CREATING THE MEDIAL AXIS TRANSFORM FOR BILLIONS OF LIDAR POINTS USING A MEMORY EFFICIENT METHOD

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Geomatics

by

Marco Lam

January 2016

Marco Lam: *Creating the Medial Axis Transform for billions of LiDAR points using a memory efficient method* (2016)

The work in this thesis was made in the:

3D geoinformation group
Department of Urbanism
Faculty of Architecture & the Built Environment
Delft University of Technology

Supervisors:	Prof.dr. J.E. Stoter
	Ir. R. Peters Msc.
Co-reader:	Ir. P. Nourian

ABSTRACT

Using Light Detection And Radar (LiDAR) large parts of the earth's geography can be captured an represented as a 3D pointcloud. The whole elevation dataset of the Netherlands (AHN2) is currently available and captured using this technique, it contains around 640 billion points. These massive dataset in its current form is well suited for visualisation and certain forms of analysis, as the 3D points are the outer boundary of objects. However, the Medial Axis Transform (MAT) is another way to represent these objects. As it represents the inner/outer skeleton of the objects some features become more easily detectable and it could be used as a tool in point cloud analysis.

The MAT can be created from pointclouds using various methods, however the shrinking ball algorithm is used as it is relative simple to implement, more storage efficient and easier to parallellize compared to other methods. Yet the computation of it for a massive dataset such as the AHN2/3 is troublesome as it does not fit inside the main memory of the computer.

This thesis focusses on how to scale the MAT so it can be computed for massive datasets using a main memory efficient approach. Two methods (i.e. tiling and streaming based algorithms) are proposed. They both subdivide the pointcloud dataset in to manageable subsets, so that the MAT can be computed on these smaller sets. However, the tiling approach relies heavily on temporal storage on the external memory (harddisk) by creating smaller tiles. whilst the streaming approach tries to manage it within the Main memory by scanning the input dataset multiple times and storing tiles in the main memory.

This thesis concludes that using both methods it is possible to compute the MAT of a dataset which is larger than the main memory. The tiling approach seems suited as the temporal storage of the external memory is about the same size as the output data, further more the main memory usage can be regulated easily as the amount of tiles which will be processed at the same time can be chosen. The streaming approach shows potential to be efficient as well in computing the MAT. However, because streaming computations loads the input data sequentially and processes it using a limited memory buffer, outputting data and freeing memory space is needed. In this thesis a first step in finding a way to achieve that is made, however it is not functioning that well. As such the data outputting can not be as rapid as it should be when using streaming algorithms.

ACKNOWLEDGEMENTS

This report is the result of a master thesis on Geomatics at the Delft University of Technology. For this result, I want to thank super visors Professor dr. Jantien Stoter and Ir. Ravi Peters for their input, knowledge and guidance.

Furthermore I want to thank my co-reader Ir. Pirouz Nourian, whose detailed comments helped me improve my report.

Additionally, I want to thank my Friends, in particular Fleur, Rene and Hans who helped me continue working during the evenings and weekends.

And most importantly, I want thank my family and Roxanne, who have always supported and encouraged me with everything I do.

CONTENTS

1 INTRODUCTION 1
- 1.1 Datasets . . . 2
- 1.2 Research objectives . . . 2
- 1.3 Workflow . . . 3
- 1.4 Scope . . . 3
- 1.5 Outline . . . 3

2 BACKGROUND INFORMATION AND RELATED WORK 5
- 2.1 Spatial data structures . . . 5
- 2.2 Medial Axis Transform . . . 9
- 2.3 Applications of MAT . . . 11
- 2.4 computation of the MAT . . . 15
- 2.5 Memory Hierarchy . . . 20
- 2.6 Strategies for scaling GIS algorithms . . . 20

3 COMPUTING BUFFERS FOR THE MEDIAL AXIS TRANSFORM 27
- 3.1 Time and Memory complexity . . . 27
- 3.2 Challenges of processing smaller datasets . . . 29
- 3.3 Regular Buffer . . . 31
- 3.4 Reduced buffer . . . 33
- 3.5 Thinned reduced buffer . . . 34
- 3.6 Summary . . . 44

4 SCALING THE MEDIAL AXIS TRANSFORM 45
- 4.1 Approach: Tiling algorithms . . . 47
- 4.2 Approach: Streaming algorithm . . . 60
- 4.3 Merging the output . . . 68
- 4.4 Differences between approaches . . . 69
- 4.5 summary . . . 70

5 IMPLEMENTATION, EXPERIMENTS AND COMPARISON 73
- 5.1 Implementation . . . 73
- 5.2 Datasets . . . 76
- 5.3 data quality . . . 76
- 5.4 External memory usage . . . 78
- 5.5 Internal memory usage . . . 79
- 5.6 Computation time . . . 81
- 5.7 Discussion . . . 84
- 5.8 Summary . . . 85

6 CONCLUSION, DISCUSSION AND FUTURE WORK 87
- 6.1 conclusions . . . 87
- 6.2 discussion . . . 89
- 6.3 Future Work . . . 90

A THINNED REDUCED BUFFER RESULTS 95

B EXTRA IMPLEMENTATION RESULTS 99

C REFLECTION 101

LIST OF FIGURES

Figure 1.1 The medial axis visualized for a building 1
Figure 1.2 Introduction: workflow 3
Figure 2.1 Region quadtree 6
Figure 2.2 Optimized Point Quadtree 6
Figure 2.3 Unbalanced and balanced kd-tree 7
Figure 2.4 Nearest Neighbour Search KD-tree 8
Figure 2.5 Subdivision of the dataset for the kd-tree and optimized kd-tree 8
Figure 2.6 Morton code 9
Figure 2.7 Creating medial axis 10
Figure 2.8 inner and outer MAT 10
Figure 2.9 MAT of a rectangle 11
Figure 2.10 Domain Decomposition Lemma 12
Figure 2.11 Powercrust 12
Figure 2.12 Surface reconstructed from strongly simplified MA . 13
Figure 2.13 Simplification using LFS 14
Figure 2.14 Regular point representation vs splat representation . 14
Figure 2.15 Visibility analysis using the medial balls 15
Figure 2.16 Radius calculation 15
Figure 2.17 MAT computation using voronoi 16
Figure 2.18 Explanation of the algorithm of Ma 17
Figure 2.19 Radius calculation 17
Figure 2.20 Noise in a dataset 19
Figure 2.21 Sequence of Shrinking ball algorithm with noise . . . 19
Figure 2.22 Minkowski Sum 22
Figure 2.23 D&C used to compute the voronoi diagram 23
Figure 2.24 Spatial coherence 24
Figure 2.25 Workflow spatial finalizer 24
Figure 2.26 Streaming Delaunay Triangulation 25
Figure 3.1 Overview methodology 27
Figure 3.2 Pointcloud dataset split up in 2 subsets order 29
Figure 3.3 Border issues when computing MAT 30
Figure 3.4 Unfinished points 30
Figure 3.5 Point needing data from outside the tile 32
Figure 3.6 Buffers on 2 subsets 32
Figure 3.7 Determining which MAT in the buffer region is final 33
Figure 3.8 Reduced buffer 34
Figure 3.9 The medial ball is tangent to more than one boundary point 35
Figure 3.10 finished or unfinished 36
Figure 3.11 Creating test sets 38
Figure 3.12 Analysis area 38
Figure 3.13 Invalid inner MAT of a point 39
Figure 3.14 Rotterdam Dataset with reduced buffer 40
Figure 3.15 Histogram: Rotterdam Dataset with reduced buffer . 40
Figure 3.16 Thin objects errors 41
Figure 3.17 Normal calculation of planar plane 41
Figure 3.18 Outliers in the air 42

Figure 3.19 Noise in the dataset . . . 42
Figure 3.20 Outer medial axis deviations . . . 43
Figure 3.21 Inner medial axis deviations . . . 43
Figure 4.1 Subdividing a Geographic Pointcloud using 2D grid 45
Figure 4.2 MAT computation for a single point with buffers . . 46
Figure 4.3 Segmentation of a dataset . . . 47
Figure 4.4 Buffers around the tiles . . . 48
Figure 4.5 Workflow tiling process . . . 48
Figure 4.6 Dataset subdivided in 200m x 200m tiles . . . 49
Figure 4.7 Buffer regions . . . 50
Figure 4.8 Buffer regions . . . 51
Figure 4.9 Dataset split in 64 tiles . . . 52
Figure 4.10 Dataset is split up in to 2 collections . . . 54
Figure 4.11 Dataset is split up in to 4 collections . . . 55
Figure 4.12 Subdivision using kd-tree . . . 56
Figure 4.13 Skinny vs square collections . . . 57
Figure 4.14 Reduced buffer . . . 57
Figure 4.15 4x4 collection of tiles . . . 58
Figure 4.16 Spatial Finalizer Workflow . . . 60
Figure 4.17 Neighbouring cells . . . 61
Figure 4.18 Illustration of the problem of using spatial finalizer with buffers . . . 62
Figure 4.19 Waiting times for tiles . . . 62
Figure 4.20 Waiting tile for tiles . . . 63
Figure 4.21 Maximum NN scan area . . . 64
Figure 4.22 Maximum NN scan area . . . 65
Figure 4.23 Merging order . . . 68
Figure 5.1 Workflow scheme for streaming and tiling approach 74
Figure 5.2 Location datasets . . . 77
Figure 5.3 Difference in output of streaming and tiling . . . 77
Figure 5.4 AHN3 datasets: acsmat main memory usage . . . 81
Figure 5.5 AHN3 datasets: acsmat Computation Time . . . 83
Figure 5.6 AHN3 c_67hz1: Process Times . . . 83
Figure 5.7 Computation time comparison (Regular buffers/Reduced buffers) . . . 85
Figure A.1 maximum NN scan area . . . 95
Figure A.2 maximum NN scan area . . . 95
Figure A.3 maximum NN scan area . . . 96
Figure A.4 maximum NN scan area . . . 97
Figure A.5 maximum NN scan area . . . 98

LIST OF TABLES

Table 3.1 MAT Datastructure 28
Table 3.2 computation memory usage for either the inner or outer MAT!, separated in amount of points 28
Table 3.3 Unfinished points Delfgauw 31
Table 3.4 Unfinished points Woerden 31
Table 3.5 Errors in MAT calculation with preprocessing 39
Table 4.1 normalizing coordinates 49
Table 4.2 MAT computation memory usage 58
Table 4.3 MAT computation memory usage (Streaming approach) 68
Table 4.4 Main differences between the tiling and streaming method 69
Table 5.1 Amount of collections per segmentation method . . . 75
Table 5.2 External memory usage MAT (800 x 800m dataset of Rotterdam puntenwolk) 78
Table 5.3 External memory usage of the tiling and streaming algorithm on parts of the rotterdam pointcloud . . . 78
Table 5.4 External memory usage of the tiling on AHN3 sets . 79
Table 5.5 Maximum memory usage for several sizes collections using the tiling approach (800 x 800 m) 79
Table 5.6 Main memory usage of streaming approach 80
Table 5.7 Maximum memory usage for several sizes collections using the tiling approach (1600 x 1600 m) 80
Table 5.8 Steps both approaches make and the description . . . 81
Table 5.9 Processing time of parts of the tiling process (800 x 800m) 82
Table 5.10 Processing time of parts of the tiling process (1600 x 1600m) 82
Table 5.11 Processing time of parts of the streaming process (800 x 800 m) 82
Table 5.12 Processing time of parts of the streaming process (1600 x 1600 m) 83
Table B.1 Maximum memory usage for several sizes collections (1600 x 1600 m) 100

LIST OF ALGORITHMS

2.1 Shrinking Ball Algorithm 18

4.1 Tiling approach 53
4.2 Streaming approach 67

ACRONYMS

AHN3 Dutch: Actueel Hoogtebestand Nederland; English: current elevation map of the Netherlands

LiDAR Light Detection And Ranging

MAT Medial Axis Transform

MA Medial Axis

SF Spatial Finalizer

1 INTRODUCTION

The shape of objects is usually described by their outer boundaries. Using Light Detection And Ranging (LiDAR) large parts of the earth's geography can be captured and represented as a pointcloud. While the outer boundaries are useful for further analysis and visualisation, another way to represent objects is Medial Axis Transform (MAT), which could be seen as the inner and outer skeleton of the object (see Figure 1.1). Using its properties, objects become more easily detectable and several features become more apparent. As such the MAT could be seen as a tool in pointcloud analysis. For instance, MAT already has the width or height of a object in its datastructure represented by a radius. The medial axis is always a dimension lower than the representation of an object using the outer boundary, as such shape characteristics can be identified more easily. Applications involving MAT are surface reconstruction [Amenta et al., 2001], Simplifying shapes [Tam and Heidrich, 2003][Berger and Silva, 2012], Simplifying point clouds [Ma, 2012][Peters, 2014b] and visibility analysis without surface reconstruction from pointclouds [Peters et al., 2015]. These will be elaborated more in Section 2.

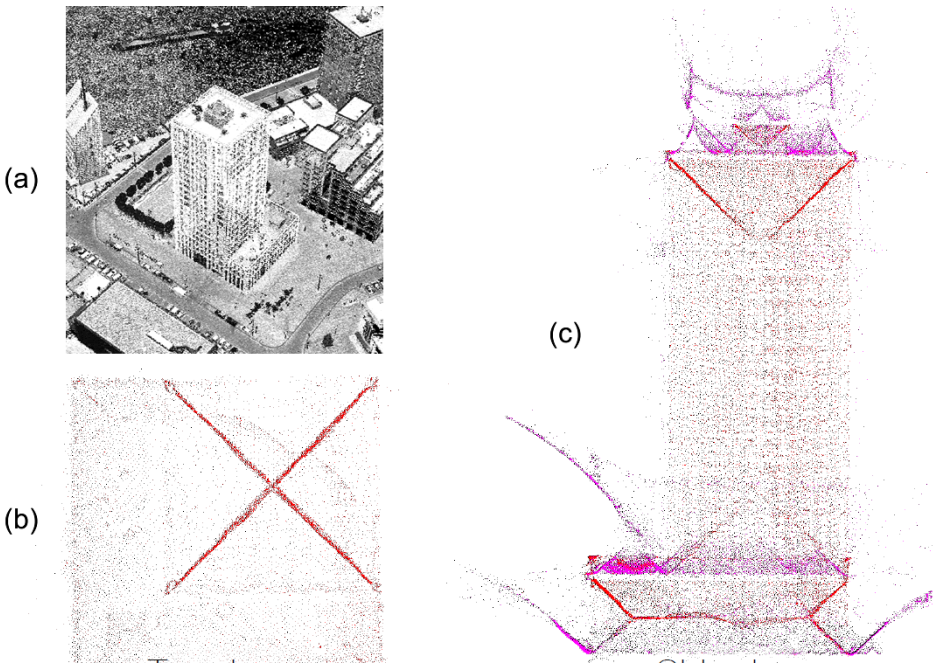


Figure 1.1: The medial axis visualized for a building (a) building represented by points of the outer boundary (b) top view of the building represented by the medial axis (c) side view of the building represented by the medial axis

1.1 DATASETS

The datasets used during this project are 3D pointclouds captured using airborne (LiDAR). This is done by using ultraviolet, visible, or near infrared light to image objects. As these distances are measured from the air, typically the horizontal planes are better captured than vertical planes (such as walls).

The Netherlands is one of the few countries which has a high resolution LiDAR dataset comprising the nations terrain, the AHN3 (dutch: Actueel Hoogtebestand Nederland; English: current elevation map of the Netherlands). The point density of the dataset is about 8 point per m^2 and therefore it contains more than 600 billion height measurements. The AHN3 is mainly necessary for water- and weir management of government agencies. Using this large dataset it can be determined how water flows from the land, how high the water level in ponds is and how much water rivers can discharge. Furthermore the dataset could be used for other kinds of management, such as 3D-mapping, permits and enforcement of it [Rijkswaterstaat, 2014].

Municipalities are maintaining their on LiDAR datasets as well. The Rotterdam dataset has an average point density of 30 points per m^2 with peak densities of 60 points per m^2 [Peters, 2014b].

However, due to the large size of the dataset it is hard to process the complete dataset. To create the 3D MAT of the dataset the current available algorithm needs to be scaled.

While algorithms to compute the 3D MAT work on small datasets, it could be troublesome for massive pointcloud datasets. This is because to run this algorithm ,all the data points are stored in the main memory. For larger datasets this is impossible to do. Therefore smart solutions need to be thought of to ensure that all the data can be processed with limited memory.

1.2 RESEARCH OBJECTIVES

1.2.1 Research question

The main research question is:

- How can the 3D medial axis point approximation using the shrinking ball algorithm be scaled in a memory efficient way for a large dataset which does not fit in the internal memory?

For the answer of the main research question the following sub questions need to be resolved:

1. What are the challenges in scaling the 3D medial axis using the shrinking ball algorithm?
2. How to design and implement several methods for scaling the shrinking ball algorithm?
3. How do the methods compare to each other?
   In terms of memory usage, computation time and output quality.

1.3 WORKFLOW

The research questions/objectives mentioned previously have been completed sequentially (See Figure 1.2). As not all ideas worked out, it was an iterative process. The MAT was first analysed, while performing a literature research on scaling methods. This has lead to several implementations on the scaling level and MAT computation level. Finally these implementations were tested and compared to each other using several real-life datasets. The benchmarks of these tests gave an insight in to how to compute the MAT for massive datasets efficiently.

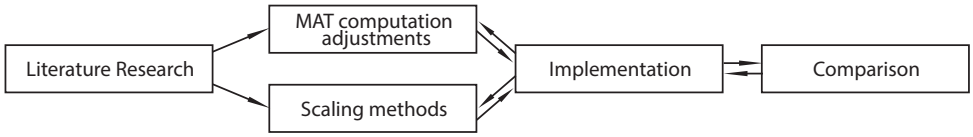


Figure 1.2: workflow

1.4 SCOPE

For this thesis only topographic point cloud datasets will be used. The resulting scaling algorithm will therefor only work on datasets with similar characteristics as topographic point clouds. This means that most of the points should be spread in 2 of the 3 directions (in the case of the AHN x and y direction) and have limited points spread in the the zenith direction (in case of the AHN the height). Therefore regions can be formed as it was a 2D dataset.

The estimation of the normal vectors should be correct to create the MAT for an object. However, as normal vectors are estimated for pointclouds, they are not always correct. As the project is about scaling the algorithm, solving this problem with incorrect normals will not be addressed.

1.5 OUTLINE

In chapter 2 the background information an related work is presented. The nature and features of the MAT are explained and applications for it are described. Several methods to compute the MAT are given, where the shrinking ball algorithm is explained in more detail as it is used in the rest of the thesis. An overview of related scalable GIS algorithms/methods is finally given.

Chapter 3 focusses on the MAT computation itself, the usage and need of buffers is explained and the possibility of a reduced buffer method where points are left out is evaluated.

In chapter 4 the two main approaches to scale the MAT are introduced, namely the Tiling algorithm and Streaming algorithm. The way they chunk the dataset in processable parts is explained and how they use it to compute the MAT (explained in the previous chapter).

In chapter 5 the two approaches are implemented on real world datasets and compared to each other in terms of data quality, process time, main memory usage as well as secondary memory usage.

Chapter 6 will provide the conclusions drawn during the research. However, as some additional research could be done to improve both approaches, some recommendations are made as well.

2 BACKGROUND INFORMATION AND RELATED WORK

This chapter will provide the background information and related work. These will form the basis used to create the methodology. As the data used for this thesis is geographical data, spatial data structures will be first explained. Followed by the MAT in which its properties and applications will be discussed. A few computation methods to get the MAT will then be introduced, with the shrinking ball algorithm explained more thoroughly, as it is the chosen method in this thesis. Finally, the possible scaling methods will be presented.

2.1 SPATIAL DATA STRUCTURES

As the large amounts of spatial data need to be accessed and processed rapidly using queries, spatial data structures are used to organize the data. To be able to access the data quickly the spatial databases sort the data based on their spatial keys, so that the sorting is applied on the space occupied by the data. These techniques are called spatial indexing methods [Samet, 1995]. Spatial data structures can be characterized in 2 types: space-driven and data-driven. The quadtree is an example of a space-driven data structure, as it is partitioning the embedded space. The kd-tree is an example of a data-driven data structure, it is partitioning the data items themselves [Vitter, 2007].

A hierarchical tree structure can be defined recursively as a collection of nodes. Each node is a data structure consisting of a value and a list of reference nodes. Simply said, a tree structure has a root value and subtrees of children with one parent node.

2.1.1 Region Quadtree

The quadtree [Finkel and Bentley, 1974] is a tree data structure in which each node has 4 children (see Figure 2.1). Each of these 4 nodes represents a bounding box, while all of those 4 together cover the entire area of its root node. The way this structure of quadtree is created is by recursively subdividing the cells in to 4 equal-sized subcells until each cell contains a point. A quadtree usually takes O(n log n) time to build with n points.

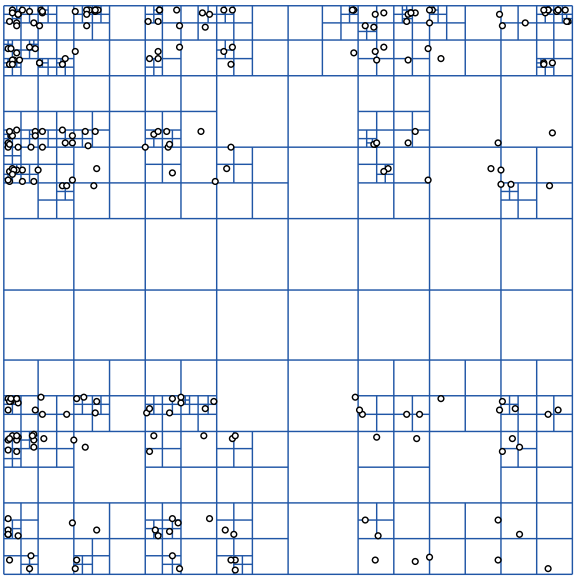


Figure 2.1: Region quadtree (Image from Wikipedia,2005)

Optimized point quad tree

The optimized point quad (Figure 2.2) tree differs from the region quadtree in that it does not subdivide in 4 equal-sized subcells. It looks at the points inside each cell and subdivides it in such a way that each of the 4 subcells does not contain more than half of the points of the parent cell. [Finkel and Bentley, 1974]

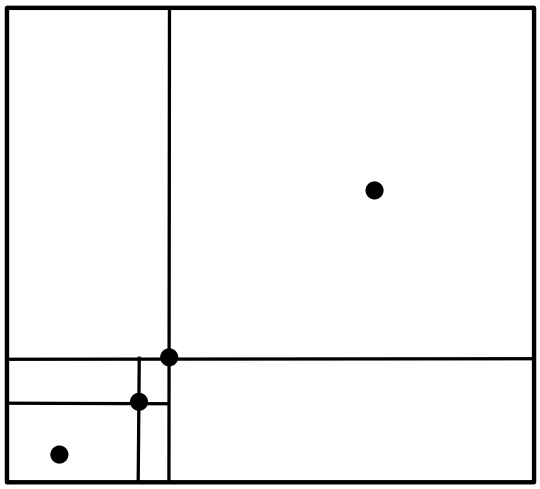


Figure 2.2: Optimized point Quadtree

2.1.2 KD-tree

The difference between a KD-tree [Bentley, 1975] and a quadtree is that it splits alternately in the x and y direction (in case of a 2 dimensional dataset). As a result each node only has 2 children. Whilst there are many ways to construct a kd tree, the one applicable for this thesis is the so called "optimal

tree". This kd-tree is created by choosing to split the dataset at the median point. Then for both sides of the subsets a median point is chosen and both datasets are split again, but in the alternative dimension/direction. Another method, is the an unbalanced kd-tree, which is made by choosing the split in such a way that both halves are subdivided uniformly in space. However, this could lead to empty nodes, see Figure 2.3. As the balanced kd-tree is a data-driven data structure, point deletion is harder than with a space-driven data structure. The splitting using a cutting plane/line will change when new points are deleted, therefore large parts of the data structure might need to be rebuilt. The creation of a kd-tree can be achieved in $O(nlogn)$ time.

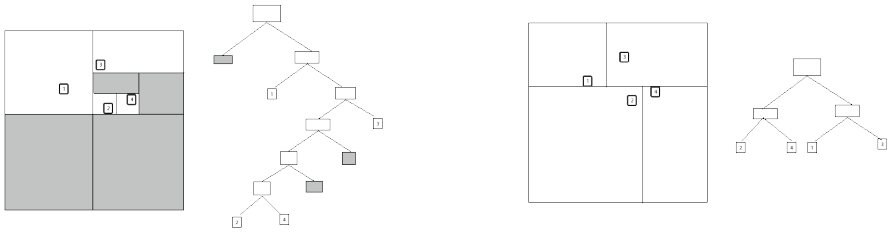


Figure 2.3: (a) unbalanced kd-tree, unnecessary partitioning can be seen in the graph, the grey regions are empty nodes. The depth is 6. (b) balanced kd-tree, depth is 2 (Image from A. Sud)

Nearest Neighbour Search

The kd-tree is probably the most popular data structure used for searching in multidimensional space [Shakhnarovich et al., 2006]. A method to get the nearest neighbour for point p is to follow the path from the root till the terminal node (i.e. the node which contains a point in the lowest level of the graph). By determining from the root whether point p is to the left or the right of the cutting line it is decided to which path it should go. When this is done iteratively, eventually the process will end with the terminal node i.e. the closest neighbour (See Figure 2.4). On average the search complexity is $O(logn)$ [Bentley, 1975]. However in a worst case scenario, the search complexity can be $O(n)$. This happens on situations where search query needs to go through every point, to find the nearest neighbour.

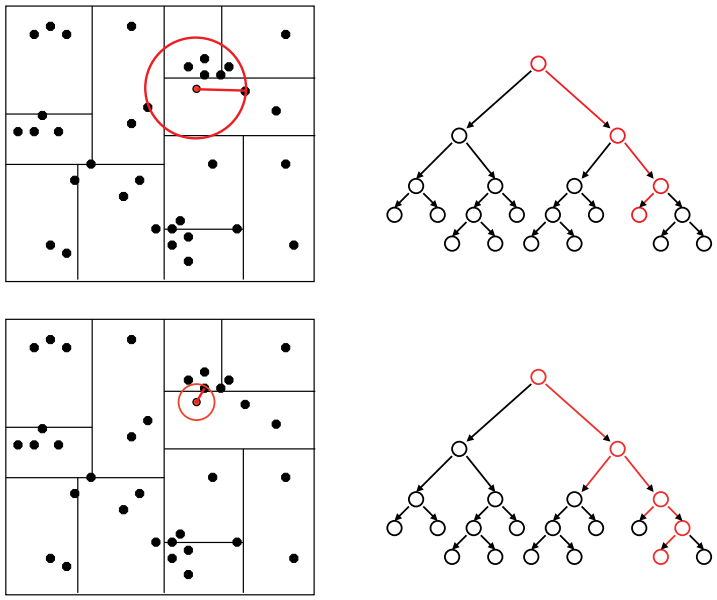


Figure 2.4: Nearest Neighbour Search KD-tree. (Adjusted image from E. Fox)

Optimized KD-tree

While the regular KD-tree is created by alternately splitting in the x and y direction. The optimized KD-tree splits the data set in the direction of largest spread (Figure 2.5). This results in subsets which are potentially less "skinny" than with the regular KD-tree generation [Friedman et al., 1977].

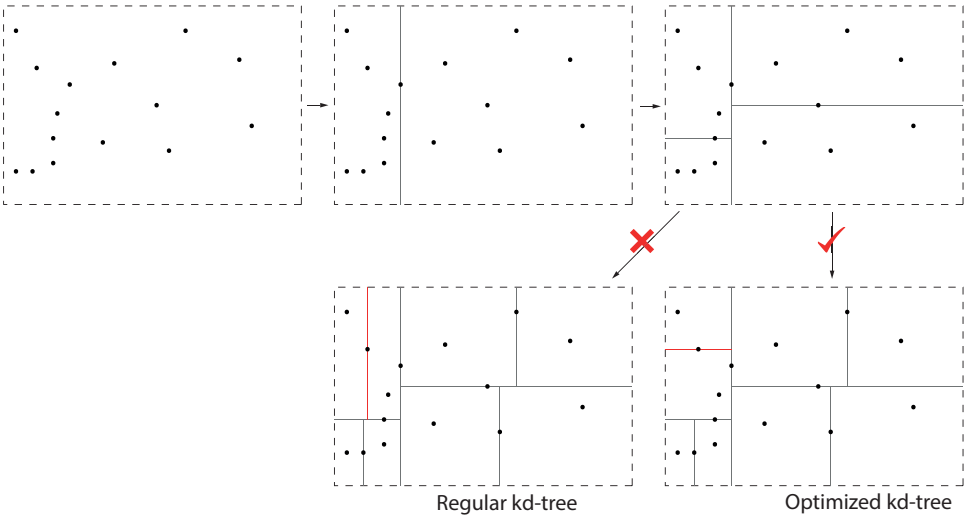


Figure 2.5: subdivision of the dataset for the kd-tree and optimized kd-tree

2.1.3 Space filling curves

A space-filling curve is a continuous curve whose range contains the entire n-dimensional space. This continuous curve can be thought of as a path of a continuously moving point through a n-d map. By doing so, the correlation between the proximity of objects in space and the proximity of their representation in the datastream (locality) is improved. In other words, it

is a method to order a n-dimensional space in to a 1-dimensional stream. While there are many forms of space filling curves, the Morton curve [Morton, 1966] is used in this case to sort the tiles (section 4.2.5). This is not the best space filling curve to preserve locality [Jagadish, 1990], as the Hilbert curve performs better. However the Morton curve is easy to compute, as it maps the 1 dimensional list of tiles by interleaving the bits of the binary representations of the X and Y values of the coordinate values (See Figure 2.6).

