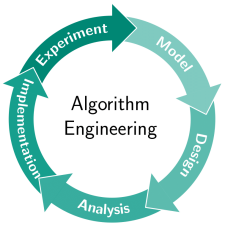

A Sweepline Algorithm for Calculating the Isolation of Mountains

Bachelor's Thesis of

Nicolai Hüning

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering (ITI)

Reviewer:	Prof. Peter Sanders
Second reviewer:	Prof. Thomas Bläsius
Advisor:	M.Sc. Daniel Funke

30. Juli 2022 – 30. November 2022

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . . . .

(Nicolai Hüning)

Abstract

The isolation is an important metric for the classification of mountain peaks. It specifies the distance between a peak and the closest point to the peak with the same elevation. The closest point is called the Isolation Limit Point (ILP).

In this thesis a sweepline algorithm to calculate the isolation for all peaks with the help of Digital Elevation Models (DEMs) is developed. The work builds upon the approach by Kirmse and de Ferrante [10], who developed a $O(n^2)$ algorithm for the isolation calculation to prepare for the more cost intensive prominence calculation.

DEM-Data is a format to store elevation data of the earth or other planets. The elevation data is stored on a grid which is divided into tiles of one latitude times one longitude. With this approach the DEM-data contains a 3D representation of a planet.

The algorithm developed for this thesis is divided in two phases, where one phase always works on one DEM-Tile at a time. During the first phase potential peaks are calculated with the help of a simple heuristic and after this an upper bound for the isolation is found. With the help of this upper bound the peak is now linked with tiles that could theoretically contain an ILP. During the second phase the Isolation Witness Points are calculated within every linked tile. The ILP with the shortest distance to the peak is now the global ILP and the distance to this global ILP is the isolation of the peak. With this approach the isolation for every peak is calculated in one run of the algorithm.

To calculate an ILP a sweepline approach is used which goes down the contour line of the DEM-Model withing one tile. With this concept the 3D-Search for the nearest point with the same elevation is now reduced to a 2D-Search in the contour-line-plane of the peak.

The two phases are then parallized by running calculations parallel per tile in one phase. This results in a very fast approach to calculate the isolation for all mountain peaks of a planet.

Zusammenfassung

Zur Klassifizierung von Berggipfeln ist die Dominanz, manchmal auch Isolation genannt, eine wichtige Metrik. Sie gibt für einen Gipfel den Abstand zum nächsten Punk eines anderen Berges an, den Dominanz-Zeugen, welcher die gleiche Höhe hat wie der Gipfel.

In dieser Arbeit wird ein Sweepline Algorithmus zur Berechnung der Dominanz mithilfe von DEM-Daten entwickelt. Dabei wird an die Arbeit von Kirmse und de Ferranti [10] angeknüpft, welche einen einfachen $O(n^2)$ Algorithmus zur Berechnung der Dominanz verwenden um damit die kosten-intensivere Prominenzberechnung vorzubereiten.

DEM-Daten sind ein Format um Höhendaten der Erde, oder anderer Planeten zu speichern. Dabei wird eine Repräsentation des Planeten in 3D erzeugt indem, Höhendaten mithilfe eines Gitters gespeichert werden. Diese Gitter sind in Kacheln, von meist einem Längengrad mal einem Breitengrad, aufgeteilt.

Der in dieser Arbeit entwickelte Algorithmus ist in zwei Phasen aufgeteilt, wobei eine Phase immer auf einzelnen DEM-Kacheln ausgeführt wird. In der ersten Phase werden potentielle Berggipfel mithilfe einer einfachen Heuristik berechnet und anschließend eine obere Schranke für die Dominanz bestimmt. Mithilfe der oberen Schranke kann nun der Gipfel auf Kacheln verteilt werden, welche potentiell einen Dominanz-Zeugen enthalten können. In Phase zwei werden dann die Dominanz-Zeugen innerhalb der Kacheln für diese Gipfel bestimmt und die Ergebnisse zum Schluss zusammengeführt. Damit wird in einem Durchlauf die Dominanz für alle Berggipfel bestimmt.

Zur Berechnung des Dominanz-Zeugen wird ein Sweepline-Ansatz verwendet, welcher die Höhenlinien des DEM-Models abtastet und damit die 3D-Suche, nach dem nächst höheren Punkt, auf eine 2D-Suche, nach dem nächsten Punkt in derselben Höhenlinie, reduziert.

Die einzelnen Phasen werden in dieser Arbeit parallelisiert, indem Berechnungen innerhalb einer Phase für einzelne Kacheln parallel ausgeführt werden. Daraus resultiert ein sehr schneller Algorithmus zur Berechnung der Dominanz aller Berggipfel eines Planeten.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The isolation of a mountain is the distance along the surface to the nearest point with a higher elevation for a given mountain peak (called the isolation limit point (ILP)). This is an important metric for mountain classification. Mr. Peter Grimm, for example, used in his book "Ansichten, Systematiken und Methoden zur Einteilung der Alpen" (Views, Systematics and Methods for the Classification of the Alps) [7] the prominence and especially the isolation to define and create a tier-list for mountains. The problem of mountain identification is however not quite easy to answer, since in real world scenarios a lot of factors, like the view from the valley and more, are used to define a peak as mountain. The work "What is a mountain?" by Peter Fisher and Jo Wood (1998) [3] deals with that question in a humorous way.

A mountain with high isolation is especially interesting for climbers, because it specifies the remoteness of a mountain peak. However, since the isolation is just the distance to the nearest point with the same elevation, remote islands with just a few metres of elevation can have a very big isolation.

A Digital Elevation Model, or DEM, is a way to represent the surface of the earth in a digital and manageable form factor. With the help of these DEMs a lot of manual classification work can be automated.

1.1 Motivation

While hiking in the beautiful Alps the question appeared how interesting metrics—like prominence or isolation—of mountains could be efficiently calculated with the help of DEMs.

Since the surface of our earth is rather static, the question arises why an efficient algorithm for such metrics would be necessary. However, as the measurement techniques to create DEMs of our earth get more evolved, higher resolution DEMs are built. For example the data from the TanDEM-X mission has a 7.5 times higher resolution compared to the data from the SRTM, which is used for testing in this work. With increasing accuracy faster algorithms are required to get a solution in reasonable time.

It could also be interesting to get this kind of metrics for other planets. There are for example efforts to create a full DEM of Mars [8].

1.2 Contribution

This solution for calculating the isolation builds on the work of Andrew Kirmse and Jonathan de Ferranti [10] (source code [11]). They developed a solution to calculate the prominence and isolation for every mountain with the help of Digital Elevation Models (DEM). The special focus of that work however lies on the more difficult task of calculating the prominence of mountains.

The isolation is used as a first step to get more viable candidates of peaks for the prominence calculation. Because of that the isolation algorithm is a rather simple $O(n^2)$ approach, with a few performance optimisations and very poor multithreading capabilities (Cf. figure 4.1).

In this work a more sophisticated approach is developed to calculating the isolation for every mountain in a DEM. It uses a sweepline / scanline approach at its core to get results with a time complexity of $N \log(N)$.

1.3 Outline

In Chapter 2 the DEM data model is introduced followed by a brief overview of work in similar areas. This chapter concludes with a short introduction in few basic mathematical concepts mostly concerning spherical calculus.

Chapter 3 explains the algorithm, for calculating the isolation for every mountain, developed for this thesis. In this chapter first a top level overview of the algorithm is presented, followed by a deeper explanation of the different data structures and phases.

The performance and results of the developed algorithm is then evaluated in Chapter 4.

2 Background

2.1 Digital Elevation Models

Digital Elevation Models or DEMs have become one of the most important tools to analyze the earths surface. They represent the earth in 3D by providing elevation-measurements on a grid. The grid is mostly stored as files of 1 square degree of coverage, called a tile. Each tile is uniformly divided into a square-grid of elevation measurements. A tile is represented by its smallest latitude and longitude. For example tile $(1°, -3°)$, which is the same as $(1°N, 3°W)$ C.f. Section 2.3, would cover the area between $1°$, $2°$ latitude and $-3°$, $-2°$ longitude.

The resolution of DEMs is given by the length of one sample at the equator in arcseconds [10, Chapter 2.1].

The data used in this work comes from viewfinderpanoramas.org [2]. The main source of that data set is from the Shuttle Radar Topography Mission (SRTM) which started in 2011 [9]. The SRTM data covers the earth between latitude $60°N$ and $59°S$ with a resolution of 3 arcseconds (about 90m at the equator). For the US a 1 arcsecond SRTM resolution is available as well.

