. The duality computation takes $O(n \log n)$ time, because for n cells the conversion needs to be done and there are n dual half-lines in the intersection counting. Then the convex hulls for both halves are computed at line 9, which takes $O(n)$ per convex hull. The convex hulls are used to update the critical rays of the other part at line 10. This means that for each cell the running time for updating is $O(\log n)$. So in total updating costs $O(n \log n)$. Hence that a single call takes $O(n \log n)$.

Total Running Time

The complete algorithm takes $O(n \log^2 n)$. Let $T(m)$ denote the time needed for a recursive call with $r - l + 1 = m$. Then $T(m)$ satisfies

$$T(m) = 2T(\frac{m}{2}) + O(m \log m) \tag{3.1}$$

By the Masters Theorem we thus get $T(m) = O(m \log^2 m)$, so the total running time is $O(n \log^2 n)$.

Chapter 4

The 2-Dimensional Problem

In this chapter we study the 2-dimensional version of the problem. The input for the 2-dimensional version of the problem is a grid of $n \times n$ cells given as $T[1..n][1..n]$, where $T[i][j]$ denotes the elevation of (the center of) the cell (i, j). As in the 1-dimensional version, the goal is to compute the visibility index of each grid cell $T[i][j]$. If we compute the visibility index for each cell using the single-viewshed approach proposed by Van Kreveld [20]. The running time per cell is $O(n^2 \log n)$ and therefore the complete running time would be $O(n^4 \log n)$. The lower bound for computing the total-viewshed of a grid terrain is $\Omega(n^4)$, as for each cell it is possible that all other cells are visible. Hence, using the total-viewshed for computing the visibility index is rather inefficient. We think that an efficient exact algorithm is not very likely to exists and we will try to approximate the visibility index by using 1DVisibilityIndex from Chapter 3. The basic idea of the algorithm is to generate lines on the grid and then for each line compute the visibility index using 1DVisibilityIndex . In the end we combine the results from the lines into a single visibility index approximation. Before we describe the general idea of 2DVisibilityIndex (Section 4.2), we first present how we define visibility of grid cells in Section 4.1.

4.1 Visibility Definition

The representation of the terrain plays an important role in the visibility of cells. As discussed in Chapter 1 there are multiple ways of representing a terrain. We will be using a Digital Elevation Model (DEM). There are two approaches for representing a terrain using a DEM. In the first approach the elevation of a cell corresponds to the elevation of its center point (the center point representation). Non-center points are interpolated based on neighboring center points. The advantages of this is that it is more realistic and that the terrain is continuous. The disadvantages is that you need to interpolate for non-center points. In the second approach all points in a cell have the same elevation (the stepped representation). The advantage of this approach is that it is easier to compute the elevations of non-center points. The disadvantages is that the elevation profile is less realistic.

We are not interested in the complete elevation model of the terrain, but in the elevation profile along a line ℓ. So for every cell that intersects ℓ, we want a vertex in the elevation profile. For the different approaches the position of the chosen vertex/vertices in the grid differs. For the stepped representation the complete elevation profile can be constructed using two vertices per step, while for the center point representation there are a lot of vertices required for the complete elevation profile. Therefore we use a representation that is a simplified version of the center point representation, as it is more realistic than the stepped representation and still only needs one vertex per intersected cell. It is a simplified variant of the triangulation of the terrain, therefore we will call it the simplified representation. An example is presented in Figure 4.2. For each cell we conceptually create a diagonal. The intersections of the line and the diagonals are included in the 1-dimensional terrain we generate along ℓ. The elevation of the intersection point is interpolated using the four neighboring center points (just like the center point representation). The interpolation is a weighted average of the elevations, where the weights are the distances from the intersection point to the center points. Note that we will address challenges for both the simplified representation as the stepped representation, as both (or a variant of) are commonly used.

Figures 4.1 and 4.2 present examples of the implications the different approaches have. All approaches can be used in the visibility definition presented in Definition 1. Each figure presents the vertices that form the elevation profile. Figure 4.1 uses the stepped representation and Figure 4.2 the simplified representation. For the stepped and the simplified representation the number of relevant points can be determined based on the number of cells intersecting the line. The stepped representation needs two points per cell and the simplified representation needs one point per cell. This means that in total we only need $O(n)$ points to create the elevation profile of a line, as a line intersects at most $2 \cdot n$ cells. Determining the location of the intersection points is also easier. We will leave other representations and the evaluations to future research (also see Chapter 8).

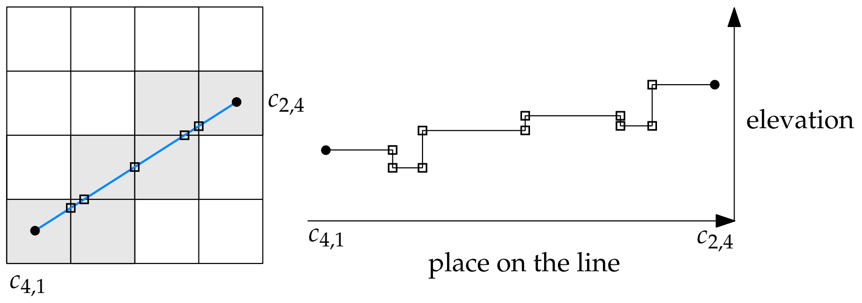


Figure 4.1: An example of the vertices on the segment $c_{4,1}c_{2,4}$ using the stepped representation. On the right the elevation profile along the line depicted on the left is shown.

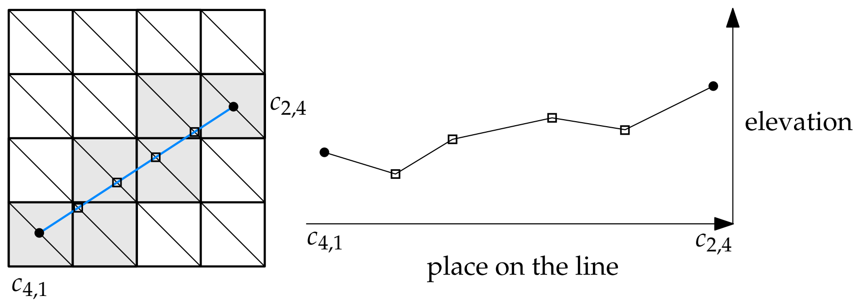


Figure 4.2: An example of the vertices on the segment $c_{4,1}c_{2,4}$ using the simplified representation. On the right the elevation profile along the line depicted on the left is shown.

4.2 General algorithm

The general approach of the algorithm is presented in pseudo code in Algorithm 3. The idea is to generate lines, according some scheme, and combine the results. The different schemes for generating lines are discussed in Section 4.2.1. Generating different lines give different results for the visibility index of a cell. How we combine these results is discussed in Section 4.2.2. The analysis of the running time can be found in Section 4.2.3. To be able to use 1DVisibilityIndex , 1DVisibilityIndex and its input needed to be modified slightly. These modifications are discussed in Section 4.2.4.

Algorithm 3 2DVisibilityIndex(T)

```
Input: grid $T$ of $n \times n$ cells with elevations.
Output: $VisIndex[i][j]$ is an approximation of the visibility index of cell $T[i][j]$.
1: $\mathcal{D} \leftarrow$ GenerateLines ($T$)          ▷ Generates lines according to some scheme
2: $VisIndex[i][j] \leftarrow 0$ for all $1 \leq i \leq n, 1 \leq j \leq n$.
3: for all $l \in \mathcal{D}$ do
4:     $\mathcal{S} \leftarrow$ CreateArray ($l$)
5:     $VisIndexLine[k] \leftarrow 1$ for $1 \leq k \leq \mathcal{S}$.size
6:     $Crits[k] \leftarrow$ initial critical rays straight down for $1 \leq k \leq \mathcal{S}$.size
7:     $result \leftarrow$ 1DVisibilityIndex ($\mathcal{S}, 1, \mathcal{S}$.size, $VisIndexLine, Crits$)
8:     for all cells $c$ in $S$ do
9:         Determine indices $i, j$ such that $c$ corresponds to the cell $T[i][j]$ and combine the visibility index from $result[c]$ with $VisIndex[i][j]$
10:    end for
11: end for
12: return $VisIndex$
```

4.2.1 Generating Lines

In this section we will present a scheme for approximating the total visibility index. The most simple approach is to generating lines randomly. Randomly generating lines can be done efficiently, but the downside is that there are cells without a line intersecting them or with only a few lines intersecting them. To overcome this, we will present a structured approach. The idea of this approach is to make sure all cells have m lines intersecting them and those lines are in m different equally-spaced directions.

There are m equally-spaced directions, which means that we have angles of $\frac{180}{m}$ degrees between consecutive directions. For each of the m lines we generate n parallel lines to make sure that each cell intersects with m lines. Figure 4.3 shows an example of generating these lines for a single cell and $m = 4$. For $m = 4$ there is a horizontal, a vertical and two diagonal lines intersecting each cell. Figure 4.4 shows the lines that are generated for $m = 3$. Using this generation scheme there are in total $m \cdot n$ lines.

4.2.2 Combining Line Results

Combining the visibility index results of different lines can be done in several ways. A simple way is to use the (unweighted) average. Although it is simple to use, it ignores the area for which the

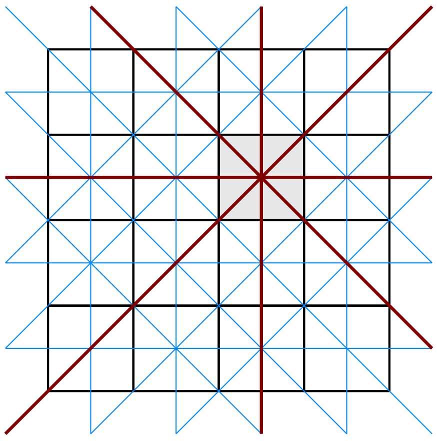


Figure 4.3: An example of generating lines (blue) in a structured manner for $m = 4$. The lines intersecting a single cell (gray) are depicted in red.

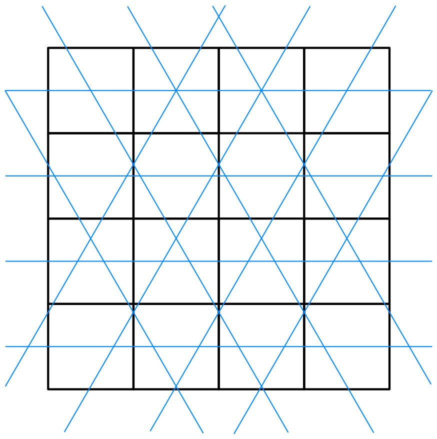


Figure 4.4: An example of generating lines (blue) in a structured manner for $m = 3$.

visibility index is averaged. Figure 4.5 shows that the area for which the results are averaged can differ a lot. Instead an weighted average, which takes into account the area for which the visibility index is averaged, is probably better. Such a weighted average can be computed in the following fashion. The visibility index of 1DVISIBILITYINDEX of a cell c is separated into two values, cells that are visible to the left of c and cells that are visible to the right of c. For each partition we compute the average visibility index based upon the two lines that envelope that partition. To compute the overall visibility index we use the unweighted average and leave this idea for improvement to future research (also see Chapter 8).

This requires 1DVISIBILITYINDEX to compute visibility index information separate for cells that have a lower index and cells that have a higher index. This modification is straightforward as the algorithm is based upon left-right separation of data (see the recursion pattern in Chapter 3). This and some other modifications are discussed in Section 4.2.4.

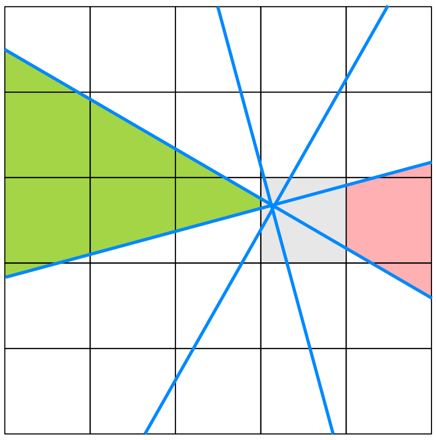


Figure 4.5: An example of partitions for a cell (gray), the overall visibility index is computed using a weighted average. The weights are the area of the corresponding partition. Two example partitions are shown in green and red.

4.2.3 Running Time Analysis 2DVisibilityIndex

As 2DVisibilityIndex is not an exact algorithm, we do not need to analyze its correctness. How good the approximation is something we do want to know and will be tested by the experiments from Section 5.3 and can be found in Section 6.2. However, analyzing the running time is straightforward. For simplicity we assume that there are m lines generated by GenerateLines. We know that T is a grid of $n \times n$ cells and therefore we know that the amount of cells covered by a line is at most $2 \cdot n$. 2DVisibilityIndex is split in two phases, the first phase is the generation of the lines and the second phase is the visibility index computation of the lines.

The running time of the first phase depends on the scheme used to generate them. We assume that we can generate m lines in $O(m)$ time.

The second phase of the algorithm consists of three parts. First we construct the array $\mathcal{S}$ of cells that are covered by l. This requires $O(n)$ time, as we can compute the indices of the cells that are below l and if needed interpolate between different (neighboring) cells in $O(n)$ time. We used a modified version of the algorithm presented by Bresenham [2].

The second part is the call of 1DVisibilityIndex . The running time of 1DVisibilityIndex is $O(n \log^2 n)$ (see Section 3.3).

The last part is to incorporate the results of 1DVisibilityIndex for m lines. If we use the unweighted average, we can combine this in $O(m)$ time per cell. If we use the weighted partitioned average, the running time per cell is also $O(m)$ per cell. For each partition we can compute the size of its area in $O(1)$, as they can be represented as a couple of triangles (also see Figure 4.5). So the running time of the third part is $O(m \cdot n^2)$.

These two examples use averages to combine the results. For averages it is possible to do some extra work after each call to 1DVisibilityIndex and thereby reducing the total amount of work in the end. There are two approaches to compute the average more efficiently. The first approach computes the running average after each call to 1DVisibilityIndex and the second approach is to sum the results and only compute the average at the end. The first approach can be done in $O(1)$ per cell and there are at most $2 \cdot n$ cells intersecting a line, resulting in $O(n)$ time per call. While the second approach can be done in $O(1)$ per cell intersecting a line and an extra step at the end of the algorithm of $O(1)$ per cell in the grid, resulting in $O(n)$ per call and a $O(n^2)$ in the end. We generate m lines, so the total time needed for the first approach is $m \cdot O(n)$ and the time for the second approach is $m \cdot O(n) + O(n^2)$.

We prefer to use the first approach as it is slightly faster and easier to analyze. It is clear that different combing approaches result in different running times. We generate m lines and therefore the total running time of the second phase is $m \cdot (O(n) + O(n \log^2 n) + O(n)) = m \cdot O(n \log^2 n) = O(m \cdot n \log^2 n)$.

Hence the total running time of 2DVisibilityIndex is $O(m) + O(m \cdot n \log^2 n) = O(m \cdot n \log^2 n)$. The total running time of a combing approach that cannot be done efficiently during execution and is therefore postponed to the end of the algorithm would be $O(m) + O(m \cdot n \log^2 n) + O(m \cdot n^2) = O(m \cdot n^2)$ (assuming $O(m)$ time needed to combine the results of m lines per cell).

4.2.4 Modifications of 1DVisibilityIndex and Its Input

Modifications to 1DVisibilityIndex

There are two small and simple modifications required to 1DVisibilityIndex to better integrate 1DVisibilityIndex into 2DVisibilityIndex . 1DVisibilityIndex requires an array with elevations, where the $T[i]$ denotes the elevation of cell i, which corresponds the point $(i, T[i])$ (see Chapter 3). 1DVisibilityIndex uses these points to determine if cells are visible from one another. We modify the algorithm slightly to use the points $(T[i].x, T[i].y)$ instead of the point $(i, T[i])$. This means that the point of cell i now corresponds to $(T[i].x, T[i].y)$.

The second modification to the 1DVisibilityIndex is, that we split the visibility index into two parts. The visibility index of cells to the left of cell i and the visibility index of cells to the right of cell i or as mentioned before, the visibility index is separated for cells with a lower index and cells with a higher index. This modification is very simple, as 1DVisibilityIndex already executes the computations separately for left and right. 1DVisibilityIndex is based upon the left-right separation of the input (see the recursion pattern in Chapter 3). The second modification is required for some of our ideas and can be used in future research.

Input Modifications for 1DVisibilityIndex

2DVisibilityIndex uses 1DVisibilityIndex as a subroutine and 1DVisibilityIndex has a few preconditions. The first precondition is that all input data must be non-negative and integral and the second precondition is that all cells must have a unique index. For the stepped DEM representation the first precondition is satisfied, as all elevations in the grid are non-negative and integral. For the simplified representation the elevations of non-center points could be non integral. Those elevations are positive, as they are computed based on positive center point elevations. We solve this by just rounding the elevations to the nearest integer (ties are rounded up). The index of a point on the line could be non-integral, as the distance from that point to the staring point of the line is non-integral. They are non-negative, as distances are non-negative. We round all the computed indices down to the nearest integer.

The second precondition requires that cells have a unique index. For the simplified representation this is not really a problem, as at each intersection of the line and a diagonal there is only one point needed to represent the elevation. For the stepped representation we cannot satisfy this precondition trivially, as we need to represent the terrain with steps. At the boundaries of the steps there are two points required to indicate the elevation change. The index of these two points is identical, as the elevation change is instant. Both 1DVisibilityIndex and the implementation do not accommodate for this. 1DVisibilityIndex initializes the critical rays straight down. Points on the critical ray are not considered visible, as the closest point obstructs the visibility of the farthest point. If all cells have a unique index this is not a problem. In our implementation we simplified this a bit further, by initializing the critical rays to go through the neighboring cell at -1 (see Appendix A.1.2).

To mediate this problem we can do the following. First scale all data with a factor f. Then we compute the points and move them slightly if there are points with the same index. The points on the line where this occurs is at the edges of a step, points where we go from one cell to the next. Consider such a point p with distance d to the starting point of the line. The point from the current step is placed at distance $d-1$ and the point of the next step at distance d. An example is presented in Figure 4.7. It could be the case that an intersection point occurs at one of the corners of a step, for example a line with angle of 45°. In that case we place the point of the current step at $d-1$, the point of the next step at $d+1$ and at the intermediate index we place a point with the maximum index of the neighboring steps (excluding the step we just processed). Figures 4.6 to 4.8 present different examples. This last point is required to make sure you cannot look beyond the edge of a step.

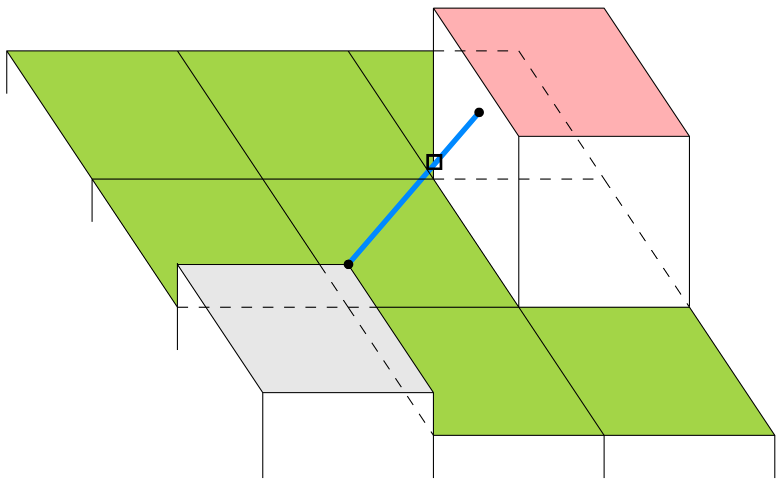


Figure 4.6: An 3-dimensional example of the stepped representation. The red cells have a higher elevation as the lower left corner cell (gray) and the green cells have a lower elevation. The visibility from the corner of the lower left corner cell to the center of the upper right corner cell (denoted by the blue line) is obstructed by the edge of the step of the red cell, the square denotes the point of obstruction.

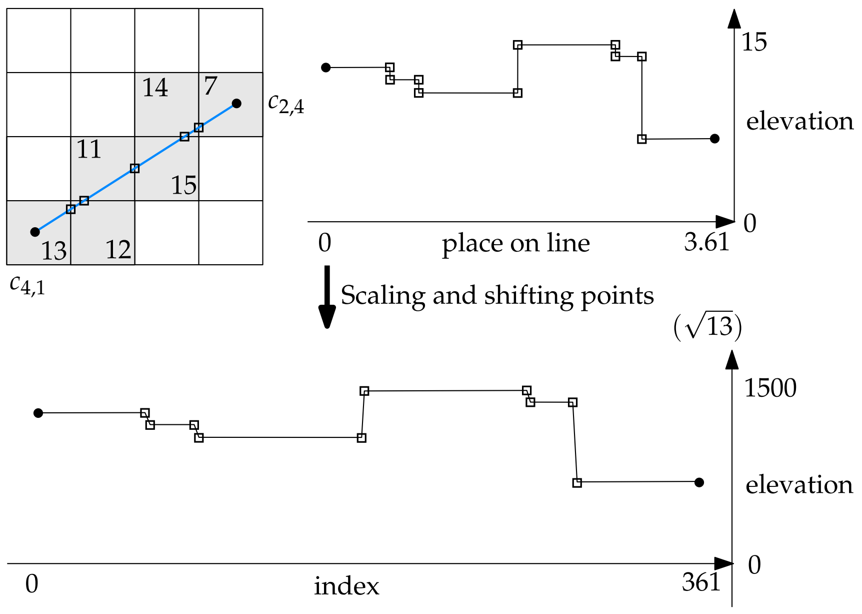


Figure 4.7: An example of the result of the steps we do to mediate the problem with non-unique indices in the stepped representation with $f = 100$. The numbers within the cell represent the elevation of that cell.

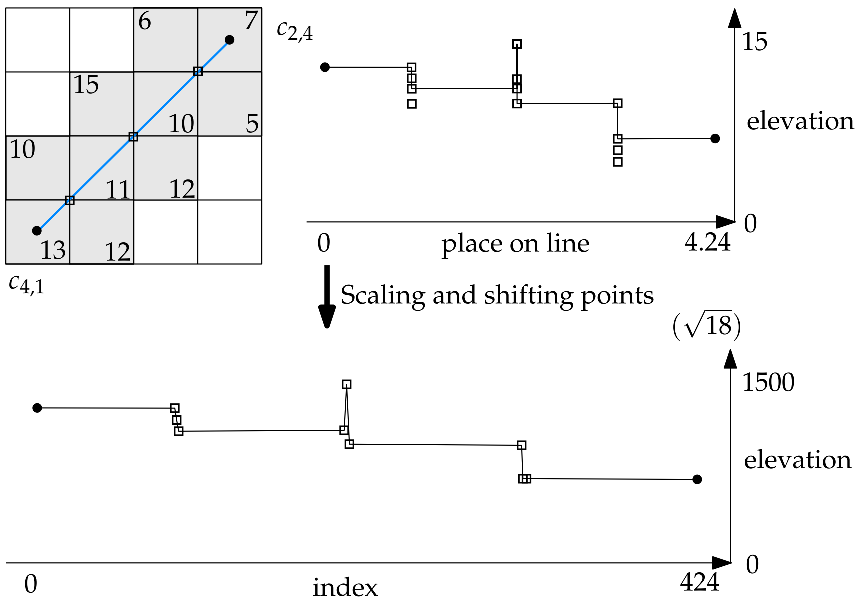


Figure 4.8: An example of the result of the steps we do to mediate the problem with non-unique indices with $f = 100$ in the stepped representation with a intersections between the line and four cells. The numbers within the cell represent the elevation of that cell.