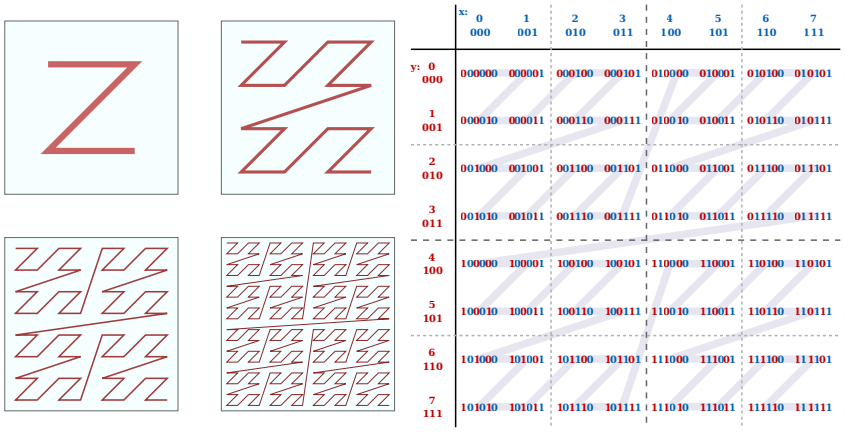


Figure 2.6: Wikipedia,2015 (a) Four iterations of the Z-order curve (continuous line) (b) mapping the 1 dimensional list of tiles by interleaving the bits of the binary representations of the X and Y values of the coordinate values

2.2 MEDIAL AXIS TRANSFORM

Blum [1967] originally spoke of the *transformation* for extracting new descriptors of shape which he called the Medial Axis (MA) and the Medial Axis Function (MAF). Philbrick et al. [1968] named it later the Medial Axis Transform (MAT). Blum [1973] posited several ways of defining what the MAT is. The one most similar to the algorithm used during in this thesis is the Maximal Disk Formulation (Figure 2.7): Against each location on the boundary a ball is placed in such a way that the ball is tangent to the boundary. When the ball starts to grow it will eventually touch another part of the boundary. When that happens the ball will have the following properties:

- It is completely within the boundaries of the shape for the interior MAT
- It is tangent to more than one boundary point

A ball with these properties is called a medial ball. Figure 2.7 shows the MAT for a surface, a continuous field. In this thesis however, a geographic pointcloud is used (a set of sampled points on the surface). As such the MAT will also consist out of a set of discrete points (see Figure 2.9). Figure 2.8 shows that the MAT can be created for the inner and outer part of an object (inner and outer MAT).

The locus of centers of the created balls is called the medial axis (MA). The medial axis including the radii (r) belonging to each point is the MAT (x, y, r). When this is done in 3D the MAT will include a z direction (x, y, z, r).

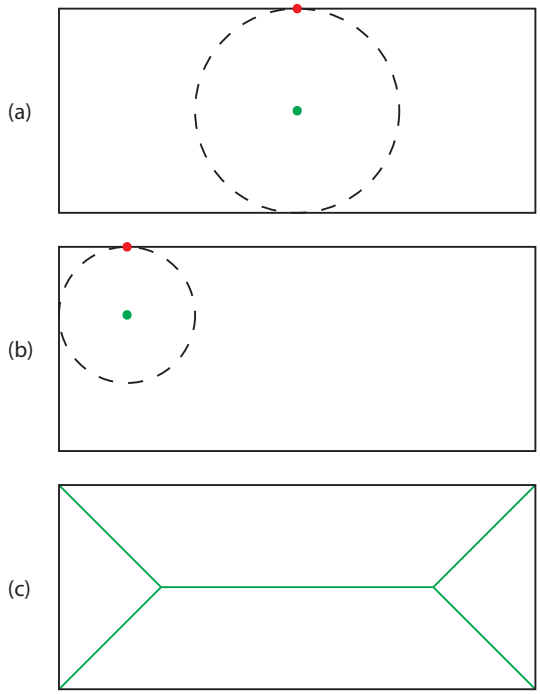


Figure 2.7: (a) The object is a rectangle. The red dot is the analysed location on the boundary. The circle is tangent to the boundary on the red dot. It touches the boundary on the other side, this is the maximum ball which will fit in the object at the location of the red dot. The center of this circle is displayed as a green dot. (b) Same procedure is done on another location. (c) When all locations are processed the locus of centers of all circles form the medial axis displayed in green

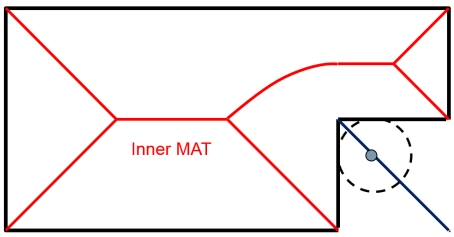


Figure 2.8: inner and outer MAT

2.2.1 Properties of MAT

Peters [2014a] names a few other valuable properties:

- Complete
  The MAT describes the shape of an object. So while the MAT is created using the bounbary representation of an object, the boundary of the collection of medial balls represents the boundary of the original shape. Figure 2.9 displays the MAT as red points, the blue points represent the outer boundary of a rectangle.

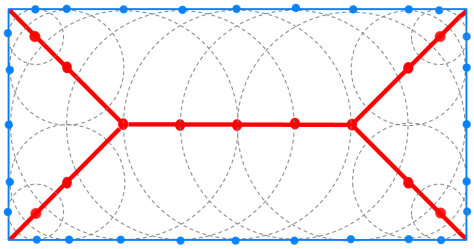


Figure 2.9: MAT of a pointcloud
gray medial balls; red medial axis; blue outer boundary representation

- Topology preserving
  The MAT preserves the topology of the original shape.
- Compact
  The dimension of the MAT is 1-dimension lower than that of the boundary representation of the object. As can be seen in Figure 2.9, the outer boundary is a 2 dimensional surface, while the MAT consists out of 1 dimensional lines (en-captured in the 2-dimensional surface). Therefore a MAT is easier to analyse.
- Hierarchy
  The hierarchical structure of the MAT allows traversal of different elements that defines an object
- Medial
  The MA is centered in an object. As it is in the center of 2 boundary points.
- Sensitivity
  Small changes in an object lead to huge changes in the MAT.

2.3 APPLICATIONS OF MAT

As mentioned before, the MAT can be used for spatial analysis. There are several advantages of using MAT to model geometrical objects [Blum, 1973].

- The location of features is known
  The MAT is the objects center
- Thickness information of objects can be extracted
- Shape characteristics (curvature, symmetry) can be identified in a intuitive and accurate way by analysing a skeleton; as such Feature recognition [Prinz, 1988] is made easier.
- If a medial axis is composed of a set of connected branches, its structure is a graph. Each individual branch of the skeleton is associated with different parts of the object. The branches of the medial axis can separate the shape into simple segments [Grsoy, 1989]. Using the Domain Decomposition Lemma a given domain can be split in to smaller subdomains (i.e. datasets can be subdivided and merged while preserving whole attributes). The MAT is represented by the union of the domains [Choi et al., 1997] (see Figure 2.10).

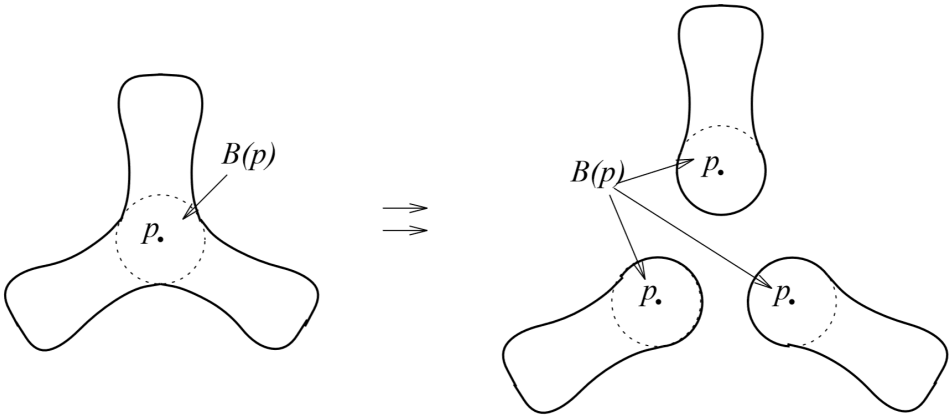


Figure 2.10: The idea of Domain Decomposition [Choi et al., 1997]. The union of the 3 subdomains will result in the MAT of the original domain

- The property that the MAT is always represented one dimension lower than the boundary representation is widely used in many applications [Liu, 2011]. For instance shape centers are used in map creation to put text annotations. It can also be used for shape matching, when the structures are similar, while their posture is different.

2.3.1 Surface reconstruction

Amenta et al. [2001] reconstructed surfaces of a 3D object by approximating the MAT of an object (see Figure 2.11). Sample points of the object are then interpolated if they lie on the union of the inner and outer medial balls (so called powercrust method).

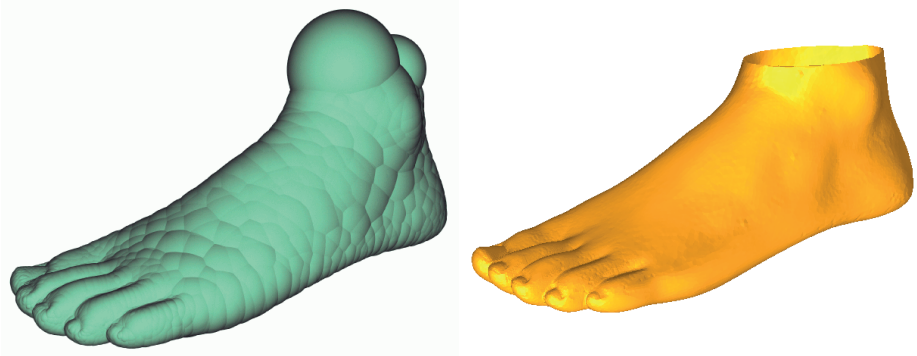


Figure 2.11: left: inner medial balls; right: interpolation of sample points using the powercrust method(Images from Amenta et al. [2001])

2.3.2 Simplifying shapes

Tam and Heidrich [2003] and Berger and Silva [2012] presented a way for simplifying the shape of 3D objects by manipulating the MAT. They decomposed the axis into parts and removed a subset to reduce the complexity of the resulting object (Figure 2.12).

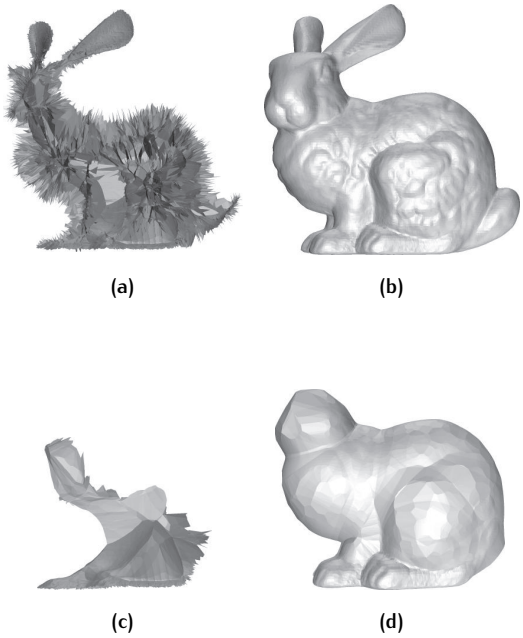


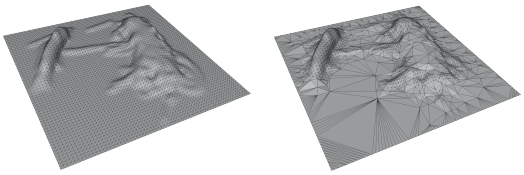
Figure 2.12: (a) Original MA, (b) Surface reconstruction from boundary points (c) Strongly simplified MA, (d) Surface reconstructed from strongly simplified MA (Images from Tam and Heidrich [2003])

2.3.3 Segmentation

Berger and Silva [2012] performed pointcloud segmentation using medial kernels. Which is a similarity measure defined as the likelihood of two points belonging to a common interior medial ball. By using these medial kernels to perform random walks in the pointcloud, restricting to regions with similar medial balls.

2.3.4 Simplifying point clouds

Ma [2012][Peters, 2014b] demonstrated using the shrinking ball algorithm a way to selectively simplify point clouds while preserving features of objects. This could be achieved by determining the local feature size (LFS) [Dey et al., 2001], which is the shortest distance to the MA for each point on the boundary of objects. Points positioned far away from the MAT are of lesser importance to the shape of an object than points nearby. By simplifying (removing less important points) using LFS the features of objects will be therefore be preserved (Figure 2.13).



(a) (b)

Figure 2.13: LFS-based simplification
(a) Original dataset (b) After simplification (Images from Peters [2014b])

2.3.5 Visualisation by point splatting

For visualization splats (1 dimensional circles) in the direction of the normal vector could be placed. The size of these splats are determined by their proximity to the LFS. This provides better visibility of objects when points are sparse [Peters, 2015] (see Figure 2.14).

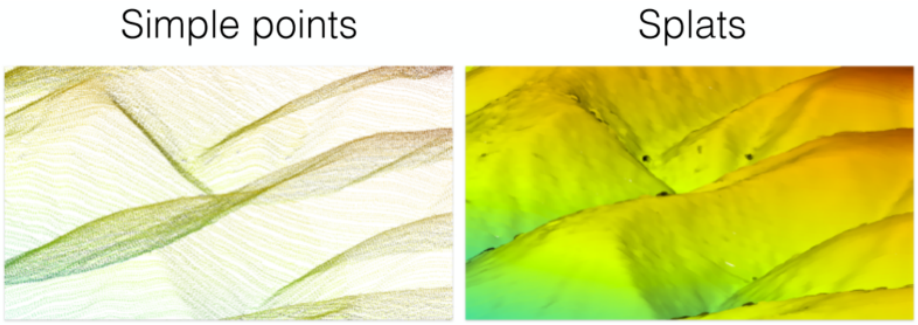


Figure 2.14: regular point representation vs splat representation [Peters, 2015]

2.3.6 Visibility analysis

Usually visibility analysis is applied on 3D city models which are often created from aerial point clouds. By using the MAT, visibility analysis be used on a point cloud directly without the need to compute a 3D city model. By modelling the urban environment as a union of medial balls, which are then used to construct a depth map that is used for point visibility queries [Peters et al., 2015] (see Figure 2.15).

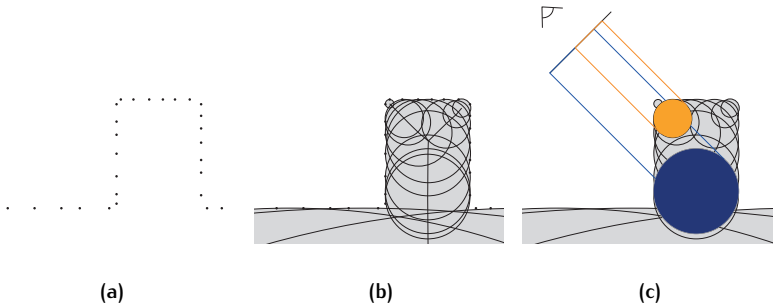


Figure 2.15: Visibility analysis using the medial balls in 3 steps (Images from Peters et al. [2015])
(a) Approximating MAT from point cloud (b) Computing depthmap (c) Point visibility querying

2.4 COMPUTATION OF THE MAT

There are several algorithms to compute the MAT. A few of these are introduced here.

2.4.1 Tracing based algorithms

The tracing algorithm [Sherbrooke et al., 1995][Reddy and Turkiyyah, 1995] computes the medial axis by tracing its paths. Observe Figure 2.16, the computation starts tracing from a seam-end Point, like a vertex of a polyhedron. It traces the seam till it encounters a junction point or Seam-end. As a junction point is a location where multiple seams intersect, the tracing continues at those seams. When the trace meets a seam-end point, the tracing for that part stops.

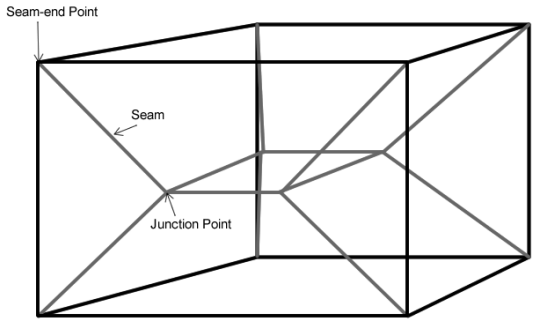


Figure 2.16: Classification of MA Points (Image from [Sherbrooke et al., 1995])
Seam point: point which is equidistant to at least 3 boundary points
Seam-end Point: location where the seam ends at the boundary of the object
Junction Point: location where multiple seams meet

2.4.2 Voronoi based algorithms

The vertices in a voronoi diagram are closely related to the MA, as 3 or more boundary points are equidistant to the voronoi vertice (similar property as

the MA has). By creating the voronoi ball (e.g. a ball which has multiple boundary points on its boundary, and a voronoi vertice as center) the inner and outer voronoi vertices can be determined. As inner voronoi balls and outer voronoi balls intersect shallowly, if at all [Amenta et al., 2001].

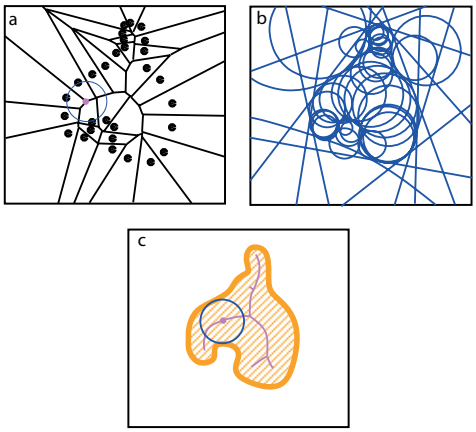


Figure 2.17: Two-dimensional example of MAT approximation [Amenta et al., 2001] (a) Pointcloud representing the outer boundary of an object, edges representing the voronoi diagram. The vertices of the voronoi diagram approximate the medial axis. (b) On the voronoi vertices a voronoi ball is placed which touches the outer boundary of the object. (c) The inner medial axis is determined by the union of inner balls and outer balls.

2.4.3 Shrinking ball algorithm

The shrinking ball algorithm uses a ball which shrinks till it fits inside an object. [Ma, 2012][Jalba et al., 2013] Let us first define that V is the set of sample points on a surface and N the corresponding normal vectors. A ball B is represented with a center α and a radius β ($B(\alpha, \beta)$). If this B touches at least 2 or more points in V and contains no points in its interior, this ball is called a medial ball. The medial axis of V is defined as the locus of centers of the medial balls.

To compute the medial ball per point p in V, a ball B is placed with its center c^0 on L_p (a line through p along the direction of normal vector n). The initial radius of B is r_{init} and is given by a sufficiently large number (Figure 2.18a). By decreasing r iteratively, the medial ball is obtained. At each iteration step a nearest neighbour search takes place, to find the point $\tilde{p}^i$ in V closets to c^i where i is denoted as the ith iteration step. Ball B is then recomputed with point p and $\tilde{p}^i$ on its boundary, while having c^i on L_p. This is iteratively performed for point p, till there are no closest points to center c^i other than p and $\tilde{p}^i$ (Figure 2.18b, c, d). The final ball B is then the medial ball.

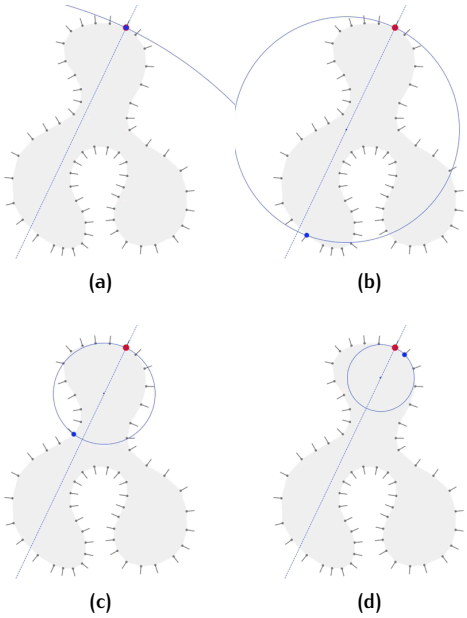


Figure 2.18: (a) largest ball; (b) blue dot is closest to the previous balls center; (c) blue dot is closest to the previous balls center; (d) blue dot is closest to the previous balls center and there is no other dot inside the ball (images from R. Peters)

To calculate the radius of the ball the following formula is used:

$$r = \frac{d(p_1, p_2)}{2cos\theta_{p_1}} \tag{2.1}$$

Where $d(p_1, p_2)$ is the distance between the 2 final points and θ is the angle between vector (p_1, p_2) and the vector (p_1, c) (see Figure 2.19). While the process and equation 2.1 are described for a 2D dataset, the same method applies in 3D.

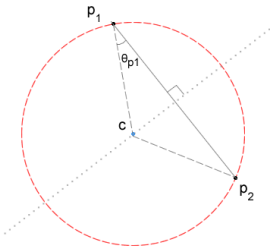


Figure 2.19: Radius calculation

The shrinking ball algorithm is chosen for the thesis because it is relatively simple to implement, more storage efficient and easier to parallelize compared to other methods (e.g. voronoi, tracing methods). The parallelization can be done by splitting the pointcloud dataset in N equal sized chunks and process each chunk per thread.

Starting radius r

As mentioned in the previous paragraph, the computation starts with a large radius medial ball and shrinks till no other points are inside the ball.

Algorithm 2.1: SHRINKING BALL ALGORITHM (V, N) Ma [2012]

Input: Pointcloud Dataset V, Normal Vectors N
Output: c_p: Medial axis, r_p radii p

Function *COMPUTE_MA_POINT(V,N)*
  for ***each*** $p \in V$ **do**
    $i \leftarrow -1$;
    if p *is the new element* **then**
      $\tilde{p}^i \leftarrow$ a randomly chosen point, $\tilde{p}^i \in V - \{p\}$;
      $r_{init} \leftarrow$ COMPUTE_RADIUS$(p, \tilde{p}^{-i})$;
      # compute sufficiently large r_{init}
    $r_p^0 \leftarrow r_{init}$
    repeat
      $i \leftarrow i + 1$;
      $c_p^i \leftarrow p - r_p^i \mathbf{n}_p$;
      $\tilde{p}^i \leftarrow$ the closest point from $c_p^i, \tilde{p}^{-i} \in V - \{p\}$;
      $r_p^{i+1} \leftarrow$ COMPUTE_RADIUS$(p, \tilde{p}^{-i})$;
    until $r_p^i = r_p^{i+1}$;
    $c_p \leftarrow c_p^i; r_p \leftarrow r_p^i; \tilde{p} \leftarrow \tilde{p}^i$;
    if p *is not the last element of* V **then**
      $q \leftarrow$ next sample in V;
      $r_{init} \leftarrow$ COMPUTE_RADIUS$(q, \tilde{p})$
  return $\{c_p\}, \{r_p\}$;

Function *COMPUTE_RADIUS*$(p, \tilde{p})$
  $\theta = arccos \frac{\mathbf{n}_p * (p - \tilde{p})}{d(p, \tilde{p})}$;
  $r = \frac{d(p, \tilde{p}}{2cos\theta_p}$;
  return r;

However, how big should the starting ball size be? As the medial axis is in the centre of the objects and the radius of it describes the width of the object itself, the starting size for shrinking should be as big as the largest object detectable. Let us assume that one wants the inner and outer MAT of the built environment. As the datasets used in are parts of the Netherlands, nearly all human-made objects are less than 200 meters tall (apart from the top 5 highest buildings [Hoving, 2012]). The optimal starting radius therefore should be 100 meters.

Normal calculation

To numerically compute the normal vector for points, a point set in the neighbourhood is usually used. By computing a plane that best fits the point set a normal vector can be extracted. Hoppe et al. [1992] proposed a method where the k nearest neighbours are being used to fit a plane by using the total least squares method. And while there are numerous improvements on this method, the normal calculation for points remains just an estimation. Errors may occur when points are not well spread or at ”corners” of objects (as corners can not have a well defined normal vector).

Other methods are Least Squares Analysis and Principal Component Analysis, the latter has an advantage, as it has the possibility to examine the quality of the fit.

Handling noise

As mentioned in section 2.2.1, the MAT is sensitive to even small irregularities on the surface (see Figure 2.20), it is necessary to use de-noising measures.

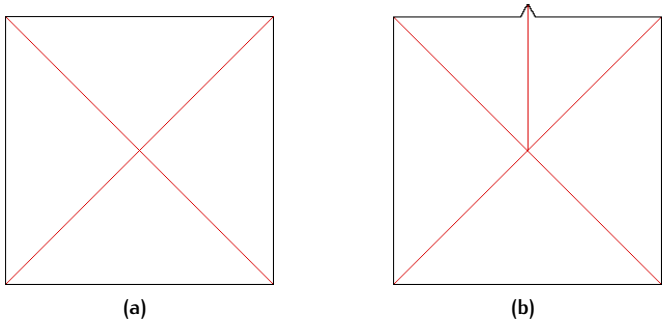


Figure 2.20: (a)Square 2D object with its inner MAT (b) Square 2D object with a outlier (noise) in its topline, MAT is effected by this

Amenta et al. [2001] suggests to make use of the so-called separation angle. Which is the angle between the medial point and the two corresponding surface points. As balls with smaller separation angles are more likely to be influenced by noise, a thresholds are set. Whenever the separation angle is smaller than the threshold or difference in two consecutive separation angles is bigger than a second threshold the ball is rejected. Observe Figure 2.21, the 5^{th} ball has a much lower separation angle than the others, therefore the last ball will be flagged as noise and the previous ball is taken.

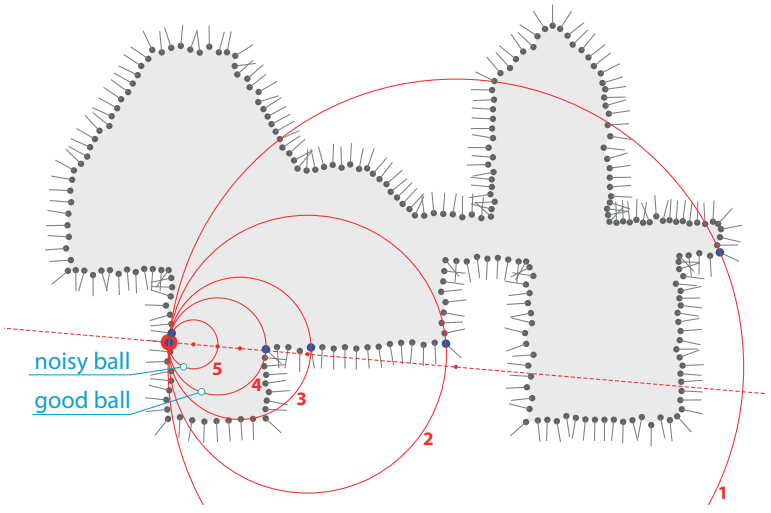


Figure 2.21: Sequence of Shrinking ball algorithm with noise [Peters, 2014b]

2.5 MEMORY HIERARCHY

Computer systems contain a hierarchy of memory levels, each of which has their own characteristics. For this thesis the simplest input/output (I/O) model (two-level memory hierarchy) will be considered, where the two levels [Nodine, 1992]:

- Small, but fast internal memory.
  Also known as the main memory or RAM of the computer.
- Large, but slow external memory.
  Also known as the hard disk (or solid state disk).

When applications process large amounts of data, the I/O communication between these two levels of memory is usually the bottleneck. This is due to that accessing the internal memory takes several tens of nanoseconds (10^{-8} seconds), while accessing data from a disk can take several milliseconds (10^{-3} seconds).

Data that is too large for the internal memory can be stored temporary on the external memory. However, the communication between the internal memory and the external memory is often a bottleneck for large-scale applications. External-memory algorithms reduce this bottleneck by optimizing the efficiency of fetching and accessing data stored in slow external memory (such as disks) [Vitter, 2007][Silvia et al., 2002].

J.S. Vitter explains that there are two general categories of problems in reducing I/O costs:

- Batched problems, no preprocessing is done and the entire file of data items must be processed (e.g. Streaming)
- Online problems, where a computation is done in response to a continuous series of query operations. (e.g. making use of a datastructure with a spatial index)

Furthermore the I/O performance of algorithms can be expressed in terms of the bounds the following 4 operations:

- Scanning (streaming), which involves sequential reading or writing of the elements in a file
- Sorting, which places the elements of a file into a sorted order
- Searching, which looks online through N sorted elements
- Outputting the answers to a query in a blocked output-sensitive manner.

The first 2 of these bounds (Scanning and Sorting) apply to batched problems, while the latter two (Search and Output) apply to online problems.

2.6 STRATEGIES FOR SCALING GIS ALGORITHMS

Voronoi diagram is a widely studied problem in computational geometry, there are many different standard algorithms used to compute it I/O efficiently. Sack and Urrutia [2000] lists a few of these.

2.6.1 Divide and conquer

The Divide and conquer (D&C) approach solves a large problem by [Dasgupta et al., 2006]:

- subdividing it in to sub-problems which are smaller instances of the same type of problem (divide)
- recursively solving these sub-problems (conquer)
- combining the answers to solve the original large problem

Therefore D&C could be used to solve problems involving datasets which do not fit in the internal memory, as it subdivides the problem in smaller pieces. In the upcoming subsections several methods for subdividing a large spatial dataset will be explained.

