THE EFFECT OF A* PATHFINDING CHARACTERISTICS ON THE PATH LENGTH AND PERFORMANCE IN AN OCTREE REPRESENTATION OF AN INDOOR POINT CLOUD.

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

Olivier Rodenberg

June 2016

Olivier Rodenberg: *The effect of A* pathfinding characteristics on the path length and performance in an octree representation of an indoor point cloud.* (2016)

ISBN 999–99–9999–999–9

The work in this thesis was made in the:
Faculty of Architecture and the Built Environment
Delft University of Technology
...

Supervisors:	Associate professor dr. Sisi Zlatanova
	ir. Edward Verbree
Co-reader:	Dr.ir. Ben Gorte
Delegate of the Board of Examiners:	Dr.ir. Heidi Sohn

ABSTRACT

This thesis presents a new workflow for path finding through an octree representation of an indoor point cloud. I applied the following steps: 1) the point cloud is processed so it fits best in an octree; 2) during the octree generation the interior empty nodes are identified and further processed; 3) for each interior empty node the distance to the closest non empty node directly under it is computed, this can be used as constraint in path finding; 4) a network graph is computed with the connectivity of all interior empty nodes; 5) a collision avoidance system is pre-processed in two steps: firstly the clearance of each empty node is computed, and secondly the maximal crossing value between two empty neighbouring nodes is computed. 6) the A* path finding algorithm is conducted. The A* uses the interior empty nodes and the network graph to find a path.

Finally, benchmark test were conducted to identify the effect of octree operators and A* operations on the path length and computation time. The A* path finding computation time is positively effected by: firstly, pre-processing a network graph, using a Manhattan distance and by keeping the octree depth to a minimum. The path length is positively influenced by: extending the path connectivity, by using an Euclidean distance type and by increasing the octree depth. Although, the octree depth depends on the minimal and maximal spatial resolution needed for path finding. And this is dependant of the size and orientation of the point cloud. By pre processing a point cloud the spatial resolution can be increased.

ACKNOWLEDGEMENTS

I would like to express my gratitude to the people who supported me either directly or indirectly, while I was working on this thesis. First, I would like to thank my main supervisor Sisi Zlatanova for her guidance and the weekly indoor lab. I would like to thank my second supervisor Edward Verbree for our weekly meetings and his in-depth feedback on my research.

I would like to thank Isabelle Spigt for her support during my graduation period and for proof reading my thesis.

Furthermore I would like to thank all participants to the indoor lab for their feedback and interesting ideas. I also would like to thank Ben Gorte for co reading my thesis. And I would like to thank Robert Voûte for his interest in my research.

Finally, I thank my parents and the rest of my family and friends for their continued support during my student life. Thank you all.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

XOR exclusive OR 13
DUT Delft University of Technology 1

1 INTRODUCTION

Pathfinding is computing an optimal path between a start and a goal point. One of the most important and challenging aspects is finding a collision free path. Path finding is well established in 2D outdoor situation, think of software behind car and phone navigation systems [Noto and Sato, 2000]. Recently there is a growing demand for 3D indoor path finding applications [Isikdag et al., 2013]. Although this has been researched in the field of robotics in the 80 and 90 these algorithms are mainly focussed on 2D navigation. Besides the algorithms were developed for relatively slow robots. Nowadays we would like to have the ability to help people find a path in buildings. Or compute a path for a drone. What all these examples have in common is that an object with a certain geometry needs to find an optimal collision free path between a start and goal point.

To manage this we, at least, need to know the geometry of the object and a model of the environment. One way of representing the latter one can be with a point cloud. A point cloud is a large collection of points having at least an X, Y and Z coordinate representing a boundary of space. Figure 1.1 shows a point cloud of the pub of the Faculty of Architecture and the Built Environment at Delft University of Technology (DUT). New mobile laser scanners make it possible to generate a point cloud in a fraction of the time of traditional point cloud acquisition methods.

Figure 1.1: Point cloud of the Bouwpub.