Chapter 5

Experiments

The theory presented in Chapters 3 and 4 is tested in practice by performing experiments. The details of the experiments and the experimental set-up are presented in this chapter. Some important design decisions and their implications on the implementation are presented in Appendix A. Section 5.1 describes how the experiments are performed, the input is generated and presents the environment specifications. The actual experiments are presented in Sections 5.2 and 5.3 for 1DVisibilityIndex and 2DVisibilityIndex , respectively.

5.1 Experimental Set-up

The running time of both algorithms, 1DVisibilityIndex (Algorithm 1) and 2DVisibilityIndex (Algorithm 3), are compared against a brute-force approach to verify that they are indeed efficient. Running the brute-force approach also means that we can check the outcome of both algorithms. 1DVisibilityIndex is an exact algorithm and 2DVisibilityIndex is an approximation algorithm, therefore we can verify the correctness of 1DVisibilityIndex and test how good the approximation is of 2DVisibilityIndex .

5.1.1 Experiments

This subsection gives a general overview of the experiments. The detailed experiments are described in Sections 5.2 and 5.3 for 1DVisibilityIndex and 2DVisibilityIndex , respectively. The experiments are executed as C++ unit tests. Each unit test, corresponding to an experiment, follows the same structure. First it loads the data into the correct structures. Then it calls the brute-force approach and measures the time it took from calling the function till returning from the function. The time needed to execute the function is then outputted. In a similar fashion 1DVisibilityIndex is called and its execution time is measured and outputted. The last step is to verify the correctness by comparing the results from the brute-force approach to the results from 1DVisibilityIndex . A similar approach is used to conduct the experiments for 2DVisibilityIndex . Instead of verifying the correctness in the last step, we check how good the approximation is.

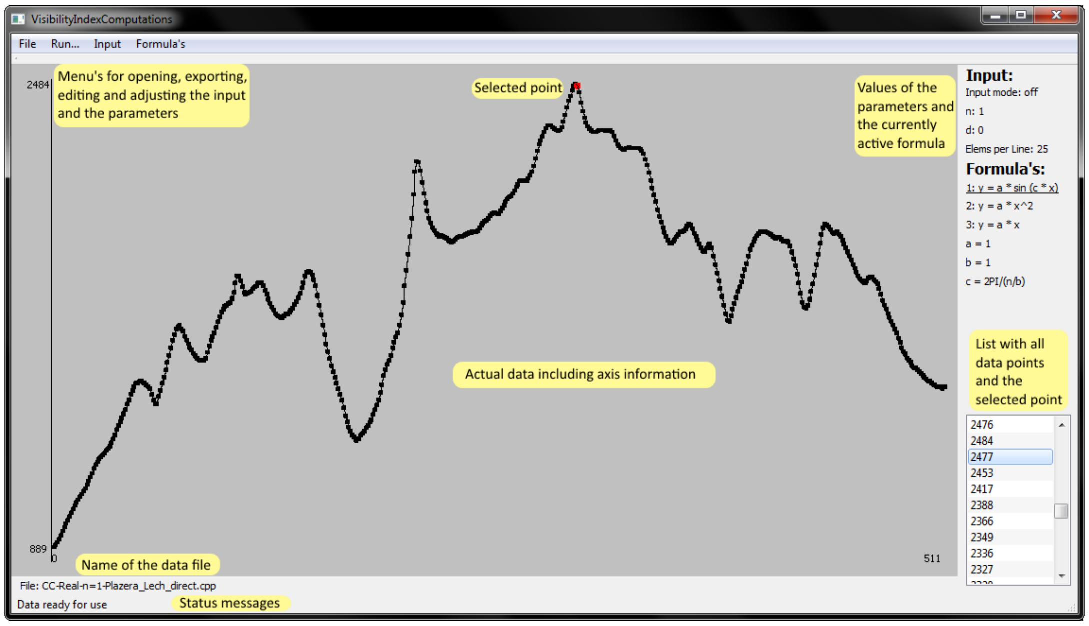
The time measurement is done in milliseconds and using the `std::chrono::steady_clock` package. Each experiment is repeated ten times, with the exception of running the brute-force approach. The brute-force approach is run only three times, due to its long running time. The implementation is compiled using the standard compiler options without any flags for optimization.

5.1.2 Data Set Generation

There are two different kinds of data, the first is used in the experiments of 1DVisibilityIndex and the second is used in the experiments of 2DVisibilityIndex . First we will discuss the data generation for 1DVisibilityIndex and then for 2DVisibilityIndex . For 1DVisibilityIndex we can generate data using a GUI. The created data sets can be exported in three different file formats. The first format is used for importing it back into the GUI and is text based, the second format is used for inclusion into the code and the last export option is used to generate Ti*k*Z output for image inclusion in this document.

Data Set Generation for 1DVisibilityIndex

To generate the inputs for the experiments, we use a graphical user interface (GUI). Screenshots of the GUI can be found in Figures 5.1 and 5.2. Screenshots showing the less interesting parts of the GUI, e.g. the menu structure, can be found Appendix D. The GUI is used for viewing and creating data sets. There are different ways of creating data sets, ranging from using formulas to simply clicking inside a drawing panel.


VisibilityIndexComputations
File
Run...
Input
Formula's
Menu's for opening, exporting, editing and adjusting the input and the parameters
Selected point
Values of the parameters and the currently active formula
Actual data including axis information
List with all data points and the selected point
Name of the data file
Status messages
2484
889
0
511

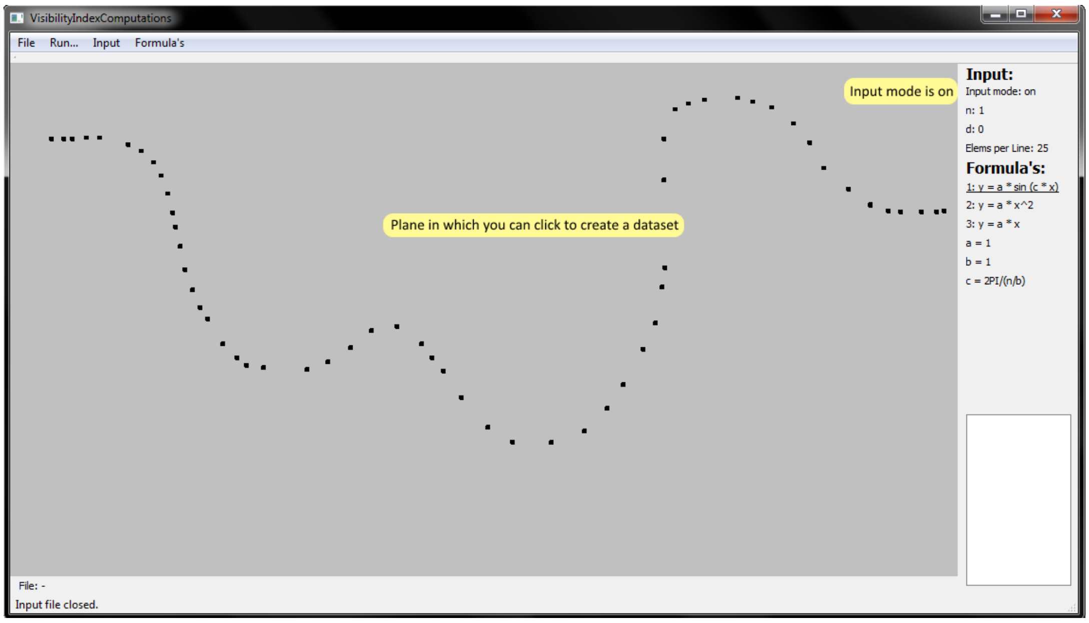
Input:
Input mode: off
n: 1
d: 0
Elems per Line: 25
Formula's:
1: y = a * sin (c * x)
2: y = a * x^2
3: y = a * x
a = 1
b = 1
c = 2PI/(n/b)

2476
2484
2477
2453
2417
2388
2366
2349
2336
2327

File: CC-Real-n=1-Plazera_Lech_direct.cpp
Data ready for use

Figure 5.1: An annotated screenshot of the GUI.

The first option is to use the drawing panel (also see Figure 5.2). Each click creates a point and repeatedly clicking creates a data set. This created data set will typically be small, as we probably will not click a thousand times. Therefore we added the option to add extra points in between two consecutive user points. These points are placed along the line between the two consecutive user points at even spacing. There is also an option to allow the points the deviate a bit from their original position. When the dispersion d is set, each point with y-value y will be placed randomly in the range from $[y - d, y + d]$ using an uniform distribution. These two values are used during the export of the data when you choose the option “export with randomization”. The original data set, without added randomization, can also be exported with the option “export”. These two options can also be used to add randomization to a data set loaded in the main program. This is especially handy for increasing the size of the real-life data sets.


VisibilityIndexComputations
File
Run...
Input
Formula's
Input mode is on
Plane in which you can click to create a dataset

Input:
Input mode: on
n: 1
d: 0
Elems per Line: 25
Formula's:
1: y = a * sin (c * x)
2: y = a * x^2
3: y = a * x
a = 1
b = 1
c = 2PI/(n/b)

File: -
Input file closed.

Figure 5.2: An annotated screenshot of the GUI when in input mode.

There is a second option to create data sets, which generates semi-random data sets. The previous two data generators are based around a skeleton input, often created by the user. This approach lacks a decent randomization. Another approach would be to generate a completely random data set. This approach is far from ideal as the elevations would hop around from low to high and vice versa. So we have tried to create a method that tries to be in the middle of these two (opposite) approaches. We fix the dispersion d and the number of points n and let the method generate a semi-random data set. Some examples are presented in Figures 5.4 and 5.5. The method generates points in the following manner. For each point it generates, it uses the elevation of the last generated point. So we have the last point (x, y), then the next point is $(x + 1, y')$ where $y' \in [y - d, y + d]$ (uniformly distributed). We repeat this step until we have n points. The first point is fixed at $(0, 0)$ and after the generation is complete we shift the data set in such a way that all elevations are non-negative.

Because both of these options generate data without a real structure, we have also included the option to generate data using three formulas. The formulas we use are a parabola, a line and a sine. The parabola allows us to generate data sets in which all cells are visible from all other cells, a line to make only the neighbors visible and the sine to vary the number of peaks and valleys present in the data set. With the parabola data sets it is possible to limit the visibility to the neighbors, but due to some implementation limitations (see Appendix A.4) we use the line data sets for that. For each of these formulas some parameters (subset of a, b and n) can be specified, where n is the number of cells, a is the coefficient or the amplitude and b the number of peaks. After the data sets are generated, the points are shifted to make sure all values are non-negative. An example of the sine data set can be found in Figure 5.3.

Name	Formula	Parameter usage
Parabola	$y = a \cdot i^2$	$i \in [-\frac{n}{2}, \frac{n}{2}], a \in \{1, -1\}$ a determines whether it is a peak or valley parabola
Line	$y = i$	$i \in [0, n-1]$
Sin	$y = a \cdot \sin \frac{2 \cdot \pi \cdot b}{n} i$	$i \in [0, n-1]$ b is the number of peaks in $[0, n-1]$ a is the amplitude

Table 5.1: Table with the different formulas used for generating data sets and their parameters

Data Set Generation for 2DVisibilityIndex

The data sets for 2DVisibilityIndex are generated using the DEM Explorer of GeoBrain [23]. The advantages of the DEM Explorer is that you just can select the cells you want to export using a web-based interface. The generated data sets we use comes from the GTOPO30 data set and are in the ArcASCII format. Other data sets and formats are available. We did some small manual modifications to the format to be able to use it in our implementation and to visualize them for inclusion in this document. The experiments for 2DVisibilityIndex only use real-life data sets and we leave the study of other (synthetic) data sets for future research.

5.1.3 Environment

The implementation of the algorithms is done in C++11 using NetBeans [29] with CygWin [22] and Qt Creator [30] with MinGW [28]. NetBeans was used to implement the algorithms and Qt Creator was used to create the GUI. For all software it holds that the 64-bit version was used, when available. The operating system running on the machine is Windows 7 Professional 64-bit with Service Pack 1.

The machine is a HP Elitebook 8540w with a Intel Core i5-540M dual core processor (2.53 GHz with hyper-threading) and 4 GB of DDR3 RAM. Besides the normal hard-disk of 320 GB it also has a Samsung 840 EVO solid-state drive of 120 GB. The software used for the implementation and the experiments is installed on the solid-state drive, just like the operating system.

5.2 Experiments 1DVisibilityIndex

The experiments for 1DVisibilityIndex only consists of running-time comparisons. The verification of the correctness is only used to test the implementation. For each of the experiments it must hold that the outcome of 1DVisibilityIndex is the same as that of the brute-force approach. The data sets used for the experiments are presented below. For each of the experiments we will present a table of the different parameters and where possible a figure of the skeleton input.

The different parameters to generate the data sets are presented in Table 5.2. The number of cells for the parabola are clipped at 2000, because using more cells would trigger an integer overflow (discussed in Appendix A.4). Figures 5.4 and 5.5 present examples of data sets generated by the semi-random approach for different values of d, more examples of semi-random generated data sets can be found in Appendix B. Figure 5.3 presents an example of the data set generated by using the sine formula.

Formula	**Parameters**	**Number of cells**
Line		for increasing n from 1,000 to 500,000.
Parabola	$a = \{1; -1\}$	for increasing n from 100 to 2,000.
Sine	$a = 25,000$ $b = \{15; 30\}$	for increasing n from 1,000 to 500,000.
Semi-Random	$d = \{10; 15\}$	for increasing n from 1,000 to 500,000.

Table 5.2: The parameters for the generated data sets.

We also conducted experiments using real-life data sets. We used several sources for generating this data. The first source is the Geocontext-Profiler [25] and we use it to generate an elevation profile from a point to another point. The second source we use has two steps. Firsts we create a route on CycleRoute [21]. Then we load this route into GPSVisualizer [27] and export the elevation profile. For each generation type we present an elevation profile below. For some data sets the number of cells is rather large and that means there are a lot of elevations points. To make the image readable we sample only a part of the elevation profile in the figure. Important to note is that the elevations can differ per source, for example Figure 5.7 versus Figure 5.8. The data is from different sources, but from the same region. To be more precise, the start and end points are the same and therefore have the same elevation in the real world, but they do have different elevations in the data sets. More examples of real-life data sets can be found in Figures 5.6 and 5.9.

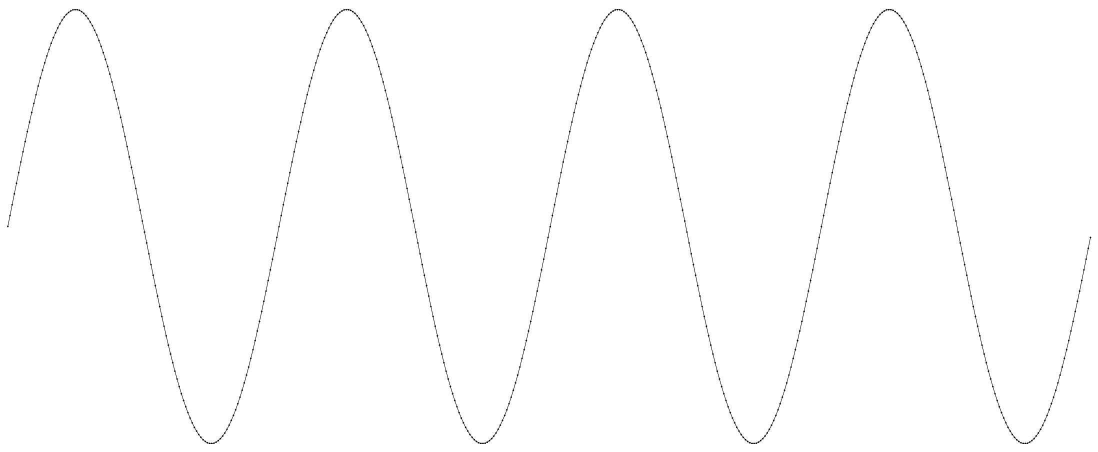


Figure 5.3: Generated data set with 500 cells using the sine formula, with a is 1,000, b is 4.

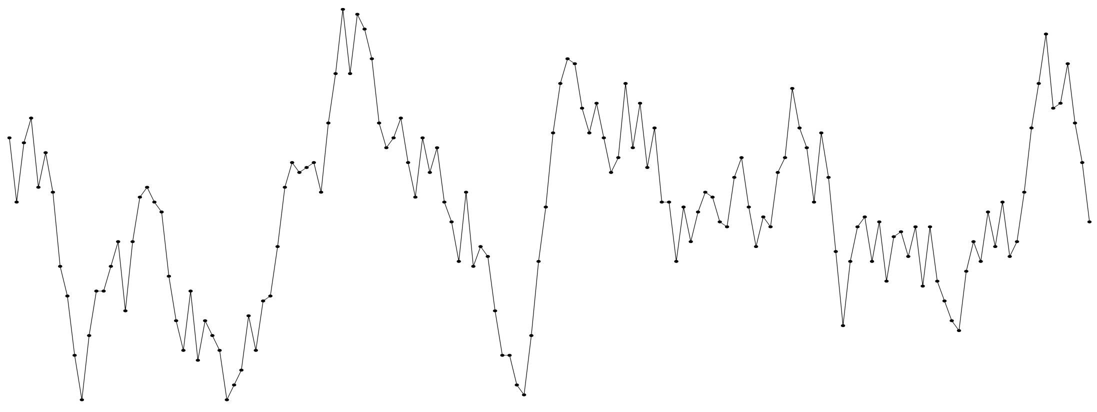


Figure 5.4: Semi-random generated data set with 150 cells and elevations in between 0 and 79, the dispersion was set to 15.

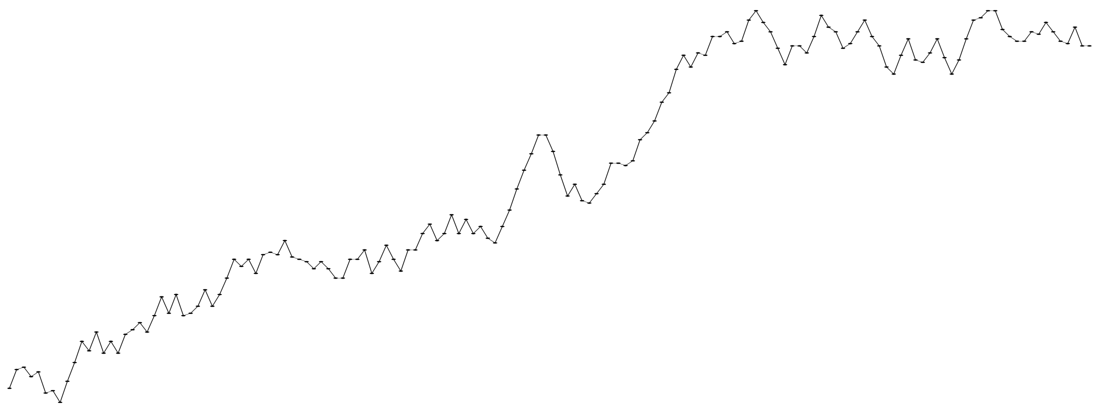


Figure 5.5: Semi-random generated data set with 150 cells and elevations in between 3 and 170, the dispersion was set to 10.

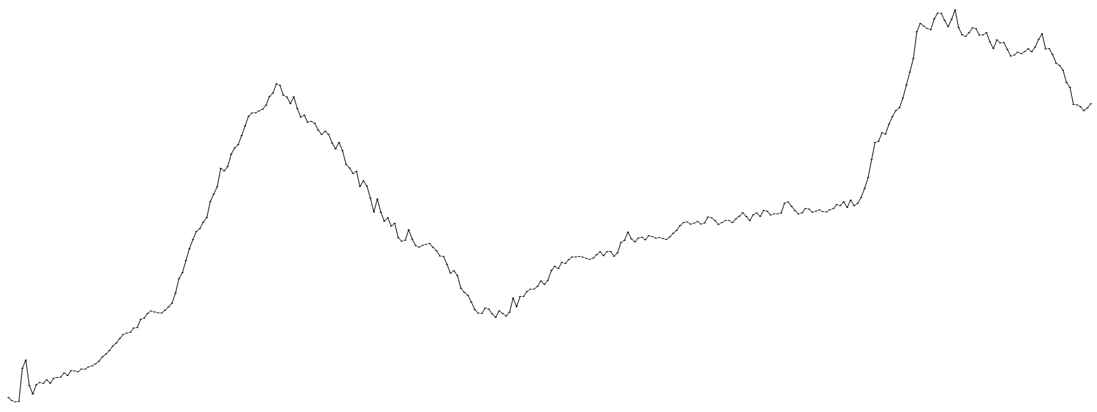


Figure 5.6: The elevation profile of the cycle route from Baden-Baden to Freudenstadt (in Germany) with 3,113 cells and elevations in between 8,000 and 42,000 (sampled at every 10^{th} cell).

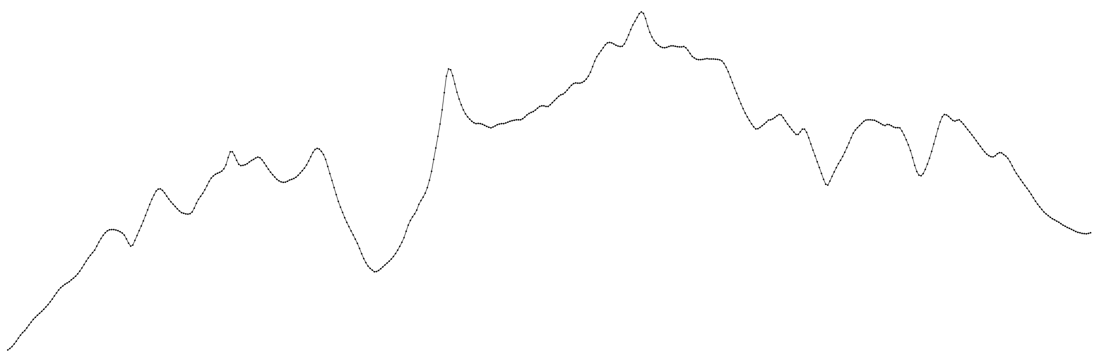


Figure 5.7: The elevation profile from Raggal-Plazera to Lech (in Austria) with 512 cells and elevations in between 889 and 2,484.