In case of a pointcloud the large amount of points form an issue, as it does not fit in the main memory when it is being processed. A top-down Level-by-Level segmentation can be used to divide the problem in to smaller parts [Danner, 2007]. The first step in this process, is to take the complete dataset as a whole. If the amount of points in the dataset is larger than a certain threshold k_{max} (the maximum amount of points a segment can have) the complete dataset is split in four parts (leafs). Then the same procedure is followed for the four leafs iteratively till the threshold of all leafs is smaller than k_{max}. Advantages of creating subproblems by this top-down segmentation is that it works well for subdividing a pointcloud into squares/boxes, however, it does not lead directly to homogeneous segments [Lindenbergh, 2014]. The spatial data structures mentioned in section 2.1 could be used to perform the segmentation. By subdividing the dataset either using a space-driven method (e.g. quadtree) or a data-driven method (e.g. kd-tree).

Buffer

Subdividing a dataset in to smaller subsets could cause issues, often points near the boundary of the subset need data from outside the subset. A way to get that data is to make use of a buffer region, where the data is loaded as well to eliminate these boundary issue. According to ESRI a buffer is defined as a zone around a map feature measured in units of distance or time [ESRI, 2012]. When one wants to create a buffer around a certain object, the Minkowski sum can be used by adding each vector of object A to each vector in object B (Image 2.23).

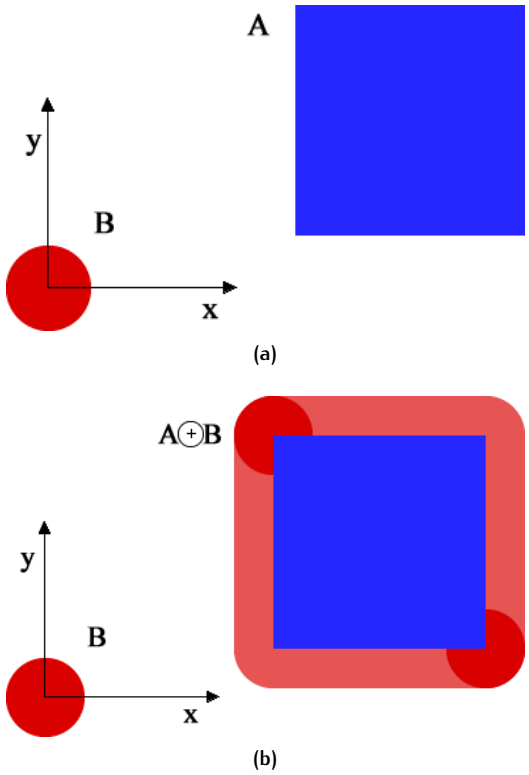


Figure 2.22: Minkowski Sum
(a) Object A and B (b) Each vector of B added to each vector in A

An example of using D&C is the computation of a voronoi diagram. To compute the voronoi diagram, [Shamos and Hoey, 1975] uses the D&C approach to subdivide a set of point sites S into subsets L and R of similar sizes. This is done by a split line in the dataset. Then for both subsets the voronoi diagrams V_L and V_R are computed recursively. The challenge is to find the split line and the merging of both voronoi diagrams. If the sites in S are presorted by the x- and y- coordinates it is easy find the split lines. The merging of V_L and V_R can be achieved by computing the voronoi edges of V(S) that separate regions of sites in L from regions in R.

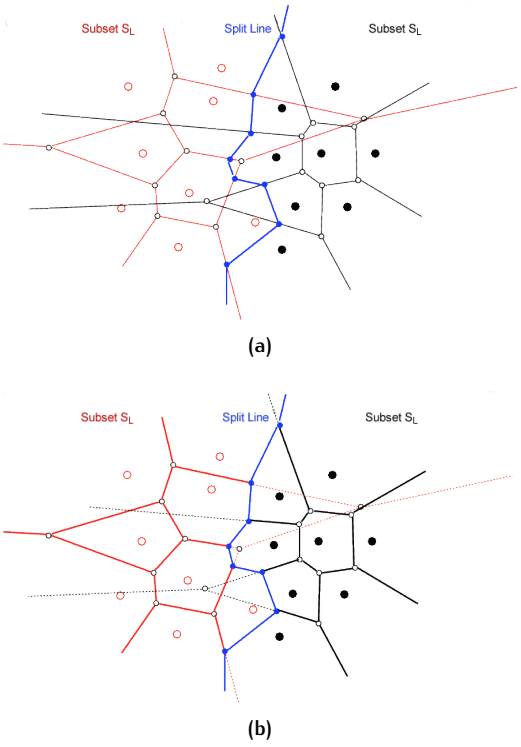


Figure 2.23: D&C used to compute the voronoi diagram (images from R.Muhamma) (a) Split line dividing V_L and V_R (b) Unwanted red lines are removed at the rightside of the split line, same is done for the blue lines

2.6.2 streaming algorithms

In streaming algorithms, data is presented as a sequence of items. The sequence streams from the external memory to the internal memory where it can be processed. As the stream can not go back to data which was already sent, therefore sometimes it needs a few extra passes. Wu et al. [2011] names four requirements in the use of a streaming algorithm:

1. Sequential data access
   Data should not be randomly accessed, but read as as one or multiple continuous data streams.

2. Linear execution
   Operations on the elements of the input stream should be chained and linked together using a pipeline.

3. Data locality
   Operations on elements involving other elements should be relative close in the datastream. Using geographic datasets, Isenburg et al. [2006a] calls this "spatial coherence". Which is correlation between the proximity in space of geometric entities and the proximity of their representation in the stream. This could be achieved by sorting all the points in such a manner (such as making use of a space filling curve) that it is usable for streaming. However Real-world datasets should already have enough of spatial coherence due to the way they are acquired (see Figure 2.24).

4. Memory recycling
   Streaming computations loads the input data sequentially and processes it using a limited memory buffer. By outputting it directly afterwards, memory space is freed and therefore it is suitable for data-intensive applications.

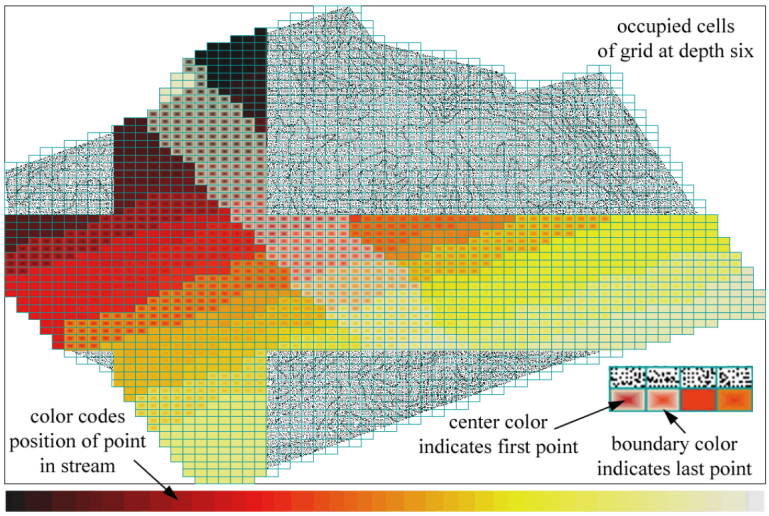


Figure 2.24: LiDAR dataset of Baisan Run at Broadmoaor, Maryland (6 million points). Colors illustrate spatial coherence in the gridcells; each center of the cell is colored by arrival time of the first point in the stream and the border is colored by arrival time of the last point in the stream. The color represents a certain moment in time [Isenburg et al., 2006a].

Spatial finalizer

Isenburg et al. [2006a] spatial finalizer works by reading data from a dataset in 3 times (Figure 2.25):

1. The first pass is used to detect the outer boundary of all the points. After the first pass a grid is created to fit within the boundaries.

2. At the second pass all points are counted within each grid cell.

3. In the final pass, the known amount of points from the step 2 is subtracted by each point which falls within that cell while reading the file. When the counter reaches zero, the points of that cell are exported out of the spatial finalizer to be processed further in the pipeline. After each exported cell a finalization tag is placed, used to notify that there are no more points within that cell.

As the spatial finalizer reads a file from start to end multiple times, the algorithm takes O(n) time.

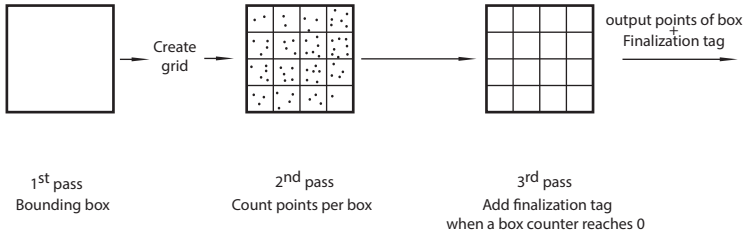


Figure 2.25: workflow spatial finalizer

Isenburg et al. [2006a] uses this streaming algorithm to compute the Delauney Triangulation in a pipeline. As an individual cell arrives from the spatial finalizer to the delauney triangulator, triangulation begins. All the points within this cell will be processed, however, as points near the border might need data from a neighbouring cell, these triangles will not be done yet. It is possible to determine which triangles are final, as the triangle circumcircle do not intersect a space which has not yet arrived from the SF (see Figure 2.26).

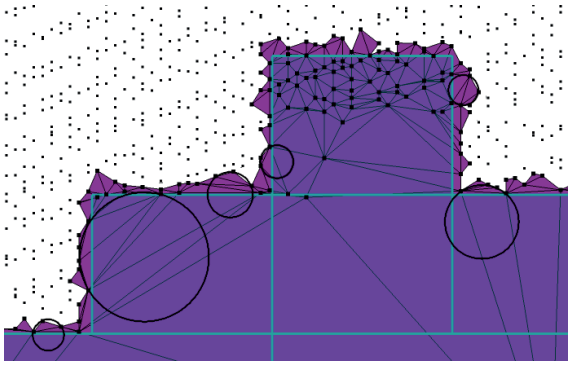


Figure 2.26: Points on the top have been processed and their triangles written out. All the visible triangles are active. A few circumcircles have been drawn to show that they intersect with unarrived space. [Isenburg et al., 2006b]

Using the streaming Delauney triangulator to create a geographic TIN, can be further processed to create a DEM raster grid [Isenburg et al., 2006b].

Constantin et al. [2010] exploits the streaming algorithm with finalizer to apply the Quadric Error Metric in conjunction with edge contraction to simplify meshes.

2.6.3 External memory algorithms

External memory algorithms use the external memory for temporal storage of datastructures which do not fit inside the main memory. External memory access is minimized by explicitly controlling the data movement and data layout, where the goal is to exploit locality in order to reduce the I/O costs Vitter [2007]. Agarwal et al. [2006] used a quadtree to compute a Digital Elevation Model.

3 COMPUTING BUFFERS FOR THE MEDIAL AXIS TRANSFORM

The computation of the MAT for massive datasets using shrinking ball algorithm could be split up into two stages. As massive amounts of points will not fit in to the main memory, in the first stage, the dataset is "chunked" to ensure that the smaller datasets processed fit in the memory. The second stage is the MAT computation itself. Figure 3.1 displays these stages with their possibilities. In this chapter the MAT computation will be discussed. Chapter 4 will describe how the chunking algorithms work.

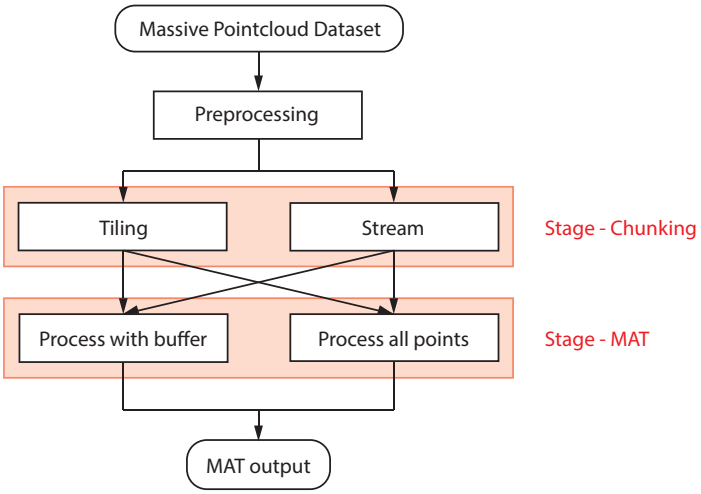


Figure 3.1: Overview of the methodology

3.1 TIME AND MEMORY COMPLEXITY

As mentioned in Section 2.4.3, the shrinking ball algorithm is used in this thesis, as it is simple to implement, storage efficient and easy to parallellize.

3.1.1 Time complexity

The shrinking ball algorithm (section 2.1) shows that each point p in dataset V is processed to get the MAT. For each of the points p iteratively the closest point to the center of the medial ball is calculated. As in the worst case all the points are passed, the time complexity is O(n^2) with n points (excluding the nearest neighbour search). However in practice it exhibits a more linear growth rate O(n) as it is uncommon that all neighbours need to be visited Ma [2012].

To calculate the nearest neighbour a kd-tree query is used. This query has a time complexity of O ($logn$) on average, yet in the worst case it is O(n). As

such, the time complexity to compute the MAT for dataset V is O (n^3) in the worst case. While in practice it might be more O($nlogn$).

3.1.2 Internal memory complexity

Excluding the kd-tree which needs to be built, the only data apart from the input which is collected is the resulting MAT. The memory complexity therefore is O(2(K + 1) n). Where K is the dimension of the input data and addition of 1 is due to the radius.

However, the true memory usage is larger, as the kd-tree itself will take a lot of memory as well (depending on the implementation method it will be larger than O(K n)). Furthermore although the normal vectors are considered as input data, it should be computed as well.

3.1.3 Time and Memory usage in practice

In practice the shrinking ball algorithm exhibits near-linear growth rates O(n) when looking for the nearest neighbours [Ma, 2012]. Querying a nearest neighbour search using a kd-tree usually can be done in O($logn$). Therefore in practice the Time complexity should be O($nlogn$).

The maximum amount of points computable is limited by the amount of main memory addressable by the program. During the process 4 groups of elements will mainly take up the memory, which are displayed in table 3.2.

The MAT algorithm loads the coordinates, normals and computes the MA. These datasets consists out of points/vectors with 3 32-bit floating points (12 byte). The KD-tree used on large datasets for fast querying neighbours takes up roughly 35 byte per point (test in practice). The resulting MAT consists out of the coordinates (12 byte) for the Medial axis and the radius (4 byte) for the inner radius as well as outer (Table 3.1). As such the hard disk space needed to store the MAT should be $2\frac{2}{3}$ of the size of the input dataset.

x	y	z	radius
32 bit float	32 bit float	32 bit float	32 bit float

Table 3.1: MAT Datastructure

Points	Coordinates	Normals	KD tree	MAT	Total
500.000	6 MB	6 MB	18 MB	8 MB	38 MB
1.000.000	12 MB	12 MB	35 MB	16 MB	75 MB
2.000.000	24 MB	24 MB	70 MB	32 MB	150 MB
4.000.000	48 MB	48 MB	140 MB	64 MB	310 MB

Table 3.2: computation memory usage for either the inner or outer MAT!, separated in amount of points

The table 3.2 shows that the amount of memory necessary to compute the MA increases in size linearly with the amount of points. Therefore it can be assumed that when one can address 2 GB of memory (on a 32-bit system), the maximum amount of points computable is about 28 million. When a 64-bit system is used, theoretically a lot more main memory could be allocated.

3.2 CHALLENGES OF PROCESSING SMALLER DATASETS

In section 2.4.3 the shrinking ball algorithm is chosen as method to compute the MAT. However, before chunking the dataset into manageable pieces, the scalability of the shrinking ball algorithm needs to be addressed. Using for example 'divide and conquer', subsets of the dataset are created, of which each has their own boundary. Yet, it is possible that points of one subset need points from another subset during the computation of the MAT. Figure 3.2 shows an example of how the medial ball could be used to determine which points might need extra data from the neighbouring tile, as it is shown in 2D the medial ball is expressed as a medial circle. Furthermore the merging of smaller datasets back to a large one could also cause problems.

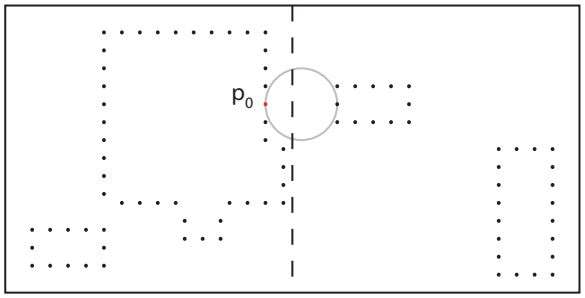


Figure 3.2: Top view of a pointcloud dataset split up in 2 subsets (expressed by the dashed lines). Some points (i.e. p_0) around the borders of both subsets need points from the other side to compute the MAT

3.2.1 Border issues when computing MAT

Points which might need points from another subset are detectable by computing the MAT with only the given subset points. If the medial circle/ball extends to a part outside the subset, it might need points from outside to be sure that the correct MAT is created. As shown in figure 3.2, the red dot indeed needs points from a neighbouring subset to compute the correct MAT. However, this is not always the case, as shown in figure 3.3. The medial circle of the green point extends to another subset, yet it did not need any points from that neighbouring subset. This can only be known when the points of the neighbouring subset are analysed.

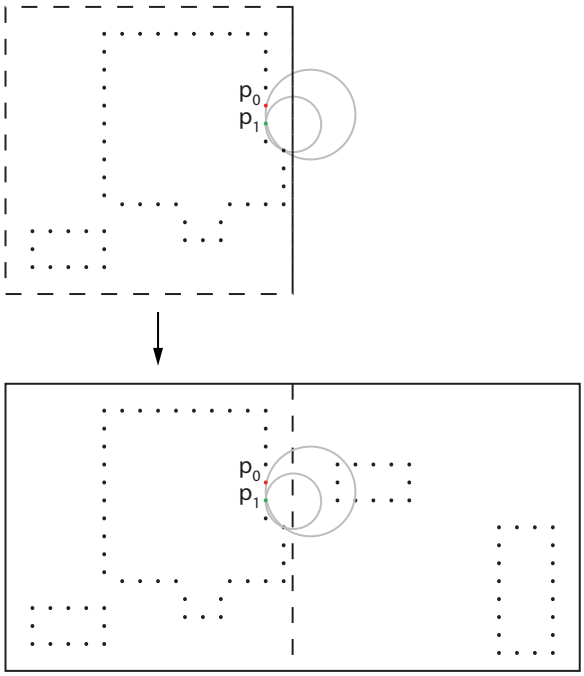


Figure 3.3: The medial ball of point p_1 was correct without the need of points in the neighbouring subset. The p_0 needs to further process the MAT with the data from the neighbouring subset.

When analysing a small piece of a real world urban dataset (see figure 3.4) it becomes apparent that many points around the border of a subset might need points from the neighbouring subset to be sure that their MAT is *final*[1].

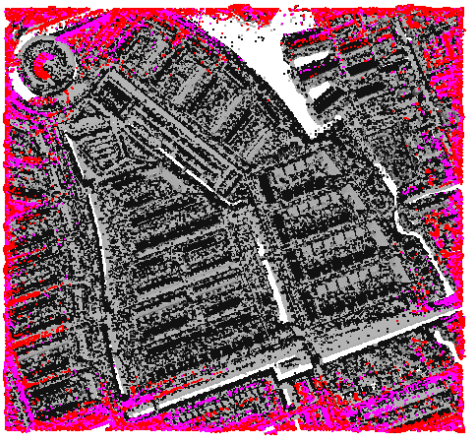


Figure 3.4: Points with a successfully created medial axis are displayed in grayscale. Points which are not finished processing because they need to know the location of points outside the region are displayed in red or purple. They represent the inner and outer MAT respectively

To test how many points will need more information around the border, a few samples of the AHN2 are used.

1 A final MAT will not change when extra points from neighbouring subsets are added in the process

	Sample 1.1	*Sample 1.2*
Area	410m x 470m	750m x 600m
Points	188530	361661
Unfinished points	27288	24457
	14,5%	8,0%
Starting radius	100m	100m

Table 3.3: Unfinished points Delfgauw

	Sample 2.1	*Sample 2.2*
Area	447m x 460m	631m x 658m
Points	200469	405068
Unfinished points	46405	65924
	23,5%	19%
Starting radius	100m	100m

Table 3.4: Unfinished points Woerden

The occurrence area is the area where unfinished points can occur. The unfinished points cannot be just anywhere in the region (see Figure 3.4). As described in Section 2.4.3 "Starting radius r_init" is the maximum size a medial ball can be. As such unfinished points can only appear at a distance of $2r_init$ to the boundary.

As discussed, MAT computations near the boundary of the subset often need additional data from outside. To solve this issue there are 2 options:

- Load a buffer region around the subset, so that points near the initial boundary have the opportunity to compute the MAT correctly using points in the buffer.
  This will be elaborated in section 3.3
- If multiple subsets are processed sequentially, of which said subset is one of, the unfinished MAT could be post-processed together with another subset, see Section 3.4.

Merging the data

Merging data could lead to boundary errors. However, as the the resulting output is the MAT, which is a list of x, y, z, r values. As such there is no link between the medial balls, or with the original input data.

This means that when a dataset is subdivided in smaller subsets, the MAT for the whole dataset is simply the collection of MATs of the subsets.

3.3 REGULAR BUFFER

As divide and conquer is focused on subdividing the problem in to individual subproblems, the large dataset will need to be subdivided in smaller portions including buffers to ensure that all points are correctly computed (Section 2.6.1). To determine the buffer region size, a closer look at the algorithm provides the answer. Observe image 3.5, in the worst case scenario, a point has a normal vector in the x direction. The medial ball created will

be aligned to the normal vector, as a result the medial ball will have the possibility to connect to a point 2*r away from the observed point.

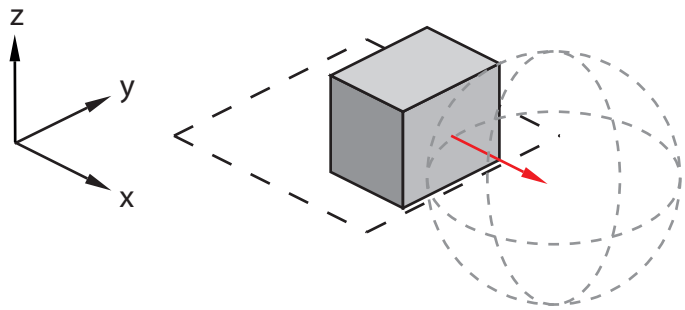


Figure 3.5: A block near the edge of a subset border with its normal vector in red and a 3-dimensional medial ball in dashed grey

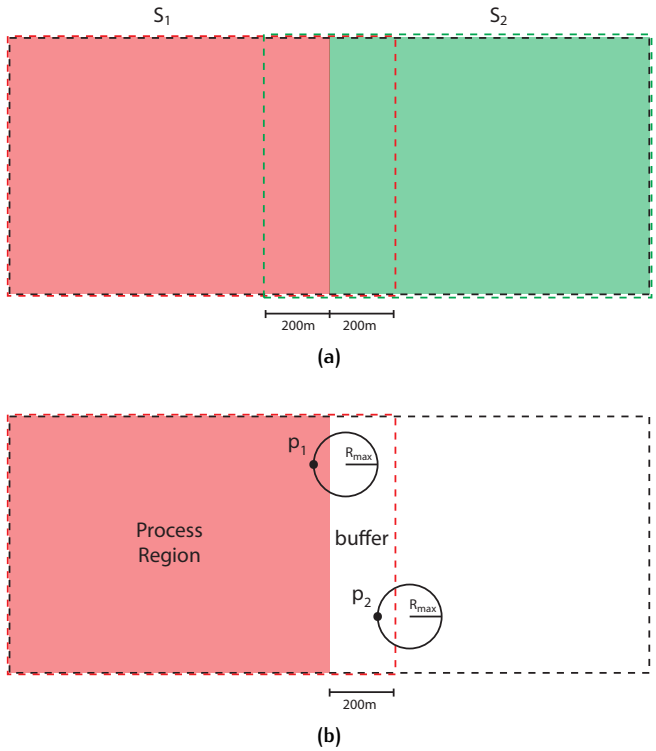


Figure 3.6: (a) To process the subset S_1, extra data is needed expressed by the dashed red line. For the Subset S_2 the same holds, it needs extra data as expressed by the dashed green line
(b) Subsets are collections of points which will be processed to retrieve the MAT and points which just function as a buffer. The process points and buffer points fall in the region bounded by the red dotted line. As shown by Points p_1 and p_2, points within the buffer region can not be processed because they might need points from outside the subset

As concluded in Section 2.4.3 (Starting Radius r), the starting radius for the current work is 100 meters. Therefore the medial ball has the possibility of being in the middle of points up to 200 meters apart. As such, the buffer region around the subsets needs to be 200 meters as well (image 3.6). The

MAT in the buffer regions will not be computed, the buffer region points are merely there to assist the MAT computations within the subset.

3.4 REDUCED BUFFER

As mentioned previously, the two subsets have their own separate, non-overlapping buffer (Figure 3.6). In the example 400 m of buffer is used. However, there is a way to reduce the buffer size, as it is possible to determine which points do not have a *final* MAT.

By determining which points are not final, the MAT of points in the buffer region could be computed as well. This is done by tagging which points in the buffer region are unfinished (points within the process region are certainly finished). When these tagged buffer points are used in another subset, they can be processed further.

Observe Figure 3.8, subset S_1 and S_2 share the same buffer region of 200 m, therefore they will both load this buffer region when they are processed, this is called a reduced buffer. Let us assume that subset S_1 will first be processed including the buffer region, sequentially followed by subset S_2. Figure 3.7, shows two points in the buffer region. Point p_1 has a *final* MAT because its medial ball lies within the bounded region of the subset. The MAT of Point p_2 is *not final* as its medial ball lies partially outside the bounded region. When subset S_2 is computed, the MAT of p_2 is processed further, while the MAT of p_1 will remain untouched.

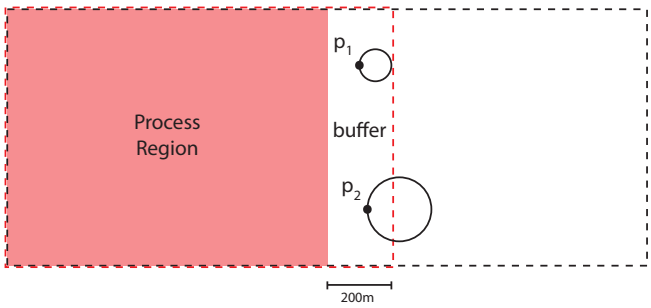


Figure 3.7: Determining which MAT in the buffer region is *final*

An advantage of using the reduced buffer instead of regular buffer, is that it is more memory efficient. Larger parts of the subsets could be removed after it is processed. In Figure 3.8, the process region of subset S_1 could be deleted after it has been computed with the shared buffer. The buffer is already pre-processed using S_1, so when it is computed again with subset S_2 there will be no boundary issues. This aspect is especially useful in cases where the subsets are small but have relatively large buffers.

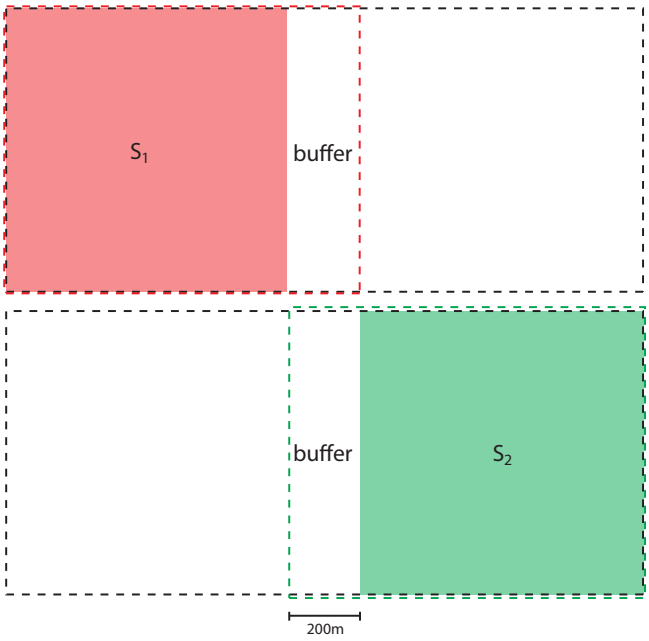


Figure 3.8: Subset S_1 and S_2 share the same buffer

3.5 THINNED REDUCED BUFFER

The problem of points from subset A needing points from subset B cannot be resolved, It is possible however, to analyse which points do not need points from outside its own subset. Furthermore it might be possible to detect which points inside a subset might not be needed for analysing points in another subset. In section 2.2 two properties of a medial ball are mentioned. Using those properties the following assumptions can be made:

1. *The medial ball is completely within the boundaries of a shape.*
   This means that there are no points outside the ball which can contribute to the computation of the MAT. Therefore a point which has medial ball that does not cross the boundary of the subset does not need any data from outside the subset. (Also explained in section 3.2)

2. *The medial ball is tangent to more than one boundary point*
   This could be interpreted as: the centre of medial ball lying on a line in the direction of the normal vector of a point. See Figure 3.9, where medial point c_1 lies on a line in the direction of the normal vector of point p_1 and p_2. When this is translated to a pointcloud, it means that for each point there is only one medial ball that is shared with points which are equidistant from the center of the medial ball ($d(p_1, c_1) = d(p_2, c_1)$). Therefore it can be said that for the surface points which form a medial ball lying completely within the boundaries of a subset, no points from outside sets will need these points to create another medial ball, see Figure 3.9.