The problem with the data from the SRTM project is that it does not cover the complete globe and contains rather big voids especially in mountainous regions (cf. fig. 2.1). Because of that the author of the viewfinderpanoramas data set uses different sources to complete the dataset. The Antarctica for example comes from the Radarset Antarctic Mapping Project [1] and most of the voids where filled with the help of the Advanced Spaceborne Thermal Emission and Reflection Radiometer global DEM (ASTER GDEM) [10, P 791]. With this the 3 arcsecond DEM data has a nearly complete coverage of the globe, whereas the 1 arcsecond dataset is incomplete.

The accuracy of the data mostly depends on the SRTM data's accuracy. There, the absolute vertical height error is not more than $16m$ for 90% of the data [14].

The data from the ESA TanDEM-X mission would be interesting as well. This data covers the entire globe with a 0.4 arcsecond coverage (approx. $12m$). Unfortunately this data is not easily accessible like the data from the SRTM mission, thus efforts to get access to this data failed for this project.

2.2 Related work

The first attempts of automated mountain classifications were conducted by the Research Institute US Army Topography Engineering in 1993 [6]. There the authors tried to replace the labour intensive work of manual classification of terrains in Mount, Plain, Basin and Flat with computer algorithms. The data used in this paper comes from the USGS typography mission, which used aerial photographs to create a DEM of the US [12].

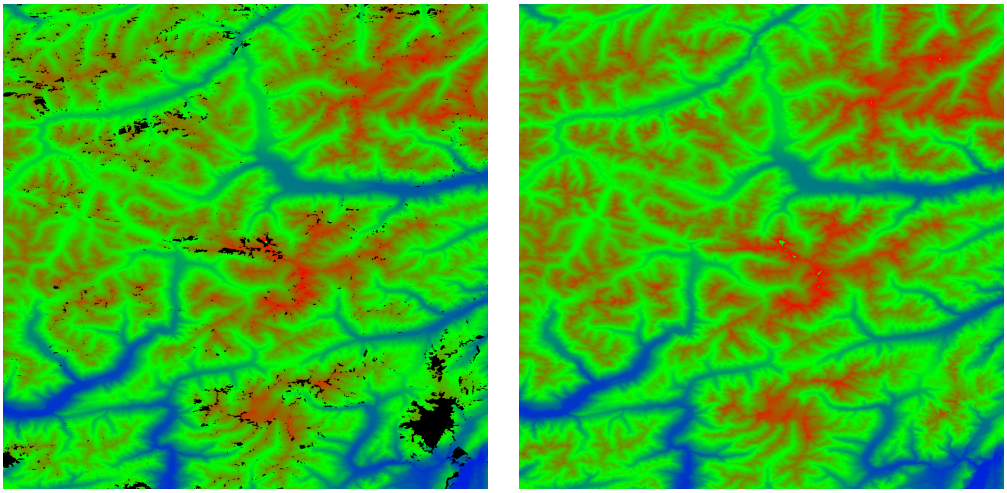

Figure 2.1: Comparison tile 46°*N* 10°*E*. Left raw data from Nasa with voids, right modified data by viewfindpanoramas.org.

In "What is a mountain?" by Peter Fish and Jo Wood [3] mountains are defined with a fuzzy concept, because the same terrain can belong to different landforms based on the scale of the DEM. With this concept the same authors created an algorithm which analyzes DEMs on different scales and classified each DEM pixel in Pass, Pit, Plane, Ridge, Channel and Peak [4]. With these attributes a multi scale fuzzy value is calculated that describes the peakness of a given DEM pixel neighbourhood.

Another more modern heuristic method to define mountains using deep learning is described in Torres at al. (2018) [16]. Here the authors train a deep neural network with the help of a gold standard data set, which is based on multiple sources, to identify mountain summits.

All of these methods try to approximate the definition of mountain by using extra free parameters like the resolution of a DEM. Another approach is to define mountains by their objective metrics, like prominence and isolation.

Using this approach Kirmse and de Ferranti [10] developed a method to find the prominence and isolation of every mountain with the help of DEM data. They used a brute force approach for calculating the isolation, where they start a search for the closest higher sample for every peak in concentric circles around that peak. To calculate the prominence they converted each DEM tile to a tree data structure where peaks and saddles are node [10, P. 794] and used this structure to find the highest saddle between two peaks, and with that the prominence of a peak.

Nahime, Milani and Fraternali (2018) [15] conducted comparison of most of these approaches where they came to the conclusion that the methods using prominence and isolation have the most accurate results for mountain identification.

2.3 Basic Mathematical Concepts

This section should give a short overview over the terms and some basic mathematical concepts used in this thesis.

Since planets are often approximated by spheres, a lot of calculations on the sphere surface take place. The geographic coordinate system is used, where points are given in latitude and longitude. The latitude is given as an angle between 0° and 90° from the equator in the north or south direction and the longitude from 0° to 180° east, west in respect to the meridian. We will also use the notation where the south and west values are given in negative degrees. So the point $(20°S, 90°W)$ can also be written as $(-20°, -90°)$.

On the sphere surface great circles correspond to strait lines in the euclidean space. Because of that there are always at least two great circle segments as strait lines between two points on the sphere surface. The smallest great circle segment between two points is called the geodesic. If the points are at polar ends of the sphere there exists an infinite amount of geodesics between them.

The following formula is used to calculate the length of the geodesic for points (λ_1, ϕ_1), (λ_2, ϕ_2) and radius R.

$$a = \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \sin^2\left(\frac{\lambda_1 - \lambda_2}{2}\right)\cos(\lambda_1)\cos(\lambda_2) \tag{2.1}$$
$$\text{distance} = R\tan^{-1}\left(\frac{\sqrt{a}}{\sqrt{(1-a)}}\right)$$

The earth and most other planets aren't perfect spheres. Because of this if a more accurate distance is necessary, the distance is calculated with formula (2.2) which approximates the planet with the help of an ellipsoid.

$$\begin{aligned} \mathrm{d}\lambda &:= (\lambda_2 - \lambda_1)/2 \quad \mathrm{d}\phi := (\phi_2 - \phi_1)/2 \\ \Lambda &= (\lambda_2 + \lambda_1)/2 \\ s &= \sin^2\mathrm{d}\phi \cdot \cos^2\mathrm{d}\lambda + \cos^2\Lambda \cdot \sin^2\mathrm{d}\lambda \\ c &= \cos^2\mathrm{d}\phi \cdot \cos^2\mathrm{d}\lambda + \sin^2\Lambda \cdot \sin^2\mathrm{d}\lambda \\ w &= \tan^{-1}(\sqrt{s}/\sqrt{c}) \\ r &= \frac{\sqrt{sc}}{w} \\ \text{distance} &= 2aw\left(1 + f\frac{3r-1}{2c}\sin^2\Lambda \cdot \cos^2\mathrm{d}\phi - f\frac{3r+1}{2c}\sin^2\Lambda \cdot \sin^2\mathrm{d}\phi \cdot \cos^2\Lambda\right) \end{aligned} \tag{2.2}$$

r and f are the equatorial radius and flattening of the World Geodetic System (WDS) [10].

3 Algorithm

The algorithm developed for this thesis builds upon the work of Andrew Kirmse and Jonathan de Ferranti [10]. They published there algorithmic approach to calculate the isolation for every mountain called “mountains” [11] on GitHub under the MIT licence. Because of this we can use basic functions and structures, like tiles, loading the tiles, calculating the length of geodesics between two points, finding potential peaks and more, and concentrate on developing the algorithm to calculate isolations quickly.

To do this we need to find the closest point with higher elevation, the Isolation Limit Point (ILP), for every peak and then calculate the distance between the peak and its ILP. We use a isolation threshold to filter out insignificant peaks. To find the ILP we divide the algorithm in two phases where each phase works on just one tile at a time.

Setup and Linking Phase During this phase the peaks are calculated, using the algorithm of Kirmse and de Ferranti. Afterwards a local ILP in the same tile as the peak is found using the sweepline approach described in Section 3.1. Such a local ILP exists for every peak that is not the highest peak in the tile because, for example, the ILP for the second highest peak would be one the slope of the highest. If we find a local ILP the distance between the peak and the local ILP serves as an upper bound, because neighbouring tiles could contain a closer one, if for example the peak is close to the edge of the tile. Because of that the peak is now linked with tiles which could contain a closer ILP. A tile could contain a closer ILP for a peak if the shortest distance to the tile is smaller than the upper bound and the maximum elevation of the tile is bigger than the elevation of the peak. If we do not find an ILP for a peak, in the tile of the peak, we first search for the closest tile that contains points with bigger elevation and use the maximum distance to this tile as the upper bound for the isolation. It is also possible that during this phase peaks at the corner of the tile are discovered which belong to the slope of a mountain peak in the neighbouring tile. This however is not a problem since during the next phase these falsely identified peaks will have an isolation below the threshold. For the linking step and in order to find the closest tile the ILP Search Area Tree described in Section 3.3 is used.