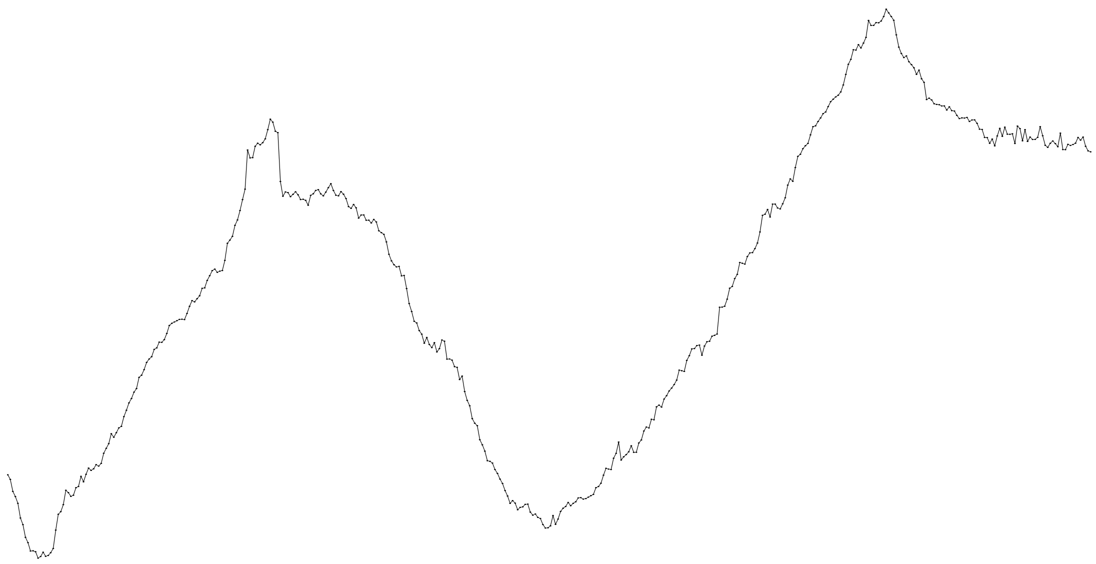


Figure 5.8: The elevation profile of the cycle route from Raggal-Plazera to Lech (in Austria) with 4,299 cells and elevations in between 8,000 and 42,000 (sampled at every 10^{th} cell).

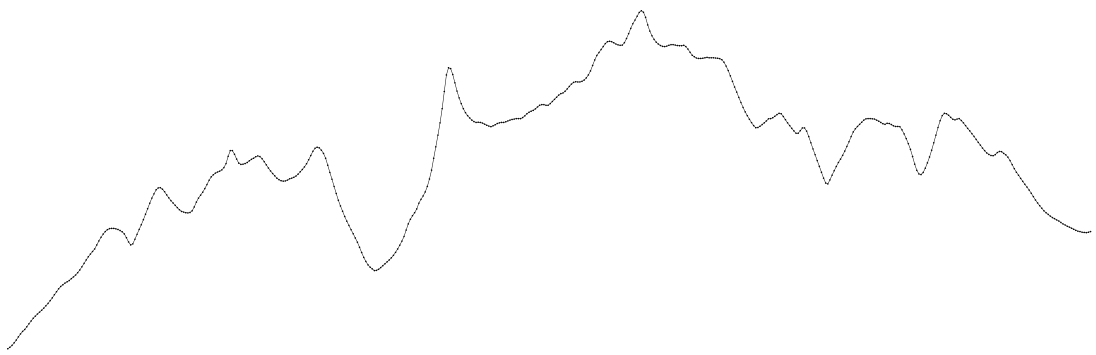


Figure 5.9: The elevation profile from Innerbraz to Warth (in Austria) with 513 cells and elevations in between 705 and 2,648.

5.3 Experiments 2DVISIBILITYINDEX

The experiments for 2DVISIBILITYINDEX are conducted to see how good the approximation is. We have conducted a specific experiment to see how many lines we need to get close to the actual visibility index. We do this by increasing m in each step. For each step we generate m lines in m different directions for the lower left corner cell. Figure 5.10 shows an example of the experiment for $m = 5$. In this experiment the brute-force approach is relatively fast, as it only requires to check the visibility of each cell with respect to the lower left corner cell. The different values for m range from 2 to 21 lines, where the horizontal and the vertical line are always present.

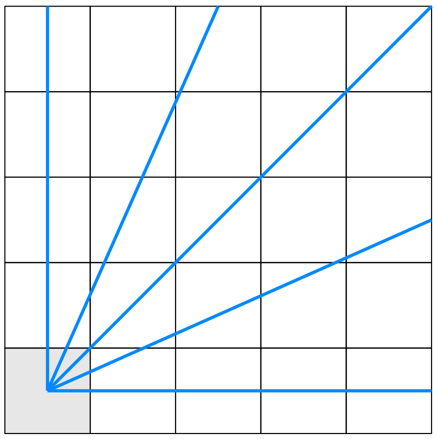


Figure 5.10: Illustrates how the lines (blue) will be generated for single corner cell (gray), $m = 5$

Several data sets are presented below, Figures 5.11 to 5.13, and the rest of the data sets can be found in Appendix C. The visualization of the data sets uses a heat map to indicate the elevation of each cell. A cell with a low elevation is yellow and a cell with a high elevation is red. The minimum and maximum elevation are presented in the legend of the heat map to indicate the boundaries of the color spectrum.

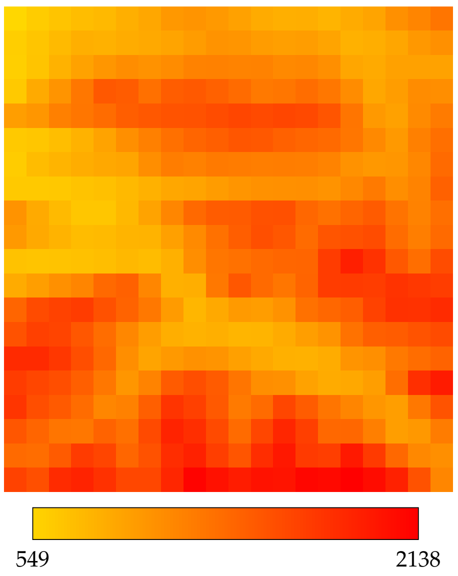


Figure 5.11: The data set with the cell of the Glatthorn (Austria) in the lower left corner. The data set consists of 20 × 20 cells

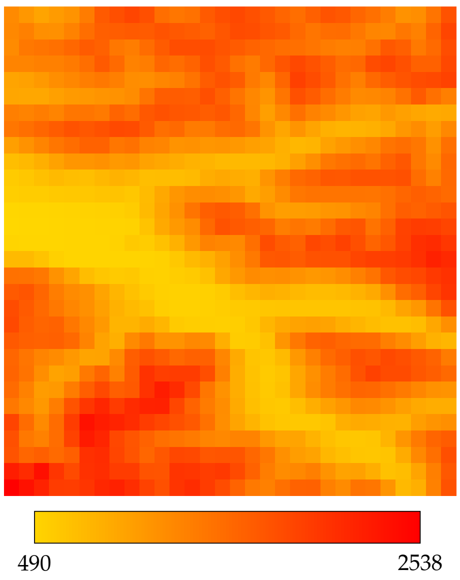


Figure 5.12: The data set with the cell of the Schesaplana (Austria) in the lower left corner. The data set consists of 30 × 30 cells

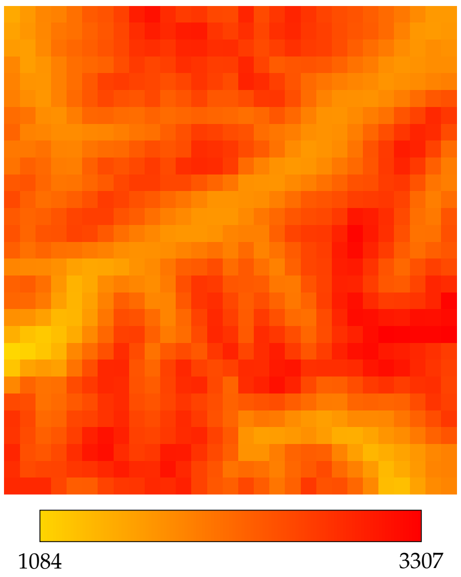


Figure 5.13: The data set with the cell of the Piz Badile (Italy) in the lower left corner. The data set consists of 29 × 29 cells

Chapter 6

The Results and Discussion

This chapter we will present and discuss the results from the experiments. Section 6.1 presents the results and their discussion for the experiments of 1DVISIBILITYINDEX . Similarly for 2DVISIBILITYINDEX , Section 6.2 presents and discusses the results.

6.1 The Results and Discussion of 1DVISIBILITYINDEX

The results for the experiments of 1DVISIBILITYINDEX are presented and discussed per aspect. The first aspect is the scalability of 1DVISIBILITYINDEX and can be found in Section 6.1.1. The second aspect, Section 6.1.2, focuses on the characteristics of the data set, in our case the average visibility of the profile. All results are presented in Appendix E

6.1.1 Running Time Plots

Figure 6.1 shows the running time of 1DVISIBILITYINDEX for various data sets of various sizes. The running time of the brute-force approach can be found in Figure 6.3. Note that for data sets with more than 10,000 cells, the brute-force approach took too long and is therefore omitted. The parabola data sets are also excluded, because their sizes are only up to 2,000 cells. Therefore Figure 6.4 shows the running time of the parabola data sets separately.

Figures 6.1, 6.3 and 6.4 show that 1DVISIBILITYINDEX scales well with respect to the data set size and that 1DVISIBILITYINDEX is very efficient compared to the brute-force approach. This is expected, as the brute-force approach has a running time of $O(n^3)$ and 1DVISIBILITYINDEX of $O(n \log^2 n)$. The running time of the brute-force approach for the line data sets and the peak ($a = -1$) parabola data sets is very low, because the neighbors are the only cells that are visible. Hence the running time of the brute-force approach becomes quadratic instead of cubed for those data sets. The standard deviation of the running time of 1DVISIBILITYINDEX is below 5% for all data sets with a size of at least 10,000 cells, except for the two largest sine (b = 30) data sets, they are just around the 6.5%. The standard deviation of the running time of the brute-force approach is in general below 3%. For the parabola data sets most of the standard deviations for the different sizes are larger, as many running times are very close to 0.

In Table 6.1 and Figure 6.2 the running time is divided by $n \log^2 n$. In most cases the value becomes constant, for large data sets. There are two data set types that deviate from this behavior, namely the sine ($b = 15$) and the real-life data sets. The reason for this is the way they are generated. The generation of the data sets can introduce some artifacts. We will discuss the artifacts and their source for each of these two types of data sets in a separate paragraph. Note that the semi-random data sets are generated with randomness in mind and therefore a rapid increase or decrease is not surprising.

For the sine data set there are only 15 peaks and valleys and therefore there are cells that have the same elevation. Especially, around the top of the peaks and the bottom of the valleys. During the generation of the data sets, non-integer values are rounded down to the nearest integer. This is needed as 1DVISIBILITYINDEX only works with non-negative integral input values. The steeper the curve is, the less likely it is that neighboring cells have a different elevation. This generation artifact is also present in the sine ($b = 30$) data sets, only there it is smaller as there are twice as many peaks and valleys.

The real-life data sets are based upon the base data set of 3,000 cells. To increase the size of the data sets, we have added extra cells in between two neighboring cells. The elevation of these cells is interpolated along the line from one neighbor to the next. The way we generate the real-life data sets decreases the average visibility index. For all types of data sets it is the case that the average visibility index decreases, when the number of cells increases (see Figure 6.5). Figure 6.5 shows that for the real-life data sets this decrease is more rapidly than for the other type of data sets.

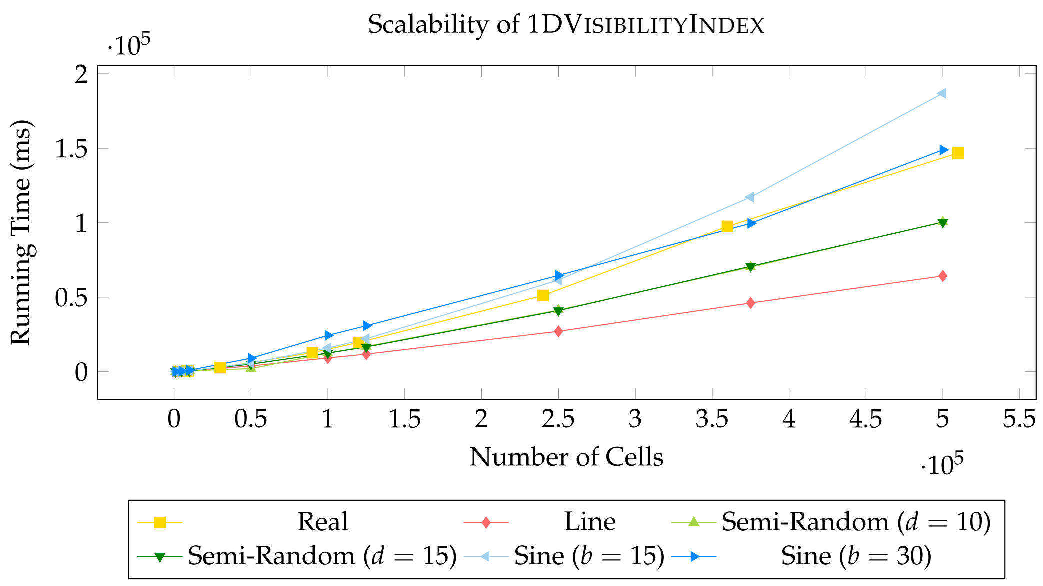


Figure 6.1: Running time of 1DVISIBILITYINDEX for various data sets of various sizes.

# cells	Line	Semi ($d = 10$)	Semi ($d = 15$)	Sine ($b = 15$)	Sine ($b = 30$)	# cells (for Real)	Real
1,000	$4.03{\cdot}10^{-4}$	$3.88{\cdot}10^{-4}$	$3.42{\cdot}10^{-4}$	$4.34{\cdot}10^{-4}$	$4.64{\cdot}10^{-4}$	-	-
5,000	$3.40{\cdot}10^{-4}$	$3.32{\cdot}10^{-4}$	$2.95{\cdot}10^{-4}$	$3.59{\cdot}10^{-4}$	$4.73{\cdot}10^{-4}$	3,000	$3.578{\cdot}10^{-4}$
10,000	$3.34{\cdot}10^{-4}$	$3.37{\cdot}10^{-4}$	$3.00{\cdot}10^{-4}$	$3.59{\cdot}10^{-4}$	$5.49{\cdot}10^{-4}$	8,997	$.45{\cdot}10^{-4}$
50,000	$4.15{\cdot}10^{-4}$	$4.18{\cdot}10^{-4}$	$3.17{\cdot}10^{-4}$	$4.74{\cdot}10^{-4}$	$7.44{\cdot}10^{-4}$	29,990	$4.18{\cdot}10^{-4}$
100,000	$4.52{\cdot}10^{-4}$	$4.53{\cdot}10^{-4}$	$3.35{\cdot}10^{-4}$	$5.76{\cdot}10^{-4}$	$8.84{\cdot}10^{-4}$	89,970	$5.26{\cdot}10^{-4}$
125,000	$4.73{\cdot}10^{-4}$	$4.64{\cdot}10^{-4}$	$3.30{\cdot}10^{-4}$	$6.17{\cdot}10^{-4}$	$8.61{\cdot}10^{-4}$	119,960	$5.73{\cdot}10^{-4}$
250,000	$5.15{\cdot}10^{-4}$	$5.10{\cdot}10^{-4}$	$3.37{\cdot}10^{-4}$	$7.66{\cdot}10^{-4}$	$8.04{\cdot}10^{-4}$	239,920	$6.68{\cdot}10^{-4}$
375,000	$5.46{\cdot}10^{-4}$	$5.50{\cdot}10^{-4}$	$3.59{\cdot}10^{-4}$	$9.11{\cdot}10^{-4}$	$7.75{\cdot}10^{-4}$	359,880	$7.95{\cdot}10^{-4}$
500,000	$5.61{\cdot}10^{-4}$	$5.60{\cdot}10^{-4}$	$3.59{\cdot}10^{-4}$	$1.04{\cdot}10^{-3}$	$8.31{\cdot}10^{-4}$	509,830	$7.99{\cdot}10^{-4}$

Table 6.1: The running time divided by $n \log^2 n$, where n denotes the number of cells.

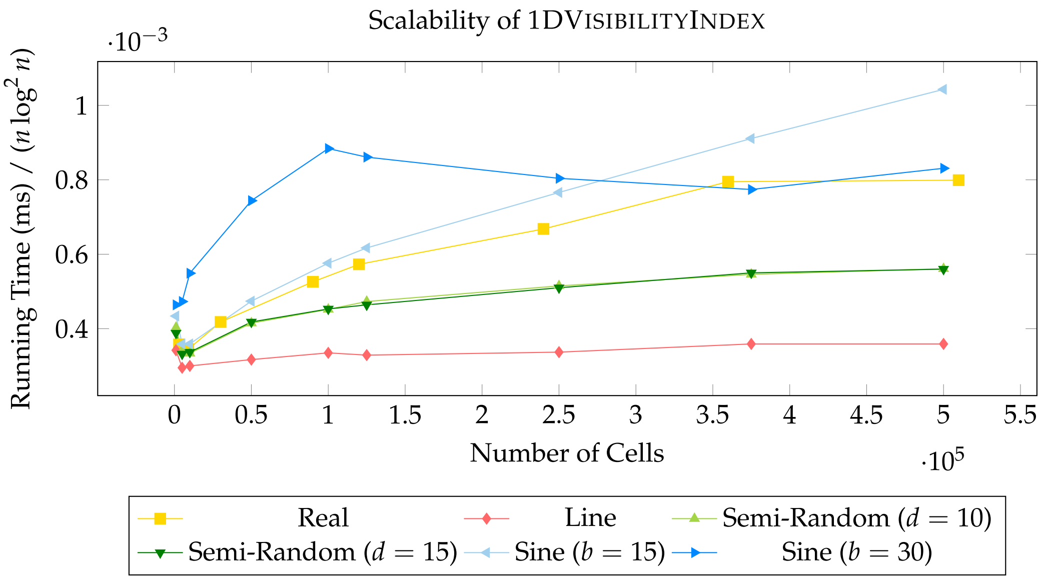


Figure 6.2: The running time divided by $n \log^2 n$, where n denotes the number of cells.

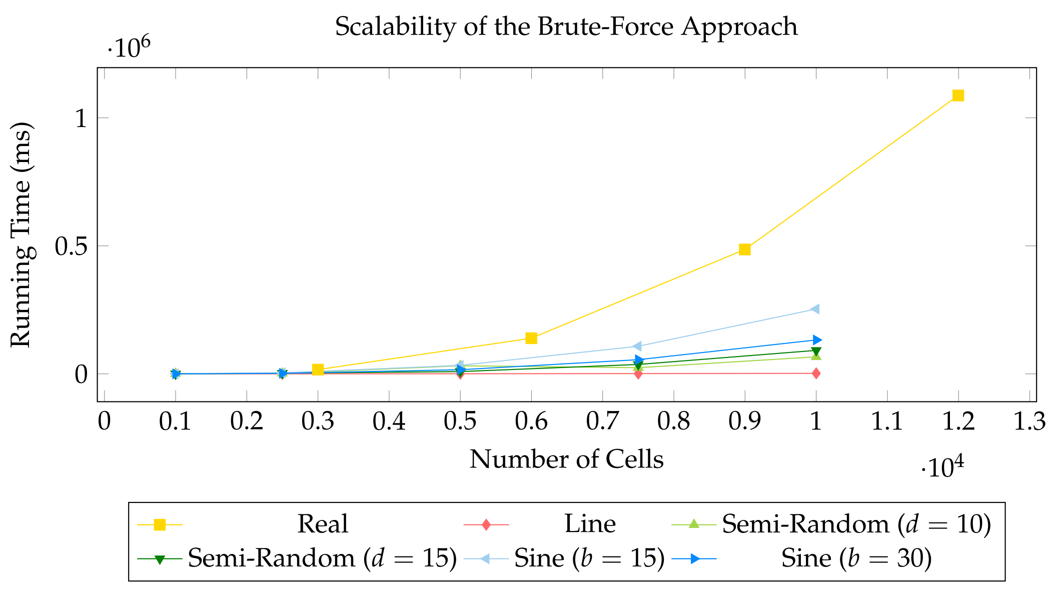


Figure 6.3: Running time of the brute-force approach for various data sets of various sizes.

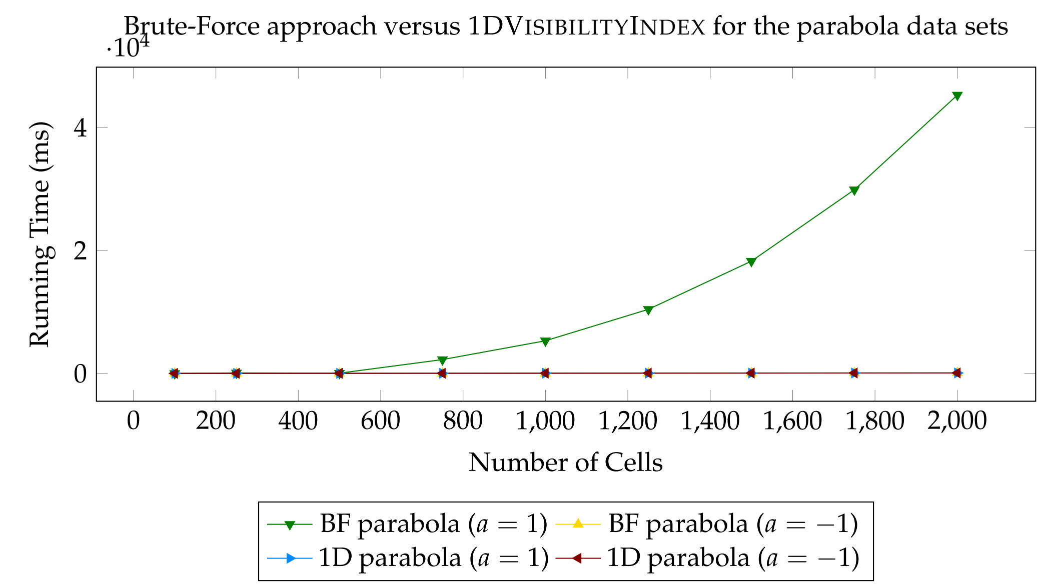


Figure 6.4: Running time of the brute-force-approach for the parabola data sets of various sizes.

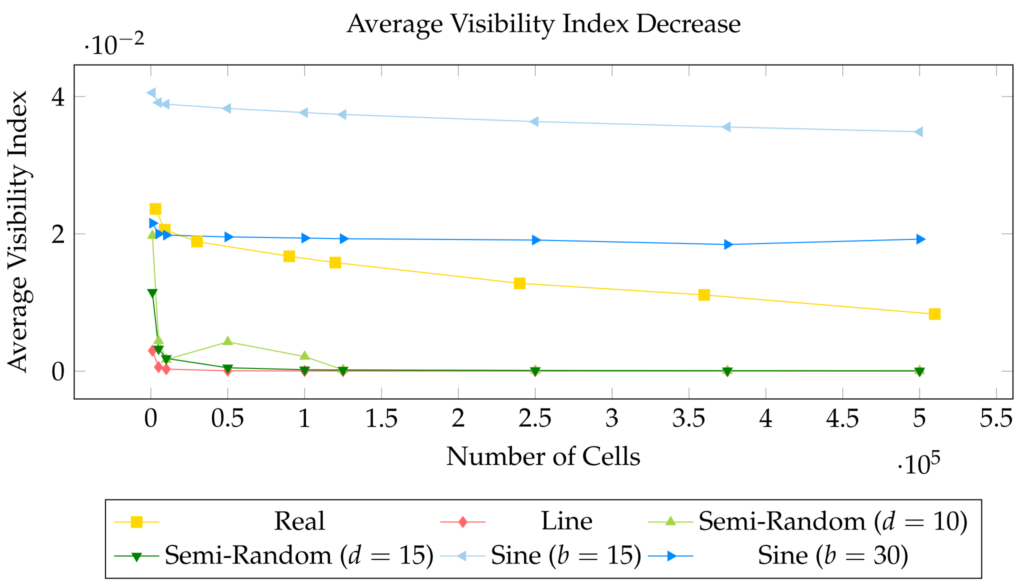


Figure 6.5: The average visibility index for various data sets of various sizes.