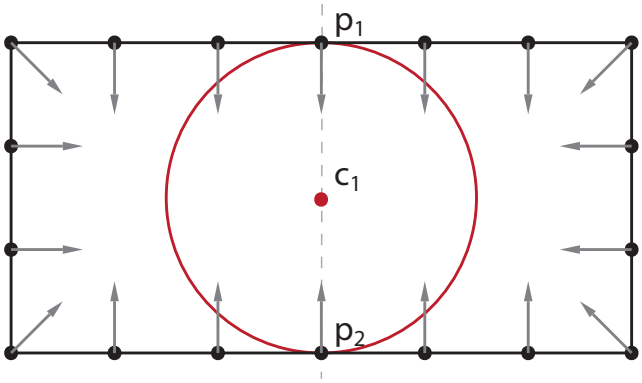


Figure 3.9: The medial ball is tangent to more than one boundary point

If it is known which MAT is final in the buffer region and which points are not needed any more in the computation of the MAT for other tiles, then these points can be removed from the dataset. In doing so a thinned buffer (i.e. a buffer which has points removed) will remain, which has two main advantages:

1. Reduction of memory usage
   As points are left out of the buffer region, because they are unnecessary, the main memory usage during MAT computation will reduce.

2. Reduction in computation time
   The creation of a kd-tree takes less time when less points are used. As nearest neighbour quering has a O($logn$) complexity with n points (Section 2.1.2), there is a slight speed up as well.

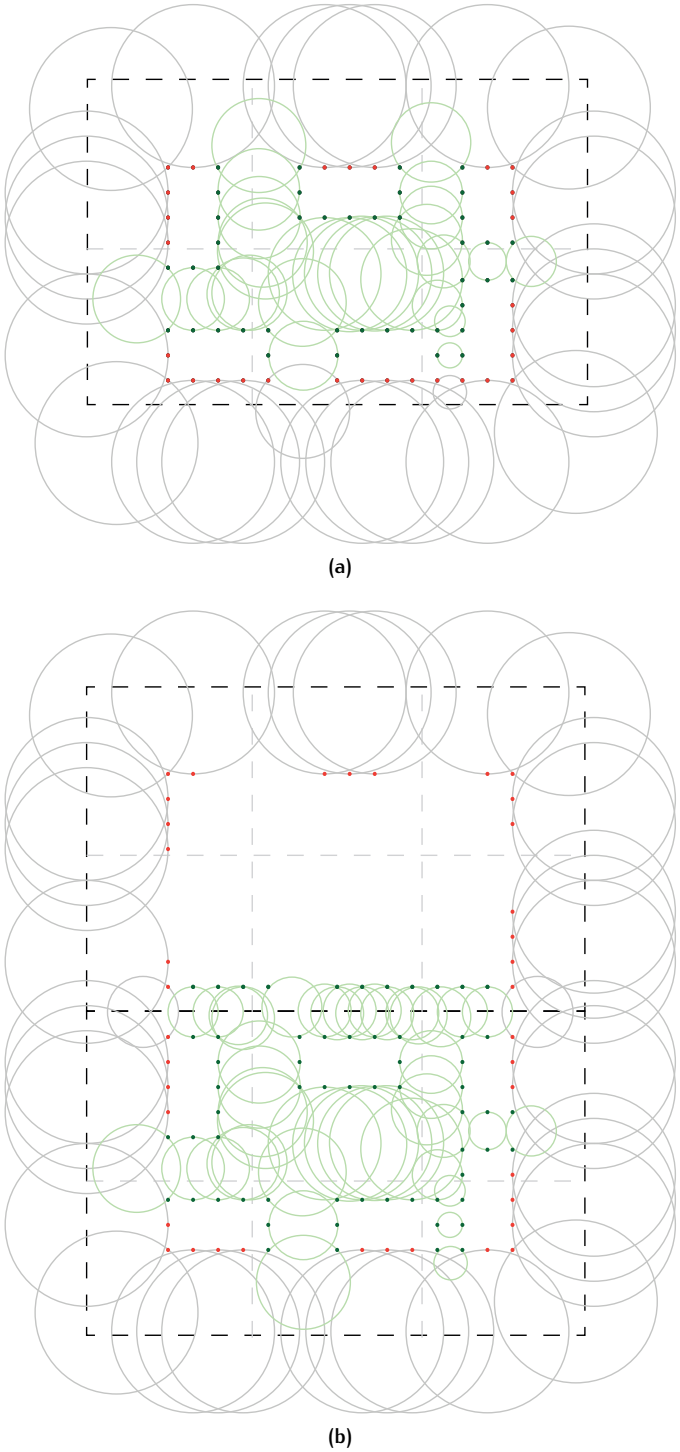


Figure 3.10: The red points might need points from other tiles to compute the final MAT. While the green dots already finished their MAT. As can be seen, the green points which have a final MAT in (a) are not needed to be able to compute the MAT of points in the additional tile added in (b). The MAT of the green points in (a) are not effected by the additional points in the new tile from (b)

3.5.1 Outliers due to making use of a thinned buffer

The previous assumptions based on the properties of the MAT work for multi-dimensional objects, as they are meant for surfaces (or higher dimensional objects). However, they do not always apply for pointclouds. The following two issues will cause deviations when using the shrinking ball algorithm on pointclouds:

- The medial ball is tangent on more than one boundary point on Surfaces and higher dimensional objects. In pointclouds the medial ball touches two points as well, but the shrinking ball algorithm only assures that the medial ball is tangent to one point.

- The normal vector computations of points are just an estimation (See section 2.4.3 "Normal calculation").

3.5.2 Data quality

The computation of the MAT is the same as with the buffer version, however, all the points will be processed. Afterwards some extra processing is needed to check which MAT computation is finalized.

check for (un)finished points

The hypothesis is that points adhering to the following two rules can be left out of the dataset without influencing the result of the computation of the MAT for other points:

1. If a point has medial ball that does not cross the boundary of the subset, it does not need any data from outside the subset.

2. If rule 1 is true for a point p_1, there are no other points from other subsets which need to use p_1 for the computation of the medial ball.

This means points of which both the inner and outer medial ball are completely inside boundary of the subset, do not need to be used any more by the process, not even for other tiles. Therefore these points can be omitted. Points of which either the inner or/and outer medial ball lie partially outside the subset, might be needed to process points from other subsets, these cannot be omitted.

Datasets were preprocessed to test the hypothesis in the following manner:

A cluster of 3x5 tiles is divided into subsets of 1x5 and 2x5 as shown in Figure 3.11 (a) and (b). After the division the MAT is computed for the 1x5 cluster. The points which have a medial ball along the right border of the cluster are stored (c and d). Then the 2x5 cluster and the points of (d) are combined as shown in (e).

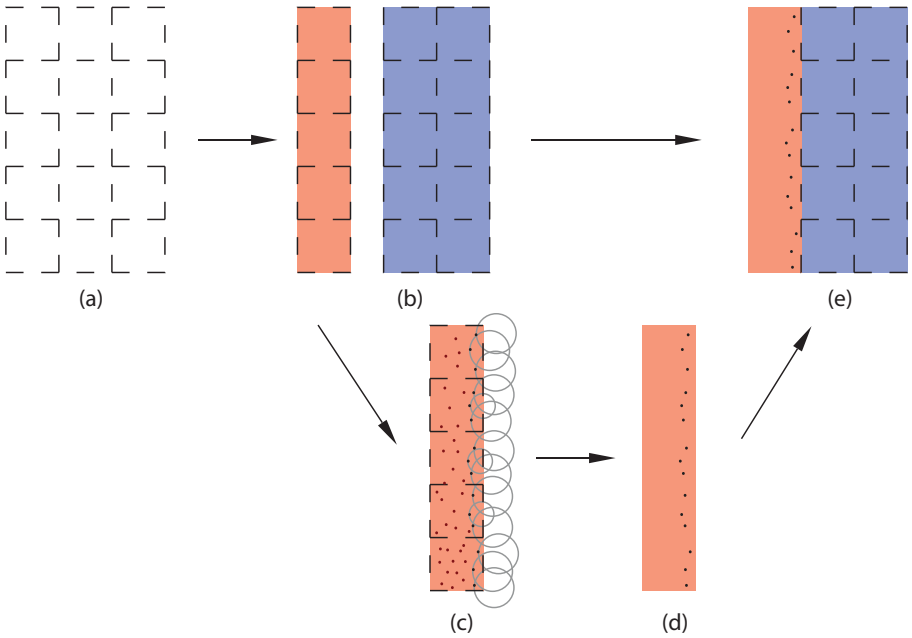


Figure 3.11: (a) Dataset consisting out of 3x5 tiles containing points
(b) Dataset is split up into 2 subsets (1x5) (2x5)
(c) The MAT of subset (1x5) is computed
(d) Based on the medial balls, it is decided which points may be left out of subset (1x5). Points near the right border have a higher chance that their medial ball cross the right boundary.
(e) The reduced subset (1x5) and complete subset (2x5) are combined to compute the MAT

The MAT will be computed for the whole resulting preprocessed dataset, described in Figure 3.11 (e). The middle part of it will be compared to the original dataset as described in Figure 3.12.

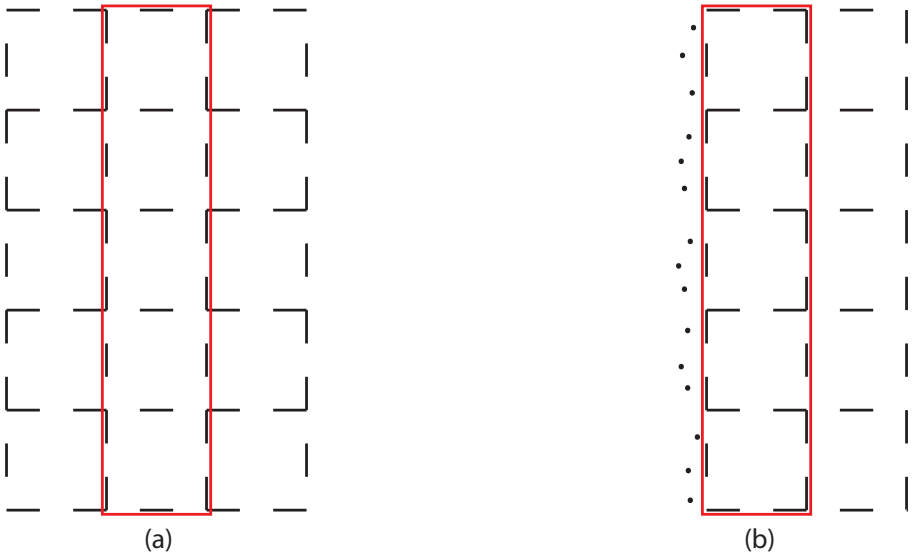


Figure 3.12: The center tiles of datasets (a) and (b) are compared to each other to test the hypothesis
(a) Original dataset
(b) Thinned dataset from Figure 3.11e

This testing procedure has been applied to the fictional pointcloud dataset as well as the Rotterdam dataset (see Section 5.2). From the analyses of the middle part it can be seen that 0.04% of the MAT has not been correctly processed for the Rotterdam subset (see Table 3.5). Errors defined as: MAT computed using the reduced thinned buffer method, which are unequal to a MAT computed without reduced thinned buffers.

	Rotterdam Dataset
MAT points	5946483
Errors outer MAT	1692
Errors inner MAT	901

Table 3.5: Errors in MAT calculation with preprocessing

Figure 3.14 displays the original dataset as well as the preprocessed dataset which will be compared to each other in order to find outliers. The preprocessed dataset clearly has fewer points in the tiles on the left side, it has been reduced in size by a half. Furthermore several objects have been removed as well.

When both of the mentioned datasets were processed some deviations in their MAT have been detected. Both the inner and outer MAT have deviations as can be seen in Figure 3.15). While most deviations are smaller than 5 meters, there are a few extreme errors reaching to 70 meters. Furthermore the inner MAT has less errors than the outer MAT. The reason for this is that the chance that a point computation actually needs another point from outside the boundary is much larger with the outer MAT. This is because the inner MAT has much less chance to compute a valid MAT, as there are less objects *underground* in a geographic dataset (see Figure 3.13). *Underground* there are less objects to create the MAT with, consequentially the MAT will not actually be computed. The result is a so called *invalid* MAT: a MAT where the radius is the initial radius (200 m).

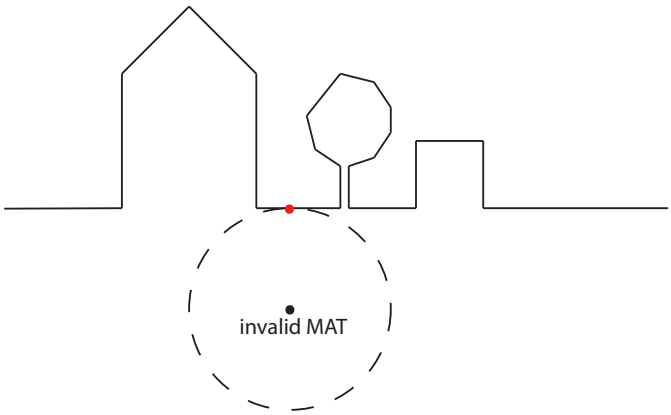
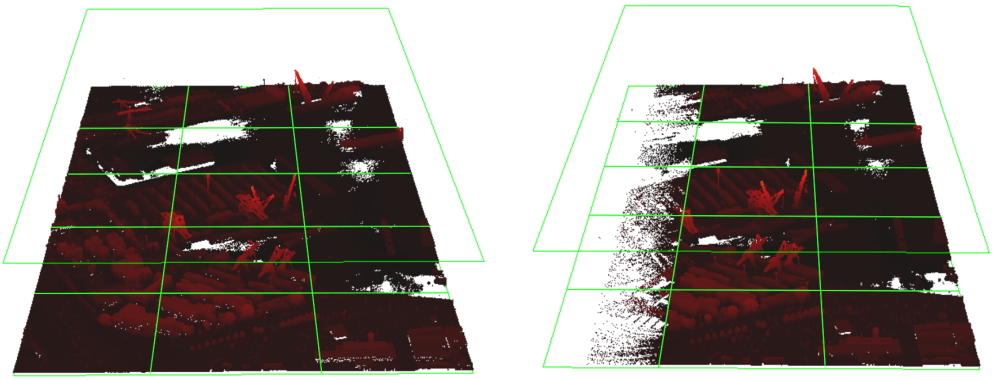
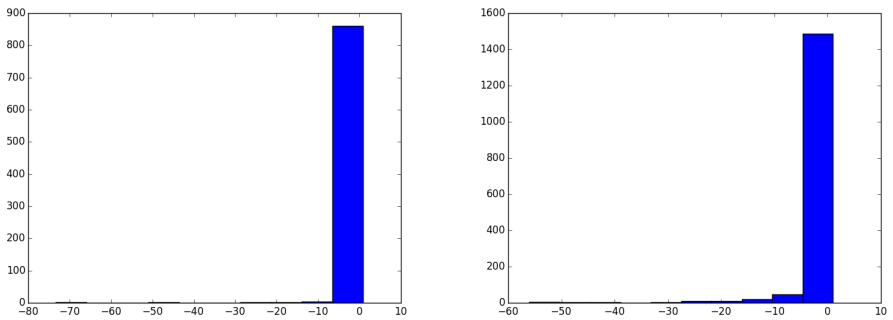


Figure 3.13: Invalid inner MAT of a point



(a) (b)

Figure 3.14: Rotterdam Dataset (a) Original dataset (b) Preprocessed dataset



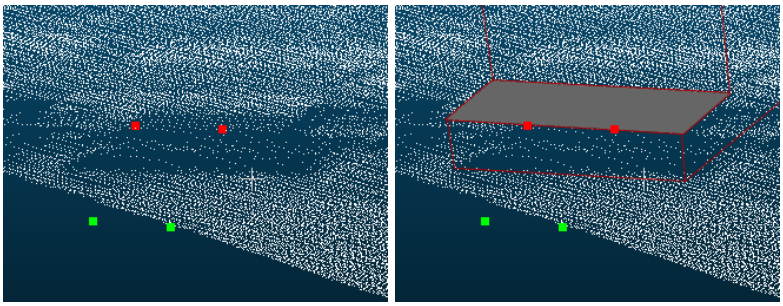
(a) (b)

Figure 3.15: Rotterdam Dataset (a) Inner medial axis deviation (b) Outer medial axis deviation

3.5.3 Thin objects

Many errors occur due to thin objects. These are objects which are represented by a set of points which lie on a planar surface. These are objects which are either thin in the real-world or objects which are represented in the dataset as a thin object. A fence expressed in points is for instance a thin object, as it lies on a planar surface and has a width of only one single point.

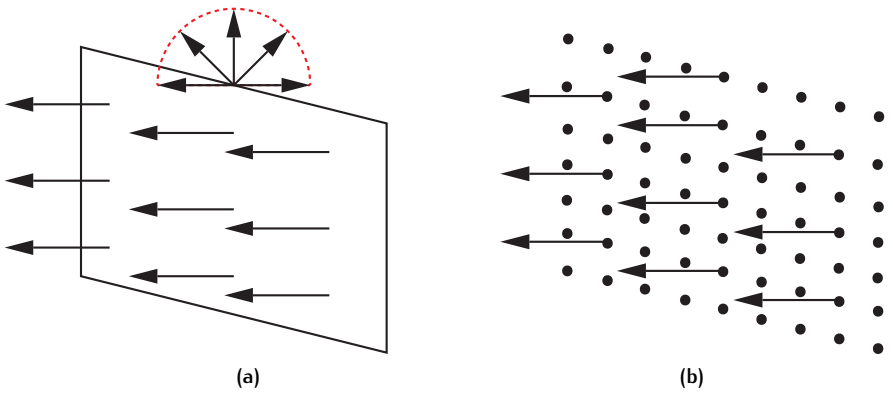
Although a building itself is not considered as a thin object, it could be that it is *represented* as a thin object in the pointcloud. Due to data capturing method, some objects are not translated well from the real-world. Pointclouds are often collected by means of an air plane, data is then captured from above, as such the horizontal planes (i.e. roofs) are well represented, however, the vertical planes (i.e. walls) are not represented equally due to the capturing angle. So although buildings do not classify as a thin object, their roofs often got a good point density, while the walls consist of considerably less points in the dataset.



(a) (b)

Figure 3.16: (a) The roof has been thinned away, because of incorrect normal estimation (b) Walls visualised

Figure 3.16 shows a situation where this forms a problem. The MAT of points on a roof near the boundary of the tile, might not be computed correctly. This issue is due to the miscalculation of the normal vector, as it is impossible to have a normal vector at the border points of the roof in case it was actually a 2D plane. As Figure 3.17 (a) shows, the plane itself has a normal pointed perpendicular to the plane. It is not possible to compute the normal for the edge of the plane, although it is imaginable in the case for computing the MAT that the edge contains a set of normal vectors pointing in all directions within a range of 180 degrees. Figure 3.17 (b) shows the normals in case the planar plane is converted in to points. All the points will have a normal direction perpendicular to the plane, when they are computed using the method described in section 2.4.3 "Normal calculation". As can be seen, the edges will also get a normal vector even though it does not actually exist in the 2D representation of the plane. As the MAT is used to determine whether a point should remain in the dataset or whether it is obsolete for further computations, this forms a problem. So although the MAT computation method might be correct, using an incorrect normal vector, the MAT result will be wrong.



(a) (b)

Figure 3.17: normal calculation of planar plane

3.5.4 Outliers in the data

The large deviations in MAT visible in the histograms (Figure 3.15) are due to outliers in the original dataset. As the red dots in Figure 3.18 show, there are points located above the city without any other close neighbours. These

points are not part of a larger object (i.e. it could be a bird) and therefore considered outliers. Apart from that they are outliers, their normals cannot be computed correctly as they do not have any close neighbours. Occasionally the normal computed is unfavourable, as the outlier might be removed in the preprocessing. This will result in large deviations in the MAT calculation. These points actually often get filtered out by the reduced buffer method. While it can be discussed that leaving the outliers out of the computation is actually the right way to process the MAT, doing it using this form of processing is not the correct way. Several of these outliers will be removed by chance, but not all of them, as such it is not a reliable way to filter outliers.

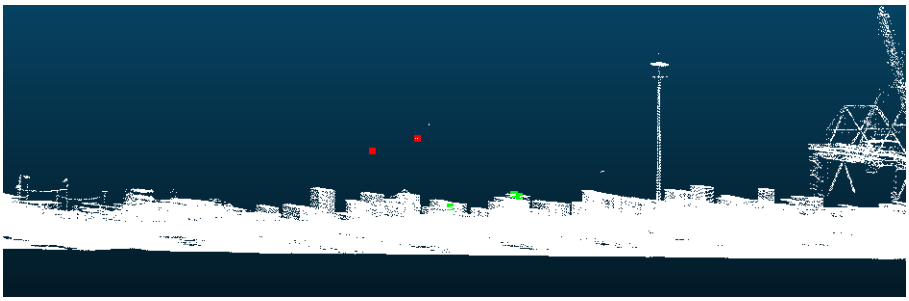


Figure 3.18: Outliers in the air

3.5.5 Noise in the Data

Several deviations in the MAT are due to noise. As explained in section 2.4.3 noise can have a huge influence on the computation of the MAT due to its inherent sensitivity. Several of the red dots have been removed (through the use of reduced buffers), while the green dots needed them for the computation of the MAT. Whilst the computed MAT would not have been necessarily correct, given that the red as well as the green points are part of the same plane. However, as the point removal is not consistent with every noisy point, it is not a reliable method.

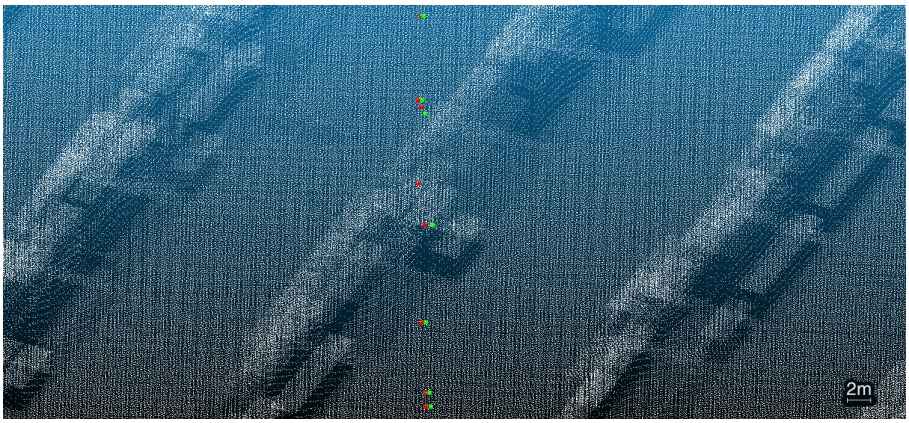
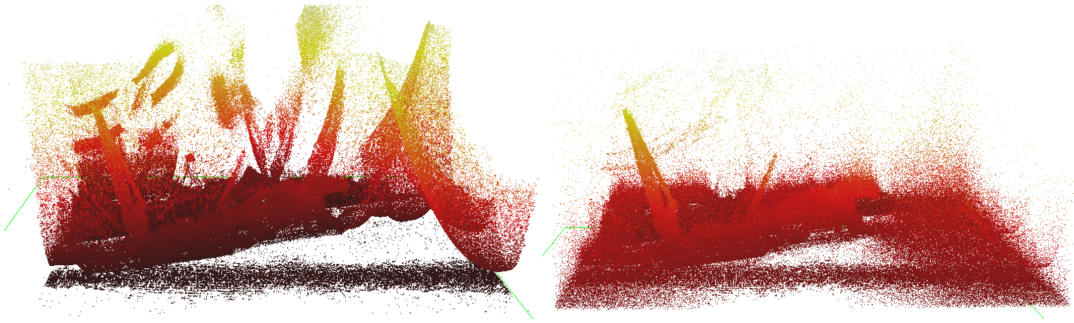


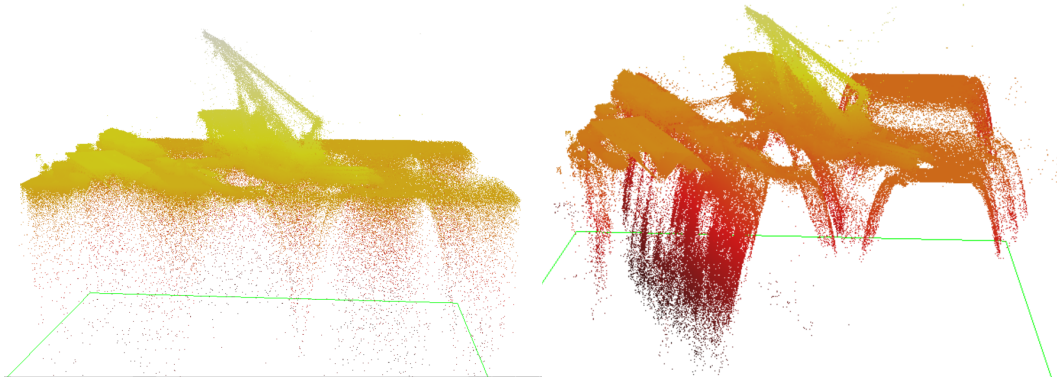
Figure 3.19: Noise in the dataset

Denoising is possible and actually necessary (as shown in section 2.4.3), as it improves the quality of the formed MAT drastically. This can be seen in Figure 3.20 for the outer MAT and in Figure 3.21 for the inner MAT.



(a) (b)

Figure 3.20: Outer medial axis deviations of the Rotterdam puntenwolk dataset (a) with noise reduction (b) without noise reduction



(a) (b)

Figure 3.21: Inner medial axis deviations of the Rotterdam puntenwolk dataset (a) with noise reduction (b) without noise reduction

However, denoising has a negative influence. During the MAT computation several iterations take place: in each iteration, a new medial ball is computed smaller than the previous ball. When the separation angle between a medial point and its two surface points is smaller than a certain threshold it will be considered an outlier. In that case the previous medial point will be used. As large amount of points are removed from the dataset by the preprocessing, there is a chance that the previous MAT is different than when all the points are there.

Furthermore, the chance that differences between two consecutive separation angles exceed the maximum threshold becomes larger the more points are left out, and the larger jumps between consecutive are.

In appendix A the same test has been performed on a fictional dataset. However, it has smaller and less errors. The difference in amounts of errors may be because the datasets are different. The fictional dataset does not have any noise and the points are homogeneously spread. Although it is hard to quantify the error causes, it seems that many errors come from the noise and the lack of points in certain areas.

3.6 SUMMARY

In this section the Shrinking ball algorithm to compute MAT was examined to evaluate its scalability. As boundary issues will occur when the MAT is computed for just one subset, three forms of buffers were introduced to eliminate that issue:

- (regular) buffer
  The miskowski sum of a 2D disc with radius r and the subset. This is done so that the point near the boundary of the subset will still have the opportunity to compute the MAT with the extra buffer with size r around the boundary.
- reduced buffer
  Unlike the 'regular' buffer, points in the reduced buffer get processed as well. By tagging which points are *final* and storing their MAT computation progress, points in these buffers will be finalized by processing them together with multiple subsets.
- thinned reduced buffer
  The hypothesis was that if points have a *final* MAT, they do not have to be used by points from another subset to compute the MAT. However, this turns out not to be true in on every occasion for point clouds, as their normals can not always be correctly computed. Because the thinned reduced buffer cannot always provide the correct MAT, it is not usable in its current state.

4 SCALING THE MEDIAL AXIS TRANSFORM

This research seeks to scale the MAT for geographic pointclouds. The previous chapter discussed the shrinking ball algorithm properties which will affect the scaling by using subsets of a dataset. This chapter will continue with the actual scaling of the shrinking ball algorithm.

As the input data is restricted to geographical pointclouds, certain advantages come along. A property of these kinds of datasets is that they are mostly flat, as they represent real world area's. The points are well spread in the x and y direction while having a relative low spread in the z direction. Because of the limited differences in the z direction the datasets can be cut as if they were 2 dimensional (see Figure 4.1).

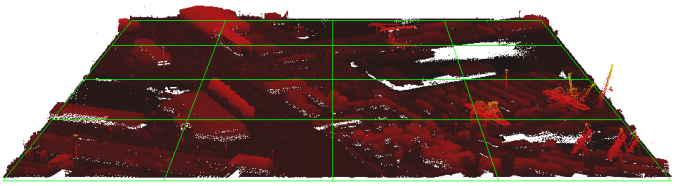


Figure 4.1: Subdividing a Geographic Pointcloud using 2D grid

Two approaches to chunk the dataset into processable parts are introduced: tiling and streaming. These approaches are different because:

- The tiling approach subdivides the dataset in to smaller datasets, storing them temporary on the external memory.

- The streaming approach reads the input file several times and tries not to create temporary smaller datasets on the external memory (preferably the whole process apart from reading the input data and writing the output data takes place inside the main memory).

Both of these methods split a large dataset in to tiles to compute the MAT. The size of these tiles are depended on the buffer size chosen, which is 200 meters in this case as discussed in section 2.4.3 "Normal calculation". To compute a single point an area of 400 x 400 m is needed as shown in Figure 4.2.