The model of the environment alone does not give enough information to find a route, for this the empty (pointless) space is needed. A common method to structure and segment a point cloud is with the use of an octree data structure. An octree consist of a cubical volume which is recursively subdivided 'into eight congruent disjoint cubes (called octants) until blocks of a uniform colour are obtained, or a predetermined level of decomposition is reached' Samet [1988]. They are used for the partitioning of space and result in a hierarchical tree structure. This make operations like neighbour finding

and indexing efficient (Vörös, 2000). Figure 1.2 shows how the subdivision an quadtree, the 2D counterpart of an octree.

Earlier research, like Wang and Tseng [2011] use the octree data structure to segment point clouds. And in the research of Zhou et al. [2011] an octree structure is used for surface reconstruction.

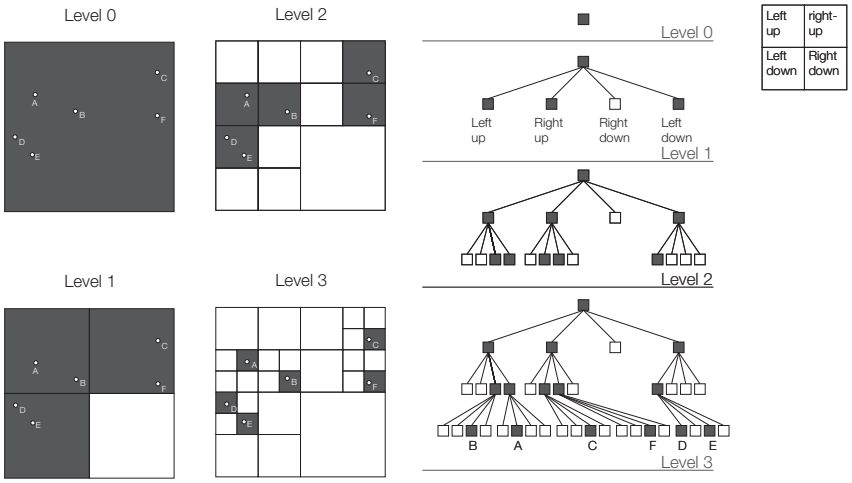

Figure 1.2: an visualization of a quadtree

One of the advantages of an octree is efficient structuring of space, large empty space can be represented by a large node high in the octree, as shown in Figure 1.3 by a quadtree, the 2D counterpart of an octree. These large empty nodes reduce the amount of nodes in the octree. This is an advantageous property of an octree for path finding. The large nodes reduces the amount of nodes and subsequently the possibilities to discover in path finding.

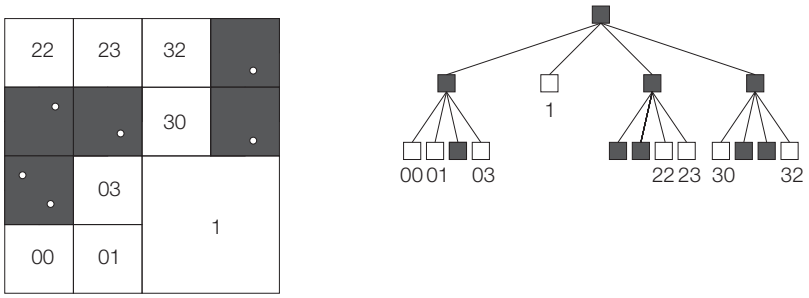

Figure 1.3: *For visualization purposes a quadtree is used instead of an octree to explain the concept.* Large empty spaces can be represented by a node which is high in the quadtree. The left image shows the a quadtree with some non empty nodes and empty nodes. The right image shows the location of these nodes in the octree. Notice that a large area in the left image refers to a higher location in the octree.