6.1.2 Characteristic Plots

Figure 6.1 hints that the visibility of the cells also plays a role in the running time, as for example 1DVISIBILITYINDEX is fastest on the line data sets. The line data sets have minimal visibility possible, as the visibility of each cell is only limited to the neighbors of each cell. It seems that this phenomenon is also visible when we compare the two sine data sets. The sine data sets with 30 peaks ($b = 30$) has a lower average visibility index then the sine data sets with 15 peaks ($b = 15$) and 1DVISIBILITYINDEX has a lower running time for the sine data sets with 30 peaks. Figure 6.7 shows this phenomenon more explicitly. The data sets are generated with the sine formula using different values for b. The size of all data sets are fixed at 100,000 cells and each run is executed 10 times.

The reason for this lower running time is the lower computation time of the red-blue intersections. REDBLUEINTERSECTIONCOUNT uses a segment tree for the actual intersection computation (also see Section 3.1.3). The segment tree is built out of slabs and each slabs contains a number of dual half-lines. During the initialization of the segment-tree dual half-lines are grouped together, as long as there are only dual half-lines of the same color in each slab. In a higher level recursion step all cells in a valley are grouped together as their critical rays have a relative large angle and go through cells around the same point. The visibility of theses cells is relatively similar, as it is limited to the valley they are part of; they cannot see beyond the peaks that enclose the valley, as illustrated in Figure 6.6. This is also visible in Figure 6.8, as the number of slabs is in general larger for the data set with the higher average visibility index. Note that the number of slabs during REDBLUEINTERSECTIONCOUNT only influences the practical running time. The data sets we have created are explicitly constructed according to a certain structure. The semi-random data sets also contain this valley structure, only on a much smaller scale and are therefore less smooth than the real-life data sets. Hence they have a lower average visibility index and running time.

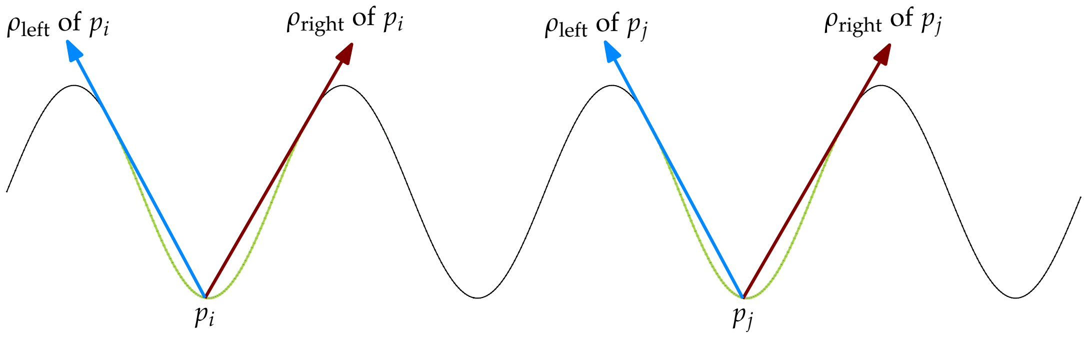

ρ_{left} of p_i
ρ_{right} of p_i
ρ_{left} of p_j
ρ_{right} of p_j
p_i
p_j

Figure 6.6: Illustration of how valleys are grouped together in the same slabs during initialization of the segment tree during REDBLUEINTERSECTIONCOUNT . The visible region for p_i or p_j are depicted in green.

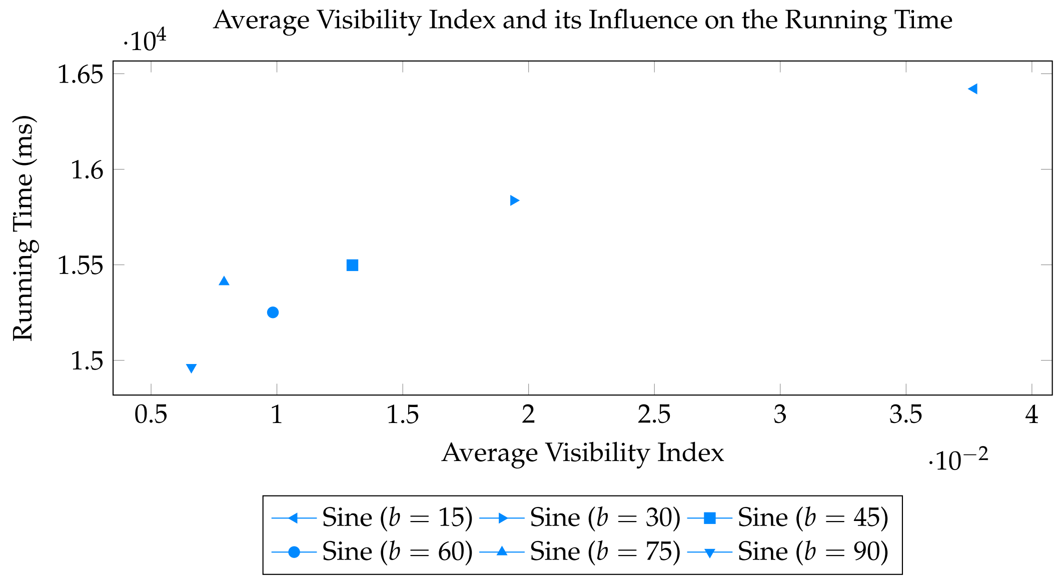


Figure 6.7: Running time and visibility index for the sine data sets with different values for b and 100,000 cells.

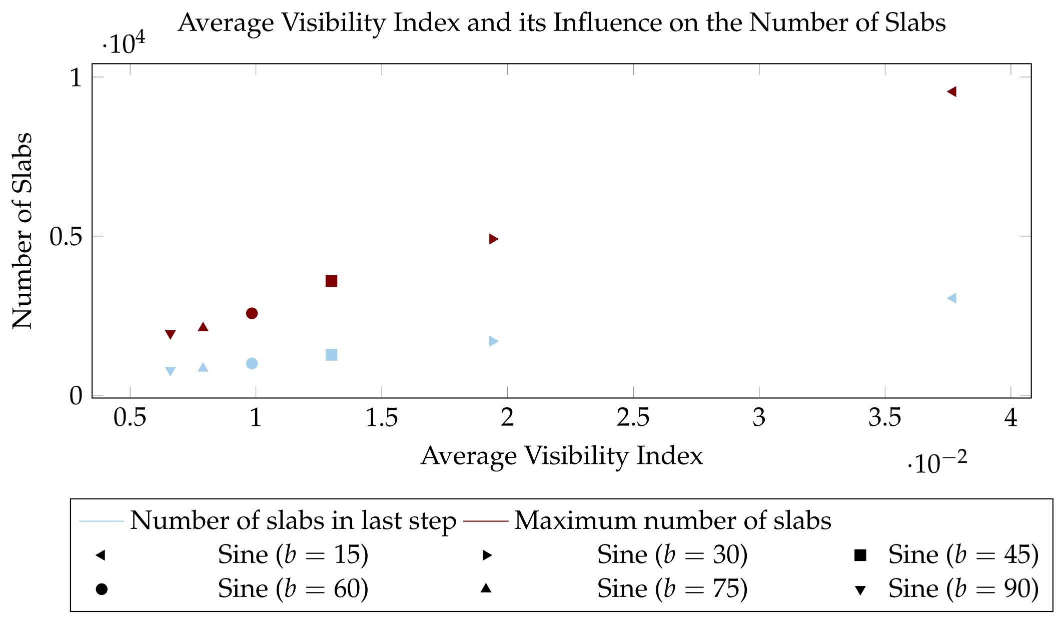


Figure 6.8: The number of slabs used during REDBLUEINTERSECTIONCOUNT and visibility index for the sine data sets with different values for b and 100,000 cells. The "Number of slabs used in the last step" denotes the number of slabs during the last recursive step and the "Maximum number of slabs" denotes the maximum number of slabs used in any recursive step.

6.2 The Results and Discussion of 2DVisibilityIndex

This section presents and discusses the results of the experiments for 2DVisibilityIndex . For each of the data sets we present a graph with the visibility index of the brute-force approach (red line) and the approximation of 2DVisibilityIndex for various m (blue lines) in Figures 6.9 to 6.11 and F.1 to F.4. Figures 6.12 to 6.14 and F.5 to F.8 presents the number of visible cells instead of the visibility index. The graphs of the data sets not presented here can be found in Appendix F.1. All results are presented in Appendix F.2.

Before we continue with the discussion of the results, we must note that the results are preliminary and therefore their discussion is limited and only serves as starting point for future research. The results show that the average of the visibility index along the different lines stabilizes relatively soon (Figures 6.9 to 6.11 and F.1 to F.4). For $m > 10$ the average visibility index is relatively stable. This would mean that the approximation is rather efficient, but therefore the approximation must also be relatively accurate.

The results show that this is not really the case (Figures 6.12 to 6.14 and F.5 to F.8). The visibility index of the brute-force approach is much lower than the visibility index of 2DVisibilityIndex .

There are two reasons that can explain this. The first reason has to do with the density of the lines. All lines start at the corner cell and therefore the density of the lines in that part of the grid is much higher. Cells that are relatively close to the corner cell are more likely to be visible, as there are less cells in between that can obstruct the visibility and cells in the neighborhood tend to have similar elevations. Figures 6.15 to 6.17 and F.9 to F.12 show the cells that are visible according to the brute force approach and they show that the cells relatively close are often visible from the corner cell. The area close to the corner cell weights heavier than other areas of the grid, because the density of the lines in the bottom left area of the gird is higher. Figure 5.10 shows this for $m = 5$, as there is only one line that covers the the top right area of the grid and 5 lines that cover the bottom left area.

The second reason has to do with ratio itself. The visibility index is the ratio of visible cells with respect to the total number of cells. The total number of cells in the brute-force approach is the entire grid of n^2 cells, while the total number of cells for each line is at most $2 \cdot n$ (and at least n). Therefore the visibility index of the brute-force approach can be substantially lower, even if the number of visible cells is similar.

What the results show that for future research a better combine metric must be used. A metric that takes into account the number of cells in the grid and the number of cells intersecting a line. The metrics we propose in Section 4.2.2 and Chapter 8 could increase the accuracy.

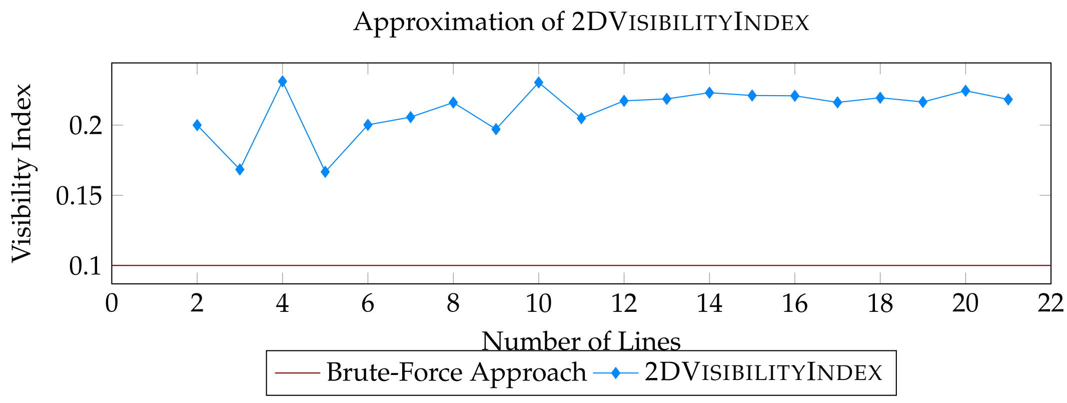


Figure 6.9: The visibility index of the brute-force approach (0.1000) and the approximation of 2DVISIBILITYINDEX for the Glatthorn data set.

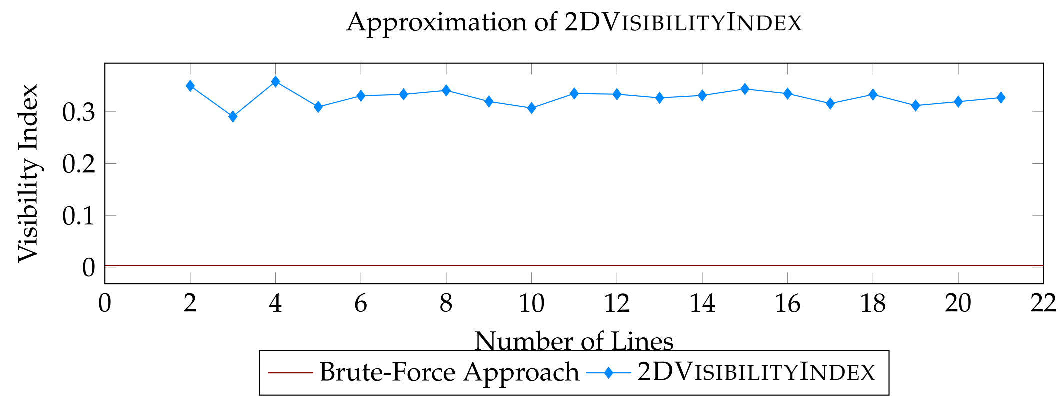


Figure 6.10: The visibility index of the brute-force approach (0.0033) and the approximation of 2DVISIBILITYINDEX for the Schesaplana data set.

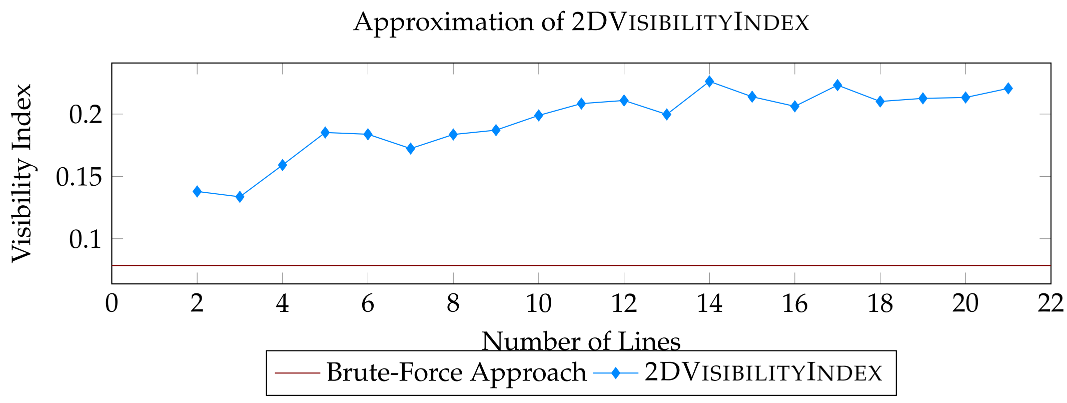


Figure 6.11: The visibility index of the brute-force approach (0.0785) and the approximation of 2DVISIBILITYINDEX for the PizBadile data set.

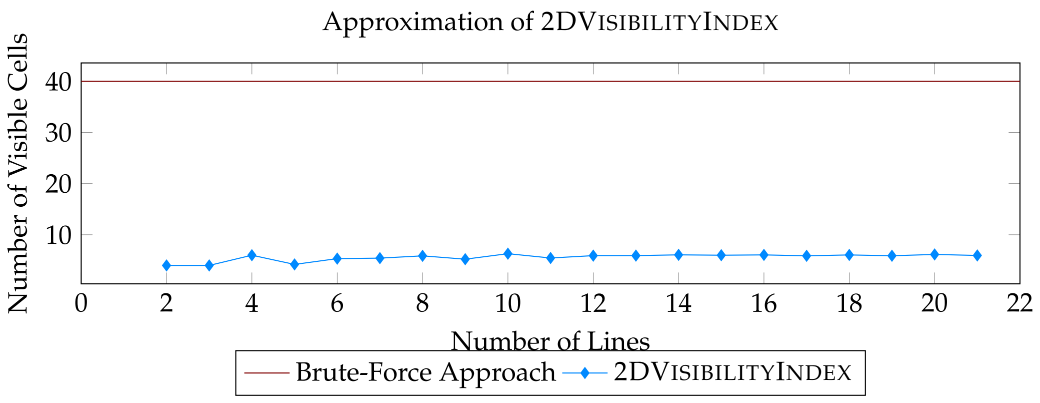


Figure 6.12: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the Glatthorn data set.

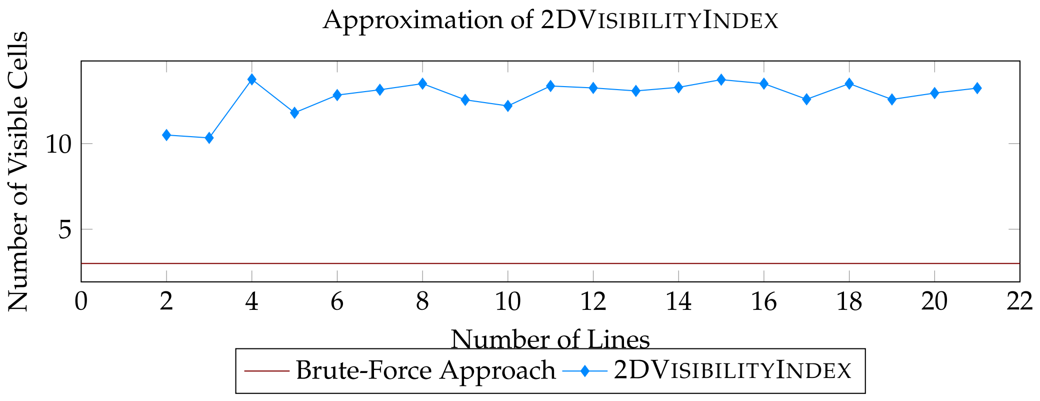


Figure 6.13: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the Schesaplana data set.

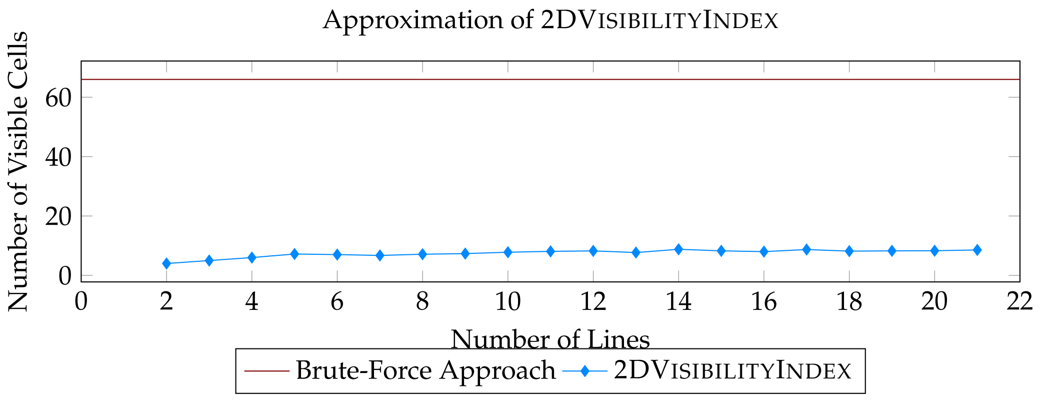


Figure 6.14: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the PizBadile data set.

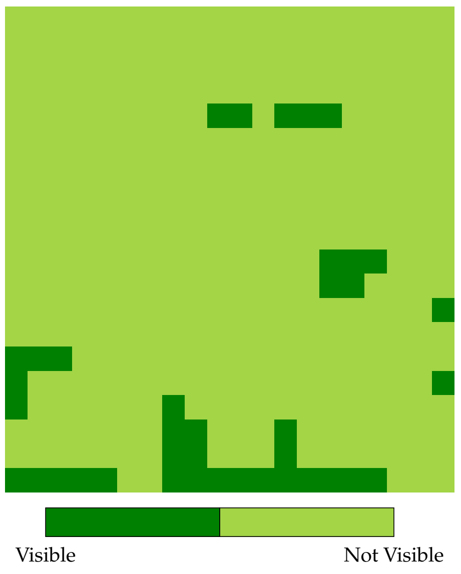


Figure 6.15: The visible cells for the brute-force approach for the Glatthorn data set.

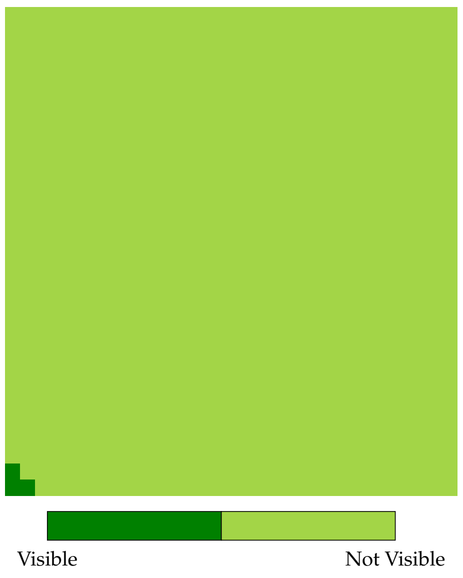


Figure 6.16: The visible cells for the brute-force approach for the Schesaplana data set.

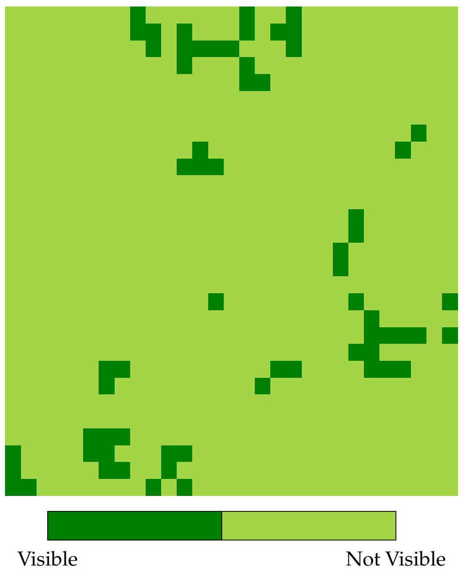


Figure 6.17: The visible cells for the brute-force approach for the PizBadile data set.

Chapter 7

Conclusions

We have studied the total visibility index problem, where the visibility index of a cell c in a grid terrain T is defined as the percentage of cells in T that are visible from c. We have proposed 1DVISIBILITYINDEX for the 1-dimensional version of the problem that runs in $O(n \log^2 n)$ time, where n is the size of the terrain. 1DVISIBILITYINDEX follows the divide-and-conquer paradigm and uses duality and REDBLUEINTERSECTIONCOUNT to compute the total visibility index. We have implemented 1DVISIBILITYINDEX and shown that the implementation adheres to the theoretical running time. 1DVISIBILITYINDEX is able to compute the total visibility index for 500,000 cells within 150 seconds, whereas the brute-force approach can only compute it for 5,000 cells within that time frame (using the real-life data sets). We have also shown that terrains with a lower average visibility index have a slightly lower running time in practice.

1DVISIBILITYINDEX forms the bases of our ideas to approximate the total visibility index of a (2-dimensional) grid terrain. 2DVISIBILITYINDEX generates m lines and calls 1DVISIBILITYINDEX on each of those lines. The resulting visibility index is the average of the visibility indices from the different lines. The theoretical running time of 2DVISIBILITYINDEX would be $O(mn \log^2 n)$, where m is the number of lines and n the size of the $n \times n$ grid. Although this depends on the way we combine the visibility indices from the calls to 1DVISIBILITYINDEX . 1DVISIBILITYINDEX is relatively fast and therefore we can choose m fairly high. For example, consider a grid of 2,000 by 2,000 cells. The size of the input for each of the calls to 1DVISIBILITYINDEX has at most 4,000 cells and 1DVISIBILITYINDEX needs less than 0.5 seconds to compute the visibility index. This means that 2DVISIBILITYINDEX can approximate the visibility index using 7,200 different lines per hour.