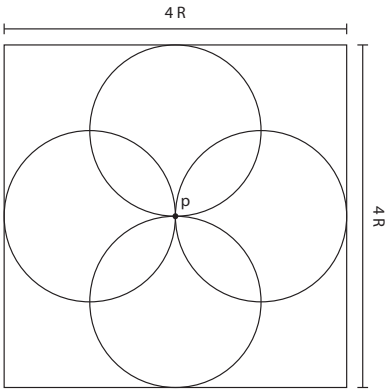


Figure 4.2: To compute the MAT for a single point p, it should have a region buffer containing surrounding points. This region buffer should be 2*r_{init}

4.1 APPROACH: TILING ALGORITHMS

The tiling approach is a D&C algorithm. It involves subdividing the dataset in to the minimum sized tiles and then merge them (segmentation) to create processable sets (collections) which will fit inside the memory during the MAT computation (see Figure 4.3). Two methods to create these collections are:

- Creating a grid
  The tiles produced in the tiling process are used to process the MAT (processing individual tiles).

- Creating collections by tree-based segmentation
  Tiles go through a segmentation process in which groups of tiles form a collection, which will be used to compute the MAT.

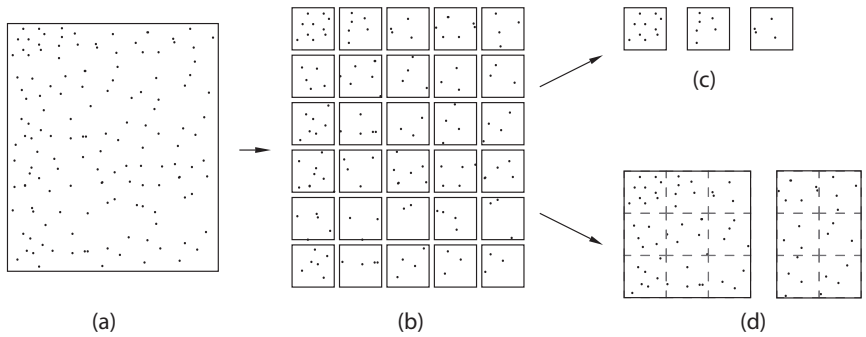


Figure 4.3: segmentation of a dataset to get collections
(a) Dataset
(b) Dataset split up to equal sized tiles (grid)
(c) The gridded tiles are used to compute the MAT
(d) The tiles are merged to form collections to compute the MAT as a set

To avoid boundary issues, buffers are placed around the subsets in the collections (see Figure 4.4). In chapter 3, two buffer types were introduced:

- "Regular" buffer method
  Buffer tiles surround the processable tiles, however, the MAT will not be computed for the points in the buffer tiles

- Reduced buffer method
  The MAT for points in the buffer will be computed as well and tagged to determine whether they are *final* or not.

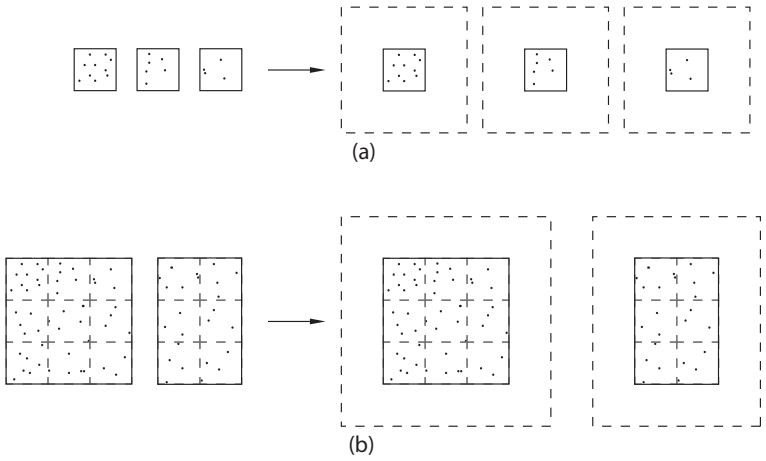


Figure 4.4: Buffers around the process tiles
(a) tiles with buffers
(b) collections with buffers

The use of the reduced buffer method will reduce the amount of collections made during the segmentation part, which should in general mean faster processing time. However, several tiles will be computed multiple times, which will slow the process down, as explained in section 3.4. Figure 4.5 shows the possibilities to use the tiling process on a geographical dataset to compute the MAT.

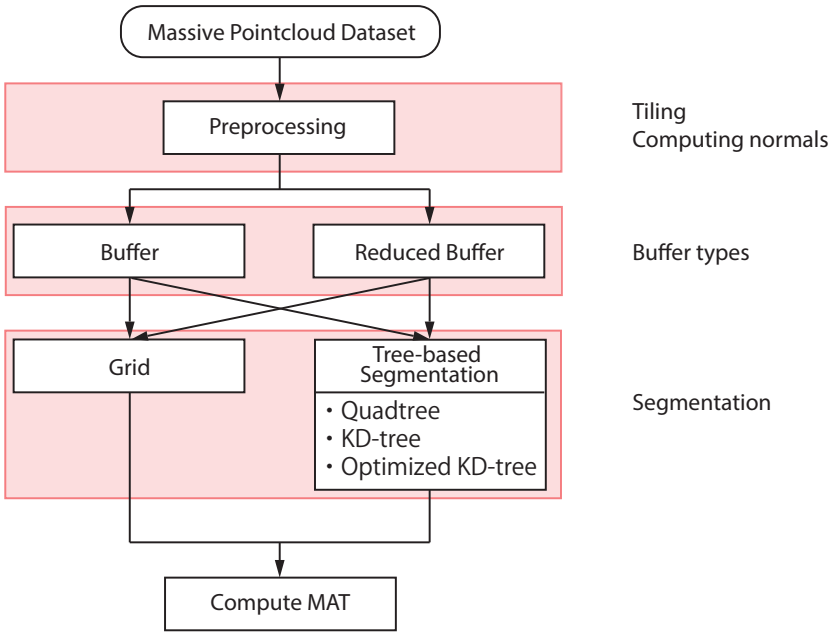


Figure 4.5: Workflow tiling process

4.1.1 Preprocessing

As determined earlier the minimum buffer is based on the maximum size of the objects which should be detected. In section 2.4.3 it was said that this should be 200 meters, as built objects in the Netherlands are usually lower or more slender than that. The complete dataset will therefore be subdivided in subdatasets of pieces of 200m x 200m. These tiles will be the smallest pieces of which collections are made of (See Figure 4.6).

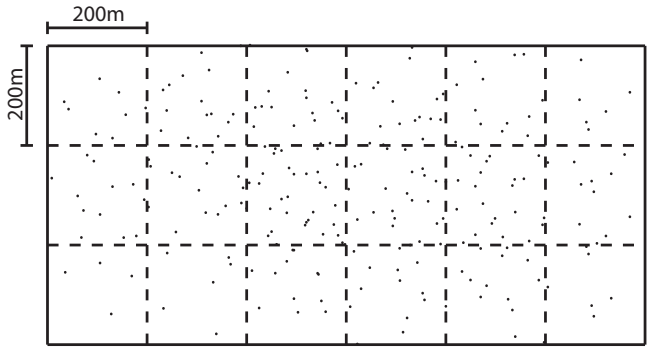


Figure 4.6: dataset subdivided in 200m x 200m tiles

The coordinates could contain high numbers, this is when the chosen coordinate is far away from the origin, which is the case when datasets are used with coordinate reference system: RD New (Section 5.2). To reduce the amount of digits of the numbers in the coordinates, the center of the dataset is chosen to be the origin of the local reference system for the x and y direction (e.g. a part of the AHN3, see table 4.1). By doing so, the absolute value of the x and y is minimal. If all the values can be expressed using six or less significant decimal integers, 32 bit floating point precision values could be used (IEEE Computer Society [2008]). This has an advantage that it is twice smaller in size than using the more precise double precision format. As can seen from the given example in table 4.1, if the pre-normalized x and y values are expressed in 32-bit float, they will have dm precision, while the post-normalized values are on mm precision.

		x	y	z
pre-normalized	min	50000.0	357824	-1.280
	max	54999.9	362499.9	71.136
post-normalized	min	2500.00	-2337.81	-1.280
	max	2499.99	2337.81	71.136

Table 4.1: normalizing coordinates

Normal vector calculation

The normal vectors of the points are calculated by simply computing it per created tile. To reduce errors near the borders of each tile, a buffer using other tiles is taken in to the computation as well, see Figure 4.7. However, as the normal calculation is just an estimation, small errors may occur (see section 2.4.3 normal calculation). This process could be optimized, but it is considered out of scope.

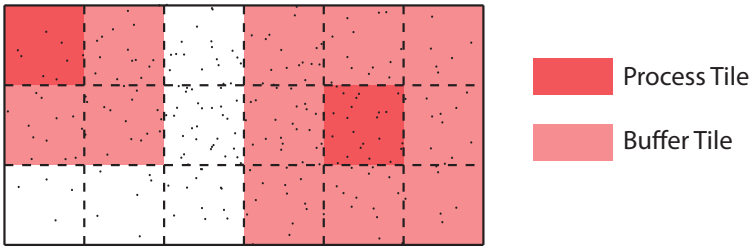


Figure 4.7: Buffer regions

4.1.2 grid-based segmentation (buffer method)

The method to compute the MAT using the grid based segmentation is similar to the normal computation. For each tile the MAT is computed individually, to remove border errors a buffer is placed around it (Figure 4.7). The process tile including its surrounding buffer tiles are processed, while only the MAT of the process tile is computed. This is not a very memory efficient method, as some tiles may contain more data, while others may contain less. This is due to that points in a dataset is are not necessary spread homogeneous.

4.1.3 tree based segmentation (buffer method)

To group the tiles in to larger collections, a space driven or data-driven segmentation can be used. The advantage of segmentation on tiles is that the process is fast. An AHN3 tile (dataset) of 5000 x 6250 m will be subdivided in 800 tiles with the size of 200 x 200 m. The segmentation of 800 tiles to create collections is much faster than grouping points in the original point-cloud which may contain millions of points. Space driven and data driven segmentations have their own advantages. The advantage of a space driven segmentation is that it creates subsets which have the same aspect ratio in size as the original dataset. A data driven segmentation assures that the each subset contains roughly half of all the points of its parent. In Figure 4.8 a data-driven segmentation has been performed on an AHN3 dataset, collections differ from shape and size. This is the effect of the data-driven segmentation, as all the collections contain roughly the same amount of points. This means that the points are not homogeneously spread over the dataset.

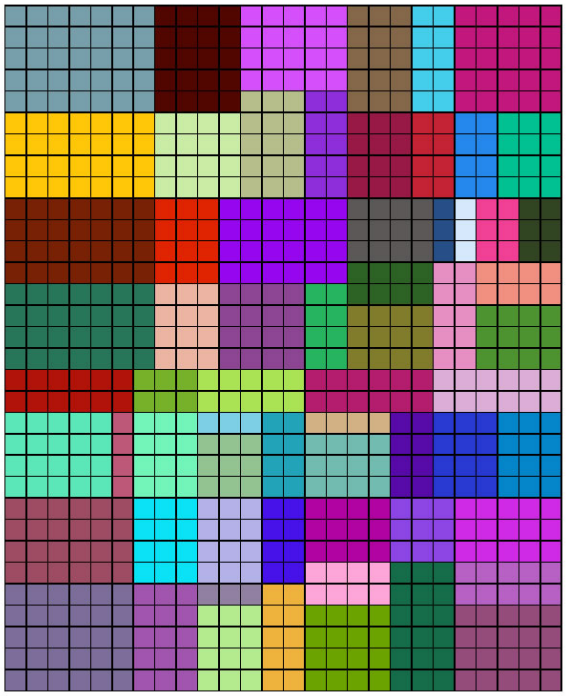


Figure 4.8: Data-driven segmentation performed on AHN3 dataset c_37en2
Different colours represent different collections, buffers are not included in the image

Segmentation

The output of the segmentation should be several collections S_i which are subsets of the dataset (where i is the number of the subset). Each collection contains 2 lists:

- List of Process Tiles (P_i)
- List of Buffer Tiles (B_i)

Both of these lists contain tiles $T_{x,y}$ where x and y are their minimum x and y value of the points in the tile respectively. The process tiles and buffer tiles are in separate lists, as the points of the buffer tiles will not be processed.

By grouping the smaller 200m x 200m tiles larger collections can be formed. We start by taking all the tiles as a single collection S_0 and keep dividing it in to smaller collections until all the tiles of the sets of collections fit inside the main memory during processing of the MAT. To decide the splitting location 2 spatial data structures are proposed, a space-driven one (based on a quadtree) and a data-driven one (based on a kd-tree).

The method to subdivide the total dataset in to smaller subsets using the top-down Level-by-Level segmentation is described in section 2.6.1. By making a small adjustment on the algorithm it can be applied so that it will create collections with buffers with the use of tiles.

Let us assume a dataset of 1600m x 1600m with the minimum x and y values of the points being 0, the dataset will first be tiled in to smaller tiles of the size 200m x 200m (see Figure 4.9). Therefore will be 64 tiles $T_{x,y}$ where x and y range from 0 to 1400 with intervals of 200. The amount of points in each tile will be stored temporary and linked to that certain tile.

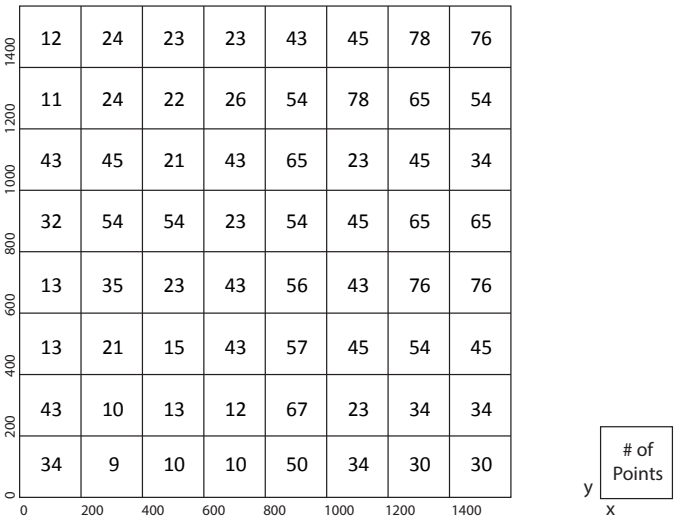


Figure 4.9: Dataset split in 64 tiles, number in each tiles represent the number of points it has

The method starts with the creation of the initial collection S_0, which has all the 64 tiles in the *process tiles* list P_0. If the amount of points of N (*process tiles* P_0 + *buffer tiles* B_0) is larger than a predefined maximum k_{max}, the collection should be split up in to 2 new collections (S_0 and S_1, where the original S_0 is replaced by a new collection). Both newly created collection collections will contain a part of the "process tiles" P_0 of the original collection, so that P_0 + P_1 contain all the 64 tiles. The newly created collections S_0 and S_1 do have to generate a list of buffers B_0 and B_1 ensuring that the collections are individually processable when the MAT is computed. The splitting of the parent dataset depends on the type of tree data structure used:

- space-driven tree structure
- data-driven tree structure

Algorithm 4.1: TILING APPROACH (S, k_{max})

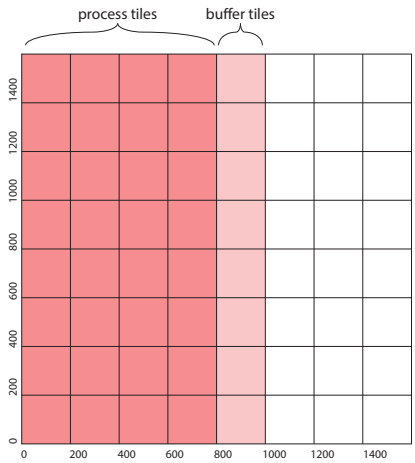
Input: List of collections S, maximum amount of points k_{max}

```
1  Function SEGMENTATION(S, k_max)
2      i ← 0;
3      j ← 0;
4      repeat
5          t ← 1 for each P, B ∈ S do
6              if COUNTPOINTS(P, B) > k_max then
7                  Subdivide S in to P_0 and P_1;
8                  B_0 ← get buffer tiles around P_0;
9                  B_1 ← get buffer tiles around P_1;
10                 Append P_0, B_0 to S;
11                 Append P_1, B_1 to S;
12                 Remove P, B from S;
13                 t ← 0
14     until t == 1;
15     for each P, B ∈ S do
16         COMPUTE_MA_POINT(P, B);
17 Function COUNTPOINTS(P, B)
18     N = 0;
19     for each T_{x,y} ∈ P do
20         N += number of points in T_{x,y};
21     for each T_{x,y} ∈ B do
22         N += number of points in T_{x,y};
23     return N;
```

Algorithm 4.1 shows a more detailed description of the algorithm of the tiling approach.

space-driven subdivision

Let us look at Figure 4.9 again. The dataset is a ready subdivided in to 64 tiles. The process starts with collection S_0, which has all the 64 tiles in its "process tiles" list P_0. If $N > k_{max}$ the collection must be split up into two smaller collections S_0 and S_1. By using a space-driven tree structure it is a simple matter of splitting the dataset in the middle in to 2 pieces. This will be done in the x direction, we will therefore get a list P_0 containing the tiles $T_{0...600,0...1400}$ and a list P_1 with $T_{600...1400,0...1400}$. Because the original dataset is split up, boundary issues might occur during MAT computations, therefore buffers should be added as well to be able to compute the MAT. Buffer tile lists B_0 and B_1, containing $T_{800,0...1400}$ and $T_{600,0...1400}$ respectively, see Figure 4.10.



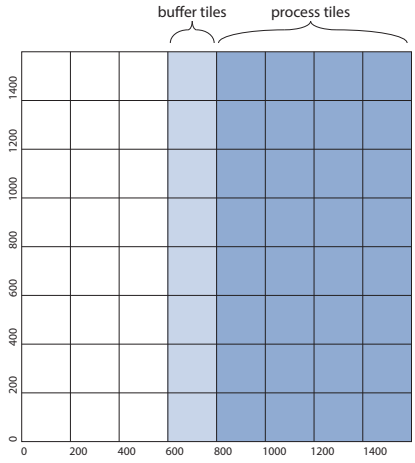
P_0

Process Tiles			
$T_{0,0}$	$T_{200,0}$	$T_{400,0}$	$T_{600,0}$
$T_{0,200}$	$T_{200,200}$	$T_{400,200}$	$T_{600,200}$
$T_{0,400}$	$T_{200,400}$	$T_{400,400}$	$T_{600,400}$
$T_{0,600}$	$T_{200,600}$	$T_{400,600}$	$T_{600,600}$
$T_{0,800}$	$T_{200,800}$	$T_{400,800}$	$T_{600,800}$
$T_{0,1000}$	$T_{200,1000}$	$T_{400,1000}$	$T_{600,1000}$
$T_{0,1200}$	$T_{200,1200}$	$T_{400,1200}$	$T_{600,1200}$
$T_{0,1400}$	$T_{200,1400}$	$T_{400,1400}$	$T_{600,1400}$

B_0

Buffer Tiles
$T_{800,0}$
$T_{800,200}$
$T_{800,400}$
$T_{800,600}$
$T_{800,800}$
$T_{800,1000}$
$T_{800,1200}$
$T_{800,1400}$

(a)



P_1

Process Tiles			
$T_{800,0}$	$T_{1000,0}$	$T_{1200,0}$	$T_{1400,0}$
$T_{800,200}$	$T_{1000,200}$	$T_{1200,200}$	$T_{1400,200}$
$T_{800,400}$	$T_{1000,400}$	$T_{1200,400}$	$T_{1400,400}$
$T_{800,600}$	$T_{1000,600}$	$T_{1200,600}$	$T_{1400,600}$
$T_{800,800}$	$T_{1000,800}$	$T_{1200,800}$	$T_{1400,800}$
$T_{800,1000}$	$T_{1000,1000}$	$T_{1200,1000}$	$T_{1400,1000}$
$T_{800,1200}$	$T_{1000,1200}$	$T_{1200,1200}$	$T_{1400,1200}$
$T_{800,1400}$	$T_{1000,1400}$	$T_{1200,1400}$	$T_{1400,1400}$

B_1

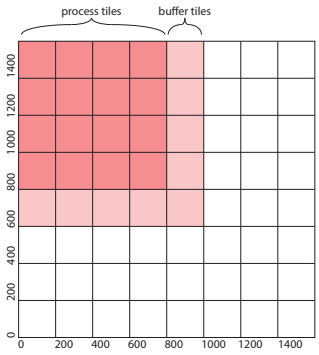
Buffer Tiles
$T_{600,0}$
$T_{600,200}$
$T_{600,400}$
$T_{600,600}$
$T_{600,800}$
$T_{600,1000}$
$T_{600,1200}$
$T_{600,1400}$

(b)

Figure 4.10: Dataset is split up in to 2 collections with an equal amount of tiles:
(a) Collection S_0, with process tiles list P_0 and buffer tiles B_0
(b) Collection S_1, with process tiles list P_1 and buffer tiles B_1

Each of the newly formed collection is then evaluated. If $N > k_{max}$ for that collection, it will be subdivided again [1], this time in the y direction, see Figure 4.11. This will iterate till all collections S_i comply to $N > k_{max}$. Each time a collection is split, the "cutting" direction switches from x to y and the other way around.

1 The subdivision takes place using only the process tiles, the buffer tiles do not have any influence on it



P_0

Process Tiles

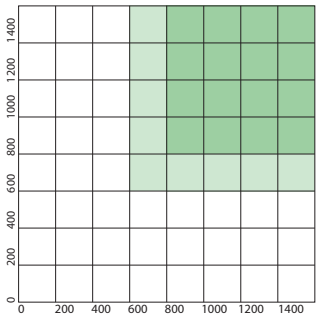
$T_{0,800}$	$T_{200,800}$	$T_{400,800}$	$T_{600,800}$
$T_{0,1000}$	$T_{200,1000}$	$T_{400,1000}$	$T_{600,1000}$
$T_{0,1200}$	$T_{200,1200}$	$T_{400,1200}$	$T_{600,1200}$
$T_{0,1400}$	$T_{200,1400}$	$T_{400,1400}$	$T_{600,1400}$

B_0

Buffer Tiles

$T_{800,600}$	$T_{0,600}$
$T_{800,800}$	$T_{200,600}$
$T_{800,1000}$	$T_{400,600}$
$T_{800,1200}$	$T_{600,600}$
$T_{800,1400}$	

(a)



P_0

Process Tiles

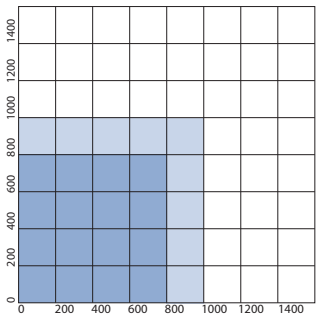
$T_{800,800}$	$T_{1000,800}$	$T_{1200,800}$	$T_{1400,800}$
$T_{800,1000}$	$T_{1000,1000}$	$T_{1200,1000}$	$T_{1400,1000}$
$T_{800,1200}$	$T_{1000,1200}$	$T_{1200,1200}$	$T_{1400,1200}$
$T_{800,1400}$	$T_{1000,1400}$	$T_{1200,1400}$	$T_{1400,1400}$

B_0

Buffer Tiles

$T_{600,600}$	$T_{800,600}$
$T_{600,800}$	$T_{1000,600}$
$T_{600,1000}$	$T_{1200,600}$
$T_{600,1200}$	$T_{1400,600}$
$T_{600,1400}$	

(b)



P_1

Process Tiles

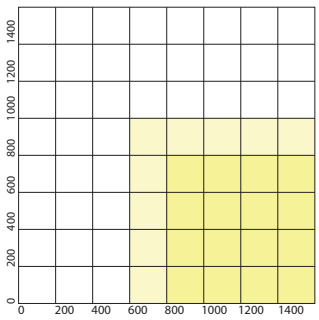
$T_{0,0}$	$T_{200,0}$	$T_{400,0}$	$T_{600,0}$
$T_{0,200}$	$T_{200,200}$	$T_{400,200}$	$T_{600,200}$
$T_{0,400}$	$T_{200,400}$	$T_{400,400}$	$T_{600,400}$
$T_{0,600}$	$T_{200,600}$	$T_{400,600}$	$T_{600,600}$

B_1

Buffer Tiles

$T_{800,0}$	$T_{0,800}$
$T_{800,200}$	$T_{200,800}$
$T_{800,400}$	$T_{400,800}$
$T_{800,600}$	$T_{600,800}$
$T_{800,800}$	

(c)



P_1

Process Tiles

$T_{800,0}$	$T_{1000,0}$	$T_{1200,0}$	$T_{1400,0}$
$T_{800,200}$	$T_{1000,200}$	$T_{1200,200}$	$T_{1400,200}$
$T_{800,400}$	$T_{1000,400}$	$T_{1200,400}$	$T_{1400,400}$
$T_{800,600}$	$T_{1000,600}$	$T_{1200,600}$	$T_{1400,600}$

B_1

Buffer Tiles

$T_{600,0}$	$T_{800,800}$
$T_{600,200}$	$T_{1000,800}$
$T_{600,400}$	$T_{1200,800}$
$T_{600,600}$	$T_{1400,800}$
$T_{600,800}$	

(d)

Figure 4.11: Collections S_0 and S_1 are split up again, both create 2 new collections based on their process tiles. Newly created collections S_i (where is ranges from 0 to 3), are displayed in (a), (b), (c) and (d) respectively

data-driven subdivision

Subdividing the dataset using a kd-tree based segmentation is similar to the previous data-driven segmentation. It only differs in where the cut should be made, as the kd-tree is a data-driven data structure. Instead of the cutting the dataset at the geometric middle of the dataset, it cuts it in such away that both newly formed collections will have roughly the same amount of points. As can be seen from Figure 4.12 the amount of points of both collections are roughly the same, while collection (a) has en-captures a larger area than (b).

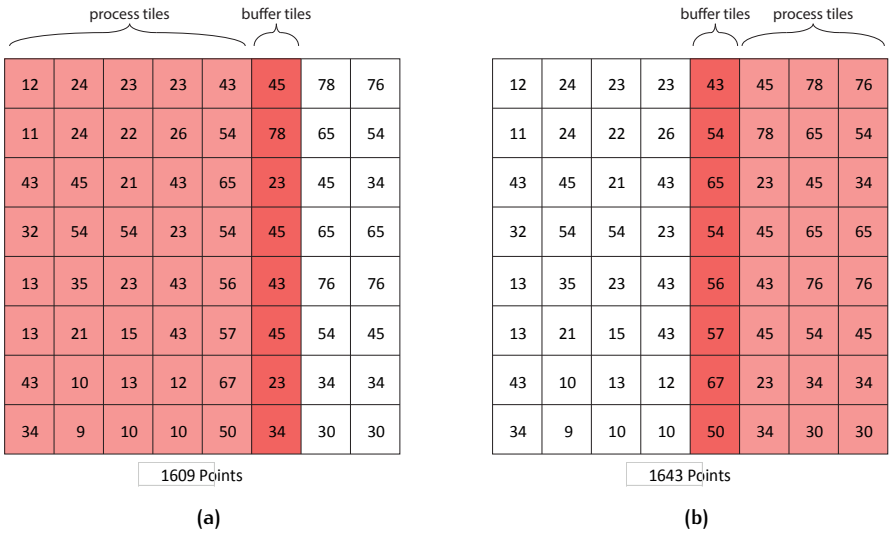


Figure 4.12: 1st subdivision using kd-tree

Optimized kd-tree

As the kd-tree is a data-driven data structure, it does not take in to account the spatial aspect ratio of the formed tiles. However, this can cause *skinny* tiles, as explained in section 2.1.2. When creating segments, square tiles are more preferred. Observe Figure 4.13, the use of the *skinny* collection (a) will need 33 tiles to be loaded, while the square collection (b) only needs to load 25. Because *skinny* subsets have more buffer tiles than square shaped subsets, splitting a dataset will in general lead to more subsets if they are skinny instead of square. This will result in to more collections and longer processing time when actually computing the MAT. Therefore square subsets are preferred. As mentioned in section 2.1.2, the optimized kd-tree generally will lead to more square subsets.

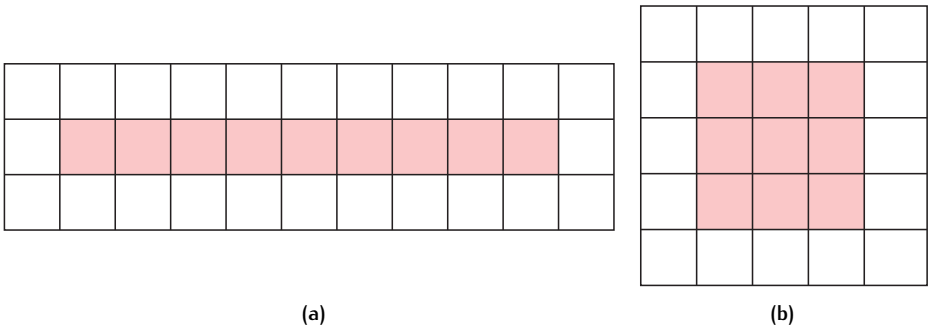


Figure 4.13: (a) skinny collection, containing 33 tiles (b) square collection, containing 25 tiles

Computing MAT

For P_i in each collection S_i the amount of points $N_{process}$ is calculated before the corresponding buffer in collection B is appended to it. The MAT is then computed for each collection for the first $N_{process}$ points in the dataset. This way the shrinking ball algorithm knows that only the first few subsets need to be processed, while the rest of the collection are the buffer tiles (see Section 2.4.3). After each tile out of the collection is processed, the MAT is directly stored on the hard disk.