Exact calculation Phase Afterwards the second phase starts. During this phase the linked peaks for a tile are used to calculate a tile-local ILP where now it is guaranteed that such an ILP exists. To calculate the ILP the same sweepline approach described in Section 3.1 is used as in the Setup and Linking Phase. The result of this is, that one peak can now have multiple possible ILPs in different tiles. The ILP with the minimum distance to the peak is now the global ILP and the distance is the isolation of the peak.

We can optimize the Setup and Linking Phase by reducing the resolution of the DEM, because we are just interested in an upper bound. If we find an upper bound that is shorter than the isolation threshold the peak can be ignored for the rest of the algorithm. This reduces the amount of ILP-searches during the Exact calculation Phase of the Algorithm.

Both phases work on one tile at a time and the results within a phase for a tile are independent to other tiles in the same phase. This makes the phases easily parallelizable by using a Thread Pool with one thread per tile. Just the ILP Search Area Tree needs to be implemented with multi threading in mind, because multiple threads can try to link peaks to one tile at the same time, causing a potential race condition.

3.1 The Sweepline

The sweepline is designed to sweep from the highest elevation in one tile down the contour of the elevation model. If the sweepline reaches the elevation of a peak it contains all points that correspond to the contour line for its elevation. For a continuous model this would mean that the sweepline includes all points that are on the same elevation as the peak. Since DEM-Data is not a continuous model also points with slightly higher elevation could represent a part of the contour line. To illustrate this we can model the discrete DEM-Data as a Voxel, where every DEM-Pixel is a cube with the grid dimension in x and y direction and the elevation value as z. In this illustration all cubes are in the sweepline where on the elevation of the sweepline-state, one of the side faces is not covered by a neighbouring block (C.f. Figure 3.1).

The shortest distance between the contour line and the peak corresponds now to the isolation of the peak. To calculate the shortest distance first the ILP needs to be found. For this a nearest neighbour search in the data of the sweepline can be conducted. Doing this reduces the 3D-Search for the ILP, where position and elevation needs to be considered, to a 2D-Search in a reduced set of points.

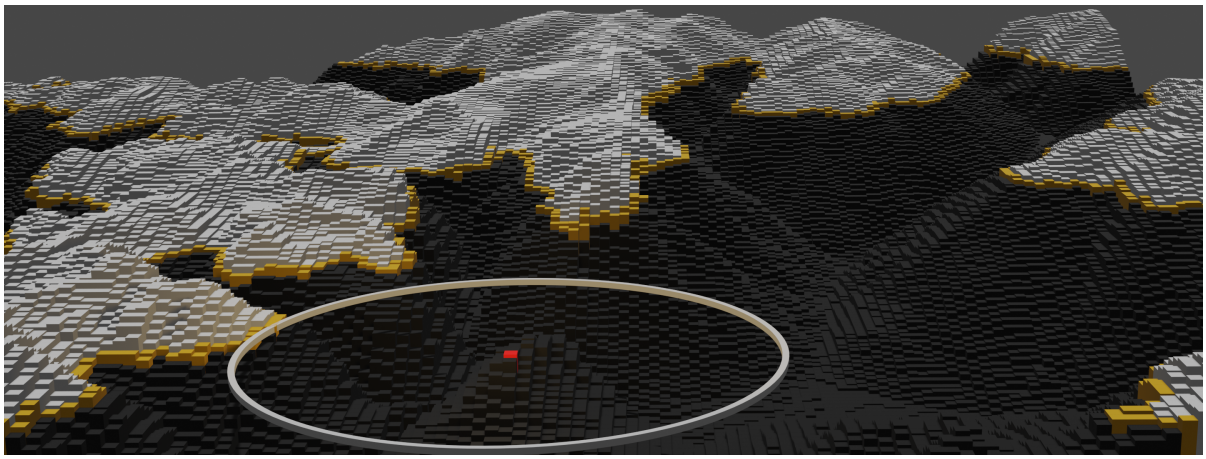

Figure 3.1: Illustration of DEM-Grid and sweepline snapshot by using a Voxel representation. Red represents current peak. Orange DEM-Pixels are contained in the sweepline. White represents already removed and black not yet added DEM-Pixels.

Algorithm 1: Top-Level algorithm to calculate the isolation of every mountain

```
Data: tiles with DEM-Pixels
1  ilpSearchTree := build ILP Search Area Tree // Section 3.3
   // Setup and Linking Phase
2  forall tile ∈ tiles do
3      peaks := tile.CalculatePeaks()
4      eventQueue := CreateSweeplineEventQueue(peaks, tile) // Algorithm 2
5      swd := kD-Tree Sweepline Data structure // Section 3.2.2
       // Run Sweepline (Section 3.1)
6      forall event ∈ eventQueue do
7          if event.type == INSERT then
8              swd.Insert(event)
9          else if event.type == REMOVE then
10             swd.Remove(event)
11         else if event.type == PEAK then
12             ilp := swd.NearestNeighbour(event)
13             ub := Distance(event, ilp)
14             ilpSearchTree.LinkPeakWithTiles(event, ub) // Algorithm 4
   // Exact calculation Phase
15 forall tile ∈ tiles do
16     peaks := ilpSearchTree.GetLinkedPeaksForTile(tile)
17     eventQueue := CreateSweeplineEventQueue(peaks, tile) // Algorithm 2
18     swd := Quad-Tree Sweepline Data structure // Section 3.2.3
       // Run Sweepline (Section 3.1)
19     forall event ∈ eventQueue do
20         if event.type == INSERT then
21             swd.Insert(event)
22         else if event.type == REMOVE then
23             swd.Remove(event)
24         else if event.type == PEAK then
25             ilp := swd.NearestNeighbour(event)
26             tileIsolations.Insert(event, ilp, Distance(event, ilp))
       // While merging keep minimum distance isolation for every peak
27     isolations.MergeIn(tileIsolations)
Result: isolations
```

To "sweep down the contour line" we insert DEM-Pixels if the sweepline reaches there elevation. If all adjacent points for a DEM-Pixel are added to the sweepline, which means that the pixel cube is covered on every side by another cube, the DEM-Pixel is out of scope and can be removed. Because of this the Sweepline always contains a one DEM-Pixel thick contour line from its elevation state (C.f. Fig 3.2).

To formalise this approach we define sweepline events. A sweepline event consists of the position of a DEM-Pixel, an elevation value and can have one of the following types:

Insert : Insert a DEM-Pixel to the data structure.

Remove : Remove a DEM-Pixel after it goes out of scope.

Peak : Calculate the nearest neighbour in the data structure to the peak DEM-Pixel.

These events can be calculated before the sweepline is executed and are stored in an event queue. To build the event queue first all Insert and Remove events and then the Peak events are added to the queue as described in Algorithm 2. The peaks are calculated in the first phase with the help of the algorithm of Kirmse and de Ferranti [10] and in the second phase the peaks are used which are linked to the tile by the ILP Search Area Tree described in Section 3.3.

Algorithm 2: Create Sweepline event Queue

Data: peaks, DEM-Pixels
1 **forall** $pix \in$ *DEM-pixels* **do**
2 $minSurrounding := \min\{elev(x) | x$ direct N,S,E,W neighboar of pix$\}$
3 **if** $minSurrounding < elev(pix)$ **then**
4 $eventQueue.insertAddEvent(pix, elev(pix))$
5 $eventQueue.insertRemoveEvent(pix, minSurrounding)$
6 **forall** $peak \in peaks$ **do**
7 $eventQueue.insertPeakEvent(peak, elev(peak))$
Result: eventQueue

In the last step the events are sorted in descending order by elevation. Adding the peak events last to the queue and using a stable sorting algorithm makes sure that every insert and remove event that is greater or equal to the peak is processed before the peak is calculated.

A DEM-Pixel gets activated when the sweepline reaches its height and deactivated if the smallest adjacent DEM-Pixel is higher than the peak. The `minSurrounding` is also checked to filter out events that would never be active.