For the 2-dimensional version we did a basic experiment and thereby we left the evaluation of different schemes to generate lines for future research. We have shown that 2DVISIBILITYINDEX has the possibility to be an efficient approximation, on the condition that it is also relatively accurate. We have several ideas to possibly improve the accuracy of 2DVISIBILITYINDEX . The most simple improvement is to use different ways of combining the results from 1DVISIBILITYINDEX . Another improvement is to use different line generating schemes. 2DVISIBILITYINDEX can also be improved by using different grid terrain representations. All of these improvements require relatively small modifications of 1DVISIBILITYINDEX and are mainly limited to 2DVISIBILITYINDEX .

Chapter 8

Future Work

This chapter discusses several aspects that could be studied more extensively or are open questions, e.g. different schemes for generating lines for the total visibility index approximation. We also discuss some modifications that could improve the running time of 1DVISIBILITYINDEX . These modifications are not necessarily improving the worst-case running time, but could improve the constants within the big-O.

The approximation of the 2-dimensional version uses lines to approximate the total visibility index. An example of generating lines was the random approach. In combination with the knowledge that the total visibility index along these lines is computed exactly, a statistical analysis could be used to determine how good the approximation is theoretical. Thereby also analyzing how many lines you need for a decent approximation (in theory).

The current approximation uses a weighted average to combine the results from the different lines. There are a lot of different possibilities in combining results. We did not test if our current idea is the best, so it is interesting to see which approach works best. We have seen that the approximation can be efficient if it would be accurate (enough). A similar, but different, approach for combining results is to use more auxiliary information of 1DVISIBILITYINDEX . Currently our idea is to partition the grid in areas between two lines. It is possible to extent that by also partitioning the line in a number of sectors. Figure 8.1 shows an example of such a sectioning of the partitions. In this sectioning the visibility index for that section is computed using the average visibility index along the lines that envelope that section. Using different ways of combining the results of 1DVISIBILITYINDEX , also means we could use other grid representations. We restricted ourselves to a simplified representation of DEM's, but for future research (and to make the algorithm more practical) it could be converted to work on DEM's using interpolation or on TIN's.

Another possible study is to compare different approaches to the approximation of the 2-dimensional problem. For example dividing the grid into four quadrants and recursively approximate the total visibility index for each quadrant and then compute the visibility index for each quadrant with respect to the other quadrants. This approach would use the divide-and-conquer paradigm and could lead to an efficient and better approximation.

To speed-up the approximation, 1DVISIBILITYINDEX could also be improved. The most simple way is to improve the performance of the implementation. Another way of possibly improving performance is to improve the algorithm. A way to improve 1DVISIBILITYINDEX is to tweak the splitting decision. 1DVISIBILITYINDEX could split the input at a peak and not halfway. Splitting at the peaks can reduce the number of slabs that occur during the REDBLUEINTERSECTIONCOUNT

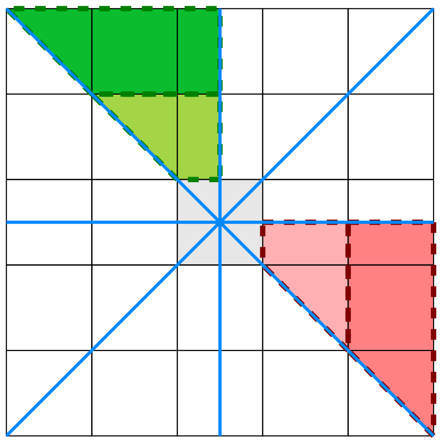


Figure 8.1: An example of dividing the partitions (green and red) into sections (dashed) for combing the results (blue) for a single cell (gray) with $m = 4$ lines

and thereby speeding up the algorithm. Another way is to replace the pre-processing step in which the topological order is computed. Currently we use a sweep from left to right and maintain a sweep-tree. Once we have handled all the points we do an in-order traversal of this sweep-tree to obtain the topological order (according to aboveness). Maybe it is possible to use a sweep from high to low to directly compute the topological order.

Both 1DVISIBILITYINDEX and 2DVISIBILITYINDEX are in-memory algorithms. To be relevant for GIS-applications they need to be able to cope with very large data sets. So it will be interesting to see what the I/O-efficiency of 1DVISIBILITYINDEX and 2DVISIBILITYINDEX is and if they can be modified to be I/O-efficient.

Bibliography

[1] L. Arge, M. de Berg, and C. Tsirogiannis. Algorithms for computing prominence on grid terrains. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 254–263, 2013. 1

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Syst. J.*, 4(1):25–30, Mar. 1965.
I used the modified version described at http://playtechs.blogspot.nl/2007/03/raytracing-on-grid.html. 27

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 13 Red-Black Trees, pages 308–338. The MIT Press, 3rd edition, 2009. 64

[4] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, chapter 10.3 Segment Trees, pages 231–237. Springer-Verlag TELOS, 3rd ed. edition, 2008. 15

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, chapter 1.1 An Example: Convex Hulls, pages 6–8. Springer-Verlag TELOS, 3rd ed. edition, 2008. 21

[6] L. De Floriani and P. Magillo. Algorithms for visibility computation on terrains: a survey. *Environment and Planning B: Planning and Design*, 30(5):709–728, 2003. 4

[7] C. Ferreira, M. V. Andrade, S. V. Magalhães, W. R. Franklin, and G. C. Pena. A parallel sweep line algorithm for visibility computation. In *GeoInfo*, pages 85–96, 2013. 5

[8] C. R. Ferreira, S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and A. M. Pompermayer. More efficient terrain viewshed computation on massive datasets using external memory. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 494–497. ACM, 2012. 4, 5

[9] J. Fishman, H. Haverkort, and L. Toma. Improved visibility computation on massive grid terrains. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 121–130. ACM, 2009. 4, 5

[10] W. R. Franklin and C. K. Ray. Higher isn't necessarily better: Visibility algorithms and experiments. In T. C. Waugh and R. G. Healey, editors, *Advances in GIS Research: Sixth International Symposium on Spatial Data Handling*, pages 751– 770. Taylor & Francis, 1994. 5

[11] M. F. Goodchild and J. Lee. Coverage problems and visibility regions on topographic surfaces. *Annals of Operations Research*, 18(1):175–186, 1989. 6

[12] H. Haverkort, L. Toma, and Y. Zhuang. Computing visibility on terrains in external memory. *Journal Experimental Algorithmics*, 13:5:1.5–5:1.23, Feb. 2009. 5

[13] D. Izraelevitz. A fast algorithm for approximate viewshed computation. *Photogrammetric Engineering & Remote Sensing*, 69(7):767–774, 2003. 6

[14] D. B. Kidner, P. J. Rallings, and J. A. Ware. Parallel processing for terrain analysis in gis: Visibility as a case study. *GeoInformatica*, 1(2):183–207, 1997. 5

[15] J. Lee. Analyses of visibility sites on topographic surfaces. *International Journal of Geographical Information System*, 5(4):413–429, 1991. 5

[16] P. Mateti and R. Manghirmalani. Morris' tree traversal algorithm reconsidered. *Science of Computer Programming*, 11(1):29 – 43, 1988.
This is the paper, but I also used the following sites: http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/, https://en.wikipedia.org/wiki/Threaded_binary_tree. 65

[17] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graphical Models and Image Processing*, 56(4):304 – 310, 1994. 13, 15, 21, 63, 64

[18] S. Tabik, A. Cervilla, E. Zapata, and L. Romero. Efficient data structure and highly scalable algorithm for total-viewshed computation. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, PP(99):1–7, 2014. 6

[19] S. Tabik, E. L. Zapata, and L. F. Romero. Simultaneous computation of total viewshed on large high resolution grids. *International Journal of Geographical Information Science*, 27(4):804–814, 2013. 6

[20] M. van Kreveld. Variations on sweep algorithms: Efficient computation of extended viewsheds and class intervals. In *Proceedings Seventh International Symposium on Spatial Data Handling*, pages 13–15, 1996. 4, 5, 23

[21] Cycleroute - create cycle routes and profiles.
http://www.cycleroute.org. 35

[22] Cygwin version 1.7.33-1 (complete).
http://cygwin.com. 34

[23] Dem explorer.
http://ws.csiss.gmu.edu/DEMExplorer, powered by GeoBrain – NASA EOS Higher Education Alliance & Georege Mason University & Center for Spatial Information Scince and Systems. 34

[24] Gmp, the GNU multiple precision arithmetic library.
http://gmplib.org. 65

[25] Goecontext-profiler.
http://www.geocontext.org/publ/2010/04/profiler/en. 35

[26] Google maps.
http://maps.google.com. 69

[27] Gpsvisualizer - create a profile from a gps track.
http://www.gpsvisualizer.com/profile_input. 35

[28] Mingw 4.9.1 (deliverd with qt creator). 34

[29] Netbeans ide version 8.0.2.
http://netbeans.org. 34

[30] Qt creator version 3.3.0 open source edition.
http://www.qt.io. 34

Appendix A

Design Decisions

This section presents some important notions and choices made during the implementation of the algorithms. In many cases it will describe the non-trivial design decision made in the implementation and their rationale. The decisions are categorized in five sections. The first four sections are limited to 1DVisibilityIndex and the last section is limited to 2DVisibilityIndex . Appendix A.1 presents the decisions with respect to the representation of the data. Appendix A.2 describes how we handle internal computations and their internal use. The third section, Appendix A.3, describes some non-trivial decisions within the implementation of the algorithms. The fourth section, Appendix A.4 describes an important limitation in the implementation due to integer overflow. The main limitation is the number elevations in the input and the value of the elevations themselves. It also presents ways to deal with those limitations. The implications of the implementation of 1DVisibilityIndex for 2DVisibilityIndex are discussed in Appendix A.5.

A.1 Data Representation

The most convenient way to show the data representation and their use is by means of a class diagram, shown in Figure A.1. The class diagrams shows that the implementation is separated in two packages, the Data package and the Algorithms package. Within these packages the classes used to implement the concepts of the algorithms are grouped together. The usage between classes is shown to show the interaction and dependencies of the different classes. The implementation is constructed in a modular fashion, such that each (part of the) algorithm can be replaced easily.

A.1.1 Input Values

The first thing to notice is that we restrict the elevation values to non-negative values. If we would use doubles, the internal representation plays a role. In C++, for example, doubles are represented using fractions. Although it is possible to use exact arithmetic, representing the input as integers is also more convenient. In the real-life setting elevations of terrains are measured within a finite accuracy, for example in meters or centimeters. So to model the elevations as integral values is representative for the real world. If an input would have negative data, it can be simply transformed by shifting the values such all values are non-negative. In summary the input data only contains non-negative integral values.

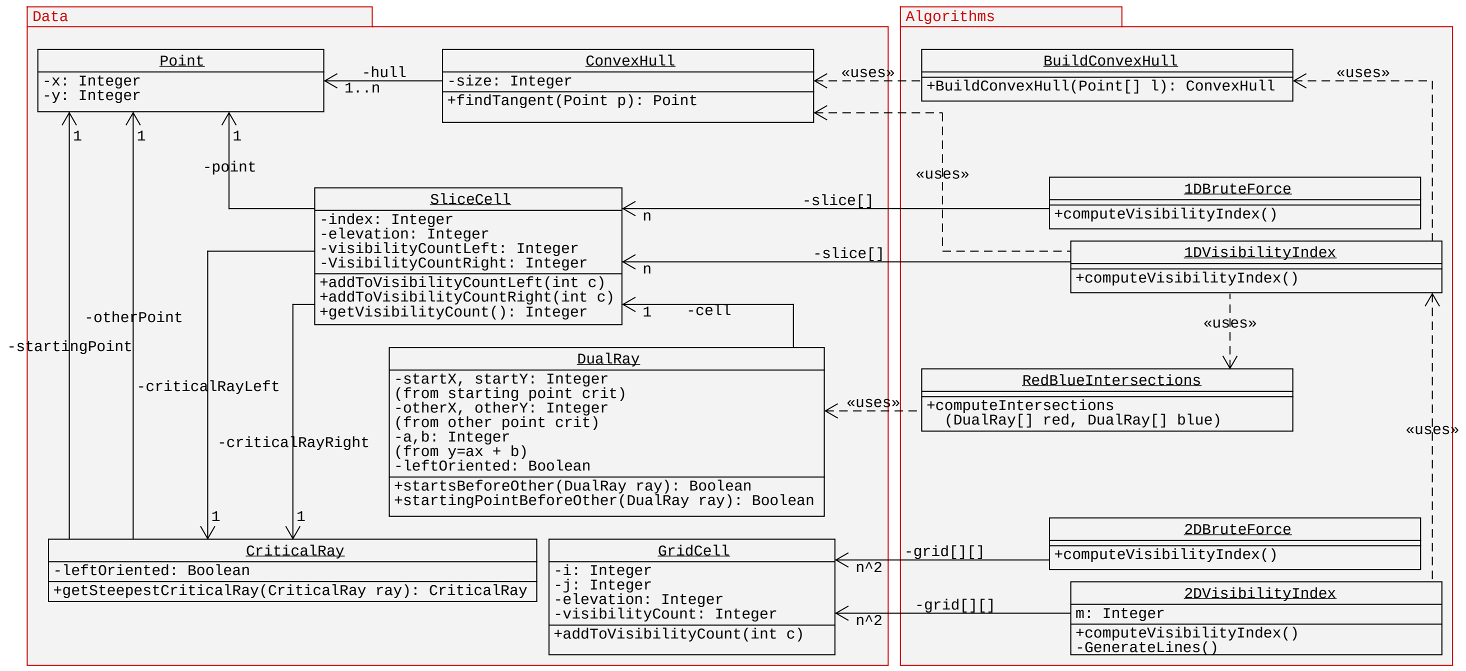


Figure A.1: Class diagram

A.1.2 Initial Critical Rays

The critical rays are represented using a starting point and another point. The initial critical rays are constructed using an artificial point. After the initialization the critical rays point downwards. In the implementation the critical rays are initialized to go through the point directly below its neighbor at $y = -1$. All elevations are non-negative and therefore any critical ray that goes through a point from the input has a smaller angle than the initial critical ray. Hence this representation does not interfere with the correctness of the algorithm. The reason for this representation is that some computations divide by Δx, which would be 0 if the ray points downwards, and an unwanted division by 0 would occur.

A.2 Computational Decisions

The first and most important decision is that no precision-reducing operations are used. Examples of precision-reducing operations are division and square-root. Normally we would compute certain values by using divisions and use them in a comparison (or other computations). To remove the division from the computations, we do not actually compute the outcome and instead rewrite the computations. An example of this is given below for a simple comparison between two fractions. To decide if $\frac{y_1}{x_1} < \frac{y_2}{x_2}$ we check if $x_2 \cdot y_1 < x_1 \cdot y$. One has to take care that the outcome remains valid, so if either $x_1 < 0$ or $x_2 < 0$ we need to flip the sign.

The second decision is that each computation has its own function, this to improve the readability of the implementation and allowing for a modular structure of computations. The rewritten computations are not necessarily as readable as the original computation, so using a modular structure increase the readability of the algorithm. Almost all of the computations (and comparisons) are used to compute whether one element is above/below or before/after another element. The outcome of the computations are represented using -1, 0 and 1. -1 represents below/before, 0 represent co-linear and 1 represent above/after. Adding the co-linearity resolution from Palazzi & Snoeyink [17] can be done easily in a modular structure without modifying the computation itself. We do this by using a wrapper around the computations and adding a call to the co-linearity resolution.

The third computational decision has to do with how we compute certain properties. As mentioned above, almost all comparisons are used to determine below/before or above/after. The above and below are used with respect to the aboveness relation defined by Palazzi & Snoeyink [17]. Computing whether a point is above a segment is done by using the counter-clock-wise test. When a point is to the right of a left-oriented dual half-line it is above that dual half-line. Symmetrically, when a point is to the left of a right-oriented dual half-line, it is above that dual half-line. The aboveness relation is not only defined on points and segments, but also on segments and segments. The aboveness relation between segments is only used to compare elements within the same colored set. Segments within the same set do not intersect and are half-lines, therefore aboveness can be determined by slope. For left-oriented dual half-lines it holds that a higher slope corresponds to a lower aboveness and for right-oriented dual half-lines a higher slope corresponds to a higher aboveness. Comparing segments from different colored sets is useless as those can intersect. To be more precise, those intersections are what we are looking for.

A.3 Algorithmic Decisions

In the implementation of the Red-Blue Intersection algorithm (Algorithm 2), there are a few non-trivial modifications. These modifications do not change the correctness, but simplify the implementation.

The first modification has to do with the aboveness order. The aboveness order within a slab σ is represented by a list $\mathcal{SL}(\sigma)$. The wall intersections of the shorts in σ need to be added to $\mathcal{SL}(\sigma)$. The modification is that the aboveness relation between the wall intersections themselves does not have to be maintained, only the aboveness order between $\mathcal{R}$ short and $\mathcal{B}$ long segments. For all red wall intersections that are above a blue long segment b_1 and below a blue long segment b_2 (where b_2 is directly above b_1 if we only consider blue long segments), the aboveness order is irrelevant. Symmetrically for blue short and red long segments. During the short-long slab sweep the ranks of red short segments and number of open red short segments are used to compute the number of intersections per element. The ranks of the red short segment is only based on the number of blue long segments above it. The number of open shorts for a blue long segment only depends on the number of starting points of red short segments for which its endpoint is not encountered in the sweep. So the internal order is not relevant for those two values. Also see the example presented in Figure 3.4. This means that we can use this modifications to speed up the implementation by only checking the list with blue long segments of σ instead of the complete $\mathcal{SL}(\sigma)$. The addition of the wall intersection to this list can be done based on the location of b_2 in $\mathcal{SL}(\sigma)$. This location is stored whit the elem itself, as $\mathcal{SL}(\sigma)$ is a linked list. Symmetrically, this modifications can be applied for blue short and red long segments.

Computing in which slab a dual half-line is long is a key part in the algorithm. Luckily Palazzi & Snoeyink [17] present a simple property of computing this: *If the start slab and the end slab are not adjacent, then and only then the segment can be long in between the start and end slab. If a segment starts in slab s and s is even, then the segment is long in slab $s + 1$ and if a segment ends in slab e and e is uneven, then the segment is long in slab $e - 1$.* Palazzi & Snoeyink [17] present some more "tricks" to simplify the algorithm. Examples of those "tricks" are that the construction of the segment tree is done on a level-by-level basis, reducing the space complexity to linear, and that merging the slabs is done by simply halving the start and end index of each element.

The sweep tree used in Palazzi & Snoeyink [17], during the first phase of the algorithm (computing the topological order), must be a balanced binary search-tree in order to get the $O(n \log n)$ running time. As a reminder the topological order is computed form red dual half-lines and blue starting points and for blue dual half-lines and red starting points. The implementation uses a red-black tree in order to get a balanced binary search-tree. A detailed explanation of red-black trees can be found in the book by Cormen et al [3]. The tree stores the aboveness order of dual half-lines of the same color and is used to locate the dual half-line that is directly above an opposing colored starting point. The dual half-lines either start at minus infinity (left-oriented) or end at plus infinity (right-oriented). For simplicity we color the left-oriented dual half-lines blue and the right-oriented dual half-lines red. Computing the topological order uses a sweep-line approach and constructs a sweep-tree that corresponds to the aboveness order at the sweep line. All blue dual half-lines start at minus infinity, so the sweep-tree is just a completely balanced binary search tree that contains all blue dual half-lines. We can construct such a tree by simply ordering the dual half-lines on slope (also see modification in Appendix A.2). These half-lines are deleted when we encounter their starting point. Therefore we can perform the normal deletion and the height of the tree remains at most $O(\log n)$. For red dual half-lines it is the other way around. We start with an empty tree and add dual half-lines when we encounter their starting point. Red dual half-lines are never deleted, as those end at plus infinity. Therefore we only need to make sure the tree is kept balanced during insertions. So we can simplify the implementation by implementing the insertions from a red-black tree and the deletions from an ordinary binary search tree.

Another modification to the sweep tree we did, was making it non-recursive and stack-less. We did this to overcome the relatively small amount of stack space available when we are traversing the sweep-tree. The algorithm we used was the algorithm of Morris [16] and it is a tree traversal algorithm without the use of a stack or any recursion.

A.4 Overflow Limitation

As mentioned in Appendix A.2, all computations with divisions are transformed in such a way that they no longer use precision-reducing operators. This has a major effect on the types used in these computations. For example computing the counter-clockwise test with input values in the range of a couple of thousand already results in a 32-bit integer overflow, as the internal computations use multiplication with three operands. The computations are relatively simple, but because of the possibly large numbers the auxiliary values can take it is very likely to overflow 32-bit integers.

We can cope with this by using 64-bit integers for all computations and auxiliary values. However, this just shifts the problem and does not solve it. A solution would be to require that computations can handle arbitrarily large integers. An example library that can handle arbitrarily large integers is GMP [24]. There are some other libraries and each has its own advantages and disadvantages. However, using such a library can reduce the performance, as there is more work to be done during computations. It also requires a lot of work to implement the current computations within this library. We do not consider the large numbers within the scope of this project and therefore we did not use such a library and used 64-bit integers instead.

The implications of this decision is that there is a maximum number of cells and a maximum elevation for those cells. To compute what the actual limitations are for the input data, we abstract from the actual meaning of the computations and only consider the ranges of variables. Thereby we can express the maximum value the computations can handle. The input or parameters for the computations are based on the input data and are therefore 32-bit signed integers. For some computations the difference between elevations and/or indices is required and because the input is always positive, as described in Appendix A.1, the absolute value of the difference also remains within the same range as the input data. Therefore we can restrict ourselves to the same range as the input, as we use signed integers to accommodate for the difference. The maximum index and elevation in the implementation can be 1,154,704. As long as the values remain below 1,154,704, we know that an integer overflow cannot occur.

A.5 Implications for the 2-Dimensional Problem

1DVisibilityIndex requires non-negative integral input. 2DVisibilityIndex interpolates the elevation for non-center points. These elevations need to be rounded, to satisfy the preconditions of 1DVisibilityIndex . Therefore we scale the generated input along the line with a factor f of 100. We picked $f = 100$, as the elevations in the order of thousands is common in real-life data sets and we do not want to cross the overflow limit (discussed in Appendix A.4). The larger the factor the more precise the visibility computations will be, as the number of significant digits increases in the computation. The scaling is also discussed in Section 4.2.4.

Appendix B

Data Sets for the 1-Dimensional Problem

In this chapter we will present some more examples of semi-randomly generated data sets. From these inputs it is clear that a decent value for *d* is important. If the value is low, you get a smoother profile and if *d* is high, you get a very rough profile.