4.1.4 reduced buffer

The segmentation process using a reduced buffer (section 3.4) is essentially the same as the regular buffer method. It only uses a smaller buffer region, see Figure 4.14. The MAT computation is different, while in the regular buffer method the MAT is not computed for points in the buffer region, the reduced buffer method does compute the MAT. This means that there will be errors in the boundary as the MAT might need points from neighbouring tiles to be computed. This is solved by computing the MAT in the buffer region multiple times with multiple collections using that buffer as explained in section 3.3. Disadvantages will be that the processing time might be increased due to that the buffer tiles need to processed multiple times.

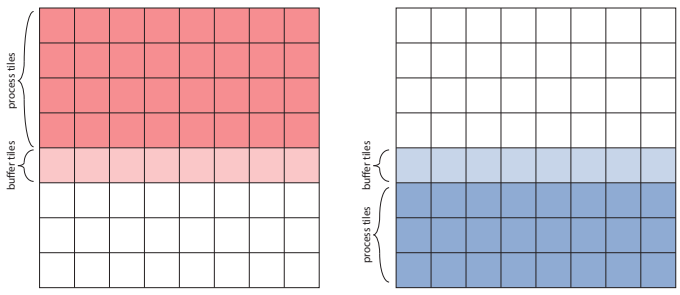


Figure 4.14: shared buffer

4.1.5 Space complexity

The tiling method makes extensive use of the internal and external memory.

Internal memory

The memory usage is directly related to the amount of points in each collection. As mentioned in the introduction of section 3, processing 1 million points uses roughly 71 MB of memory in case all those points were to be processed.

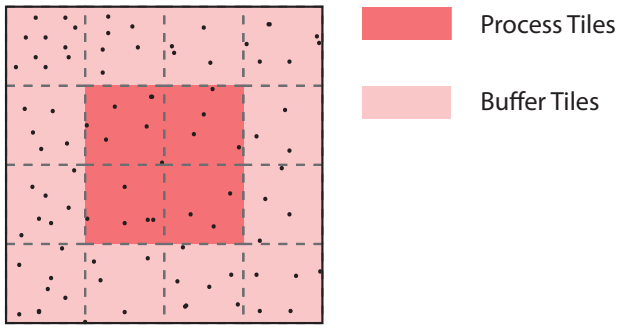


Figure 4.15: 4x4 collection of tiles: 4 process tiles, 12 buffer tiles

Let us assume a cluster consisting of 4x4 tiles of 1 million points each (see Figure 4.15). In this case 12 tiles are buffer tiles and 4 tiles are process tiles. The complete dataset of coordinates ($3 * 4$ byte floating point) needs to be loaded in. The KD tree needs to be computed for all these points as well (35 bytes per coordinate), however only the normals for 4 processable tiles need to be loaded. And the MAT output can be stored and thus removed from the memory after each tile is processed. In total 816 MB of main memory is needed to process these 4 million points, see table 4.2. The amount of main memory usage grows more or less linearly with the amount of input points.

Type	Points	Coordinates	Normals	KD tree	MAT
Process Points	4 million	48 MB	48 MB	140 MB	16 MB
Buffer	12 million	144 MB	0 MB	420 MB	0 MB

Table 4.2: MAT computation memory usage

External memory

The tiling approach makes extensive use of the external memory, as the temporary tiles as well as normal vectors are stored. This results in the need of twice the space the original dataset uses. However, the inner and outer MAT could be up to 33% bigger than the tiles and normal vectors. The tiles and normal vectors are 3 32-bit floating numbers {x, y, z}, while the inner and outer MAT consist out of 4; {x, y, z, r}.

It would seem as if the temporal external memory usage of the tiles and normals are irrelevant as in the best case they are smaller than the inner and outer MAT. However, not all computed MAT need to be stored, if the MAT can not be correctly computed because there are no neighbouring points within the initial radius, the MAT output is not valid. These final MAT outputs with a radius which is just the initial value, should be removed from the dataset. As such, the MAT could be much smaller than expected.

4.1.6 time complexity

Tiling could be done by sequentially reading an input dataset, therefore it can be done in O(n) time when there are n points in the dataset. Usually there are not many tiles created in the tiling process, for instance, a AHN3 tile can usually be subdivided in 800 tiles (or less) of 200m x 200m.

The segmentation performed on these tiles therefore will not take long either. Of each tile the amount of points per tile needs to be known before the segmentation can take place. As the space-driven segmentation splits a collection in half, it is has a O(1) complexity. Using the data-driven segmentation, the amount of points in both halves of the splitted collection should be roughly the same, this involves summing points of tiles T and has therefore a O(T) complexity. Do notice that the amount of tiles T is much smaller than the amount of point n in the tiling process. Therefore the time complexity of the segmentation process is inferior to the tiling method.

The time complexity of the MAT computation of the tiling approach will be discussed later in section 4.4.1.

4.2 APPROACH: STREAMING ALGORITHM

In streaming algorithms, data is presented as a sequence of items. The data is streamed from the external memory to the internal memory where it can be processed. As the stream can not go back to data which was already sent, therefore sometimes it needs a few extra passes. The main advantage of using a streaming algorithm is the limited use of the hard drive, the relative slow memory (as mentioned in section 2.6.2)

4.2.1 Spatial finalizer

The spatial finalizer (SF) mentioned in section 2.6.2, is implemented for the streaming approach, this computes the MAT in 3 steps (see Figure 4.16):

1. Spatial finalizer
   The SF creates a grid of empty tiles fitting within the boundary of the pointcloud dataset, followed by counting how many points are in each grid tile. Then the pointcloud file is read sequentially again, while each point is stored in the empty tiles, when a tile has all its points, it is send out to the next step.

2. Collector
   The collector receives tiles of points from the SF, these are stored in the memory as well. It waits till it can form collections (1 process tile and its buffers) and sends the collection to compute the MAT. Afterwards the process tile is omitted from the memory while the buffer tiles remain. These buffer tiles will eventually become process tiles as well when their surrounding tiles have arrived as well.

3. MAT computation
   The MAT is computed using the collections send from the collector.

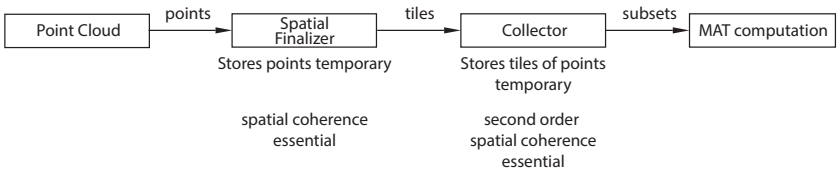


Figure 4.16: Workflow from pointcloud to MAT computation

4.2.2 Second order spatial coherence

A huge advantage of the streaming approach over the tiling approach is that data (tiles) does not need to be stored temporary on the hard disk, they are "streamed" from the file, to the SF, to the collector. The spatial coherence of datasets are usually good enough (section 2.6.2). However, this is only taking in to account the spatial coherence of points within a tile.

In section section 2.6.2 the spatial coherence is defined as:
correlation between the proximity in space of geometric entities and the proximity of their representation in the data file stream.

As explained earlier in section 3.2.1 the shrinking ball algorithm is not able

to compute the MAT for all the points in a single tile with just the data from that same cell. This means that for the SF to work, the streamed tile needs to wait for neighbouring tiles to arrive in the collector to be sure that the MAT can be computed. If a streamed tile does not need to wait long for neighbouring tiles to arrive, it has a good second order spatial coherence.

The second order spatial coherence is then defined as:
The correlation between the proximity in space of the geometric tiles and the proximity of their representation in the stream of tiles.

In this case, the stream of tiles is provided by the SF.

regular buffer method

Using the regular buffer method, computable points within a certain area need to have a buffer around them to ensure that MAT computed will be correct. As Figure4.17 shows, the points in the red tile need a buffer shown as green tiles around itself to ensure that the correct MAT is computed.

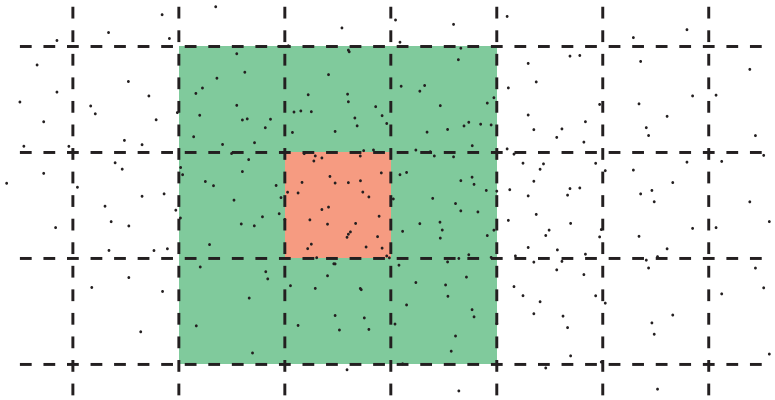


Figure 4.17: The red process tile needs points from the neighbouring green buffer tiles to be sure that it has created the MAT completely

As the use of the streaming algorithm with SF only outputs each tile once, the red process tile can not be removed from the memory, until it is not needed any more in the process. This means that it can only be removed after the its surrounding buffer tiles are processed as well. However, to process the buffer tiles as a process tile, the tiles around it need to be in the application as well (see Figure 4.18(a)). This means that before the red tile can be deleted, its 24 surrounding tiles need to have arrived (see Figure 4.18(b)). Using this knowledge a complete dataset is analysed (see figure 4.19), about half of the tiles need to wait for 50 sequential other tiles to arrive before they can be removed. It would seem inevitable that with huge datasets many tiles need to be stored temporary on the harddrive. As streaming methods tries to avoids extensive harddrive access, this is unwanted. Therefore the regular buffer method will not be used in the streaming approach.

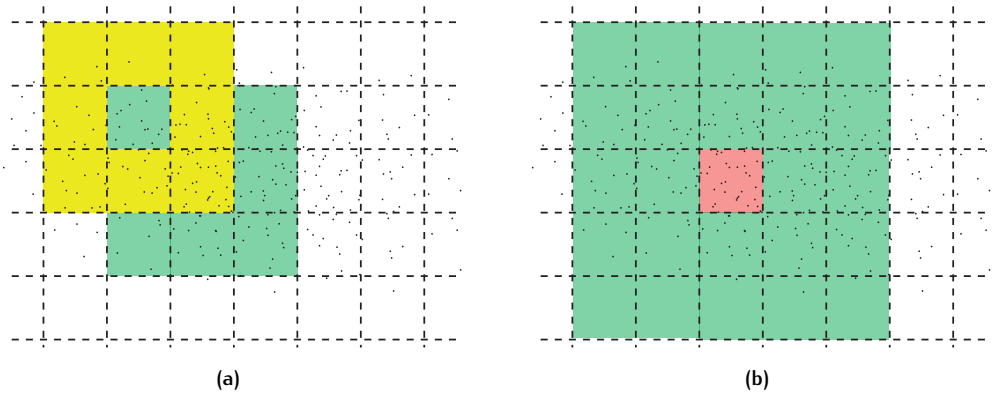


Figure 4.18: (a) Green cell needs its surrounding cells as well (b) All the green cells need to have arrived, before the red cell can be deleted from the memory

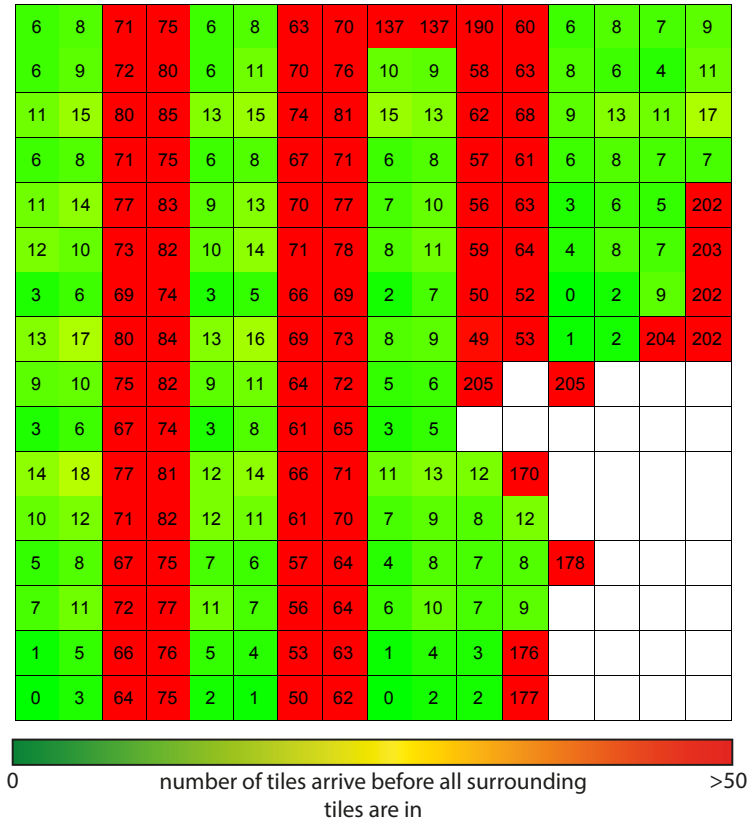


Figure 4.19: Tiles of the AHN3 dataset (c_67hz1.laz), the number in each tile represents the wait time. The wait time is expressed as how many sequential tiles need to arrive from the SF before all 24 surrounding tiles have returned.

reduced buffer method

Using the reduced buffer method, all points including the buffer itself will be computed. Points in the buffer area however, might need further computation to ensure that they get the correct MAT.

To check whether this may cause a problem, the time it takes for all the surrounding tiles to arrive needs to be found. Time is expressed here as how many tiles arrive before all the surrounding tiles are in. Image 4.20 shows how long each tile needs to wait till all the 8 surrounding tiles arrive from the SF for a part of the Rotterdam pointcloud. While many tiles are displayed in red indicating that they have a long waiting time, most of the tiles are green. The amount of active tiles at the most was 121, meaning that the SF probably still needs to store certain tiles temporary on the hard disk.

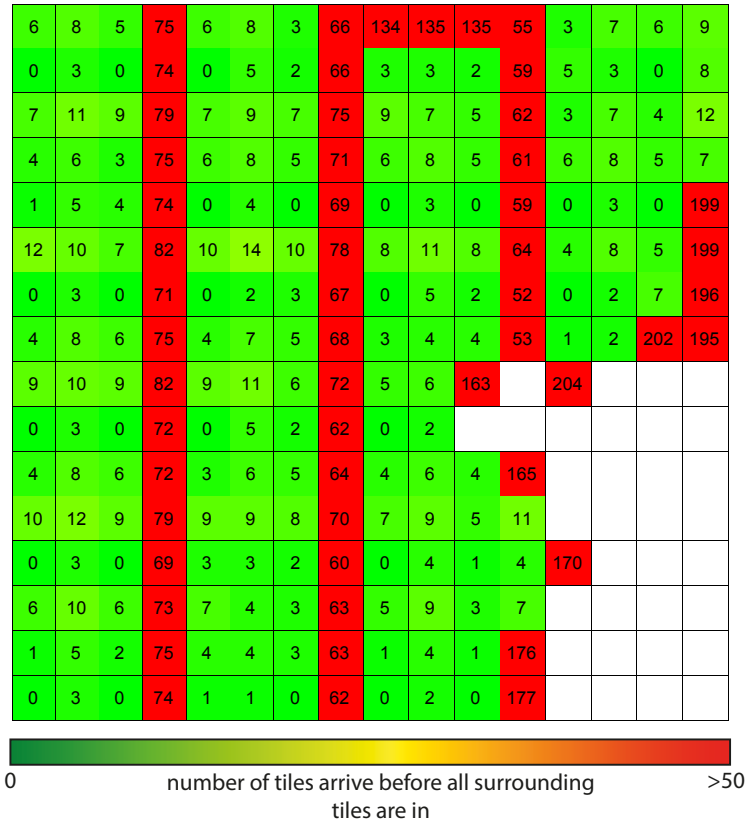


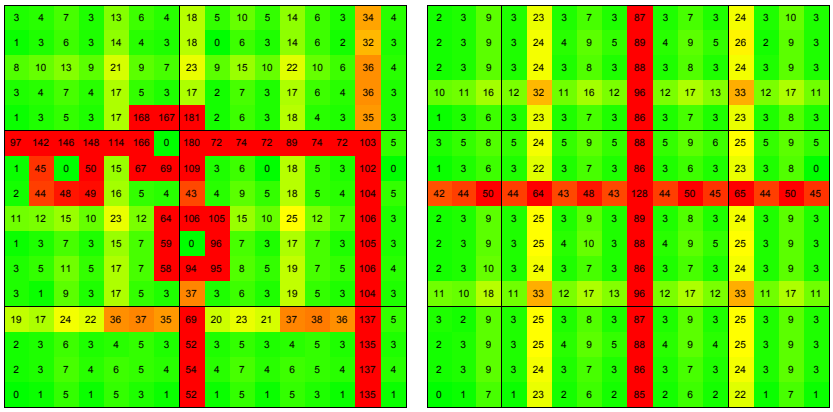
Figure 4.20: Tiles of the AHN3 dataset (c_67hz1.laz), the number in each tile represents the wait time. The wait time is expressed as how many sequential tiles need to arrive from the SF before all 8 surrounding tiles have returned.

4.2.3 Improving second order spatial coherence

As can be seen from the buffer method (Figure 4.19) and the reduced buffer method (Figure 4.20), the second order spatial coherence is not everywhere good, several tiles need to wait more than 50 sequential tiles, before their surrounding tiles have arrived. This is to be expected, because whilst the data is processed in 2 dimensions, it is stored as well as read as a 1 dimensional list/clusters of points. Insuring that this 1 dimensional stream of

points is sorted, could therefore improve the second order spatial coherence (locality). By grouping the points in tiles and sorting the tiles a space filling curve of tiles can be created.

After sorting the dataset using the z order curve (see Section 2.1.3), the locality of individual tiles seems to have improved slightly. Still many tiles need to wait a long time before the surrounding tiles arrive as shown in Figure 4.21. Before sorting a tile needs to wait on average 25.28 tiles before it can be computed. The maximum of it lies at 181. After sorting the average waiting time is 15,82 tiles and the maximum lies at 128. Figure 4.22 shows the amount of tiles in the memory during processing of each tile, these numbers are lower than the waiting time numbers because although some tiles might need to wait for 100 other tiles to be loaded, in the mean time some are removed from the memory after processing. Yet the maximum waiting time for a tile in a unsorted dataset it 39, after sorting the maximum is 24.



(a) (b)

Figure 4.21: The number in each tile represents the wait time. The wait time is expressed as how many sequential tiles need to arrive from the SF before all 8 surrounding tiles have returned.
(a) unsorted
(b) sorted using Z-order index

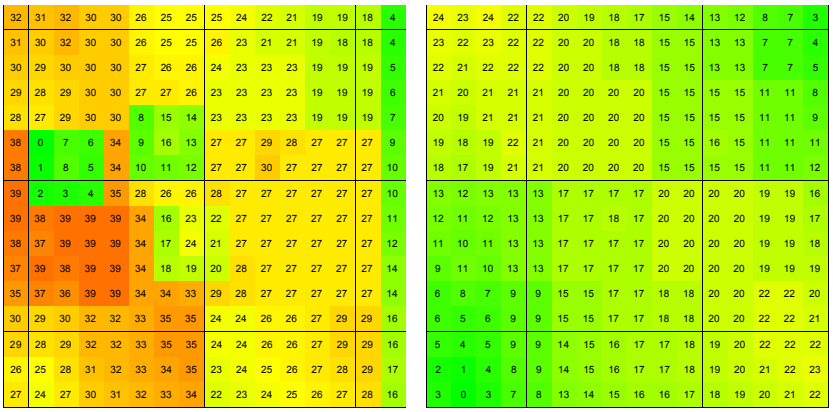


Figure 4.22: The number in each tile T represents the amount of tiles already in the main memory, when that tile T arrives from the SF
(a) unsorted
(b) sorted using Z-order index

4.2.4 Streaming process

The reduced buffer method is chosen to be used in this approach instead of the buffer method. As the reduced buffer method has a higher likelihood that cells will be released from the main memory earlier than with the buffer method is used.

general overview

The process starts with the streaming application, loading the data and piping the points including a finalizer indicating that all the points from a cell have been transferred. The collector reads the points from the cell and stores them in the memory, when all surrounding cells have been transferred, it sends it out to compute their normals when needed and computes the MAT. The MAT of the center cell (the only cell which is not a buffer cell) is then stored on the secondary memory and all its attributes/elements are removed from the main memory. This process continues, till all cells have been processed.

The tasks of the collector can be subdivided in four parts:

- collection of tiles
- detection of surrounding tiles
- computing normals and MAT
- storing MAT results

These will be explained in more details.

collection of tiles

As the points are received from the SF, they are stored in the main memory. When the finalizer tag arrives (which indicates that all points from a certain tile has been passed along), it is known that all the points of that certain cell

have arrived. Along with this tag comes a ID number, which is the location of the cell in the dataset. This ID number is the location on the Z-order curve. All the points will be stored as 32-bit float in an instance with the ID number as the key.

detecting surrounding cells

After a new tile has arrived, the process starts of checking surrounding tiles. For each tile $T_{x,y}$ currently in the memory, the availability of surrounding tiles[2] is checked. If all are available, $T_{x,y}$ and its surrounding tiles (which functions as the buffer) are grouped and send to the next stage for computing the normals and MAT.

computing normals and MAT

For each of the buffer tiles and $T_{x,y}$ the normals are computed and stored temporary in the main memory as well, if necessary. It is possible that several of the tiles have already been in this stage, and therefore their normals have been already computed. After the normals have been computed the MAT is computed. If some of the buffer tiles have already been in this stage, they also already will have computed their MAT before. This means that the radii of the previous computation(s) is still stored for each point of that cell in the main memory. Instead of using the initial radius (of 100 meter), the stored radius from the previous computation is then used in the new computation of the MAT. When points within a buffer tile has a MAT which lies with in the boundary of the subset, it does not need to be computed any further (as it is final). To reduce computation time, that point will get tagged, so that in upcoming datasets, the MAT will not be processed again.

storing MAT results

After the MAT has been computed, the radii results of buffer tiles are stored in the main memory, the MA is omitted, as it can be directly computed from the coordinates in computation with the normals and radii. The MAT of the $T_{x,y}$ is stored on the hard disk and the complete instance is omitted from the main memory, as it will not be needed any more in any future processes.

Algorithm 4.2 shows a detailed description of the algorithm. Please note, that DS is a finite stream of points with intermediate finalization tags. These finalization tags got a x and y value of the tile captured in its Morton code (section 2.1.3) and indicates that all the points from tile $T_{x,y}$ has arrived. Before "COMPUTE_STREAMING_MA" previous calculated radii of the points is loaded to replace the standard r_{init}, if they are available. After "COMPUTE_STREAMING_MA" the radii of the MAT is stored for the points to be used in future computations, as is needed when the reduced buffer method is used.

2 8 surrounding tiles, which will function as buffer in the subset: $T_{x-1,y-1}$, $T_{x-1,y}$, $T_{x-1,y+1}$, $T_{x,y-1}$, $T_{x,y+1}$, $T_{x+1,y-1}$, $T_{x+1,y}$, $T_{x+1,y+1}$

Algorithm 4.2: STREAMING APPROACH (DS)

Input: Spatial finalizer output: Datastream element DS is either Point p or finalization tags z

```
1  T = emptyarray;
2  Tiles = emptyarray;
3  while true do
4      if DS == type p then
5          Append DS to T;
6      if DS == type z then
7          Determine x, y value of DS;
8          T_{x,y} ← T ;
9          Append T_{x,y} to Tiles;
10         Empty T;
11         CHECK_SURROUNDING();
12     if DS == empty then
13         Break;
14 Function CHECK_SURROUNDING()
15     for each T_{x,y} ∈ Tiles do
16         t = 1;
17         L_p = Empty list;
18         L_n = Empty list;
19         for i = -1; i ¡= 1; i ++ do
20             for j = -1; j ¡= j; i ++ do
21                 if T_{x+i,y+j} exists then
22                     if normals not calculated yet for T_{x+i,y+j} then
23                         N_{x+i,y+j} ← COMPUTE_NORMALS(T_{x+i,y+j});
24                     append T_{x+i,y+j} to L_p;
25                     append N_{x+i,y+j} to L_n;
26                 else  t = 0;
27         if t == 1 then
28             COMPUTE_STREAMING_MA(L_p, L_n);
29             Storing MAT of T_{x,y} to hard disk;
30         remove T_{x,y} from Tiles;
31         remove N_{x,y};
```

4.2.5 Memory consumption

Unlike the tiling method it is much harder to estimate how much memory the process takes. While tiling method loads only the tiles which it actually needs to process certain points, the streaming method uses the tiles/points which are outputted by the SF. This does not mean that the tiles can be used directly to process points, as it might need to wait for neighbouring tiles.

Observe Figure 4.22.a, the maximum amount of active tiles in the main memory during the streaming process is 39. Depending on whether they are already pre-processed as a reduced buffer they might already have their normals calculated and their temporary MAT radius computed. If each of those tiles would have 1 million points and 9 tiles will be processed, it is estimated that it will take between the 1 and 2 GB of main memory.

This depends on whether other parts of the dataset already computed the Normals and MAT's (see Table 4.3).

Type	Points	Coordinates	Normals	KD tree	(MA)T
Process Points	9 million	108 MB	108 MB	315 MB	144 MB
Buffer Points	30 million	360 MB	0∼360 MB	0 MB	0∼240 MB

Table 4.3: MAT computation memory usage (Streaming approach)

4.2.6 external memory usage

Predicting the external memory usage by the streaming method is hard to do. Preferably the method will not use the external memory at all to store data temporary. However, if the dataset is large, the locality (second order spatial coherence) is bad or the main memory is not sufficiently large enough, storing on the external memory is needed.

4.2.7 Time consumption

The streaming approach, is very efficient when the spatial coherence is present. As it creates and outputs tiles after just reading the input file 3 times. Therefore the time complexity is O(n) for n points. The time complexity for the use of reduced buffers will be explained in section 4.4.1.

4.3 MERGING THE OUTPUT

The output MAT of both methods is a collection of subsets the size of the tiles input. These can be merged to form a larger dataset. Instead of just merging them randomly, using the z-order curve locality along the tiles can be created as shown in Figure 4.23 (see Section 2.1.3).

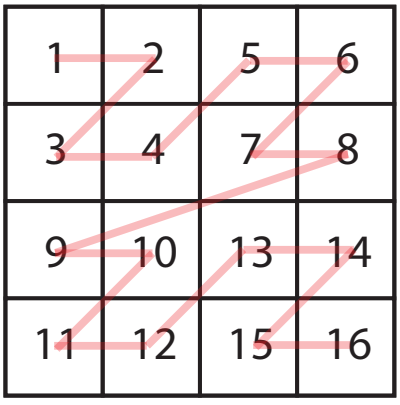


Figure 4.23: Merging the 16 tiles in the presented order is gives an overall better locality over the dataset compared to random merging

4.4 DIFFERENCES BETWEEN APPROACHES

In this chapter 2 different approaches were discussed:

- The tiling method
- The streaming method, using a spatial finalizer

The main differences between these 2 methods are shown in table 4.4.

Tiling	Streaming
Extensive use of external memory	*Avoids* extensive use of external memory
Making *large* collections of tiles possible	Making *small* collections of tiles
Can make use of *regular* buffers	To be efficient, should make use of *reduced* buffers

Table 4.4: Main differences between the tiling and streaming method

4.4.1 Difference in main memory usage and computation time

Both approaches can be split up in separate stages. As already shown in chapter 3, the first stage is the chunking stage, the second is the actual MAT computation. As the approaches differ a lot in the chunking stage, the computation time cannot be compared as time complexity. They both use the same MAT computation however, where the tiling approach uses regular buffers and the streaming approach uses reduced buffers.

chunking stage: tiling vs streaming

Using the tiling approach the amount of points each collection may have can be chosen during the segmentation, and therefore the maximum main memory usage can be easily regulated. As the tiles are stored temporary on the hard disk, collections can be made by loading the tiles in to the main memory. The streaming approach however reads the input data sequentially, and stores the tiles temporary in the main memory. It depends on the spatial coherence as well as the secondary spatial coherence on how much main memory is used. Therefore it is expected that the streaming approach will have a higher main memory usage than the tiling approach, while the tiling approach makes more extensive use of the hard disk.

Because the tiling approach writes and reads tiles from the slower hard disk, it will be more time consuming than the streaming approach, which stores tiles as well, but inside the faster main memory.

MAT stage: regular buffer vs reduced buffer

The regular buffer and reduced buffer both make use of the same MAT algorithm. As discussed in section 3.1.1, the worst case scenario time complexity of the MAT computation is $O(n^3)$. However, this assumes all n points are being processed as well as used in the kd-tree. As this is not the case for the tiling approach (as buffers are not being processed) as well as the streaming approach (although buffer tiles are being processed as well, when a MAT is final, the point might be used in the kd-tree, but the MAT does not need to be recomputed). Therefore the worst case scenario for computing the MAT can

be expressed as $O(n^2m)$ where n is the amount of points which are going to be processed (process points) and m the amount of points in total which are in the kd-tree (process points + buffer points).