It is sufficient to simply look at the von Neumann neighbourhood and not the complete Moor neighbourhood, because on a small scale the grid of the DEM can be approximated as rectangles, where the distance along the diagonal is always longer than along the main axis.

To run the sweepline the events from the `eventQueue` are processed one after the other. This is shown in Algorithm 1. The state of the sweepline is defined by the elevation of the last sweepline event.

Our tests in Section 4.3 show that removing events does not make sense in the second phase of the Algorithm, because in that phase there are a lot more Insert than Peak events and the cost to remove all inserted DEM-Pixel is higher than gain from a cheaper nearest neighbour search. This does not effect the result since the contour line will always be closer to peaks than points of the space between a contour line.

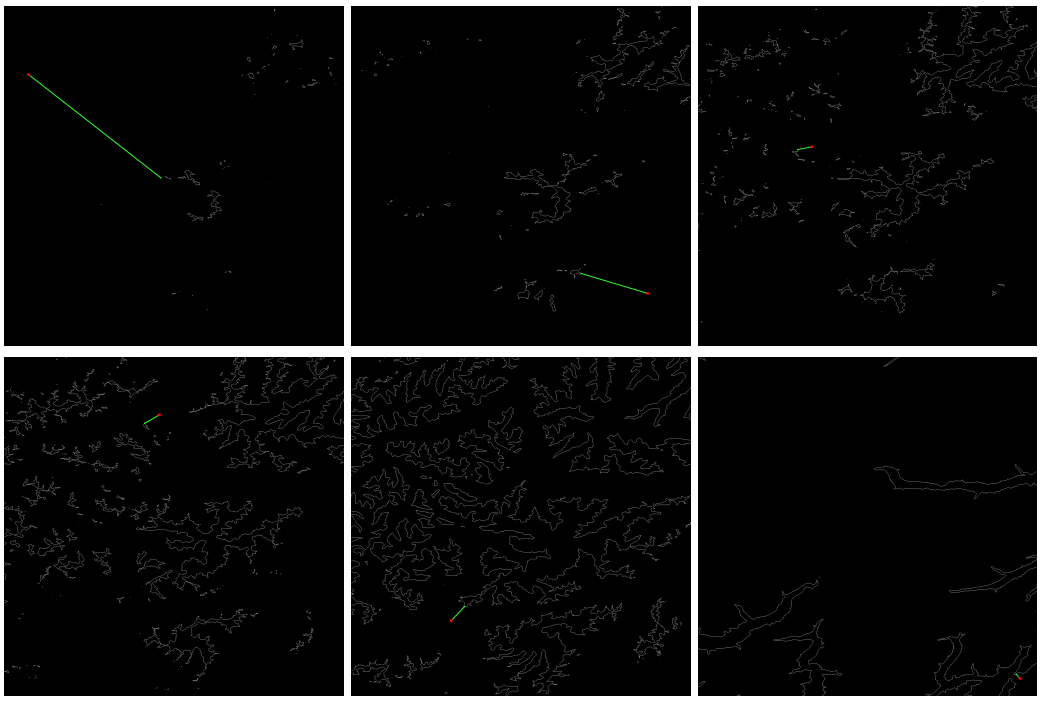

Figure 3.2: Example steps of the sweepline with peak (red point) and nearest neighbour. White pixels represent DEM-Pixels that are in the data structure of the sweepline.

3.2 Sweepline Data Structure

3.2.1 Basic concepts

The data structure which the sweepline is based on, needs to have a few key features. Most importantly it needs to support a fast nearest neighbour search operation. Because of this a space partitioning tree is the obvious choice. The data structure also needs to be dynamic, since between every peak calculation, a lot of DEM-pixels are inserted.

The first test was conducted with the KD-Tree implementation of the Computational Geometry Algorithms Library (CGAL). It was dismissed because according to the CGAL documentation [13], the implementation doesn't support insertions after the tree is built, which caused the tree to be rebuilt on every nearest-neighbour peak calculation.

Because other implementation did not satisfy the requirements, a dynamic *k*D-Tree was implemented, which had good performance on nearest neighbour queries, and a fixed-size Quad-Tree with better performance for inserts. The *k*D-Tree was used in the first phase of the algorithm. This was done because a lot of peaks need to be calculated in this step since the peak finding algorithm generates a lot of potential peaks. The Quad-Tree is used in the second phase, where the insertion operation dominates C.f. Section 4.2.

Subdivision in Quadrilaterals The concept of both data structures is to create a space-partitioning tree for a spherical surface. For that the space was divided into quadrilaterals which are aligned with latitude and longitude (C.f. Fig. 3.3). We will call them just quadrilaterals in the rest of this chapter. These quadrilaterals are defined by their north-west and south-east corners.

Each quadrilateral is then further subdivided into smaller quadrilaterals by using a center-split approach which guarantees a max tree-depth of $O(\log(N))$.

In this thesis the biggest quadrilateral is one tile, since we use the tile-bucket-tree to divide further.

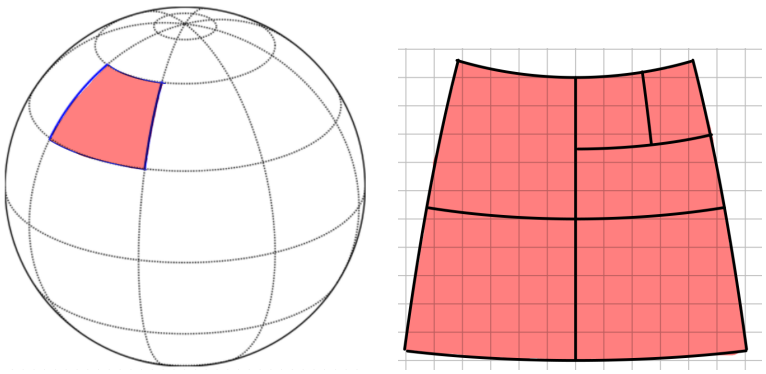

Figure 3.3: Representation of a lat-lng aligned quadrilateral.

The points are saved in a vector at the leaves of the tree. Testing if a point is inside a quadrilateral is now done by comparing the latitude and longitude with the north-east and south-west corner of the quadrilateral.

Shortest Distance between Point and Quadrilateral To conduct a nearest neighbour search we need to calculate the shortest distance of a pint to a quadrilateral. During the first phase of the algorithm the quadrilaterals are approximated as trapezoids. Because of this euclidean distance calculations can be used. For the second phase since points are further apart a more sophisticated approach is necessary.

For this an algorithm was developed to find the point s in the quadrilateral Q with the shortest distance to point p. Afterwards Equations (2.1) or (2.2) can be used to calculate the distance between s and p.

There are four different cases for the relative location between p and Q.

1. p is inside Q (red area in Fig. 3.4).
2. p is between the longitude lines of Q (green area in Fig. 3.4).
3. p is outside the longitude lines of the Q, but between the lines that are perpendicular to the longitude circles of Q and go through the corners of Q (blue area in Fig. 3.4).
4. All other options (white area in Fig. 3.4).

In case 1 the distance is defined to be zero.

In case 2 the point with the shortest distance s is on the intersection between the longitude line of p and one of the latitude-edges of the quadrilateral (Fig. 3.4 left). This is because all latitude circles are parallel to each other, and the shortest distance between two latitude circles is along the longitude lines. Because of this there is no point with a shorter geodesic on the

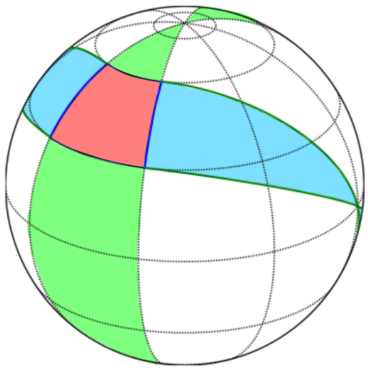

Figure 3.4: Areas for different min-distance cases between quadrilateral and point.

latitude circles of Q and the latitude circle p lies on. The shortest distance to Q is now the shorter distance between these two intersection points and p.

If none of the above was the case, the point s is on one of the longitude lines of the quadrilateral. To find out which longitude line has a shorter distance to p, the point p is rotated by the angle it would take to get the center longitude of the quadrilateral aligned with the meridian. If the longitude of p is now positive, the west longitude-edge of Q is closer to p. Otherwise, the east longitude-edge of Q is closer.[1]

Now point s with shortest distance to p on the longitude edge is calculated with linear calculus. For this point p and the edge-points defining the longitude edge l_1, l_2 are transferred into Cartesian space using the formula:

$$\begin{aligned} x &:= \cos(latitude)\cos(longitude) \\ y &:= \cos(latitude)\sin(longitude) \\ z &:= \sin(latitude) \end{aligned} \tag{3.1}$$

Then the following formula is used to find the point that is closest to the longitude circle:

$$\begin{aligned} A &:= l_1 \times l_2 \\ B &:= p \times A \\ S &:= A \times B \end{aligned} \tag{3.2}$$

Note, S is normalized, because (3.1) produces normalized vectors and (3.2) uses only vector products.