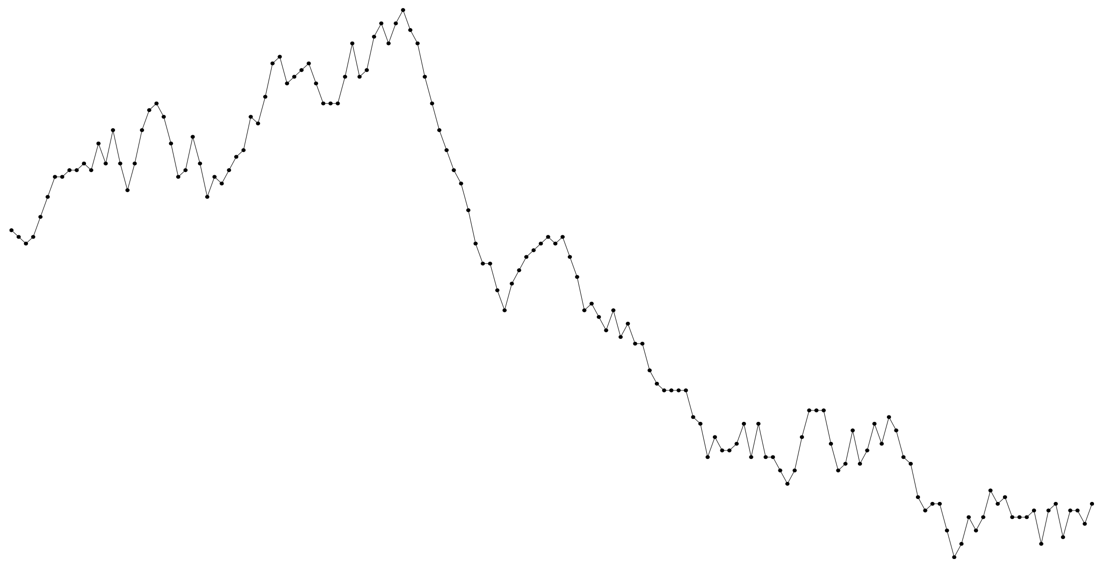


Figure B.1: Semi-random generated data set with 150 cells and elevations in between 0 and 82, the dispersion was set to 5.

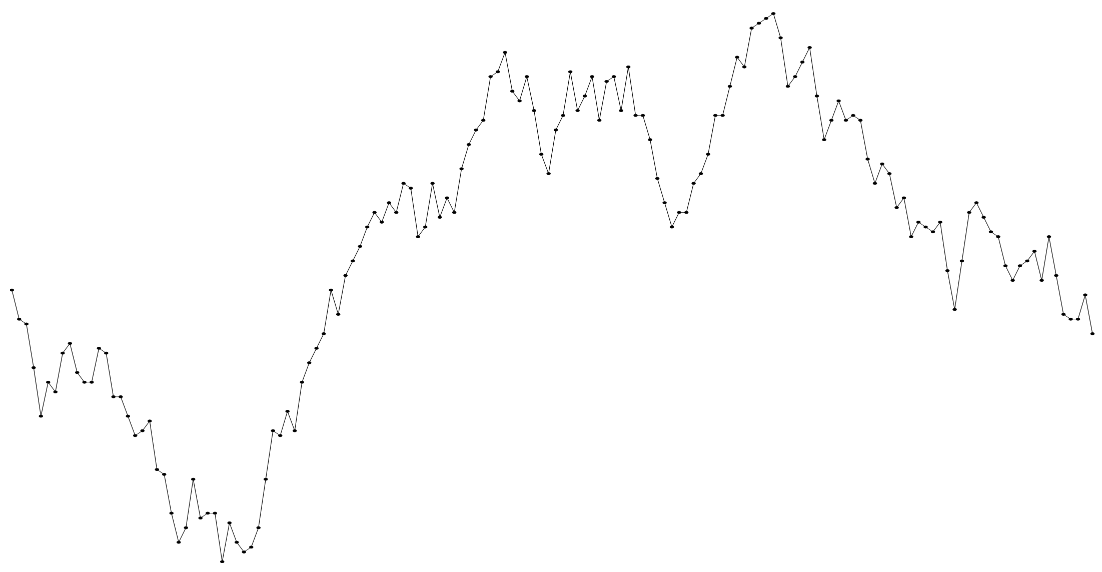


Figure B.2: Semi-random generated data set with 150 cells and elevations in between 0 and 113, the dispersion was set to 10.

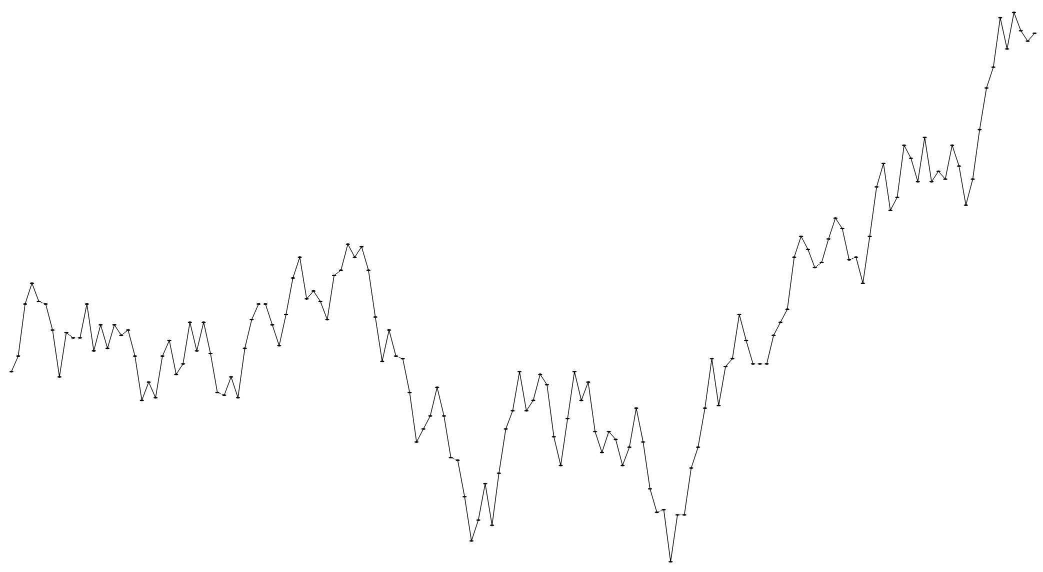


Figure B.3: Semi-random generated data set with 150 cells and elevations in between 0 and 211, the dispersion was set to 20.

Appendix C

Data Sets for the 2-Dimensional Problem

The visualization of the data sets uses a heat map to indicate the elevation of each cell. A cell with a low elevation is yellow and a cell with a high elevation is red. The minimum and maximum elevation are presented in the legend of the heat map to indicate the boundaries of the color spectrum. The visualization is on the left-hand side and on the right-hand side is a screenshot of the map [26] of the area depicted. The map has two markers, the red one (with dot) shows the bottom left corner and the other one (often gray) shows the top right corner. Note that the markers are not placed at the border of the cells, but are placed around the center of the cell. More specifically, when the bottom left corner cell includes a mountain peak (also included in the name) the marker is placed upon that peak. The map is included to give the data sets more context.

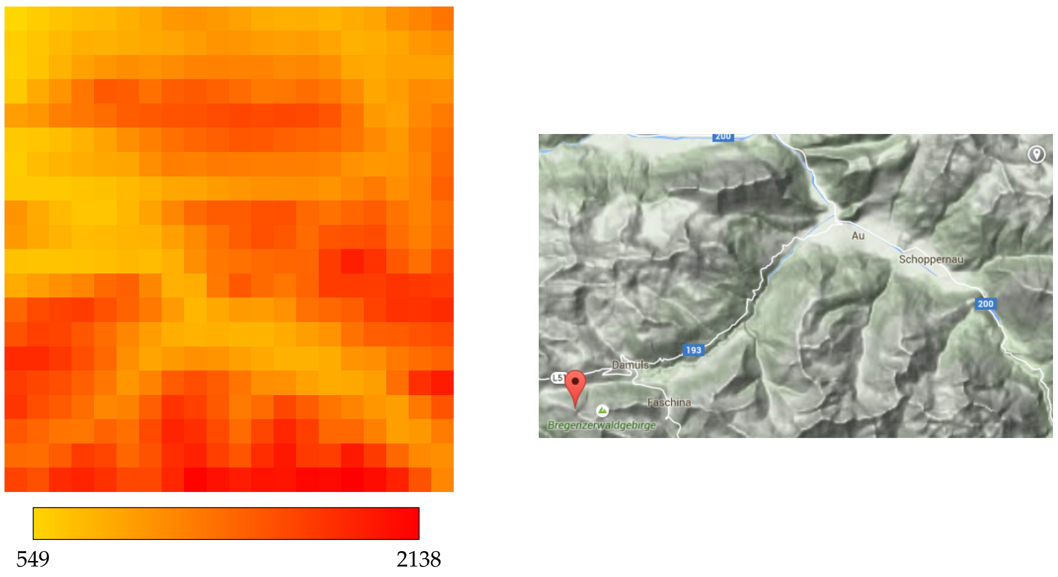


Figure C.1: The data set with the cell of the Glatthorn (Austria) in the lower left corner. The data set consists of 20 × 20 cells

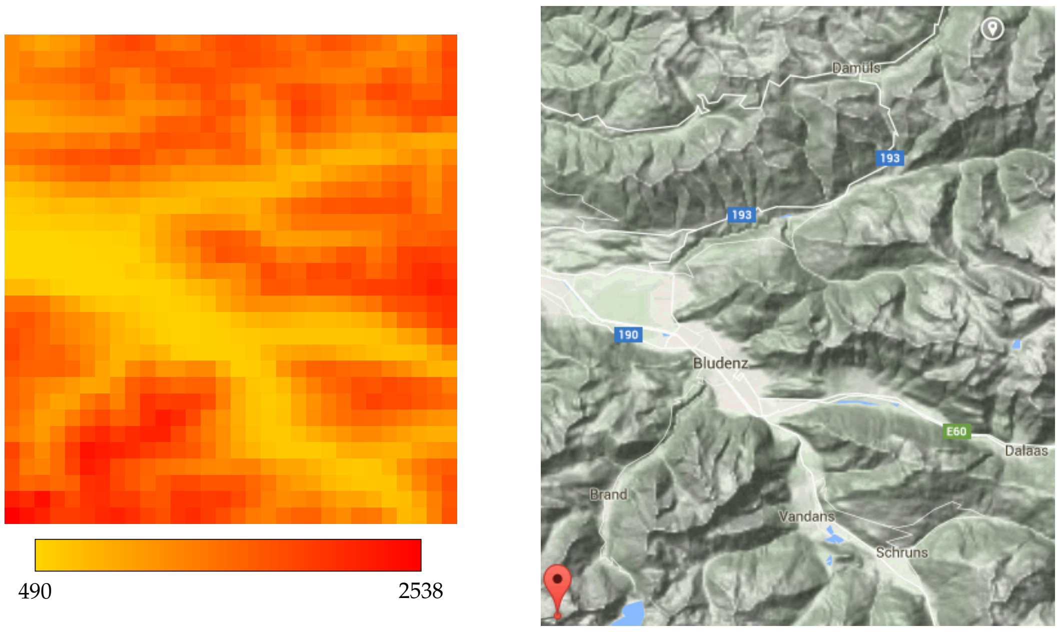


Figure C.2: The data set with the cell of the Schesaplana (Austria) in the lower left corner. The data set consists of 30 × 30 cells

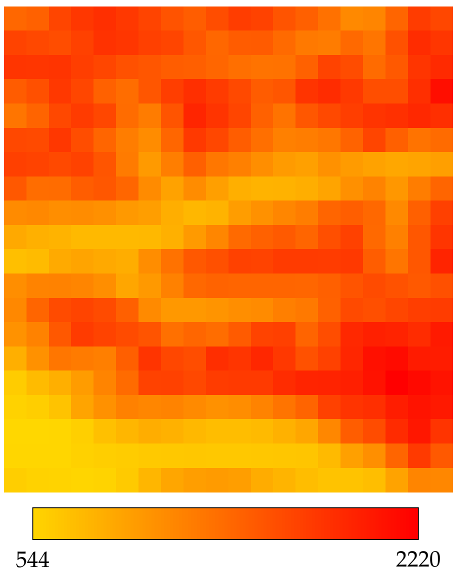


Figure C.3: The data set of the top right 20 × 20 cells of the Schesaplana data set (Figure C.2).

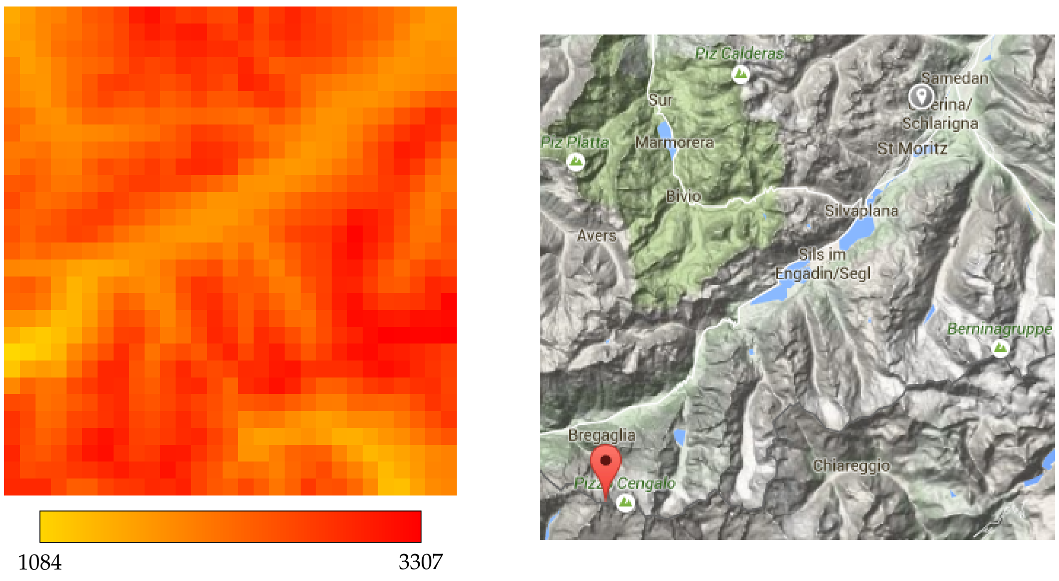


Figure C.4: The data set with the cell of the Piz Badile (Italy) in the lower left corner. The data set consists of 29 × 29 cells

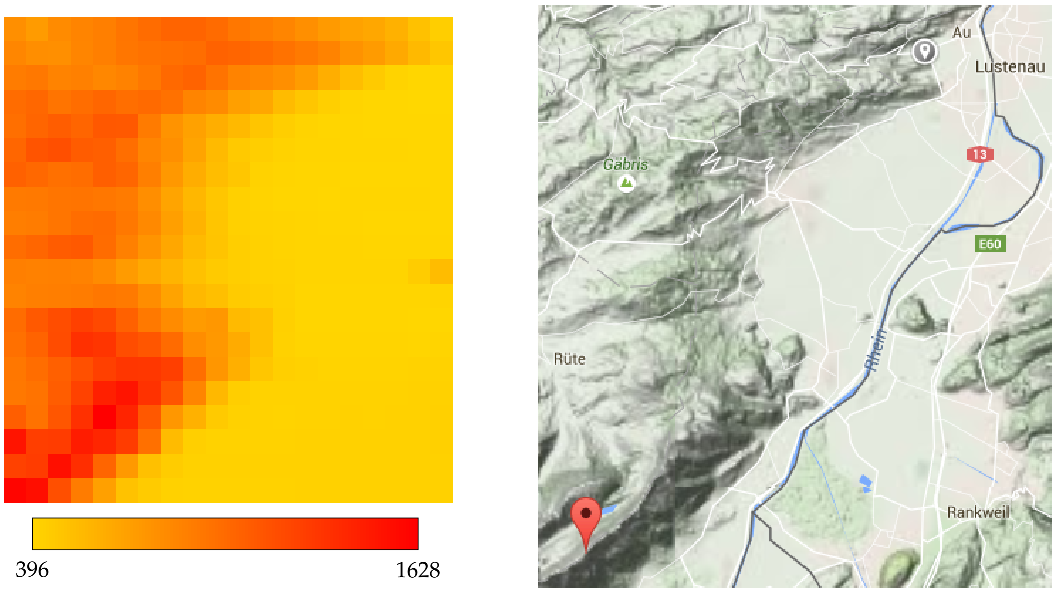


Figure C.5: The data set with the cell of the Furgglenfirst (Austria) in the lower left corner. The data set consists of 20 $\times$ 20 cells

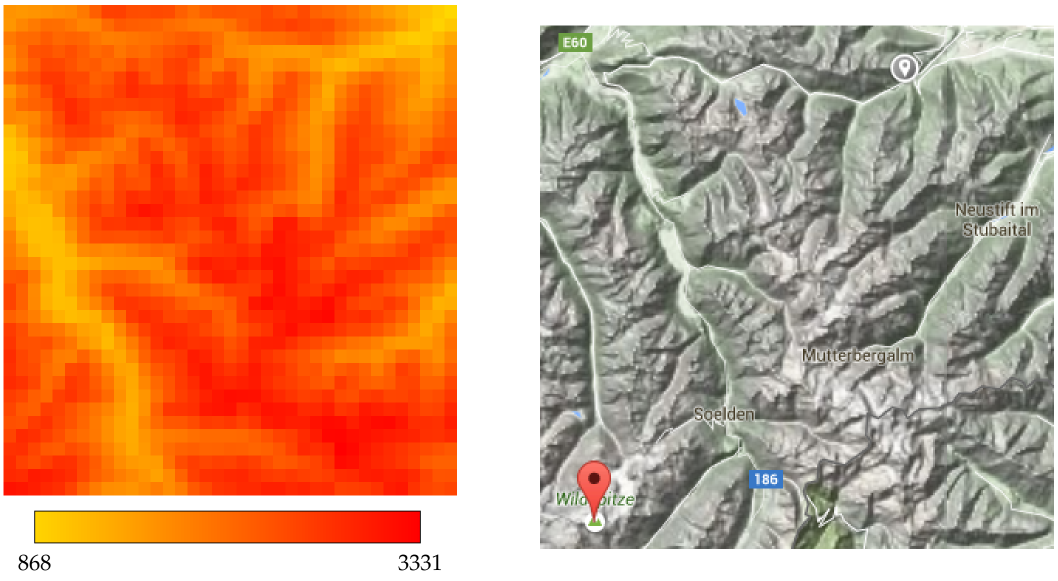
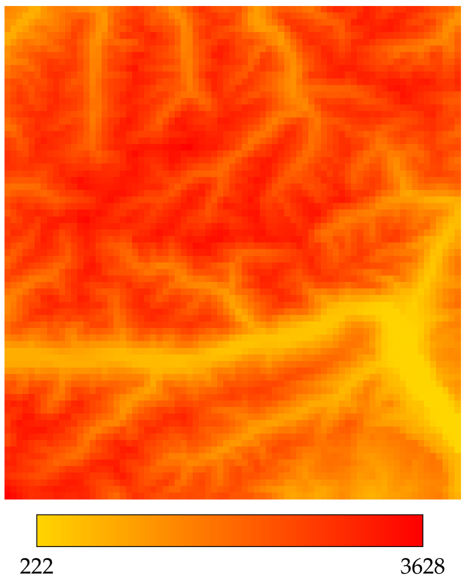


Figure C.6: The data set with the cell of the WildSpitze (Austria) in the lower left corner. The data set consists of 37 $\times$ 37 cells



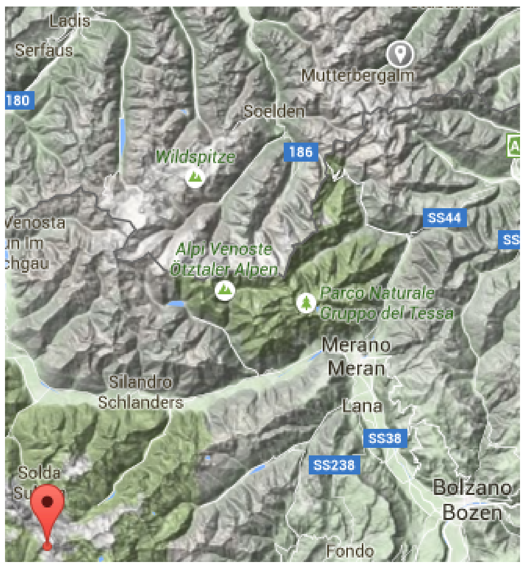
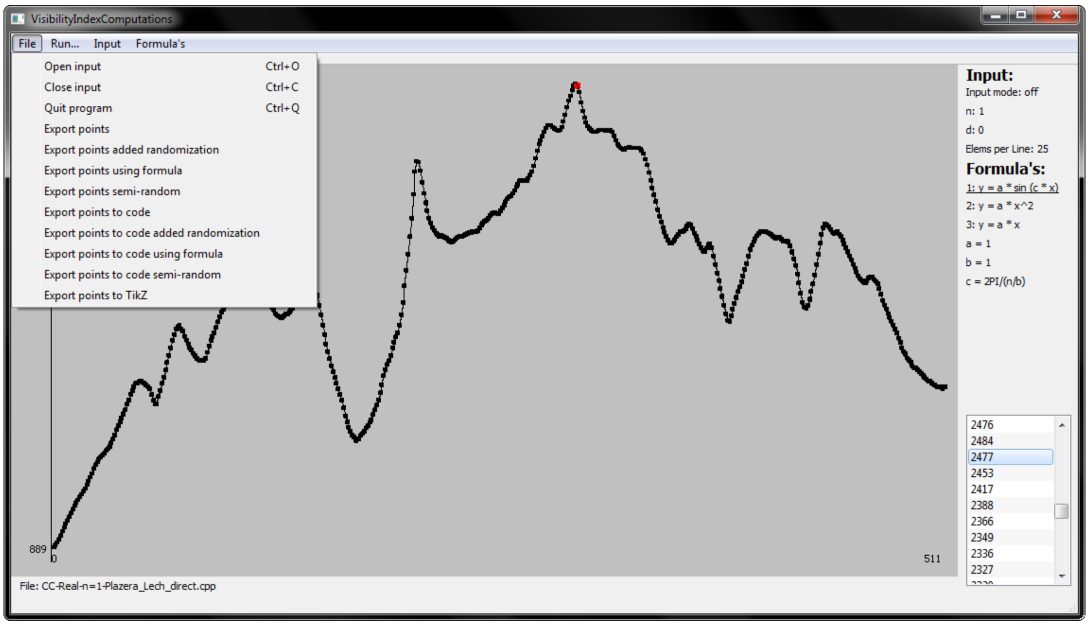


Figure C.7: The data set containing a part of the Italian Alpes. The data set consists of 75 $\times$ 75 cells

Appendix D

GUI Screenshots

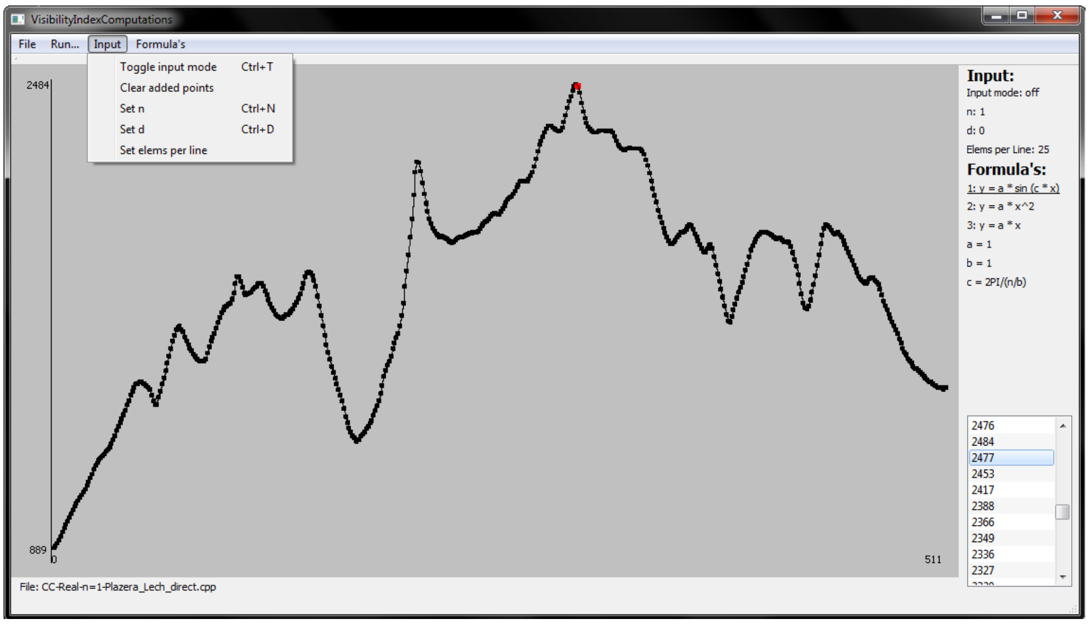
In this chapter we will show some more screenshots of the GUI. Many of these screenshots show the less interesting features of the application.