The MAT of dataset V is the sum of computations of all the collections. It is not easy to compare the regular buffer and reduced buffer with regards to time complexity, because although they use the same algorithm, their n and m inputs are not the same. However, it can be assumed that the reduced buffer will be slower than the regular buffer because of the following reasons:

- With the regular buffer approach $n_{regular}$ is always smaller than $m_{regular}$. While in the worst case scenario, $n_{reduced} = m_{reduced}$ with the reduced buffer approach (as buffer points are processed as well).
- Let us assume that the number of processable points is $n_{regular,i}$ and $n_{reduced,i}$ for the regular buffer method and reduced buffer method respectively, where i is the ith collection. The reduced buffer approach processes buffer points multiple times to retrieve the *final* MAT. The regular buffer approach processes all points a single time, therefore it can be said that:

$$\sum_{i=0}^{c} n_{regular,i} < \sum_{i=0}^{c} n_{reduced,i}$$

- The reduced buffer method omits tiles after each computation, the number of points in later collection decreases, meaning that the first collections will have a significant higher $n_{reduced,i}$ than the last few collections. Furthermore as the reduced buffer method computes buffer tiles as well, $n_{reduced,i}$ at the beginning will have a much higher value than $n_{regular,i}$. The regular buffer method has a $n_{regular,i}$ which does not deviate much further in the process, as no tiles are removed.

Based on combination of these 3 assumptions it can be said that the reduced buffer approach will probably be slower. As the time complexity is exponential, high fluctuations in input size will take longer to compute than a constant size input. Furthermore the expected input n in general is larger in the case of the reduced buffer approach.

4.5 SUMMARY

4.5.1 Tiling approach

The tiling approach makes extensive use of the hard disk, by storing tiles $T_{x,y}$ (where x, y are the geographic location of each tile) temporary on it. As each tile $T_{x,y}$ is easily accessible, collections S_i could be made. Collections S_i consist out of a list of process tiles P_i and a list of buffer tiles B_i.

To make these collections (clusters) a top-down segmentation is suggested, either a space-driven or data-driven datastructure is used. This is achieved by first placing all the tiles $T_{x,y}$ in a single collection S_0 and keep iteratively splitting the amount of processable tiles $T_{x,y}$ in half, where both halves of the data are stored in new collections S_i replacing the parent one. When all the collections contain a amount of points which will fit inside the main memory, the process is ended.

The internal memory usage can easily be regulated by just taking larger or smaller collections of tiles. As tiles are stored temporary on the hard disk and the normals will be stored there as well, the external memory usage will be two times the size of the input dataset. However, as the inner and outer MAT will take a maximum of 2.6 times the input dataset of disk space, the external memory used during the computation should be available anyway.

4.5.2 Streaming approach

Unlike the tiling method, the streaming approach minimizes writes to the hard disk. The streaming does this by reading sequentially each data entry in a file multiple times. Using the spatial finalizer, the MAT is computed in 3 steps:

- Spatial Finalizer (SF)
  Reads the data entries sequentially from the input file multiple times and outputs the tiles.
- Collector
  Receives sequentially the tiles from the spatial finalizer and stores them temporary in the main memory. When its surrounding tiles, which function as the buffer arrives, it computes the MAT. After the MAT is computed, tiles which are not needed further in the process are omitted from the main memory.
- MAT computation
  Computes the MAT and stores the radii of points in the buffer tiles temporary, so that they can be used in future computations.

The use of the reduced buffer method is necessary, as using *regular* buffers will increase the main memory use. This is because:

- Using regular buffers, the process tile $T_{x,y}$ can not be deleted until its 24 surrounding tiles have arrived. (its surrounding tiles need $T_{x,y}$ as well when the MAT is computed for them).
  Using the reduced buffer method only the surrounding 8 tiles of process tile $T_{x,y}$ need to available and $T_{x,y}$ can be omitted after the MAT is computed.
- The second order spatial coherence could be better, tiles which are close to each other in geometric sense, are not necessary close to each other in the sequence of tiles outputted by the SF.

5 IMPLEMENTATION, EXPERIMENTS AND COMPARISON

In this chapter the previous described approaches are implemented on several real-world datasets. The results are evaluated based on their the output quality, computation time and memory usage (main memory and hard disk).

5.1 IMPLEMENTATION

Figure 5.1 shows the workflow for both approaches. Apart from code written for this thesis, masbcpp[1] has been implemented/adjusted and lastools[2] has been used as well.

In the tiling approach, the datasets were cut in to 200m x 200m tiles using las tile from the lastools package. The normal computations are computed using compute_normals from masbcpp. The segmentation has been written in the python programming language (using pointio[3] library). The MAT is computed using code based on compute_ma from masbcpp.

In the streaming approach, the tiles were piped from the spatial finalizer (SF) to the collector. The collector (analyze input) is written in C++ which was merged with compute_normals from the masbcpp. The MAT is then computed using the same compute_ma from tiling approach. The code makes use of the libraries: cnpy, kdtree2, psapi, tclap and vrui.

Finally the quality control and checks were performed using Python.

The developed code is available at https://github.com/Rissos/scalingMAT.

1 masbcpp by R. Peters
https://github.com/tudelft3d/masbcpp
2 lastools by M. Isenburg
http://www.cs.unc.edu/ isenburg/lastools/
3 pointio by R. Peters
https://github.com/Ylannl/pointio

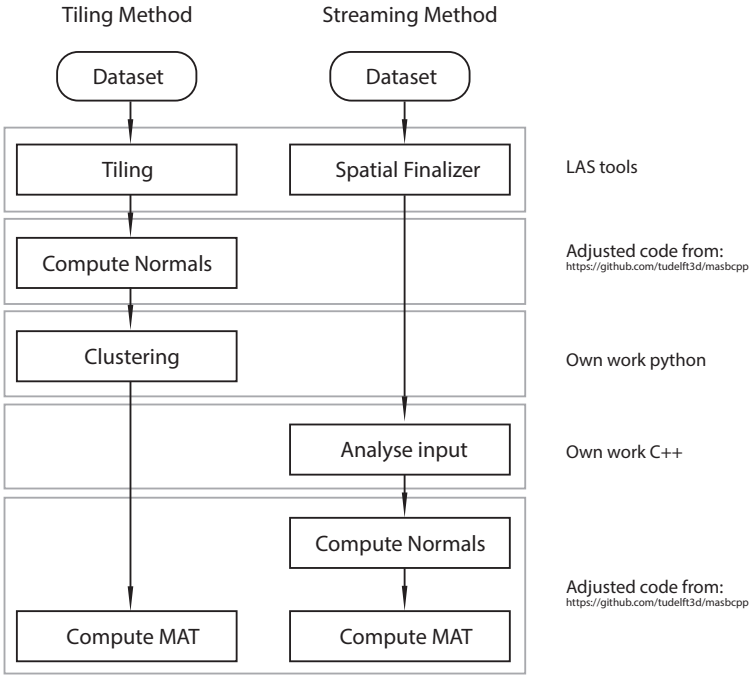


Figure 5.1: Workflow scheme for both approaches used

5.1.1 hardware

The tests were performed on a laptop (windows) equipped with an Intel Ci5-4300U @ 1,9 GHz and SK Hynix 256 GB Solid State Disk. The amount of main memory used is limited by the use of a c++ compiler for 32-bit (therefore only 2 GB of main memory could be addressed).

Tests on larger datasets were performed on a system (linux) equipped with Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz and 512 GB Solid State Disk.

5.1.2 Tiling Implementation

The tiling approach has been implemented through a python script as well as a program build in c++. The python script does the preprocessing and builds the collections, while the c++ program computes the MAT based on the output of the python script. It starts by first using lastile to split the dataset in to smaller las[4] tiles of 200 by 200m. The python script then reads the las datafiles and converts them to npy[5] while adjusting each coordinate to comply with a local reference. After that it computes the normal vectors for each individual tile. Then the python script forms grid-like collections or collections based on a quadtree/kd-tree. For each collection of tiles including buffer it creates a ASCII file containing the path to the npy files in the collection and calls the c++ program to compute the MAT based on the list of npy files.

4 LAS file format is a public file format for the interchange of 3-dimensional point cloud data data between data users ASPRS [2013]

5 npy: binary file in NumPy a fundamental package for scientific computing in Python

preprocessing

Because it is known that the bare minimum size of each subset should be 400m x 400m and the buffer size should be 200 meters, it is chosen to precut the dataset in to pieces of 200m x 200m. These tiles are the smallest subsection of the binary tree. To produce these tiles lastile of lastools is used, which cuts a dataset or list of datasets in to tiles with a size predefined by the user. Then all the .las tiles are converted to .npy.

The Principal Component Analysis (using k=10 nearest neighbours) is used to compute the normals of each single tile is computed using its surrounding tiles are buffers.

Segmentation method

All available metadata of the individual tiles are then loaded in to a matrix.

The 3 segmentation methods are tested on 4 large datasets of the AHN3, see table 5.1. The maximum amount of point per collection indicates the size of the collection (a dataset can be subdivided in a few large collections or many smaller collections) These datasets differ from each other in content as some of them are smaller in geometric size. There is no best segmentation method for collections with many points ($>$ 20 million), however, the optimized KD-tree method worked best with smaller sized collections ($<$ 18 million points). The quadtree method does seem to work in all occasions (when one of the others worked as well), the kd-tree and optimized kd-tree method struggled sometimes with smaller sized collections. Overall there are no significant differences.

	c_37en2			c_67hz1			c_11hz2			c_37gn1		
Max Points per collection	qt	kd	okd	qt	kd	okd	qt	kd	okd	qt	kd	okd
50 mil	23	21	23	12	11	12	23	25	25	10	9	9
40 mil	31	33	32	14	16	16	35	32	31	14	15	15
30 mil	48	50	47	26	24	25	51	53	50	24	21	21
20 mil	90	89	91	44	44	44	95	103	100	41	41	41
18 mil	111	114	109	56	50	48	122	121	117	50	50	48
15 mil	154	-	147	74	74	-	-	-	-	66	68	65

Table 5.1: Amount of collections per segmentation method; qt = Quadtree; kd = KD-tree; okd = Optimized KD-tree
The highlighted values indicate the best solution for the size of collection

5.1.3 Streaming Implementation

One of the main advantages of the streaming method is that writing to the hard disk is minimized. As such only the reduced buffer method is implemented, section 4.2 shows that the regular buffer method longer waiting times introduces per cell and thus also take more main memory. The downside is that processing time will increase, as certain parts need to be processed multiple times. As the SF loads the datafile and pipes the tiles to the collector, no preprocessing is needed.

5.2 DATASETS

To test the approaches 2 pointclouds are used, the AHN3 and Rotterdam puntenwolk. The AHN3 has a point density on average of 8 points/m², while the Rotterdam puntenwolk has on average a point density of 30 points/m². More details about both datasets has been presented in section 5.2. The AHN3 is downloadable in tiles, while Rotterdam puntenwolk is downloadable as a whole. Figure 5.2 shows the datasets used:

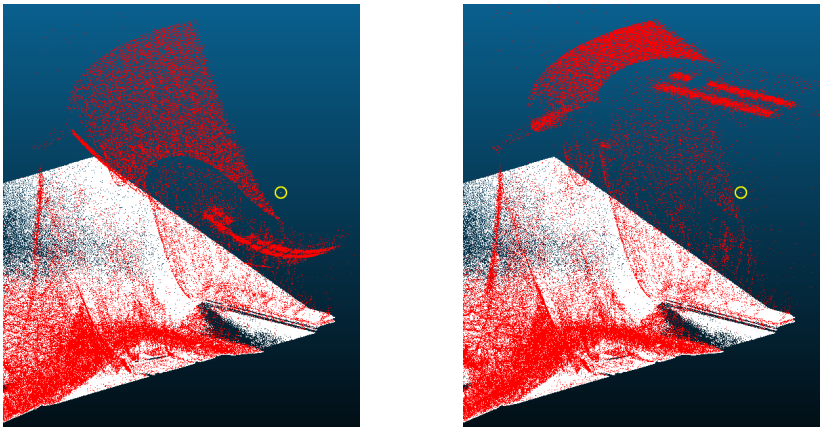
- AHN3
  - c_67hz1 : Zeeland, Zuiddorpe
    Relative small dataset (5000m x 4700m) ($2,8 * 10^8$ points)
  - c_37gn1 : Zuid Holland, Rotterdam Relative small dataset (5000m x 3000m) ($2.4 * 10^8$ points)
  - c_37en2 : Zuid Holland, Delft
    Regular dataset (5000m x 6250m) ($5,2 * 10^8$ points)
  - c_11hz2 : Groningen, Appelscha
    Regular dataset (5000m x 6250m) ($5.1 * 10^8$ points)
- Rotterdam dataset

From the Rotterdam Dataset two smaller subsets are extracted. These were used on the consumer laptop to compute the MAT. The one of these subsets was specifically chosen to fit completely in the main memory during processing, whilst the other dataset does not fit. The AHN3 datasets are contain large quantities of points, whilst it possible to compute the MAT on the consumer laptop using the scaling options, it would take significantly more time than the smaller ones. The AHN3 has a accuracy up to 0.05 m [Rijkswaterstaat, 2015].

5.3 DATA QUALITY

The output of the tiling and streaming method should be the same. However, there are some apparent differences. Whilst both methods use the same algorithm to process the data, the output data is not the same. The de-noising which takes place, does not work similar in both approaches. Although they do have the same method for de-noising, the way the streaming approach handles the input data differs from the tiling method. Removing tiles from the collection (which is done by the streaming approach) has influence on the de-noising. The tiling method is in this case effected by noise in Figure 5.3, whilst the streaming method is able to filter it out. This issue will not be further discussed, as it is an issue which is not effected by the "basic" workings of the MAT algorithm.

Figure 5.2: location datasets



(a) (b)

Figure 5.3: Difference in output due to a single point inside the yellow circle(a) Tiling (b) Streaming

5.4 EXTERNAL MEMORY USAGE

The memory usage can be subdivided in 2 parts; main memory and external memory. Both approaches have first been tested on small size datasets of the Rotterdam puntenwolk. However, the streaming approach did not seem suited in its current state to be tested on larger datasets (will be explained in section 5.8), therefore only the tiling method has been further analysed using the larger AHN3 datasets.

In section 4.4 the distinct differences in how the 2 approaches external memory were summarized. The tiling approach makes extensive use of the external memory, by storing tiles temporary on it. The streaming approach avoids writing too much data on the external memory as it stores the tiles on the main memory. Ideally the streaming approach will just read the input datafile and outputs the MAT on the external memory, without needing to make use of temporary writes during the computation.

As the MAT consist out of x,y,z,r, it is expected that the output (i.e. inner and outer MAT)should be 2.6 times the size of the input datafile (i.e. point-cloud x,y,z). However, this should still be filtered as some values are not valid (as explained in section 3.5.2). In table 5.2 can be seen that the Inner MAT has much less valid values than the Outer MAT.

	Outer [points]	Inner [points]	Memory usage [MB]
Total MAT	21889415	21889415	700
valid	21334948	10706325	513

Table 5.2: External memory usage MAT (800 x 800m dataset of Rotterdam puntenwolk)

5.4.1 Small size datasets

Both approaches have first been tested on the small sized datasets of the Rotterdam puntenwolk, table 5.3 presents the results.

As can be seen the streaming method does not need to make use of storing data temporary on the external memory during the processing of subset (800 x 800m). However, a processing the larger subset (1600 x 1600m) does result in storing temporary data on the hard disk as the internal memory runs full. This is because the second order spatial coherence drops as the datasets grow larger.

The tiling method does make extensive use of the external memory. Compared to the output MAT the difference is not very big.

Dataset	Approach	Computation [MB]	Output MAT
800 x 800	Tiling	494	513
	Streaming	-	510
1600 x 1600	Tiling	1828	1706
	Streaming	340	1747

Table 5.3: External memory usage of the tiling and streaming algorithm on parts of the rotterdam pointcloud

Large size datasets

The MAT of the AHN3 datasets is sometimes larger and sometimes smaller than the temporary tiles used by the streaming method (see Table 5.4).

Dataset	Computation [GB]	Output valid MAT
c_67hz1	13.6	8.2
c_37gn1	11.4	10.8
c_37en2	25.2	22.7
c_11hz2	24.5	25,3

Table 5.4: External memory usage of the tiling on AHN3 sets

The MAT output size on the external memory can be smaller than the temporary tiles which are needed during the computation. Hypothetically it is possible that the MAT turns out to be non-existing (0 MB), for instance when the dataset is just a straight plane and no MAT can be computed. As the radius of each point will be infinite when the MAT is computed and thus none of those MAT will be included in the output. In this situation the temporary tiles stored by the tiling method would need much more space than the output MAT. However, without pre-analysing a dataset, it cannot be known how much big the output MAT will be. Therefore one has to assume the worst case: all the MAT are valid. In this case a much larger part of the external memory needs to be reserved for the MAT, which is much more than the temporary files need.

5.5 INTERNAL MEMORY USAGE

Both approaches subdivide the dataset in to tiles and recombine collections of tiles to form subsets. For these subsets the MAT is then computed. In both situations, the amount of points per subset will determine the internal memory usage. As the tiling approach will only have the loaded subset in the memory, the main memory usage is solely occupied with that. The streaming approach has both the subset as well as other tiles which come from the SF in the main memory. Therefore it is forced to make small collections, while the tiling approach does not necessarily need to do this.

5.5.1 small size datasets (800 x 800 m)

Using the tiling method, the size of the collections can be chosen. As expected, when more collections are formed the size of the maximum memory usage decreases as well (See Table 5.5). There is not a large difference between the memory usage of 16 collections and 4 in this dataset (1.3% increase in internal memory usage). This is due to the buffers formed around the computed tiles, as the use of smaller sized collections become less effective due to the buffer process point ratio.

# collections	1	2	4	16
Max memory	1342	1068	835	824

Table 5.5: Maximum memory usage for several sizes collections using the tiling approach (800 x 800 m)

The streaming approach performs slightly worse than the tiling approach when using 16 collections. The maximum memory usage of the streaming approach takes place during the computation of the first collection, in which tile # 13 is processed including its buffer tiles (See Table 5.6). Because buffers are processed as well and tiles are removed after a collection has been processed, the amount of memory usage drops along the process. This can be seen in table 5.6, as the later collections need less memory to be processed (the last processed collection only consumes 158 MB of memory, as the collection only contains 1 tile).

Part #	Max memory [MB]	tiles in collection	Tiles in memory
13	953	6	7
15	641	3	6
0	736	4	9
1	849	5	8
2	681	4	9
3	805	6	8
12	825	6	8
14	506	3	7
8	539	4	7
9	590	4	6
11	406	2	5
10	290	1	4
4	524	4	4
6	411	3	3
7	246	2	2
5	158	1	1

Table 5.6: Main memory of parts of the streaming process (small dataset)
Part # is the number of the tile being processed together with its buffer tiles (collection)
Max memory is the max main memory usage during the computation of the collection, this takes place just after the MAT is computed
Tiles in collection, is the number of tiles in the collection
Tiles in memory, is the total number of tiles temporary stored in the main memory regardless whether they are in the current collection or not

5.5.2 small size dataset (1600 x 1600 m)

The 1600 x 1600 m dataset is 4 times larger than the previous set. As such the tiling method cannot compute the MAT with less than 8 collections (as the main memory is not large enough to compute larger subsets). Table 5.7, shows that using 64 collections does not give a huge saving in main memory use compared to using 16 collections.

# Collections	8	16	64
Max memory [MB]	1569	1059	927

Table 5.7: Maximum memory usage for several sizes collections using the tiling approach (1600 x 1600 m)

The streaming approach has a maximum memory use of 1277 MB (see Appendix B for the complete table), which is much higher than the 927

MB of the 64 collections tiling approach. The streaming approach actually reaches the limits of main memory capacity, as such it had to store 340MB of data temporary on the hard disk to be able to continue processing data (see Section 5.4.1).

5.5.3 large size datasets

Similar as with the smaller sized datasets, the larger AHN3 subsets uses less main memory during the computation when more collections are used. From Figure 5.4 it can be clearly seen that there is a exponential drop in memory usage.

However, at a certain moment the memory usage stabilizes in the graph. This is because the decrease of points by splitting collections in to smaller collections is less when the collections are already small, as buffers still need to be applied.

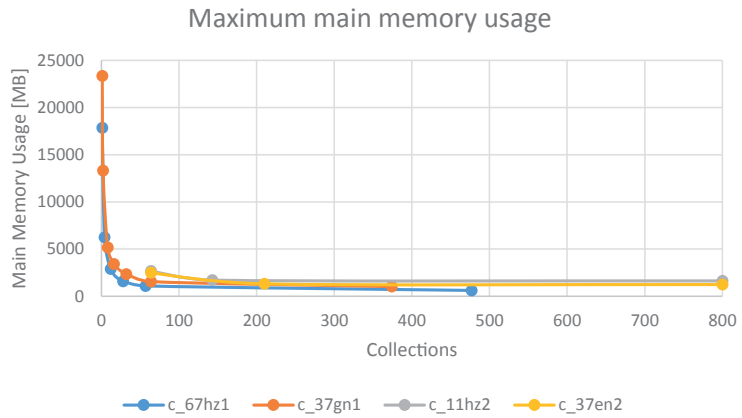


Figure 5.4: AHN3 datasets: acsmat main memory usage

5.6 COMPUTATION TIME

This section presents the computation time of both approaches. The time complexity has been described in chapter 4. As can be seen in Figure 5.1, the steps both approaches uses are not the same. However, both approaches do make use of the same basic steps (see Table 5.8). Apart from the total computation time, these 4 steps will be also evaluated, to see where time differentiations between the 2 approaches derives from.

	Tiling	Streaming
Creating Tiles	Create and store tiles on the hard disk	Tiles are being outputted by SF
Loading Tiles	Loading the tiles from the hard disk	Reading tile output of SF
Create KD-tree	Create the KD-tree	
Compute MAT	Using *regular* buffers	Using *reduced* buffers

Table 5.8: Steps both approaches make and the description

5.6.1 Small datasets

Tiling

The tiling approach does not show big differences in the total computation time, as can be seen from both Table 5.9 and 5.10. However, the loading time as well as KD-tree creation time increases when more collections are formed. This is to be expected, as having more collections leads to having more buffer tiles and thus more points needing to be loaded; e.g. just having 1 collection means in this case that there are no buffer tiles, while having 16 collections means that there are 84 buffer tiles.

# Collections		1	2	4	16
Tiles creation	[s]			2.91	
loading	[s]	0,736	0,892	1,346	3,757
KD Tree	[s]	7,946	12,197	16,557	44,788
MAT compute	[s]	583,121	546,087	556,834	546,004
Total	[s]	591,803	559,176	574,737	594,549

Table 5.9: Processing time of parts of the tiling process (800 x 800m)

# Clusters		8	16	64
Tiles creation	[s]		20.6	
loading	[s]	8.829	6.122	17.332
KD Tree	[s]	55.275	77.413	183.402
MAT compute	[s]	1621.603	1542.722	1553.975
Total	[s]	1685.707	1626,257	1754,709

Table 5.10: Processing time of parts of the tiling process (1600 x 1600m)

Streaming

Using the streaming approach, collections contain less tiles further in the process (as tiles are omitted during the process). This causes a huge difference in processing times for each collection as can be seen in Table 5.11, the first collection to be processed takes 314.10 seconds, while the twelfth collection only takes 0.53 seconds. This is also because tiles which have been processed before (as a buffer tile), process faster then tiles which have not been processed before. The four times larger 1600 x 1600 m dataset shows same results (Table 5.12), the individual collections have the same maximum and minimum computation time, while the total time takes roughly 4 times longer.

	Process # [s]	Create and load tiles	KD Tree [s]	MAT compute [s]
Max	1	-	3.55	312.55
Min	12	-	0.39	0.14
Total		4.62	23.93	749.83

Table 5.11: Processing time of parts of the streaming process (800 x 800 m)

	Process #	Create and load tiles	KD Tree [s]	MAT compute [s]
Max	44	-	4.61	312.30
Min	64	-	0,34	0.05
Total		14.03	97.23	2789.06

Table 5.12: Processing time of parts of the streaming process (1600 x 1600 m)

5.6.2 Large datasets

Performing the tiling approach on the AHN3 datasets, shows that using more collections result in to longer processing times. As can be seen in Figure 5.5, the increase in computation time grows linearly with the amount of collections used. Subsets c_11hz2 and c_37en2 need about twice the amount of time to process compared to c_67hz1 and c_37gn1, this is because the former has about twice the amount of points the latter has.

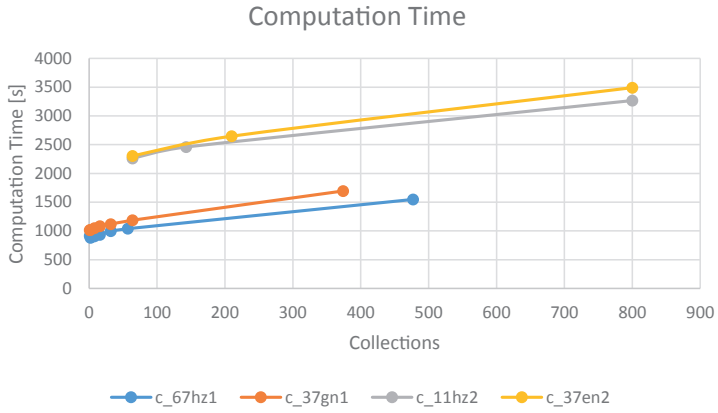


Figure 5.5: AHN3 datasets: acsmat Computation Time

The large increase in computation times when more collections are used, is caused by the kd-tree building (see Figure 5.6). The loading tiles as well as the MAT computation itself do not increase a lot.

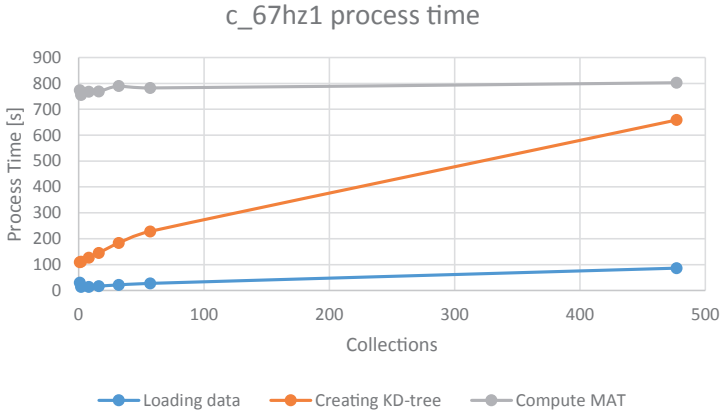


Figure 5.6: AHN3 c_67hz1: Process Times

5.7 DISCUSSION

In section 4.4.1 some theoretical estimations were made on memory usage and computation time. The result of the implementation should reflect that.

5.7.1 Memory usage

As expected, the streaming approach uses more memory than the tiling approach. With larger datasets, which will have a worse locality (having a larger 2-dimension space converted to 1-dimension space to be able to store it as a file). As the streaming approach is dependent on the locality, large datasets will quickly fill up the main memory and eventually it needs to store the tiles temporary on the hard disk when the main memory is full.

The tiling approach stores all the temporary tiles on the hard disk and has therefore a lower main memory consumption.

5.7.2 Computation time

As discussed in section 4.4.1, the first stage in the tiling approach (e.g. tiling and segmentation) should be a slower process than the streaming approach (e.g. reading the input file multiple times and output the tiles without writing temporary files on the hard disk). This is also reflected in practice, as table 5.10 shows that the first stage in the tiling approach takes 20.6 + 17.332 seconds to create the tiles and load them when the dataset is split up into 64 collections. The streaming approach is able to do that in 14.03 seconds (Table 5.12).

The second stage, i.e. the actual MAT computation, should be in favour of the buffer approach (section 4.4.1). As the tiling approach uses regular buffers and streaming uses reduced buffers, not all results can be compared to see differences with the use of regular buffers and reduced buffers. The tiling approach has the possibility to create large collections, while streaming approach will create small collections. There are 16 of these small collections in the 800 x 800 m dataset, and 64 in the 1600 x 1600 m. They consist out of the same collections as the tiling approach, when using 16 and 64 collections respectively. Therefore the relative computation time of both buffer methods can be compared using these values. Figure 5.7, shows the computations for both the regular buffer and reduced buffer using data from Tables 5.9, 5.10, 5.11 and 5.12. It shows that although the reduced buffers have lower kd-tree building times, the MAT computation time is much larger compared to the regular buffers.

The lower kd-tree building times are caused due to that reduced buffers remove tiles during the process, therefore collections will have less tiles. Thus the kd-tree will get smaller and faster to build. However, to be able to remove tiles, more processing has to take place; the MAT will be also computed for buffer tiles and checks have to take place to see if they are final or not. This will make the MAT computation slower.

As the streaming approach using reduced buffers performs worse than the tiling approach in terms of computation time and memory usage, it was not applied on the larger datasets.

5.7.3 tiling approach on larger datasets

Figure 5.4 and 5.5 shows that the memory usage drop exponentially while the computation time rises linearly with increasing collections. As such it would seem wise to just create more collections, to reduce the memory usage. However, there is not much difference in memory usage when 800 collections are made or 150, while the processing time does increase when more collections are used. This is because points in a geographical datasets are not homogeneously spread. Furthermore the 800 collections are not obtained by segmentation, but by using all the tiles individually together with a buffer. Of these 800 tiles, probably just a few have a high amount of points (a higher point density). If the collection with the largest amount of points can be found, this should be the threshold k_{max} for when the actual segmentation is performed on the dataset. This will ensure that the all the collections will have a similar small amount of points (minimal memory usage) and relative fast processing time. The focus lies on the memory part, as that is the one which drops exponentially, so more advantages can be obtained from it.