The idea behind this formula is the following: The vector product A calculates the plane of the great circle defined by l_1 and l_2. The vector product B calculates the plane of the great circle that is perpendicular to A and goes through the point p. Lastly the intersection between these two great circles is calculated ($A \times B$). That means that the geodesic between s and p

[1] This is just possible because we split the sphere at the anti meridian. Because of this the value of the most west longitude of a quadrilateral is never bigger than the most east one.

is perpendicular to the longitude circle and with that the shortest possible geodesic between them.

S is now transferred back to longitude and latitude using

$$\begin{aligned} latitude &:= \arcsin(z) \\ longitude &:= \arctan_2(x, y) \end{aligned} \tag{3.3}$$

If the latitude of s is between the top and bottom latitude of Q the point with the shortest distance is s (case 3). Otherwise one of the corners is the point with the shortest distance (case 4).

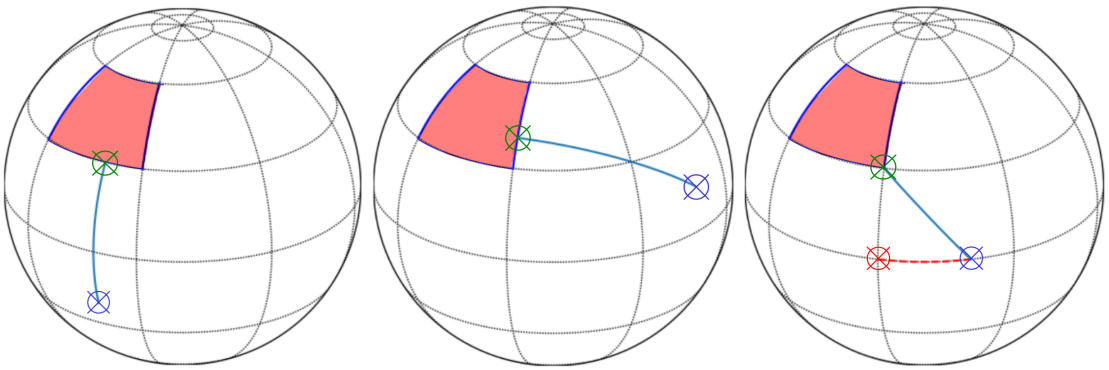

Figure 3.5: Min-distance between quadrilateral and point examples.

3.2.2 Dynamic kD-Tree

For the first phase of the algorithm a dynamic two dimensional kD-Tree was implemented. This was done because in this step a lot of nearest neighbour queries need to be calculated and having a growing tree proofed to have a runtime benefit over the static Quad-Tree described in Chapter 3.2.3 even thou the Quad-Tree has better insert performance (C.f. Section 4.2).

The kD-Tree design follows a basic concept. Each inner node has two children that represent one half of the quadrilateral of the parent. For the split policy, a center-split approach was used which splits the quadrilateral at latitude or longitude, depending on which edge-distance is longer. Each leaf node has a fixed capacity of points C it can hold.

To decrease the amount of expensive allocation and free operations a memory manager was implemented that allocates memory for nodes in chunks.

The insertion operation is done by checking on every level of the tree which child contains the point. Because of the center-split policy this has a complexity of $O(\log_2(N))$, where N is the number of insertions. If a leaf node is found with available capacity, a linear search on the array of the leaf is executed to find a free spot and the point is inserted in that spot. If the leaf reaches its maximum capacity a split operation is executed. Because of the center-split policy it is possible that this needs to be done $\log_2(N)$ times, however the sum of insertions and split operations will still be in $O(\log_2(N))$.

It is often the case that the first C points that are inserted to the data structure are very close to each other since they belong to the biggest mountain in the tile. That causes an imbalanced tree, but more importantly this causes one path down the tree to be in one chunk of the memory manager, what by itself means that nodes in the first layers of the tree, that will be called the most, could end up in different chunks. This causes bad caching behaviour and to improve this the first n layers of the tree are built during the initialisation process.

For the remove operation the tree is traversed down to the leaf which contains the point that needs to be removed. Then a linear search is again executed to find the point in the array of the leaf and the point is then removed from the leaf.

To find the closest point for peak p a textbook nearest neighbour search, described in [5], is used. This search algorithm has also a time complexity of $O(\log_2(N))$.

3.2.3 Static Quad-Tree

Our tests in Section 4.3 show that removing events in the second phase would be too cost intensive. Because of that the data structure will always need the maximum space to hold all available points. Since all data needs to be allocated anyway, a static quad tree was implemented that uses one fixed-size array to store all events. The space of the array is then grouped in leaves which correspond to one small portion or quadrilateral of the DEM tile area. To build this tree two metrics need to be available:

1. L := the maximum level of the quadtree and
2. MAX_V := the number of event points that correspond to one leaf.

These metrics depend on the number N of DEM-Pixels in a tile[2] in the following way:

$$L := \lceil \log_4(N/MAX\_V) \rceil \mid L < \log_4(N) \tag{3.4}$$

$$MAX\_V := \left\lceil \frac{N}{4^L} \right\rceil \tag{3.5}$$

(3.5) is used to calculate the maximum number of points that will be added to one leaf if the max Level is L. That means that the tree consists of 4^L leaves where each leaf can hold MAX_V event.

Algorithm 3 calculates the leaf index for a given point p of a tile. This algorithm splits for every level the bucket—and by that the quadrilateral—in 4 parts and calculates which part includes p. With this approach the index is just calculated without traversing the tree.

Analysing algorithm 3 revealed a high amount of branch miss predictions. Because of this, the if-else statements where replaced with predicated instructions. This however would make the algorithm harder to read.

To guarantee an efficient nearest neighbour search it is necessary to know which part of the tree is occupied by values and which can safely be ignored. Because of this a Boolean-array is

[2]This number is fixed per resolution, for example the 3 arc-second tile has a maximum of 1200^2 = 1.440.000 DEM-Pixels.

Algorithm 3: Find index for latitude and longitude

Data: lat, lng of p and min/max latitude and longitude for quadrilateral
level, index := 0
bucket := 4^L
while $level < L$ **do**
    midLat := (maxLat + minLat)/2
    midLng := (maxLng + minLng)/2
    if $lat \geq midLat$ **then**
        minLat := midLat
    else
        index := index + bucket/2
        maxLat := midLat
    if $lng \geq midLng$ **then**
        minLng := midLng
        index := index + bucket/4
    else
        maxLng := midLng
    bucket := $\sqrt[4]{\text{bucket}}$
    level := level + 1
Result: index

used with size $(4^L - 1)/3$ which stores for every inner node if points where inserted to any leaf inside the boundary of this node.

To insert a point to the Quad-Tree Algorithm 3 is used with the addition that on every step the corresponding node is marked that it now contains data. When the index is found the point is inserted to the leaf.

The nearest neighbour query is essentially the same as for the kD-Tree, with the addition that now one level contains four child nodes and that the boundaries are calculated similar to the method described in Algorithm 3.

3.3 ILP Search Area Tree

The ILP Search Area Tree has a few requirements:

- It needs to be able to create a relationship between peaks and tiles that could contain an ILP.
- It needs to be able to find tiles that are withing the upper bound of a peak.
- And lastly it needs to find the closest tile with bigger elevation than a peak, if no upper bound was found.

Because of this a kD-Tree, similar to the Dynamic kD-Tree from Section 3.2.2, with center-split was implemented. This tree splits the space in longitude and latitude and has one leaf for

every tile. Since a tile has always the size of one latitude and one longitude and there are 360 longitudes and 180 latitudes, the maximum depth of this tree is $\log_2(390 \cdot 180)$.

With that the ILP Search Area Tree divides the complete search area in quadrilaterals on a tile level where the Sweepline Data Structure divides a tile in quadrilaterals on a DEM-Pixel level. So one node on the ILP Search Area Tree represents a quadrilateral on the spherical search space, where the root node would represent the whole globe.

Every inner node in this tree contains the maximum elevation of the area it represents. Because of this while building the tree the maximum elevation for every tile needs to be available.