VisibilityIndexComputations
File
Run...
Input
Formula's
Open input Ctrl+O
Close input Ctrl+C
Quit program Ctrl+Q
Export points
Export points added randomization
Export points using formula
Export points semi-random
Export points to code
Export points to code added randomization
Export points to code using formula
Export points to code semi-random
Export points to TikZ
889
0
511
2476
2484
2477
2453
2417
2388
2366
2349
2336
2327

Input:
Input mode: off
n: 1
d: 0
Elems per Line: 25
Formula's:
1: y = a * sin (c * x)
2: y = a * x^2
3: y = a * x
a = 1
b = 1
c = 2PI/(n/b)

File: CC-Real-n=1-Plazera_Lech_direct.cpp

Figure D.1: The file menu structure for opening and exporting data.

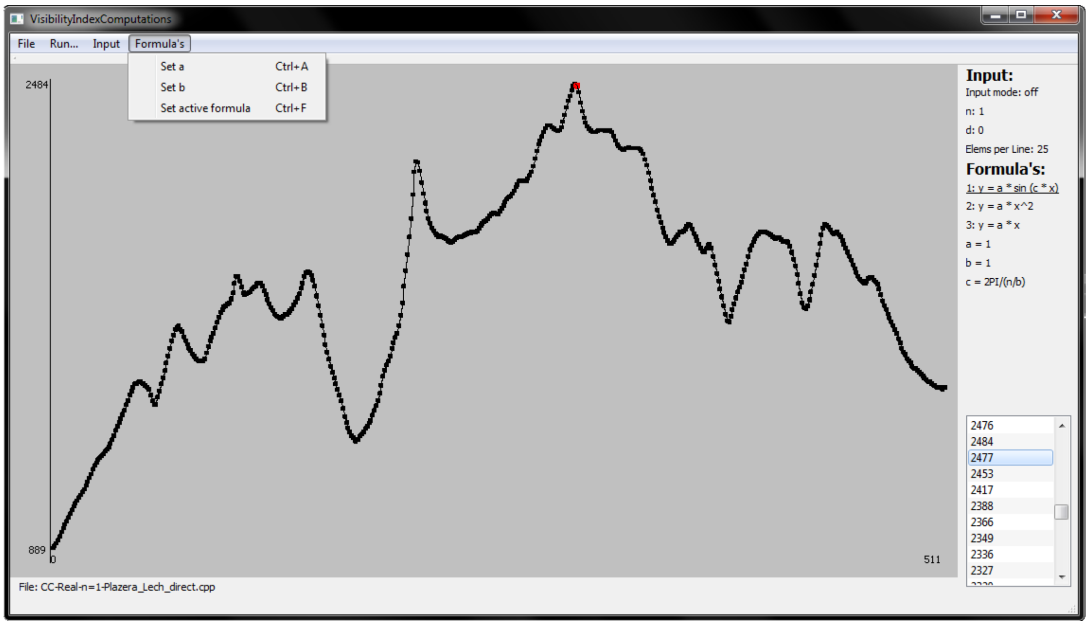


Input:
Input mode: off
n: 1
d: 0
Elems per Line: 25
Formula's:
1: y = a * sin (c * x)
2: y = a * x^2
3: y = a * x
a = 1
b = 1
c = 2PI/(n/b)

2476
2484
2477
2453
2417
2388
2366
2349
2336
2327

File: CC-Real-n=1-Plazera_Lech_direct.cpp

Figure D.2: The input menu structure for creating data sets.



Input:
Input mode: off
n: 1
d: 0
Elems per Line: 25
Formula's:
1: y = a * sin (c * x)
2: y = a * x^2
3: y = a * x
a = 1
b = 1
c = 2PI/(n/b)

2476
2484
2477
2453
2417
2388
2366
2349
2336
2327

File: CC-Real-n=1-Plazera_Lech_direct.cpp

Figure D.3: The formula menu structure for adjusting the formula parameters.

Appendix E

Results of the Experiments for the 1-Dimensional Problem

In this chapter all results of the experiments from Section 5.2 are presented. The results are presented per data set. A brief description of the table format is presented in Appendix E.1. After the description we first present the results of 1DVISIBILITYINDEX in Appendix E.2 and then the results of the brute-force approach in Appendix E.3. In the last section, Appendix E.4, we present the results for the experiment where we look at the visibility index and its influence on the running time.

E.1 Description of the Tables

Before we present the tables with the results, we give a brief description of the format of the tables and the values within.

# cells The number of cells.

# slabs in last step The number of slabs in the segment tree used by REDBLUEINTERSECTIONCOUNT in the last step of the recursion (the step in which the two halves of the (complete) data set are merged).

Max. # slabs The maximum number of slabs in the segment tree in any call to REDBLUEINTERSECTIONCOUNT during the execution of 1DVISIBILITYINDEX .

Avg. visibility The average visibility index.

Max. visibility The maximum visibility index.

Avg. running time The average running time of the the different runs in milliseconds.

St. dev. The percentage of standard deviation of the different runs with respect to the average running time.

Run *i* The running time of the i^{th} run of the experiment in milliseconds.

E.2 1DVisibilityIndex Running Times and Results

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. dev.
3,000	66	488	$2.36 \cdot 10^{-2}$	$1.43 \cdot 10^{-2}$	143	4.71
8,997	158	1318	$2.06 \cdot 10^{-2}$	$1.38 \cdot 10^{-2}$	535	1.45
29,990	408	1318	$1.89 \cdot 10^{-2}$	$1.34 \cdot 10^{-2}$	2,772	2.37
89,970	1,144	11,336	$1.67 \cdot 10^{-2}$	$1.24 \cdot 10^{-2}$	12,806	3.20
119,960	1,534	14,644	$1.58 \cdot 10^{-2}$	$1.22 \cdot 10^{-2}$	19,584	1.07
239,920	2,740	26,542	$1.28 \cdot 10^{-2}$	$1.15 \cdot 10^{-2}$	51,168	3.04
359,880	1,534	35,862	$1.11 \cdot 10^{-2}$	$1.13 \cdot 10^{-2}$	97,518	2.48
509,830	396	44,840	$832 \cdot 10^{-3}$	$3.29 \cdot 10^{-2}$	146,493	1.79

Table E.1: The results of 1DVisibilityIndex on the real-life data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
3,000	140	140	140	140	140
	156	140	140	140	156
8,997	530	530	530	546	530
	546	530	530	530	546
29,990	2,652	2,698	2,730	2,761	2,745
	2,839	2,808	2,808	2,854	2,823
89,970	12,027	12,370	12,495	12,682	12,823
	12,916	13,150	13,135	13,182	13,275
119,960	19,078	19,390	19,624	19,702	19,702
	19,796	19,656	19,531	19,656	19,702
239,920	48,001	49,296	50,200	50,949	51,495
	52,431	52,072	52,197	52,400	52,634
359,880	93,054	94,738	96,189	98,794	96,922
	96,704	99,668	100,741	99,231	99,138
509,830	140,727	142,963	146,952	148,761	147,357
	147,389	147,607	147,248	147,950	148,247

Table E.2: The running times of the different runs of 1DVisibilityIndex on the real-life data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
1,000	6	6	$3.00{\cdot}10^{-3}$	$3.00{\cdot}10^{-3}$	34	18.60
5,000	6	6	$5.99{\cdot}10^{-4}$	$6.00{\cdot}10^{-4}$	223	3.47
10,000	6	6	$3.00{\cdot}10^{-4}$	$3.00{\cdot}10^{-4}$	530	4.39
50,000	6	6	$6.00{\cdot}10^{-5}$	$6.00{\cdot}10^{-5}$	3,859	3.37
100,000	4	6	$3.00{\cdot}10^{-5}$	$3.00{\cdot}10^{-5}$	9,232	2.24
125,000	4	6	$2.40{\cdot}10^{-5}$	$2.40{\cdot}10^{-5}$	11,800	1.29
250,000	4	8	$1.2{\cdot}10^{-5}$	$1.2{\cdot}10^{-5}$	27,112	1.83
375,000	12	12	$8.00{\cdot}10^{-6}$	$8.00{\cdot}10^{-6}$	46,132	1.87
500,000	16	16	$6.20{\cdot}10^{-6}$	$6.20{\cdot}10^{-6}$	64,295	0.78

Table E.3: The results of 1DVISIBILITYINDEX on the line data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
1,000	31	31	46	31	31
	31	31	46	31	31
5,000	218	234	218	218	234
	218	218	218	218	234
10,000	499	499	514	546	514
	546	520	561	530	561
50,000	3,681	3,759	3,837	3,837	3,884
	3,822	3,837	3,837	3,915	4,180
100,000	9,718	9,188	9,172	9,204	9,469
	9,219	9,063	9,094	9,094	9,094
125,000	11,559	11,809	11,637	11,637	11,887
	11,934	11,762	12,027	11,934	11,809
250,000	26,161	26,738	26,988	27,112	26,910
	27,393	27,237	28,080	27,112	27,393
375,000	44,304	45,364	45,879	46,958	45,786
	46,581	46,737	47,158	47,158	45,957
500,000	63,242	64,218	64,584	64,209	65,104
	63,975	64,272	64,849	64,209	64,287

Table E.4: The running times of the different runs of 1DVISIBILITYINDEX on the line data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
1,000	122	122	$1.97 \cdot 10^{-3}$	$1.24 \cdot 10^{-1}$	40	19.36
5,000	212	212	$4.41 \cdot 10^{-3}$	$5.90 \cdot 10^{-2}$	257	4.31
10,000	60	210	$1.68 \cdot 10^{-3}$	$2.16 \cdot 10^{-2}$	589	2.43
50,000	570	570	$4.26 \cdot 10^{-3}$	$1.33 \cdot 10^{-2}$	5,059	1.76
100,000	300	494	$2.13 \cdot 10^{-3}$	$6.41 \cdot 10^{-2}$	12,467	1.13
125,000	1,102	1,102	$2.06 \cdot 10^{-4}$	$1.26 \cdot 10^{-2}$	16,938	0.80
250,000	1,326	1,326	$9.69 \cdot 10^{-5}$	$5.81 \cdot 10^{-3}$	41,359	1.09
375,000	438	1,076	$6.38 \cdot 10^{-5}$	$9.31 \cdot 10^{-3}$	70,229	1.80
500,000	1,402	1,402	$4.83 \cdot 10^{-5}$	$3.75 \cdot 10^{-3}$	100,568	1.55

Table E.5: The results of 1DVISIBILITYINDEX on the semi-random ($d = 10$) data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
1,000	46	46	31	46	46
	46	31	31	46	31
5,000	249	265	249	249	265
	249	265	280	249	249
10,000	577	577	592	592	577
	592	577	624	592	592
50,000	4,867	4,976	4,992	5,054	5,070
	5,116	5,132	5,148	5,116	5,116
100,000	12,168	12,292	12,402	12,542	12,495
	12,511	12,573	12,495	12,573	12,620
125,000	16,816	16,894	16,957	17,019	17,050
	17,191	16,770	16,894	16,770	17,019
250,000	40,950	41,589	41,386	41,464	41,968
	41,636	41,542	41,277	41,464	40,310
375,000	67,953	68,842	69,420	69,638	70,231
	71,214	71,463	71,136	71,931	70,465
500,000	98,155	98,514	98,779	99,964	101,306
	101,571	101,977	102,008	101,743	101,655

Table E.6: The running times of the different runs of 1DVISIBILITYINDEX on the semi-random ($d = 10$) data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
1,000	44	44	$1.15{\cdot}10^{-2}$	$6.00{\cdot}10^{-2}$	39	20.53
5,000	36	140	$3.25{\cdot}10^{-3}$	$4.66{\cdot}10^{-2}$	251	4.53
10,000	154	184	$1.86{\cdot}10^{-3}$	$2.65{\cdot}10^{-2}$	595	3.21
50,000	290	598	$4.94{\cdot}10^{-4}$	$1.17{\cdot}10^{-2}$	5,088	2.03
100,000	422	634	$2.10{\cdot}10^{-4}$	$5.74{\cdot}10^{-3}$	12,505	0.79
125,000	358	970	$1.65{\cdot}10^{-4}$	$6.94{\cdot}10^{-3}$	16,628	0.94
250,000	506	1,236	$1.04{\cdot}10^{-4}$	$6.64{\cdot}10^{-3}$	41,032	1.99
375,000	2,732	2,732	$7.12{\cdot}10^{-5}$	$4.83{\cdot}10^{-3}$	70,752	1.22
500,000	812	1,264	$4.96{\cdot}10^{-5}$	$3.14{\cdot}10^{-3}$	100,378	1.25

Table E.7: The results of 1DVISIBILITYINDEX on the semi-random ($d = 15$) data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
1,000	46	46	31	46	31
	31	46	31	31	46
5,000	249	249	280	249	249
	249	234	249	294	249
10,000	577	592	592	639	608
	592	592	577	608	577
50,000	4,867	4,960	5,038	5,086	5,148
	5,148	5,132	5,163	5,163	5,179
100,000	12,292	12,386	12,480	12,604	12,589
	12,526	12,511	12,573	12,573	12,511
125,000	16,660	16,489	16,536	16,816	16,894
	16,801	16,567	16,536	16,520	16,458
250,000	39,592	40,154	40,653	40,856	40,856
	40,950	41,125	41,917	41,995	42,135
375,000	70,215	70,590	69,934	70,371	71,448
	71,650	71,728	71,838	70,418	69,326
500,000	98,592	99,450	99,684	99,590	99,715
	100,074	102,086	102,164	101,164	100,494

Table E.8: The running times of the different runs of 1DVISIBILITYINDEX on the semi-random ($d = 15$) data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
1,000	68	122	$4.05 \cdot 10^{-2}$	$5.00 \cdot 10^{-2}$	43	22.51
5,000	320	620	$3.91 \cdot 10^{-2}$	$4.96 \cdot 10^{-2}$	271	6.78
10,000	570	1,182	$3.89 \cdot 10^{-2}$	$4.96 \cdot 10^{-2}$	635	3.28
50,000	1,680	4,910	$3.83 \cdot 10^{-2}$	$4.94 \cdot 10^{-2}$	5,776	3.22
100,000	3,050	9,542	$3.77 \cdot 10^{-2}$	$4.92 \cdot 10^{-2}$	15,882	4.23
125,000	3,738	11,774	$3.74 \cdot 10^{-2}$	$4.89 \cdot 10^{-2}$	22,118	4.56
250,000	1,382	22,808	$3.63 \cdot 10^{-2}$	$1.48 \cdot 10^{-1}$	61,544	4.65
375,000	1,240	27,048	$3.56 \cdot 10^{-2}$	$1.25 \cdot 10^{-1}$	117,192	4.77
500,000	112	20,450	$3.49 \cdot 10^{-2}$	$4.31 \cdot 10^{-1}$	186,939	4.95

Table E.9: The results of 1DVISIBILITYINDEX on the sine ($b = 15$) data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
1,000	46	62	46	31	46
	46	46	46	31	31
5,000	296	249	265	265	312
	265	265	265	265	265
10,000	592	624	624	639	639
	624	670	639	655	639
50,000	5,366	5,596	5,662	5,928	5,756
	5,818	5,881	5,928	5,896	5,928
100,000	14,570	15,022	15,459	15,787	15,943
	16,286	16,348	16,473	16,520	16,411
125,000	20,014	20,904	21,724	22,105	22,354
	22,354	22,620	23,259	22,651	23,197
250,000	55,645	58,406	59,946	60,902	61,916
	62,431	63,944	64,116	63,913	64,225
375,000	106,984	108,997	113,419	116,766	119,215
	120,759	121,305	121,305	121,524	193,362
500,000	171,959	171,725	179,010	188,838	188,675
	193,721	192,285	193,580	196,232	193,362

Table E.10: The running times of the different runs of 1DVISIBILITYINDEX on the in ($b = 15$) data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
1,000	40	72	$2.15 \cdot 10^{-2}$	$2.60 \cdot 10^{-2}$	46	15.86
5,000	166	304	$2.00 \cdot 10^{-2}$	$2.48 \cdot 10^{-2}$	357	5.64
10,000	316	620	$1.98 \cdot 10^{-2}$	$2.48 \cdot 10^{-2}$	969	2.45
50,000	998	2,754	$1.96 \cdot 10^{-2}$	$2.48 \cdot 10^{-2}$	9,065	3.25
100,000	1,698	4,910	$1.94 \cdot 10^{-2}$	$2.47 \cdot 10^{-2}$	24,393	1.99
125,000	2,036	6,054	$1.93 \cdot 10^{-2}$	$2.47 \cdot 10^{-2}$	30,859	1.54
250,000	516	11,774	$1.91 \cdot 10^{-2}$	$1.41 \cdot 10^{-1}$	64,662	4.90
375,000	112	14,230	$1.84 \cdot 10^{-2}$	$4.61 \cdot 10^{-1}$	99,610	6.40
500,000	78	22,808	$1.92 \cdot 10^{-2}$	$4.78 \cdot 10^{-1}$	148,902	6.36

Table E.11: The results of 1DVISIBILITYINDEX on the sine ($b = 30$) data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
1,000	46	31	46	46	62
	46	46	46	46	46
5,000	343	343	343	343	343
	390	358	343	390	374
10,000	936	936	967	967	1,014
	967	967	967	998	967
50,000	8,439	8,642	9,110	9,126	9,204
	9,172	9,204	9,157	9,438	9,157
100,000	23,384	24,226	24,523	24,694	24,897
	24,944	24,788	24,398	24,117	23,961
125,000	30,662	31,090	30,778	31,590	30,544
	31,125	31,137	31,153	30,560	29,905
250,000	59,794	59,155	61,900	65,223	66,175
	65,894	66,799	67,501	67,298	66,877
375,000	86,112	93,334	95,752	98,233	100,651
	102,195	103,178	104,816	105,751	106,080
500,000	128,591	139,183	144,830	147,342	150,041
	152,069	153,223	155,735	158,355	159,650

Table E.12: The running times of the different runs of 1DVISIBILITYINDEX on the sine ($b = 30$) data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
100	4	4	1	1	3	210.82
250	4	4	1	1	5	161.02
500	4	4	1	1	15	0.00
750	4	4	1	1	25	33.59
1,000	4	4	1	1	31	0.00
1,250	4	4	1	1	46	15.86
1,500	4	4	1	1	57	26.11
1,750	4	4	1	1	65	10.35
2,000	4	4	1	1	70	12.05

Table E.13: The results of 1DVisibilityIndex on the parabola ($a = 1$) data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
100	0	0	0	15	0
	0	0	0	0	15
250	0	0	15	0	15
	0	15	0	0	0
500	15	15	15	15	15
	15	15	15	15	15
750	31	15	31	31	15
	31	15	31	15	31
1,000	31	31	31	31	31
	31	31	31	31	31
1,250	46	46	46	46	46
	31	46	46	46	46
1,500	62	46	46	46	46
	46	31	46	46	46
1,750	93	46	62	62	46
	46	62	46	62	46
2,000	62	62	78	62	78
	78	62	78	62	78

Table E.14: The running times of the different runs of 1DVisibilityIndex on the parabola ($a = 1$) data set.

# cells	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. Dev.
100	6	6	$2.98{\cdot}10^{-2}$	$3.00{\cdot}10^{-2}$	2	316.23
250	6	6	$1.20{\cdot}10^{-2}$	$1.20{\cdot}10^{-2}$	6	129.10
500	6	6	$5.99{\cdot}10^{-3}$	$6.00{\cdot}10^{-3}$	15	0.00
750	6	6	$4.00{\cdot}10^{-3}$	$4.00{\cdot}10^{-3}$	23	36.66
1,000	6	6	$3.00{\cdot}10^{-3}$	$3.00{\cdot}10^{-3}$	34	18.60
1,250	6	6	$2.40{\cdot}10^{-3}$	$2.40{\cdot}10^{-3}$	45	19.58
1,500	6	6	$2.00{\cdot}10^{-3}$	$2.00{\cdot}10^{-3}$	54	15.62
1,750	6	6	$1.71{\cdot}10^{-3}$	$1.71{\cdot}10^{-3}$	71	19.35
2,000	6	6	$1.50{\cdot}10^{-3}$	$1.50{\cdot}10^{-3}$	76	9.79

Table E.15: The results of 1DVISIBILITYINDEX on the parabola ($a = -1$) data set.

# cells	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
100	0	0	0	15	0
	0	0	0	0	0
250	15	0	0	15	0
	15	0	15	0	0
500	15	15	15	15	15
	15	15	15	15	15
750	15	31	31	15	31
	31	15	15	31	15
1,000	46	31	31	31	46
	31	31	31	31	31
1,250	46	46	31	62	31
	46	46	46	46	46
1,500	62	46	46	62	46
	46	62	46	62	62
1,750	62	62	62	62	62
	78	93	96	60	70
2,000	70	70	80	80	70
	70	71	78	78	93

Table E.16: The running times of the different runs of 1DVISIBILITYINDEX on the parabola ($a = -1$) data set.

E.3 Brute-Force Approach Running Times

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
3,000	16,572	0.24	16,567	16,614	16,536
8,997	485,739	0.31	485,769	484,240	487,207

Table E.17: The results of the brute-force approach on the real-life data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
1,000	15	0.00	15	15	15
5,000	473	1.83	483	468	468
10,000	1,955	0.44	1,950	1.965	19.50

Table E.18: The results of the brute-force approach on the line data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
1,000	556	1.56	561	561	546
5,000	31,605	1.56	31,044	31,964	31,808
10,000	66,501	2.31	67,251	67,516	64,738

Table E.19: The results of the brute-force approach on the semi-random ($d = 10$) data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
1,000	166	5.22	171	156	171
5,000	8,923	0.70	8,860	8,965	8,923
10,000	91,759	1.14	92,960	91,057	91,260

Table E.20: The results of the brute-force approach on the semi-random ($d = 15$) data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
1,000	315	4.76	300	315	330
5,000	33,312	8.56	36,603	31,761	31,574
10,000	253,432	0.18	253,032	253,921	253,344

Table E.21: The results of the brute-force approach on the sine ($b = 15$) data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
1,000	145	6.36	140	156	140
5,000	16,187	0.95	16,364	16,114	16,083
10,000	132,459	0.74	133,411	132,506	131,461

Table E.22: The results of the brute-force approach on the sine ($b = 30$) data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
100	5	173.21	0	15	0
250	83	10.43	93	78	78
500	41	21.12	46	31	46
750	2,246	0	2,246	2,246	2,246
1,000	5,303	0.51	5,319	5,272	5,319
1,250	10,415	0.61	10,452	10,452	10,342
1,500	18,241	0.94	18,142	18,142	18,439
1,750	29,848	0.06	29,858	29,827	29,858
2,000	45,234	0.46	45,286	45,411	45,006

Table E.23: The results of the brute-force approach on the parabola ($a = 1$) data set.