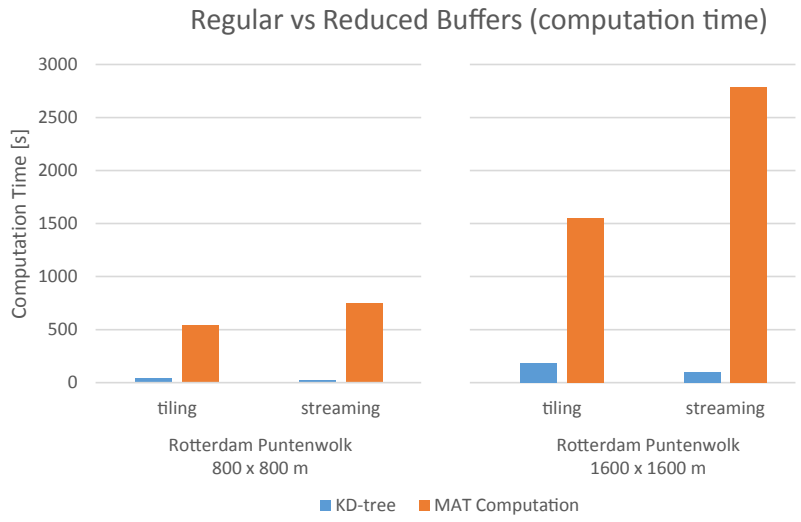


Figure 5.7: Computation times using regular buffers and reduced buffers

5.8 SUMMARY

Out of the three presented segmentation methods none was the best in every occasion (see Figure 5.1). The Optimized KD-Tree implementation did perform the best in cases where there were smaller sized collections. The KD methods in general were not always computable on smaller sized collections when the Quadtree implementation was.

In general the test results has shown that the tiling approach outperforms the streaming approach in computation time as well as main memory usage. The streaming approach does use less external memory (on small datasets). However, as the possible output MAT is larger than temporary tiles stored by the tiling approach, the temporary external memory usage is less important.

In section 4.4 it was discussed that the streaming approach would be faster in creating collections compared to the tiling approach and that the

reduced buffer method should be slower than the regular buffer method. Section 5.6.1 shows that this is indeed true, as the tiling approach needs to write the tiles and then load tiles from the hard disk, it is slower than the streaming approach, which stores tiles in the faster main memory. The streaming approach does seem to be significantly slower as well.

While transferring data through a pipeline from program to program (done by the streaming method) is much faster than reading and writing from the external memory (done by tiling). These differences are negligible compared to the computation time of the shrinking ball algorithm. For instance in the medium sized rotterdam dataset, the creation of tiles takes 20.6 seconds, the segmentation took 0.1 second and the loading of collections in the worst case 17 seconds. The streaming method is able to transfer the data in binary in 14 seconds. However these differences are quite small compared to the MAT compute time of 1754.7 seconds.

6 CONCLUSION, DISCUSSION AND FUTURE WORK

Using the methodology and implementations presented in this thesis it was possible to compute the MAT using the shrinking ball algorithm on massive datasets. In chapter 5 the 2 basic scaling methods were implemented and compared to each other. In section 6.1 conclusions are drawn from these experiments. However both methods do have their limitations, these will be discussed and several recommendations can be made for future research 6.2.

6.1 CONCLUSIONS

In the search for a memory efficient way to compute the MAT for large datasets several sub-questions have been answered.

What are the challenges in scaling the 3D medial axis using the shrinking ball algorithm?

In the computation of the MAT using the shrinking ball algorithm a certain starting radius should be chosen. This will be the maximum size of an object you can detect. This start radius can be used to create buffers for each subset, as no other points outside of the dataset + buffer can be of influence on the MAT result.

An advantage of the MAT using the shrinking ball algorithm is that the computed medial points of coordinates are independent on the results of other medial points. They only rely on the normal vectors of the computed coordinate and the surrounding points. As such, using a divide-and-conquer method will not introduce merging problems when the MAT computed using collections with buffers.

Predicting which coordinates are not needed in the buffer to compute the MAT for a tile is difficult. Based on whether the radius of a certain MAT crosses the boundary of a tile has been proven not to be reliable enough. Therefore the thinned reduced buffer will not work.

How to design and implement several methods for scaling the shrinking ball algorithm?

Using the conclusions of the previous sub-question essentially different 2 methods can be created:

- Tiling method (D&C)
  The dataset is subdivided in to minimum sized tiles (based on the buffer size) and stored on the hard disk. From there collections in-

cluding buffers are made based on a quadtree/kdtree/grid. For each individual collection MAT is then computed.

- Streaming method Instead of relying heavily on hard space, the streaming method reads the complete dataset multiple times while computing the MAT on the fly.

How do the methods compare to each other?
In terms of memory usage, computation time and output quality.

The output of both methods are not the same. Some points in the original dataset might be classified as noise or not depending on the chosen method. Without noise reduction the output should be the same.

In general the tiling method outperforms the streaming method in both computation time and main memory usage.

- The streaming method stores tiles in the main memory, till its surrounding tiles have arrived. Therefore it will use up more main memory.
- While the tiling method makes more use of temporal secondary drive storage, the MAT will eventually take about the same space.
- The tiling, segmentation and reading processes take longer than the streaming process, however the differences are relatively small compared to the whole process of computing the MAT.

Conclusions specific to tiling method:

- Using the tiling method the computation time becomes longer when more small sized collections are created instead of less large sized collections. This is mostly due to building up a kd-tree. As having more collections means that there are more buffer tiles as well, although the MAT is not computed for them, they are used in the building of the kd-tree.
- The computation time grows linearly with the amount of collections, while the memory usage decreases exponentially with the amount of collections. Therefore it is more advantageous to compute many small sized collections of similar size.
- In the top-down segmentation method to create collections during tiling process there is no best separator, the space-driven (quadtree based) method ensures that it will work more often, while the data-driven (optimized kd-tree based) method clearly gives better results when smaller sized collections are made

Conclusions specific to the streaming method (in the current stage):

- The streaming method has large MAT computation times, this is because most tiles are computed multiple times. And several checks are performed to determine whether the MAT computed are final.
- The streaming method minimizes secondary drive writes, however, with larger datasets, the secondary spatial coherence (locality) becomes worse and temporary storage will be needed.

- The streaming method can not be used to its full potential yet (see Discussion).

How can the 3D medial axis point approximation using the shrinking ball algorithm be scaled in a memory efficient way for a large dataset which does not fit in the internal memory?

Based on this research, creating the MAT using the shrinking ball algorithm should be performed using the tiling method with the segmentation method based on the optimized KD-tree by making many similar small sized collections with buffers. However, if the reduced buffer method could be improved, the streaming method could be optimized which will improve the memory usage as well as computation speed.

6.2 DISCUSSION

The use of 32-bit float reduces the amount main memory usage during the processes. As the AHN3 has subdecimeter precision, the 32-bit float representation can only contain values between -9999.99 and 9999.99 without the loss of precision. When larger datasets are inserted with similar point precision, the use of 32-bit float will cause precision errors. Two solutions could be useful:

- Make use of double precision
  Higher precision values can be used, however, it cost twice as much memory
- Use a local reference system per collection
  Instead of using a local reference system for the whole dataset, create one per collection. Do note that if collections differ in size a lot, it can have differences in precision.

The shrinking ball algorithm makes use of a KD-tree to query nearest neighbour searches. However deletion and insertion in the KD-tree is a time consuming process as the whole tree might need to be rebuilt at every step, another spatial index could perform better (space driven index e.g. octree).

While the Streaming method does not seem to perform very well. This is because the shrinking ball algorithm is not well suited for it (yet). In Section 2.6.2, the four requirements in the use for a streaming algorithm are mentioned. Memory usage however, is not very well implemented. If the thinned reduced buffer method from Section 3.5 worked, then each subset piped from the streaming algorithm could be instantly processed instead of waiting for it neighbours to arrive. MAT's which are finalized could be written away and the corresponding points can be omitted from the memory.

Both the streaming and tiling methods make use of the same de-noising method (see Section 2.4.3). This is because the de-noising method is embedded in the shrinking ball algorithm algorithm itself (not the scaling). However, the outputs differ in how they handle noisy points, this means that the de-noising works differently depending on the sequence of data that comes in. The tiling method loads all the points at once, while the streaming algorithm can compute subsets multiple times and compute subsets without finalizing MAT's.

6.3 FUTURE WORK

6.3.1 Improving the streaming approach

At the current state the tiling method seems to outperform the streaming method. However, some optimizations could be made to make the streaming method perform better in computation time as well as memory usage.

- The reduced buffer method could be extended, in section 3.4 an attempt is made to use the domain decomposition lemma on pointclouds. On its own it does not perform all the time, however some with some additions it might be possible to determine which coordinates are needed for other tiles. If this works, less coordinates have to be stored in the main memory by the streaming method and possibly slightly reduce computation time as well. Some suggestions on what might produce better results:
  - Make sure edges remain in the dataset, in figure 3.17 it is shown that edges do not really have a discrete normal vector. Although it could be seen as a range of normal vectors. This also means that many other points need these edges.
  - leave a random thinned set of points, as the pointclouds are getting higher point density, leaving out a few points will not affect the results badly.
- While the optimization of the computation of the normal vector is out of scope, it should be improved in the streaming approach using reduced buffers. To compute the MAT the normal vector should be known a priori. However, in the streaming approach, the MAT is also computed for the buffers. Yet the normal vectors cannot be correctly computed for the buffer tiles, as points on the boundary of the buffer might need points from the neighbouring tiles to compute the normals.

6.3.2 Improving the tiling approach

- To reduce the hard disk usage during the computation of the MAT the normals could be computed between the segmentation and computation of the MAT.
- The collections are made using a "top down" segmentation method, while it is suited for subdividing a 2D/3D pointcloud into square boxes. A "bottom up" segmentation method might lead to more homogeneous segments.
- If the reduced buffer with removal of points (i.e. thinned reduced buffer) proves to be effective by applying the previous mentioned alterations (or by any other means), this method could also be applied on the tiling method. [1].

1 It has already been implemented in the segmentation stage, the MAT computation level should be adjusted to work with it

BIBLIOGRAPHY

Agarwal, P. K., Arge, L., and Danner, A. (2006). From point cloud to grid dem: A scalable approach 1. In *Progress in Spatial Data Handling—12th International Symposium on Spatial Data Handling*. Springer.

Amenta, N., Choi, S., and Kolluri, R. K. (2001). The power crust. *Proceedings of 6th ACM Symposium on Solid Modeling*, pages 249–260.

ASPRS (2013). *Las specification version 1.4 - R13*. ASPRS.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

Berger, M. and Silva, C. T. (2012). Medial kernels. *Computer Graphics Forum*.

Blum, H. (1967). A transformation for extracting new descriptors of shape. *Models for the Perception of Speech and Visual Form*.

Blum, H. (1973). Biological shape and visual science (part I). *Journal of Theoretical Biology*.

Choi, H. I., Choi, S. W., and Moon, H. P. (1997). Mathematical theory of medial axis transform. *pacific journal of mathematics*, 181(1).

Constantin, C., Brown, S., and J., S. (2010). Implementing streaming simplification for large labeled meshes. In *Proceedings Algorithm Engineering and Experiments*.

Danner, A. (2007). *I/O Efficient Algorithms and Applications in Geographic Information Systems*. PhD thesis, Department of Computer Science Duke University.

Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. (2006). *Algorithms*.

Dey, T., J Diesen, J., and Hudson, J. (2001). Decimating samples for mesh simplification. *Proc. 13th Canadian Conf. Comput. Geom.*

ESRI (2012). knowledge base - gis dictionary. Online article.

Finkel, R. A. and Bentley, J. L. (1974). Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*.

Grsoy, H. N. (1989). *Shape interrogation by medial axis transform for automated analysis*. Phd paper, Massachusetts Institute of Technology.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. (1992). Surface reconstruction from unorganized points. *SIGGRAPH Comput. Graph.*, 26(2):71–78.

Hoving, D. (2012). De 7 hoogste constructies van nederland. Online article.

IEEE Computer Society (2008). *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society.

Isenburg, M., Liu, Y., Shewchuk, J., and Snoeyink, J. (2006a). Streaming computation of delaunay triangulations. *ACM Transactions on graphics 25*.

Isenburg, M., Liu, Y., Shewchuk, J., Snoeyink, J., and Thirion, T. (2006b). Generating raster dem from mass points via tin streaming. In *Geographic Information Science—GIScience 2006*.

Jagadish, H. V. (1990). Linear clustering of objects with multiple attributes. *ACM SIGMOD Conf*.

Jalba, A. C., Kustra, J., and Telea, A. (2013). Surface and curve skeletonization of large 3d models on the gpu. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*.

Lindenbergh, R. C. (2014). Pattern recognition and classification. Lecture slides course Geo-signal analysis, Dept. of Geoscience & Remote Sensing TU Delft Netherlands.

Liu, L. (2011). *Multi-Dimensional Medial Geometry: Formulation, Computation, and Applications*. PhD thesis, Washington University in St. Louis.

Ma, J. (2012). 3d medial axis point approximation using nearest neighbors and the normal field. *VC*.

Morton, G. M. (1966). A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM.

Nodine, M. H. (1992). *Minimizing the Input/Output Bottleneck*. PhD thesis, Brown University.

Peters, R. (2014a). *Feature aware Digital Surface Model analysis and generalization based on the 3D Medial Axis Transform; PhD Research Proposal*. PhD thesis, TU Delft: Delft University of Technology.

Peters, R. (2014b). Towards medial axis-based point cloud simplification for lidar point clouds.

Peters, R. (2015). Point clouds: 3d medial axis transform. Presented at Geomatics Synthesis Project Symposium: Point Clouds 2015.

Peters, R., Ledoux, H., and Biljecki, F. (2015). Visibility analysis in a point cloud based on the medial axis transform. In *Eurographics Workshop on Urban Data Modelling and Visualisation*. Eurographics Association.

Philbrick, O., Cheng, G., Lley, R., Pollock, D., and Rosenfeld, A. (1968). Shape description with the medial axis transform. In *Pictorial Pattern Recognition*. Thompson Book.

Prinz, F. B. (1988). Geometric abstractions using medial axis transformation. Technical report, Geometric abstractions using medial axis transformation.

Reddy, J. and Turkiyyah, G. (1995). Computation of 3d skeletons using a generalized delaunay triangulation technique. *Computer-Aided Design*.

Rijkswaterstaat (2014). Het ahn.

Rijkswaterstaat (2015). Ahn3 voor iedereen online beschikbaar (ahn3 for everybody online available). http://www.ahn.nl/nieuws/2015/september-2015/ahn3-voor-iedereen-online-beschikbaar.html.

Sack, J.-R. and Urrutia, J. (2000). *Handbook of Computational Geometry*. Elsevier.

Samet, H. (1995). Spatial data structures. In Kim, W., editor, *Modern Database Systems*, pages 361–385. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Shakhnarovich, G., Darrell, T., and Indyk, P. (2006). *Nearest-Neighbors methods in Learning and Vision: Theory and Practice*, chapter Introduction, page 3. MIT Press.

Shamos, M. and Hoey, D. (1975). Closest-point problems. *IEEE Symposium on Foundations of Computer Science*.

Sherbrooke, E. C., Patrikalakis, N. M., and Brisson, E. (1995). Computation of the medial axis transform of 3d polyhedral. *Symposium on Solid Modeling and Applications*.

Silvia, C., Chiang, Y., Corra, W., El-sana, J., and Lindstrom, P. (2002). Out-of-core algorithms for scientific visualivisual and computer graphics. *In Visualization'02 Course Notes*.

Tam, R. and Heidrich, W. (2003). Shape simplification based on the medial axis transform. *VIS '03 roceedings of the 14th IEEE Visualization 2003*, page 63.

Vitter, J. S. (2007). External memory algorithms and data structures: Dealing with massive data. *ACM Computer Sureys*.

Wu, H., Guan, X., and Gong, J. (2011). Parastream: A parallel streaming delaunay triangulation algorithm for lidar points on multicore architectures. *Computers & Geosciences*.

A THINNED REDUCED BUFFER RESULTS

Figure A.1 displays the original dataset as well as the preprocessed dataset which will be compared to each other in order to find outliers. The preprocessed dataset has clearly fewer points in the tiles on the left side, it has been reduced in size by half. Further more some objects have been removed as well.

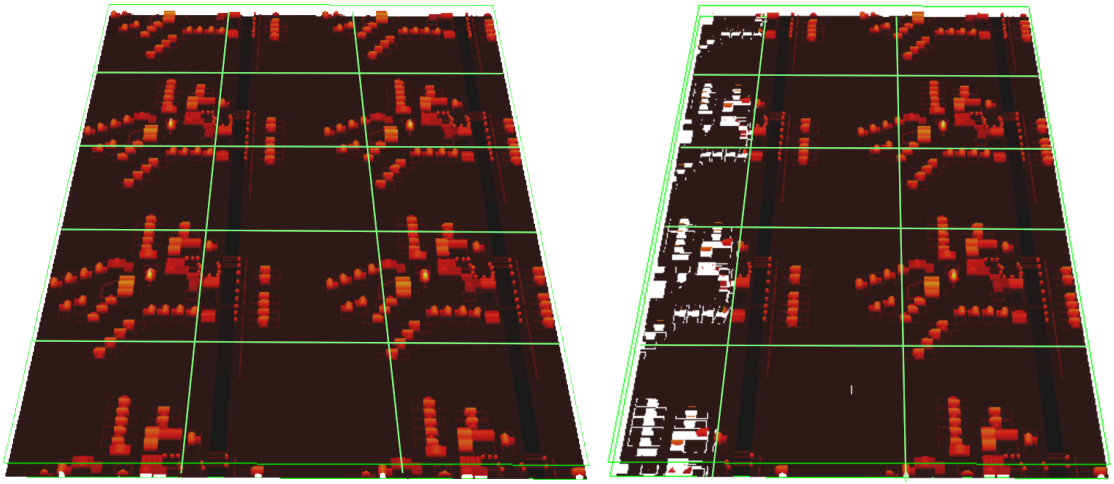


Figure A.1: (a) Original dataset (b) Preprocessed dataset

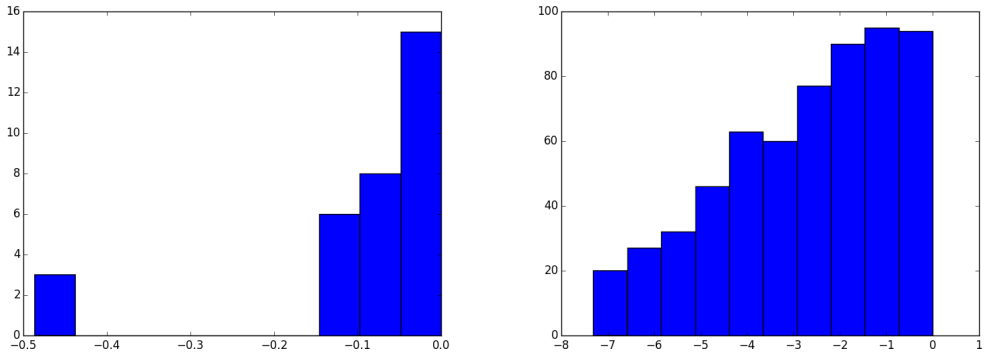


Figure A.2: Deviation of medial axis transform radius between the original dataset and the preprocessed dataset (a) Inner medial axis radius deviation (b) Outer medial axis radius deviation

When both of the mentioned datasets are processed some deviations in their MAT have been detected (see Figure A.2). The inner MAT seems to give good results, there are few deviations and the deviations themselves are quite small. The outer medial axis transform show much larger deviations, their radius deviates up to 7 meters. The reason for this is that the chance that a point computation actually needs another point from outside the boundary is much larger with the outer medial axis transform.

A.0.3 Thin objects

One of the reasons for large deviations is that thin objects disappear. These objects are a planar set of points with a width of just one point. Observe the first case in Figure A.3, it displays partially 2 neighbouring sets of point clouds. The red dashed line represents the border between them, the p_i the "to be computed" point, the q_o is the original" point to which the MAT is formed (e.g. in the original dataset), the q_t is the "actual" point to which the MAT is formed (e.g. in the thinned dataset).

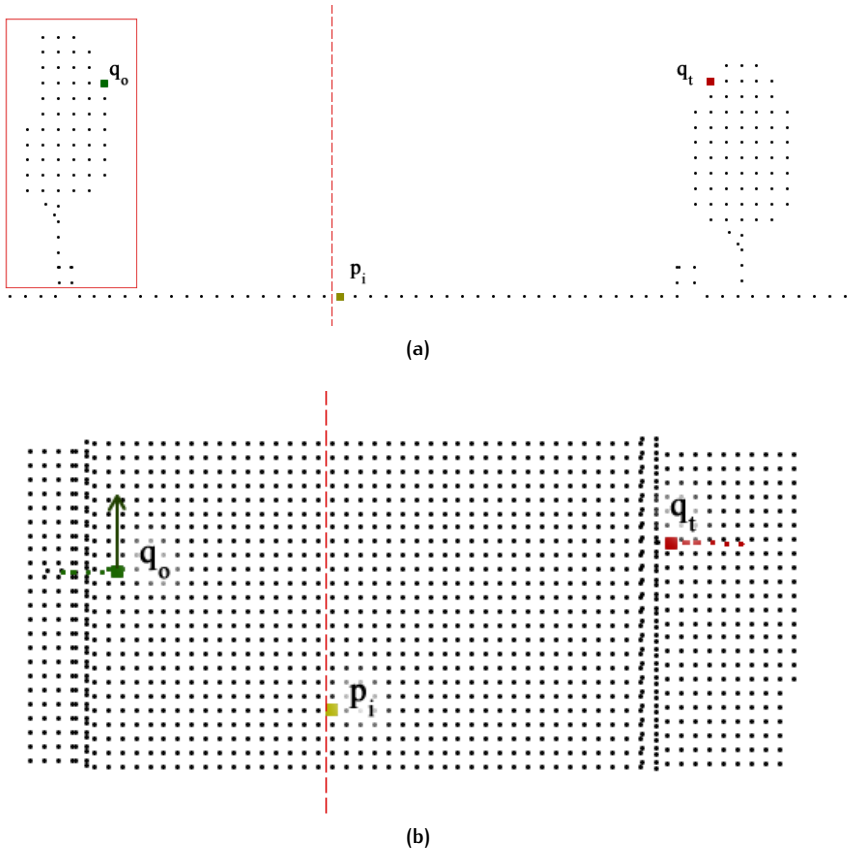


Figure A.3: (a) Front view (b) Top view

(a) shows at the left set a tree, marked by the red rectangle. In the original dataset it is present, however, in the preprocessed dataset it was removed. As a result q_o to which the p_i should form the MAT is also gone. The MAT is therefore created with the q_t.

Another case of incorrect normal computation is when points are very close each other. While the object has a width of more than just 1 point, the proximity of the points in the width is so small that errors occur. Observe

Figure A.4, it displays a short wall with q_0 and its normal vector highlighted. As a result of the formation of neighbouring points the normal vector is different than one would expect. This is caused by the short width of this wall. Because of this, the wall disappears during the preprocessing.

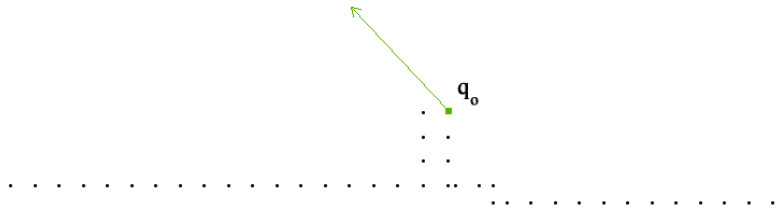
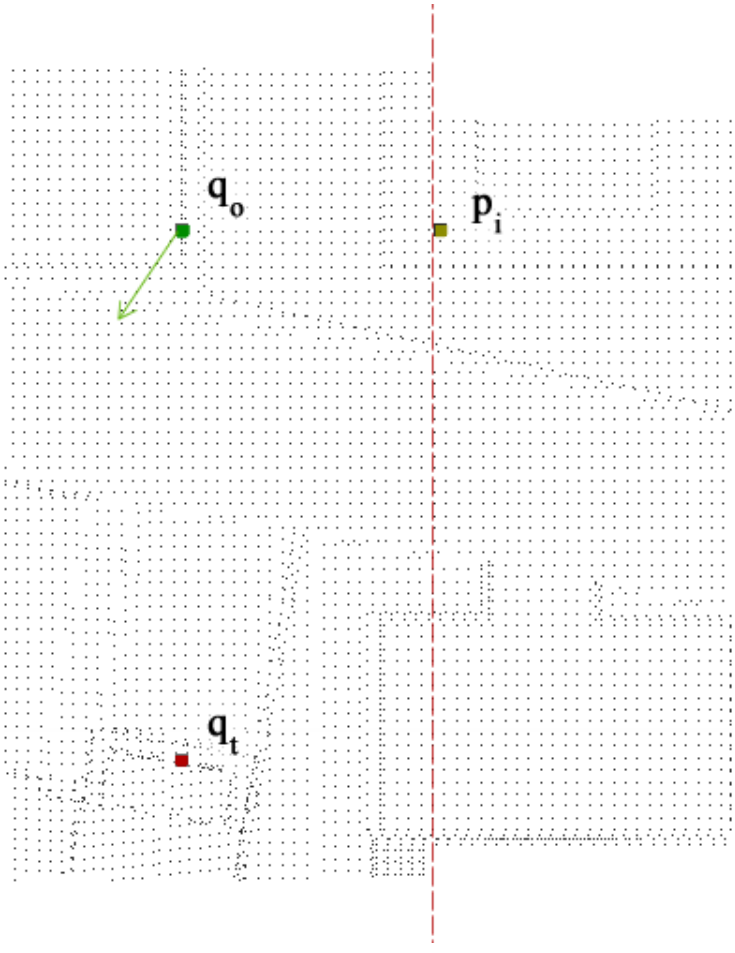


Figure A.4

(a)



(b)

Figure A.5: (a) Front view (b) Top view

B EXTRA IMPLEMENTATION RESULTS

Process #	Main Tile #	Max memory [MB]
1	21	497,319936
2	20	585,859072
3	23	582,144
4	22	719,93344
5	29	559,468544
6	28	613,482496
7	31	444,100608
8	53	642,510848
9	55	790,102016
10	61	916,811776
11	63	703,873024
12	60	1099,747328
13	62	835,260416
14	17	782,942208
15	16	859,623424
16	5	872,558592
17	19	807,366656
18	25	785,825792
19	18	766,980096
20	7	905,224192
21	24	719,937536
22	13	827,707392
23	4	800,309248
24	0	823,72608
25	1	774,750208
26	6	794,15296
27	2	799,1296
28	3	774,029312
29	12	816,61952
30	8	786,112512
31	9	742,125568
32	10	984,600576
33	11	1032,35584
34	32	1088,524288
35	33	1276,903424
36	34	1045,643264
37	35	1202,753536
38	44	1242,29632
39	46	926,06464
40	40	718,143488
41	41	774,701056
42	43	587,42784

43	42	470,929408
44	14	704,282624
45	36	772,132864
46	38	709,943296
47	15	584,871936
48	37	675,92192
49	26	548,94592
50	48	786,624512
51	27	571,199488
52	30	547,360768
53	52	711,135232
54	54	731,648
55	49	586,715136
56	39	668,504064
57	45	976,994304
58	47	761,532416
59	50	548,12672
60	58	775,96672
61	51	425,586688
62	57	439,23456
63	59	376,025088
64	56	151,298048

Table B.1: Maximum memory usage for several sizes collections (1600 x 1600 m)

C REFLECTION

The MAT is an alternative representation of geographical entities as massive pointcloud. It is retrieved by converting (manipulating) a outer boundary representing pointcloud. Two approaches for computing the MAT for large pointclouds using the shrinking ball algorithm were proposed in this thesis. The tiling approach however, proved to be the most efficient. The research was conducted from November 2014 to January 2015. The initial planning timeslots for literature research, studying of the existing algorithms, implementation of the approaches and comparison of them. In a later stadium, I decided to partly switch from the use of the python programming language to C++ as it would improve the processing time for the scaling approaches. Looping over large volumes of data tends to be faster in C++. However, this involved learning a new programming language took much time as well. The research period was extended to include further experiments, analysis and writing the report.

The methodical line of approach in Geomatics involves data capture, storage, analysis and visualization, along with quality control. As the datasets used were already available, there was no need to do it myself. Both scaling implementations had to deal with storage of the large datasets either temporary in on hard disk or in the memory, in such a way that it easy to use during the process. But to get to the approaches, the effects of scaling the shrinking ball algorithm on a pointcloud had to be analysed. Not all methods (buffers) turned out to be usable, as geographical pointclouds do have their flaws (i.e. points not homogeneously spread and noise). Quality control for both implementations took place in sense of memory usage and processing time.

Many applications and analysis are performed on the massive pointclouds representing the outer boundaries of geographic entities. However, the MAT enables faster and more intuitive methods for certain analysis and applications (i.e. analysing shape characteristics, retrieving thickness information and pointcloud simplification). By scaling the MAT, these possibilities can also be performed on massive pointclouds.

The final result are implementations of the tiling and streaming approach, they were optimized to be used on geographical pointclouds. While the tiling approach is currently the most efficient one, the streaming approach has many opportunities to be improved.

COLOPHON

This document was typeset using LaTeX. The document layout was generated using the `arsclassica` package by Lorenzo Pantieri, which is an adaption of the original `classicthesis` package from Andr Miede.