A link of a peak to a tile is represented by a vector inside the leaf, which contains the position of the peak. To ensure multi-threaded inserts, the `concurrent_vector` from the Thread Building Blocks (TBB) library is used.

The recursive Algorithm 4 is used to distribute peaks that have been identified in phase one to all leaves that could contain the ILP. If the upper bound (ub) does not exist, because the peak is the highest in its tile, the peak is marked with an ub of -1. With the help of Algorithm 4 the upper bound for such cases is calculated by using the maximum distance to the closest tile with higher elevation than p.

Algorithm 4: Link peaks to tiles that could contain an ILP

```
// ub is upper bound for peak p
Data: p, ub
Function LinkPeakWithTiles(p):
    if thisNode.maxElev < elev(p) then
        return
    if thisNode.isLeaf then
        links.PushBack(p)
        if ub < 0 then
            ub := max distance to this tile
        return
    // c1, c2 are childs of the current node
    if min distance to c1 < min distance to c2 then
        if ub < 0 or min distance to c1 < ub then
            c1.LinkPeakWithTiles(p)
        if ub < 0 or min distance to c2 < ub then
            c2.LinkPeakWithTiles(p)
    else
        if ub < 0 or min distance to c2 < ub then
            c2.LinkPeakWithTiles(p)
        if ub < 0 or min distance to c1 < ub then
            c1.LinkPeakWithTiles(p)
```

For the minimum distance calculations between a point and an inner node, the approach described in Section 3.2.1, is used. The maximum distance between a point and a tile is defined

as the minimum distance plus the length of the diagonal of the tile. Because this distance is an upper bound, it is not problematic if it is a bit bigger than the real maximum distance.

3.4 Analysis

To calculate the isolation for every mountain the space is divided into tiles of one latitude times one longitude. An ILP Search Area Tree is created to link peaks to tiles which could hold a nearest higher ground. Building the ILP Search Area Tree is in $O(T \cdot \log(T))$ where T is the amount of tiles. Because T is relatively small and has a maximum of $360 \cdot 180 = 64800$ this time is negligible.

Afterwards for every tile a sweepline run is conducted with a reduced set of DEM-Pixels and quick distance approximations. Peaks which are discovered during this run are then linked to tiles that could contain a closest higher ground with the help of the ILP Search Area Tree. This would be in $O(2n \cdot \log(2n) + P \cdot \log(T))$, where n is the reduced number of DEM-Pixels and P is the number of peaks.

Now the accurate sweepline is executed for every tile using the linked peaks as peak events. The time complexity for this step is in $O(N \cdot \log(N) + T \cdot \log(N))$.

Lastly the results of the tiles are merged using a maximum spanning tree for the position of the peaks, and the smallest found isolation for one peak is saved. This step has an approximate time complexity of $O(P \log(P))$, for the insertions in the maximum spanning tree. The time for this step is however again negligible in comparison to the sweepline steps, since the amount of peaks P is relatively small.

With that the time complexity of the algorithm would be in $O(2n \cdot \log(2n) + P \cdot \log(T)) + O(N \cdot \log(N) + T \cdot \log(N)) \approx O(N \cdot \log(N))$.

4 Evaluation

For the following evaluation the solution described in the paper by Kirmse and de Ferranti [10] serves as a reference algorithm. This was the only other available algorithm to calculate the isolation of every mountain. The algorithm and the reference algorithm were implemented in C++ and compiled using g++9.4.0 with the -03 flag.

The benchmarks were conducted on a machine running Ubuntu 20.04 with the following specifications:

- AMD EPYC Rome 7702P - 64-core + HT, 2.0-3.35GHz
- 1024GB DDR4 ECC, 2966MHz
- L3: 256MB
- 2TB NVMe Daten-SSD, Intel P4510
- With Hyperthreading enabled

As test data the SRTM-DEM3 data set from viewfinderpanoramas [2] was used. For multi threading testing and to compare the data structures for the sweepline we also used a reduced set of this data. The reduced data set includes all tiles of the quadrilateral between $(27N, 56E)$ and $(91N, 120E)$ which corresponds to 3 126 tiles which is roughly 10% of the total test data set. Since 1 arc-second DEM data was only available for the US, this section was used to compare performance on higher resolution DEMs.

Every time-critical test was conducted multiple times and the mean of the results was taken. For all tests we used an isolation threshold of one km.

4.1 Execution time

For the basic execution time displayed in Figure 4.1 left, the test data set was split in quadrilaterals that include an approximately exponential growing number of tiles. These quadrilaterals are specified in table 4.1.

The new approach was faster than the previous solution by Kirmse and de Ferranti. However the approach by Kirmse and de Ferranti had surprisingly good execution times, especially with a lower number of tiles, considering its brute force approach. This is caused by the fact that most peaks have a relatively short isolation. In fact 99.996% of peaks have an isolation of below 50km and more than 99% below 10km. Since one tile covers on the earth on average an area of 70km $\times$ 111km, the nearest higher point is most of the time in the same tile as the peak, where fast approximations for the distance calculation can be used.

Min latitude	Min longitude	Tile count
75°	151°	4
74°	149°	9
72°	145°	21
71°	145°	35
70°	144°	62
68°	142°	151
66°	140°	249
61°	132°	550
53°	118°	1.025
42°	98°	2.027
26°	67°	4.094
0°	17°	8.245
−55°	−91°	16.387
−90°	−180°	26.095

Table 4.1: Min latitude and longitude used for execution time testing. Max latitude and longitude were fixed at 90°, 180°

The "search for every peak separately in concentric circles" approach of the previous solution causes that for peaks with higher isolation a lot of neighbouring tiles need to be loaded. Because of this Kirmse and de Ferranti added a tile cache to reduce the number of tile-loads from the hard drive. We increased the capacity of that cache to the maximum number of tiles, since the testing machine had enough internal memory to store every tile.

The results of the execution time testings for the test data is displayed in Figure 4.1. A notable jump in the execution time of the previous solution is between the 2^{13} and 2^{14} tile marks. Here the Atlantic ocean and a big part of the southern hemisphere is added to the test data, which includes a lot of smaller islands with high isolation due to their remoteness. Because of this the previous solution needs to perform a lot more expensive distance calculations, since many ILPs are now in different tiles than the peak. This causes the execution-time for the previous solution to explode where our implementation is still growing as expected Cf. Fig. 4.1 right.

To test how the new approach performs on higher resolution DEMs the DEM data of the USA was used in 3 and 1 arch-second resolution. The results in Figure 4.1 show that for higher resolution the new approach performed better from the beginning, where the old solution was already rather strong.

The multi-threading execution time was tested with the reduced test data set. In this case the old implementation had some issues with the caching implementation. Analysing the code revealed that the lock used in the tile cache caused some problems for multi-threaded executions (C.f. fig 4.1 right).

Our algorithm performed well and reached a speedup of 22 for 64 physical threads. The Hyperthreading step to 128 did not improve the execution time. To confirm the values the speedup for 64 threads for the complete data set was also tested and showed as well a value of 22 which correlates in this case to about 4min of execution time.

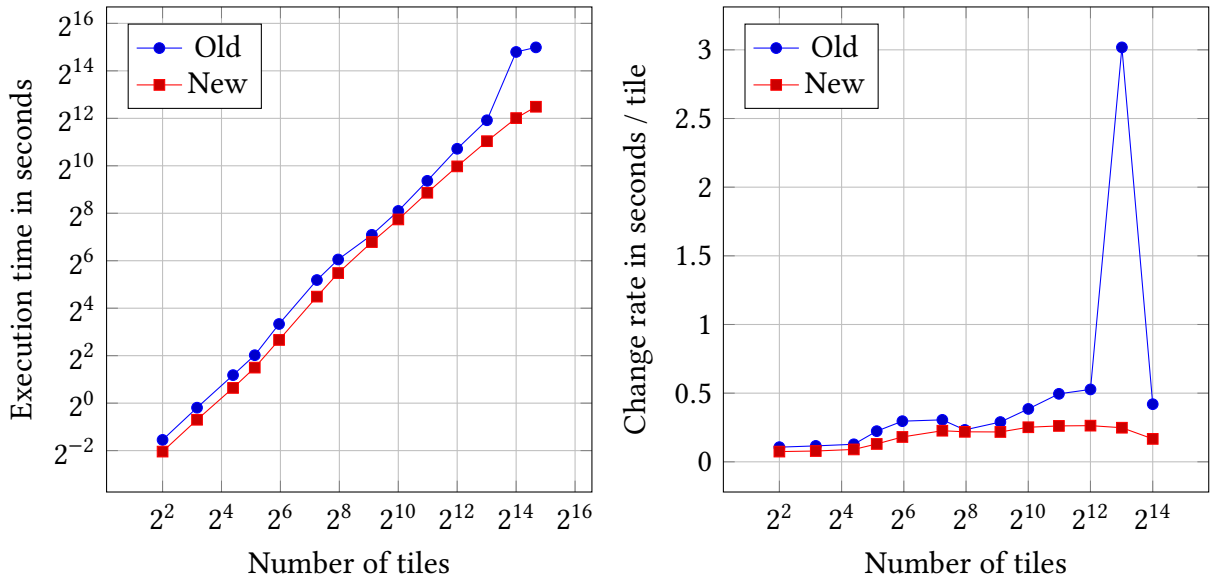

Figure 4.1: Left: Execution time single threaded. Right: Rate of change between test steps.