# cells	Avg. running time	St. dev.	Run 1	Run 2	Run 3
100	0	$\times$	0	0	0
250	5	173.21	0	15	0
500	5	173.21	0	15	0
750	10	86.60	0	15	15
1,000	15	0	15	15	15
1,250	26	35.99	31	15	31
1,500	36	24.06	31	31	46
1,750	57	16.30	62	62	46
2,000	78	0	78	78	78

Table E.24: The results of the brute-force approach on the parabola ($a = -1$) data set.

E.4 Average Visibility Index and its Influence on the Running Time Experimental Data

b	# slabs in last step	Max. # slabs	Avg. visibility	Max. visibility	Avg. running time	St. dev.
15	3,050	9,542	$3.77{\cdot}10^{-2}$	$4.92{\cdot}10^{-2}$	16,421	4.40
30	1,698	4,910	$1.94{\cdot}10^{-2}$	$2.47{\cdot}10^{-2}$	15,837	3.83
45	1,270	3,588	$1.30{\cdot}10^{-2}$	$1.65{\cdot}10^{-2}$	15,498	3.95
60	998	2,574	$9.84{\cdot}10^{-3}$	$1.24{\cdot}10^{-2}$	15,251	3.90
75	842	2,112	$7.90{\cdot}10^{-3}$	$9.91{\cdot}10^{-3}$	15,410	4.33
90	788	1,942	$6.60{\cdot}10^{-3}$	$8.27{\cdot}10^{-3}$	14,964	4.57

Table E.25: The results of 1DVISIBILITYINDEX on the sine data sets of size 100,000 cells and various values for *b*.

b	Run 1/2	Run 3/4	Run 5/6	Run 7/8	Run 9/10
15	15,027	15,586	16,009	16,198	16,429
	16,756	16,805	16,937	17,209	17,258
30	14,767	15,352	15,808	15,726	15,953
	15,124	16,423	16,593	16,274	16,379
45	14,446	14,752	14,993	15,252	15,493
	14,780	15,939	16,139	16,098	16,090
60	14,045	14,584	14,912	15,136	15,264
	15,469	15,676	15,822	15,793	15,809
75	14,211	14,434	16,304	15,008	15,536
	15,399	15,696	15,887	15,869	15,758
90	16,969	14,039	14,372	14,979	15,290
	15,255	15,327	15,483	15,605	15,597

Table E.26: The running times of the different runs of 1DVISIBILITYINDEX on the sine data sets of size 100,000 cells and various values for *b*.

Appendix F

Results of the Experiments for the 2-Dimensional Problem

The graphs of the results for the data sets not presented in Section 6.2 are presented in Appendix F.1. All results of the experiments from Section 5.3 are presented in Appendix F.2.

F.1 Graphs of data sets not already presented

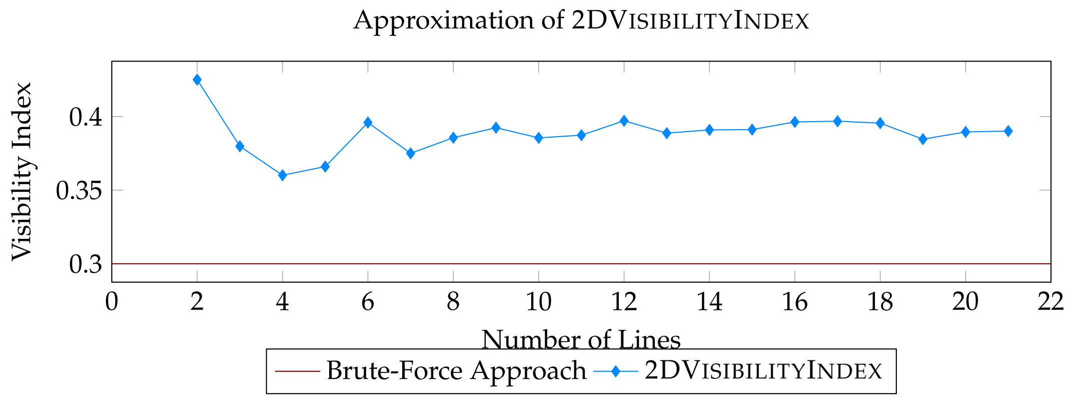


Figure F.1: The visibility index of the brute-force approach (0.3000) and the approximation of 2DVisibilityIndex for the top right part of the Schesaplana data set.

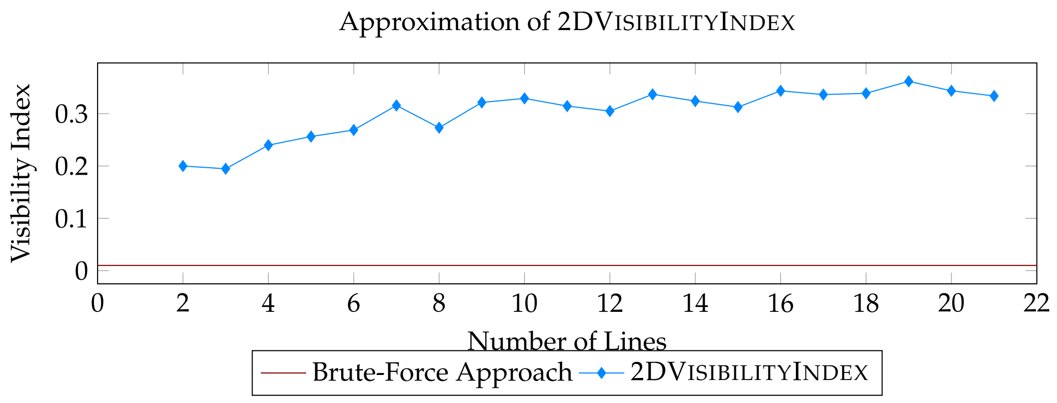


Figure F.2: The visibility index of the brute-force approach (0.0100) and the approximation of 2DVisibilityIndex for the Furgglenfirst data set.

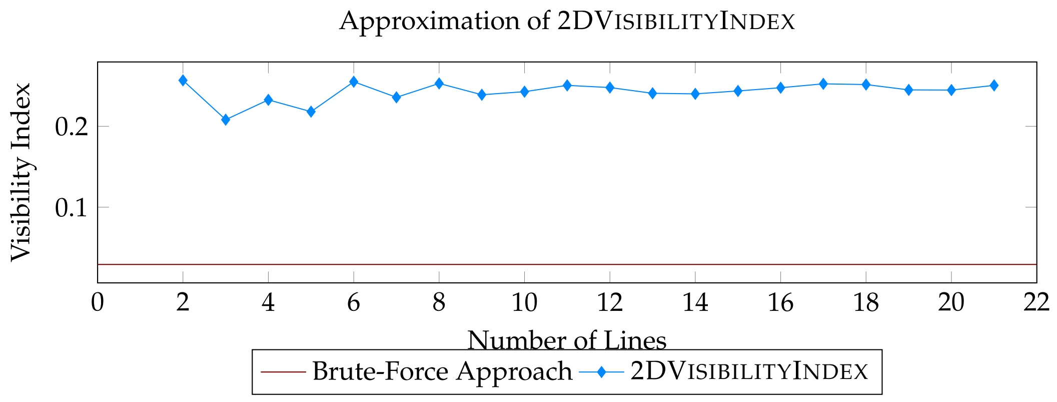


Figure F.3: The visibility index of the brute-force approach (0.0292) and the approximation of 2DVisibilityIndex for the Wildspitze data set.

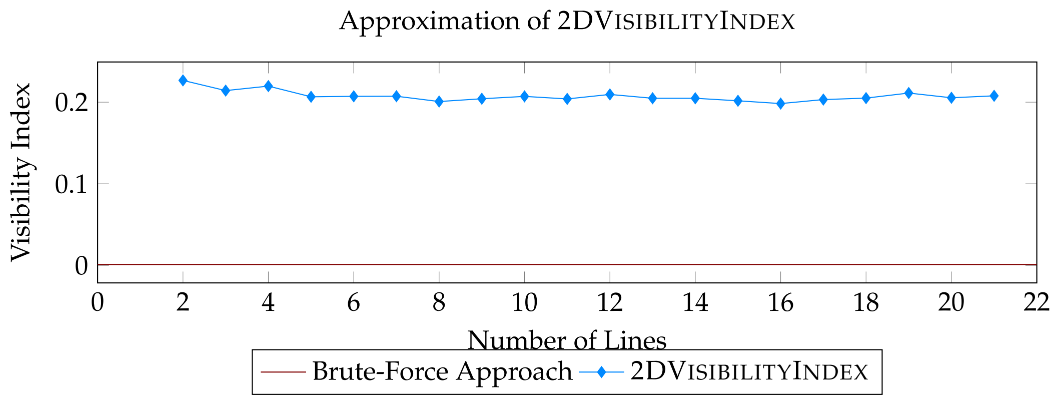


Figure F.4: The visibility index of the brute-force approach (0.0011) and the approximation of 2DVisibilityIndex for the Alpes data set.

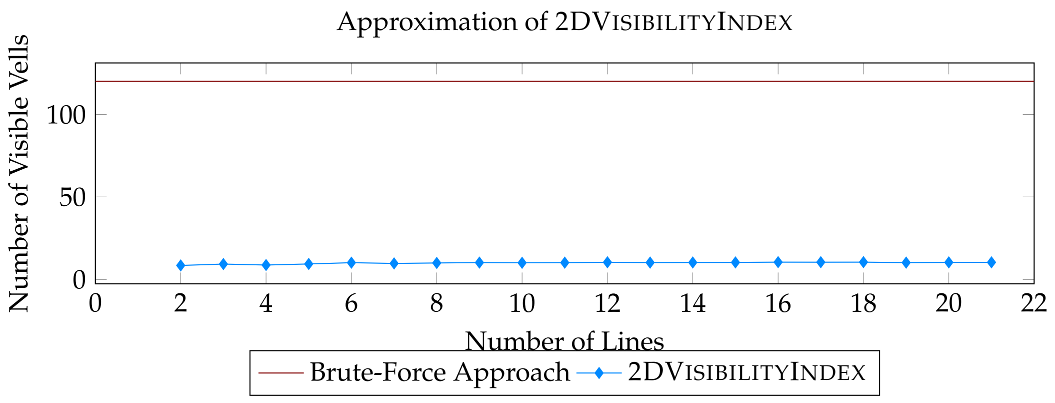


Figure F.5: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the top right part of the Schesaplana data set.

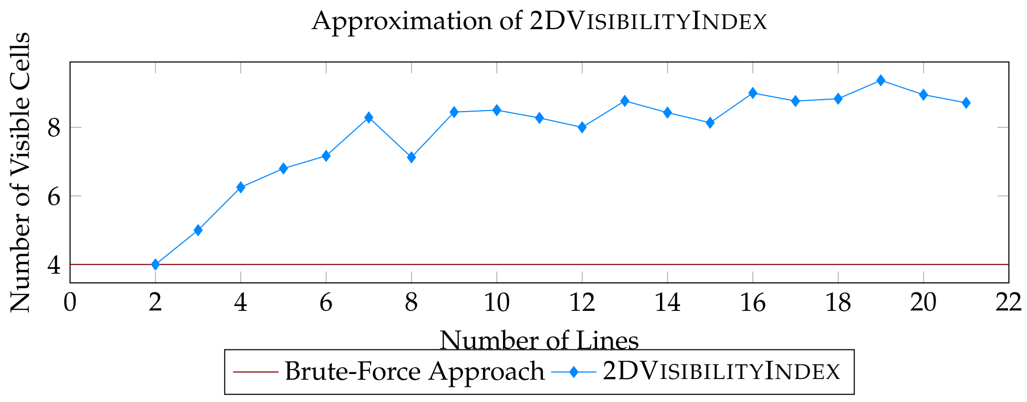


Figure F.6: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the Furgglenfirst data set.

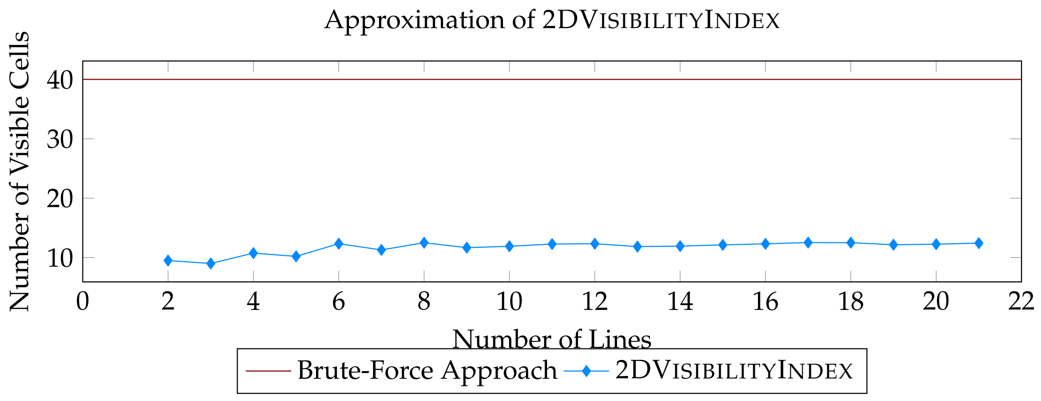


Figure F.7: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the Wildspitze data set.

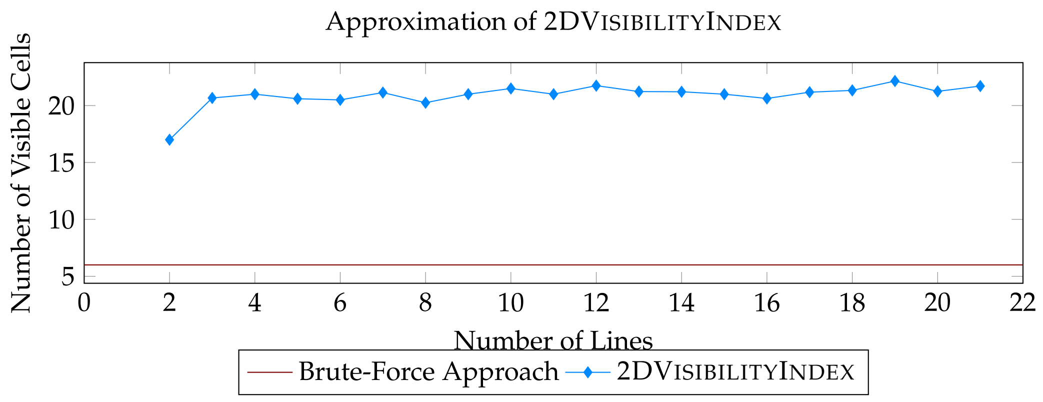


Figure F.8: The number of visible cells for the brute-force approach and the approximation of 2DVisibilityIndex for the Alpes data set.

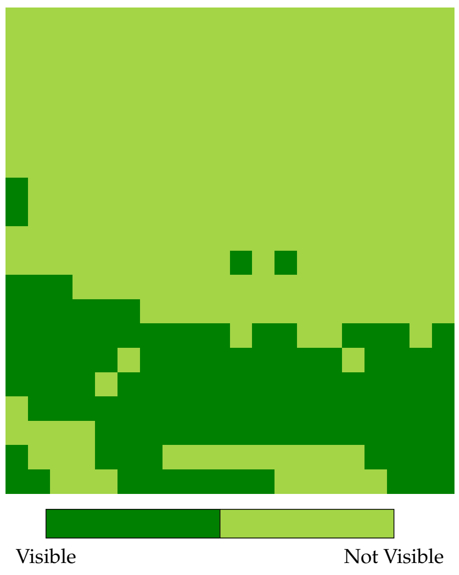


Figure F.9: The visible cells for the brute-force approach for the right top part of the Schesaplana data set.

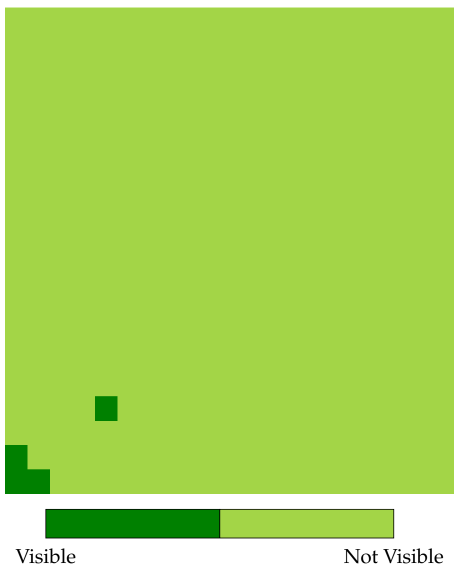


Figure F.10: The visible cells for the brute-force approach for the Furgglenfirst data set.

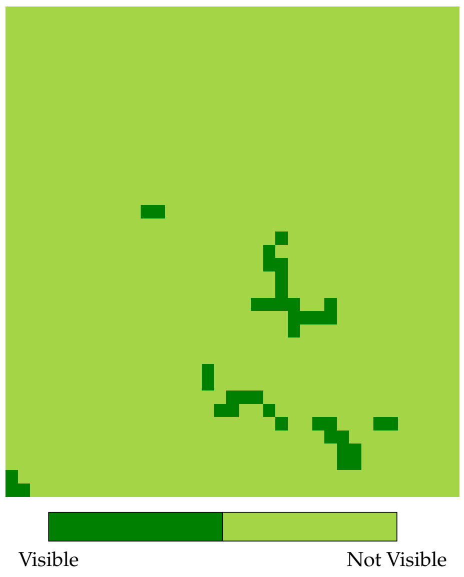


Figure F.11: The visible cells for the brute-force approach for the right top part of the Wildspitze data set.

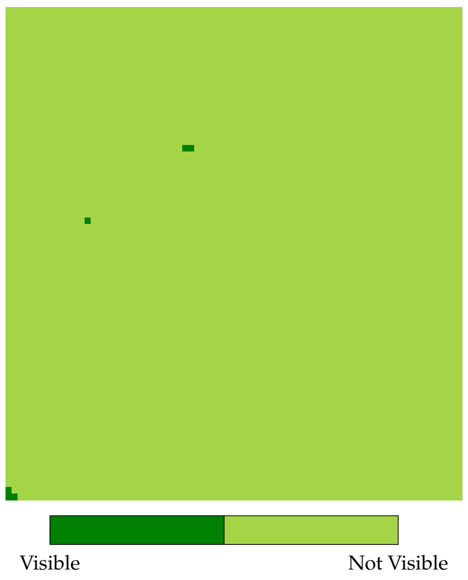


Figure F.12: The visible cells for the brute-force approach for the Alpes data set.

F.2 All Results of the Experiments for 2DVisibilityIndex

# lines (m)	Glat-thorn	Schesa-plana	Near Schesa-plana	Piz Badile	Furgglen-first	Wild-spitze	Alpes
Brute-force	0.1000	0.0033	0.3000	0.0785	0.0100	0.0292	0.0011
2	0.2000	0.3500	0.4250	0.1379	0.2000	0.2568	0.2267
3	0.1684	0.2908	0.3798	0.1336	0.1947	0.2082	0.2142
4	0.2312	0.3583	0.3601	0.1591	0.2398	0.2327	0.2197
5	0.1667	0.3094	0.3660	0.1852	0.2563	0.2181	0.2066
6	0.2002	0.3308	0.3959	0.1838	0.2688	0.2551	0.2072
7	0.2057	0.3336	0.3750	0.1723	0.3157	0.2358	0.2073
8	0.2161	0.3412	0.3856	0.1836	0.2732	0.2530	0.2008
9	0.1971	0.3198	0.3924	0.1871	0.3214	0.2390	0.2042
10	0.2304	0.3072	0.3855	0.1989	0.3292	0.2428	0.2071
11	0.2049	0.3352	0.3873	0.2084	0.3145	0.2506	0.2040
12	0.2173	0.3339	0.3971	0.2109	0.3050	0.2479	0.2095
13	0.2187	0.3268	0.3887	0.1997	0.3370	0.2408	0.2048
14	0.2231	0.3315	0.3909	0.2262	0.3241	0.2401	0.2048
15	0.2211	0.3440	0.3911	0.2139	0.3126	0.2437	0.2017
16	0.2209	0.3350	0.3963	0.2062	0.3436	0.2477	0.1983
17	0.2162	0.3159	0.3968	0.2232	0.3365	0.2525	0.2032
18	0.2195	0.3333	0.3955	0.2101	0.3389	0.2516	0.2050
19	0.2165	0.3120	0.3846	0.2126	0.3618	0.2450	0.2112
20	0.2245	0.3195	0.3895	0.2133	0.3437	0.2448	0.2054
21	0.2183	0.3273	0.3901	0.2206	0.3338	0.2506	0.2078

Table F.1: The visibility index of the brute-force approach and the visibility index of 2DVisibilityIndex for the various data sets.

# lines (m)	Glat-thorn	Schesa-plana	Near Schesa-plana	Piz Badile	Furgglen-first	Wild-spitze	Alpes
Brute-force	40	3	120	66	4	40	6
2	4.0000	10.5000	8.5000	4.0000	4.0000	9.5000	17.0000
3	4.0000	10.3333	9.3333	5.0000	5.0000	9.0000	20.6667
4	6.0000	13.7500	8.7500	6.0000	6.2500	10.7500	21.0000
5	4.2000	11.8000	9.4000	7.2000	6.8000	10.2000	20.6000
6	5.3333	12.8333	10.1667	7.0000	7.1667	12.3333	20.5000
7	5.4286	13.1429	9.7143	6.7143	8.2857	11.2857	21.1429
8	5.8750	13.5000	10.0000	7.1250	7.1250	12.5000	20.2500
9	5.2222	12.5556	10.2222	7.3333	8.4444	11.6667	21.0000
10	6.3000	12.2000	10.1000	7.8000	8.5000	11.9000	21.5000
11	5.4546	13.3636	10.1818	8.0909	8.2727	12.2727	21.0000
12	5.9167	13.2500	10.4167	8.2500	8.0000	12.3333	21.7500
13	5.9231	13.0769	10.2308	7.6923	8.7692	11.8462	21.2308
14	6.0714	13.2857	10.2857	8.7857	8.4286	11.9286	21.2143
15	6.0000	13.7333	10.3333	8.2667	8.1333	12.1333	21.0000
16	6.0625	13.5000	10.5000	8.0000	9.0000	12.3125	20.6250
17	5.8824	12.5882	10.4706	8.7059	8.7647	12.5294	21.1765
18	6.0556	13.5000	10.5000	8.1667	8.8333	12.5000	21.3333
19	5.8947	12.5789	10.2105	8.2632	9.3684	12.1579	22.1579
20	6.1500	12.9500	10.3500	8.3000	8.9500	12.2500	21.2500
21	5.9524	13.2381	10.3810	8.5714	8.7143	12.4286	21.7143

Table F.2: The number of visible cells of the brute-force approach and the average number of visible cells of 2DVisibilityIndex for the various data sets.