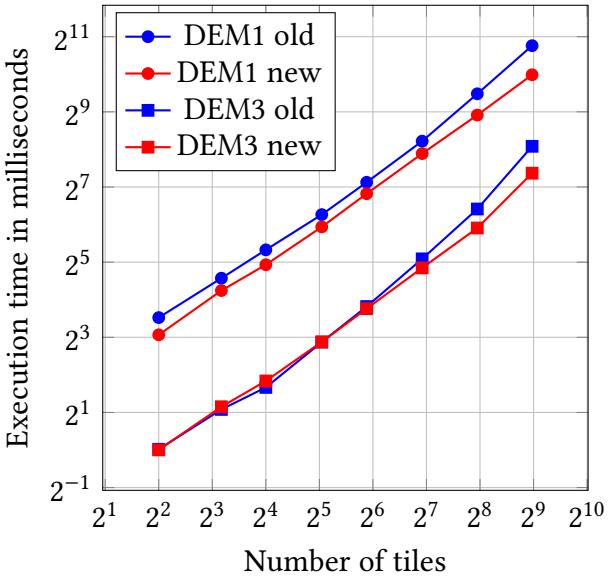

Figure 4.2: Execution time of the US on 3 arc-second (DEM3) and 1 arc-second (DEM1) resolution.

4.2 Comparison of implemented Sweepline Data Structures

In this section a short comparison between the KD-Tree described in Section 3.2.2 and the Quad-Tree described in Section 3.2.3 is conducted. Both of these space partitioning trees perform the same goal and use exactly the same interface.

We used the reduced test data set to test the execution time of the insert and nearest neighbour operation.

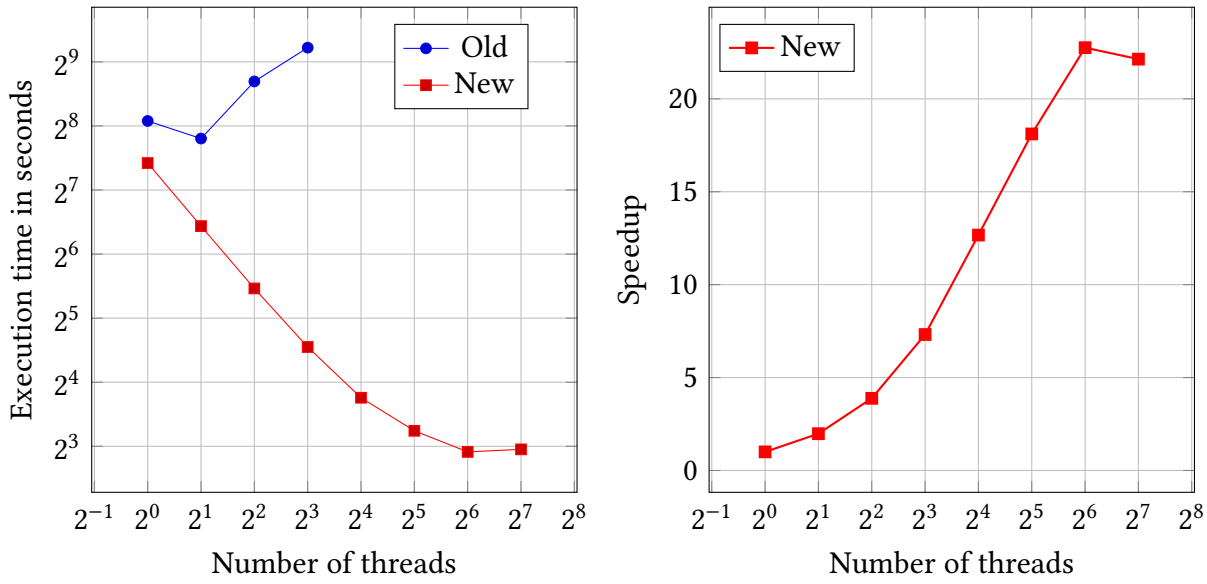

Figure 4.3: Miltithreded execution time and speedup.

First we measured the time for one insert and nearest neighbour operation in both trees and took a global average. The results are displayed in Table 4.2 and show that the *k*D-Tree performs better for the nearest neighbour query and the Quad-Tree for insertions. Table 4.2 also shows, that the cost for one nearest neighbour search is a lot higher than one insert.

Data Structure	Insert	Nearest Neighbour
*k*D-Tree	141ns	5 458ns
Quad-Tree	79ns	6 687ns

Table 4.2: Average insert and nearest neighbour operation times for *k*D- and Quad-Tree

Table 4.3 displays the number of Insert and Peak events, which causes thees operations, during the two phases. In phase one a lot more Peak events need to be processed and in phase two a lot more Insert events.

Algorithm Phase	Insert Events	Peak Events
one	134 699	43 479
two	1178 665	2 451

Table 4.3: Average number of insert and peak events, which cause a nearest neighbour search, in the different phases of the algorithm.

Using this data we can now calculate the theoretical execution time for the first and second sweepline run. The dynamic *k*D-Tree would take about 256.3ms in phase one and 179.5 ms for in phase two of the algorithm and the static Quad-Tree about 301.4 ms for phase one and 109.5

ms for phase two. These numbers correlate also with the tests of the average total execution time for one tile displayed in Table 4.4.

Algorithm Phase	*k*D-Tree	Quad-Tree
one	190ms	252ms
two	166ms	73ms

Table 4.4: Total operation time for one tile per algorithm phase and data structure.

We also conducted these tests on a set of 1 arc-second resolution data, where the difference was even more noticeable (Table 4.5).

Algorithm Phase	*k*D-Tree	Quad-Tree
one	1145ms	2405ms
two	1522ms	1194ms

Table 4.5: Total operation time per data structure on the different phases using DEM1 data from the USA.

4.3 Removing Events after they go out of scope

In this section the question if events should be removed is tested for the different requirements in phase one and two of the algorithm.

To be able to execute these tests for the Quad-Tree we replaced the Boolean array, which indicates if the branch below an inner node contains data, with a counter array. The counter of a node is now increased for every insertion and decreased on every removal from the branch below that node. If the counter of a node during the nearest neighbour search is zero, the branch below that node can be ignored, which should improve the nearest neighbour operation.

To test if the removal of events is beneficial, the total time of the operations for one tile was taken once with and without removal. This was done for every tile in the reduced test data set and the average time for these operations was than calculated. Because Section 4.2 showed that both data structures perform well on one of the phases, just the well performing phase for one data structure was tested.

The test results for phase one are displayed in Table 4.6. The results show that removing events is beneficial, because the time for nearest neighbour searches dominates during this phase. This can be explained by the relatively high cost for one nearest neighbour search (C.f. Table 4.2) and the high amount of peak events during this phase (C.f. Table 4.3), which cause a nearest neighbour search.

In phase two of the algorithm a lot more insert events need to be processed, which causes also a lot more remove events and because most peaks are already sorted out during the first phase, there are not many peak events during this phase. This also explains the results displayed in Table 4.7 where the cost to remove events is a lot higher than the benefit.

Remove	Insert	Nearest Neighbour	Total
19.12ms	12.35ms	173.67ms	205.14ms
-	18.43ms	195.44ms	213.87ms

Table 4.6: *k*D-Tree times of operations with and without removing of events with data from phase one.

Remove	Insert	Nearest Neighbour	Total
67.90ms	75.82ms	12.75ms	128ms
-	75.47ms	13.01ms	88.48ms

Table 4.7: Times of operations with and without removing of events with data from phase two.

This is also the case using a the more dynamic *k*D-Tree for the second phase as table 4.8 displays.

Remove	Insert	Nearest Neighbour	Total
182.95ms	119.39ms	11.03ms	313.37ms
-	181.26ms	11.55ms	192.81ms

Table 4.8: Times of operations with and without removing of events with data from phase two using the *k*D-Tree.

4.4 Evaluating the results

4.4.1 Comparing results from one tile

To compare the results of the new Algorithm, first a accuracy test on one tile was conducted. For this the isolation of tile $46°N, 10°E$ was calculated with the old approach, implemented by Kirmse and de Ferranti [11] and the new Sweepline approach. The old approach found 604 mountains with higher isolation than one km and the new one 551. Finding less peaks with isolation bigger than one km is, in this case, the better solution. This means that for peaks with isolation bigger than one in the old approach, the new found an ILP with isolation of less than one km.

Analysing the results revealed that for 481 peaks both algorithms found the exact same isolation, for 36 peaks the new algorithm found a closer isolation, 86 peaks where not identified by the new algorithm and one peak was not identified by the old algorithm.

The peaks where the new approach found a smaller isolation where tested against a linear search approach which revealed, that the new isolation's are exactly the closest ones. The differences are caused by the fact that the old solution uses the quick approximation, where the tile is represented as an planar trapezoid, to find the closest point and than calculates the

distance to the final point with the more accurate distance formula. The new approach uses the more accurate distance formula already during the search operation.

For the point, that was identified by the new and not the old algorithm, the ILP was on the the part of the tile which overlapped with the neighbouring one. This part is ignored in the new approach, because during the second phase of the algorithm the ILP would been found in the next tile. Because of this the peak has an isolation bigger than one km in the new approach and below one km in the old one.

4.4.2 Global comparison

The calculation of every mountain took about four minutes on the testing machine using 128 threads.

A lot of mountains were registered multiple times, if multiple peaks were found with similar elevation and slightly different positions. This is caused by the fact that the elevation of a registered peak is slightly increased during the peak finding process, to avoid that on flat peaks neighbours with the same height are registered as nearest neighbours. To filter out these cases, the sorted list was traversed and if neighbouring solutions were to close to each other, the solution with the higher isolation was removed. This works especially for big isolation well, because there most of the peaks have a unique isolation value. For smaller isolations a few duplication's will still be in the data, because a lot of peaks with the same isolation are found which breaks the neighbouring guarantee in the sorted list.

After this, a list of about 12 million peaks with isolations remained. This is halfe the amount of peaks as the authors Kirmse and de Ferranti found with their algorithm. This is caused by the fact that the old approach uses the inaccurate distance calculations to find the ILP, which causes that for a lot of peaks a slightly higher isolation is found and by that a lot more peaks have isolation higher than the threshold of one km. This was already demonstrated in Section 4.4.1.

The authors Kirmse and de Ferranti [10] published a list of isolations for every mountain. The results in that list suggest that a slightly different set of data was used, since the position of a lot of peaks were slightly of and a few isolations were found at areas where our data showed a too small elevation for this to be possible. Because of that we tried to execute the mountains algorithm [11] on our test data, however were not able to create meaningfull results. This suggests that a lot of post processing of the results was done to create the published list.

Due to this the list generated by Kirmse and de Ferranti [11] was used to assess the results.

To get a picture of the accuracy of our approach the 82 mountains with a bigger isolation than 1000km were compared. The results with the biggest differences are displayed in Table 4.9.

After analyzing the ILP for the peaks from Table 4.9 we found that all of the differences come within the data source. The two isolation limit points for the mountains in Alaska from the old source can't be isolation limits points in our source, since the maximum elevation of that area is too low. The same is the case for Sand Island, where the ILP of the old source is in the pacific ocean near Hawaii and the new ILP is on the necker islands (as well near Hawaii). For the other cases we found slightly closer ILPs which could also be caused by the different DEM data or the fact that less heuristic approaches where used.

Rank	Mountain	Old isolation	New Isolation	New Rank
14	Kljutschewskaja Sopka	2750	2746	14
18	Kinabalu	2510	2505	18
22	Silisili	2245	2502	19
-	Antarctica 83.9°S 168.38°E	54	1599	43
63	Sand Island	1217	1367	51
52	Mt Washignton	1319	1290	56
55	Putorana State Natural Reserve	1300	1284	58
68	Tahat	1162	1153	67
-	Antarctica 69.78°S 69.79°W	294	1032	79

Table 4.9: Biggest differences in isolation for mountains with more than 1000km of isolation.

We were not able to find a closer ILP for any peak in our dataset and with that we are confident that for a given data set our approach always finds the closest ILP.

5 Conclusion

In this work we introduced an algorithm to find the isolation for every mountain with the help of Digital Elevation Models. The algorithm calculates first an upper bound for every peak within one tile and links this peeks, with the help of the upper bound, to tiles that could contain an Isolation Limit Point (ILP). In the second phase the ILP for all tiles and all linked peaks is calculated and the results are merged by only keeping the ILP which results in the smallest isolation for one peak. To find the ILP in both phases a sweepline approach is used. The sweepline sweeps from the highest elevation to the lowest in one tile and contains just DEM-Pixels which represent the contour line for the elevation state of the sweepline. If the sweepline reaches the elevation of a peak, a nearest neighbour search in the data contained in the sweepline is conducted.

We were not able to find space partitioning trees that was dynamic enough or designed to run on the surface of a sphere and because of that two approaches for both of the phases of the algorithm were developed. A special challenge there was the development of an approach to calculate the shortest distance between a point and a quadrilateral on the 2D surface of a sphere.

At the end, we were able to calculate results that are closer to the reference data than the old solution in a shorter amount of time.

Bibliography

[1] Alaska Satellite Facility. *Radarset Antarctic Mapping*. https://asf.alaska.edu/data-sets/derived-data-sets/radarsat-antarctic-mapping-project-ramp/. 2008.

[2] Jonathan de Ferranti. *Digital elevation data*. http://viewfinderpanoramas.org/dem3.html. Online; accessed 05-Juli-2022. 2011.

[3] Peter Fisher and Jo Wood. "What is a mountain? Or the Englishman who went up a Boolean geographical concept but realised it was fuzzy". In: *Geography* (1998), pp. 247–256.

[4] Peter Fisher, Jo Wood, and Tao Cheng. "Where is Helvellyn? Fuzziness of multi-scale landscape morphometry". In: *Transactions of the Institute of British Geographers* 29.1 (2004), pp. 106–128.

[5] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. "An algorithm for finding best matches in logarithmic expected time". In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pp. 209–226.

[6] Graff et al. "Automated classification of generic terrain features in digital elevation models". In: *Photogrammetric Engineering and Remote Sensing* 59.9 (1993), pp. 1409–1417.

[7] Peter Grimm. *Die Gebirgsgruppen der Alpen: Ansichten, Systematiken und Methoden zur Einteilung der Alpen*. Deutscher Alpenverein, 2004.

[8] Klaus Gwinner et al. "Topography of Mars from global mapping by HRSC high-resolution digital terrain models and orthoimages: Characteristics and performance". In: *Earth and Planetary Science Letters* 294.3-4 (2010), pp. 506–519.

[9] Heather Hanson. *Shuttle Radar Topography Mission (SRTM)*. Tech. rep. Online; access 21-september-2022. NASA, 2019.

[10] Andrew Kirmse and Jonathan de Ferranti. "Calculating the prominence and isolation of every mountain in the world". In: *Progress in Physical Geography* 41.6 (2017), pp. 788–802.

[11] Andrew Kirmse and Jonathan de Ferranti. *montains on GitHub*. https://github.com/akirmse/mountains. Online; access 02-may-2022. 2017.

[12] Larry Moore. "The US Geological Survey's Revision Program for 7.5-Minute Topographic Maps". In: *Digital Mapping Techniques' 00—Workshop Proceedings, US Geological Survey Open-File Report 00-325*. 2000, pp. 21–26.

[13] The CGAL Project. *CGAL 5.5.1 - dD Spatial Searching documentation*. https://doc.cgal.org/latest/Spatial_searching/classCGAL_1_1Kd__tree.html. Online; access 28-may-2022. 1995.

[14] Ernesto Rodriguez et al. "An Assessment of the SRTM Topographic Products". In: (2005). Online; access 21-september-2022.

[15] Rocio Nahime Torres, Federico Milani, and Piero Fraternali. "Algorithms for mountain peaks discovery: a comparison". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 667–674.

[16] Rocio Nahime Torres et al. "A deep learning model for identifying mountain summits in digital elevation model data". In: *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE. 2018, pp. 212–217.