In the final year of the master Geomatics I worked in a team on a project about explorative point clouds. An explorative point cloud holds as advantage no need for an intermediate geometric model [Verbree and Van Oosterom, 2014]. During the project we created a work flow to identify

the empty space inside a point cloud using an octree data structure [Broersen et al., 2016]. And we developed a rudimentary A* path finding algorithm using the empty nodes in the octree data structure. This thesis continuous the work of Broersen et al. [2016]. Section 2.1 describes the work in detail.

1.1 CONTRIBUTION

The scientific and social contributions:

1.1.1 Scientific contribution

The scientific contribution of this research is twofold. Firstly an indoor path finding application through the empty space of a point cloud is developed. This application uses an octree data structure as a kind of catalyst for path finding.

The method consist of the following components: a work flow to pre process a point cloud for optimal classification into an octree structure. A novel method to identify interior empty nodes and at the same time the downward distance from each interior empty node to the closest non empty node is computed. A new method for pre-processing a collision avoidance system. A method to process a network graph during octree generation. And finally a path finding application, which uses the network graph and collision avoidance system.

Often assumptions are made in a design processes. For example, having less nodes in an octree reduces the amount of computation in a path finding application. However, the exact influence of an assumption is often ignored and therefore not known. This is also the case in path finding applications. Thus the second contribution aims to identify the effects of octree operators and A* operations on A* path finding. It will research the impact on the computation time and path length. To my best knowledge no research has been conducted identifying the effects of these components in A* path finding through an octree data representation of an indoor point cloud.

1.1.2 Societal contribution

Recently there is a growing interest and demand of 3D indoor path finding applications [Isikdag et al., 2013] [Liu and Zlatanova, 2011]. Examples of this are:

When a fire fighter enters a building where sight is blocked by smoke, he always walks along a wall. It could be useful for a fire fighter to know how to navigate through a building by the use of a small screen in his/hers helmet. For this, the path should be constraint to nodes with a certain distance to a wall. Or in cases where it is to dangerous to send in a fire fighter a drone could be used to asses the situation. For this a collision free path should be computed. In this cases the building geometry should be available.

In emergency situation an indoor path finding method can be used to direct persons to the closest emergency exit. In an hospital it can be used to direct patients to the room of their doctors appointment. In big buildings like airports or congress buildings indoor path finding can be used to navigate

people to certain locations or compute a closest meeting point. Furthermore, an indoor path finding application can be used for automatic monitoring of building. A unmanned airborne vehicle can be used to inspect hard to reach areas in an automated way, think of inspecting the condition of pipes on the celling. Even in underwater situations it is very useful to have an path finding method generating a collision free route [Guang-lei and He-Ming, 2012].

With an indoor path finding method outdoor and indoor navigation could be seemingly merged into one system. Think of situations where someone enters an unknown city via the train station and needs to go to a certain room in a building. Nowadays the navigation stops at the door step of the building. Especially in large buildings it might be useful to extend this navigation to the room.

1.2 RESEARCH OBJECTIVES

The goal of this research is to identify the effects of geometrical point cloud operations, octree operators and A* operations on A* path finding. Therefore the following research question is formulated:

What is the effect of A path finding characteristics on the path length and performance in an octree representation of an indoor point cloud?*

Where the A* path finding characteristics in an octree representation of a point cloud are defined by: Distance type, connectivity, is a network graph pre-processed and by the octree characteristics. Where the octree characteristics are defined by the octree depth and the point cloud characteristics. Which are defined by the direction, location and scale of the point cloud.

To answer the research question the following three sub questions are formulated:

1. *What geometrical point cloud processing operations are important for the generation of an octree and what is their effect?*

2. *What octree operators influence the performance and path length and what is their effect?*

3. *What A* algorithm operations influence the performance and path length and what is their effect?*

1.3 RESEARCH SCOPE

The scope of this research focusses on path finding through the interior empty space of an indoor point cloud. Although the world is not static, it assumed that the models in this research are static. The research focusses solely on A* path finding. An octree data structure will be used to classify the point cloud and identify the interior space.

1.4 OVERVIEW OF THE THESIS

The remainder of this thesis is structured as follows: Chapter 1 introduces the subject and provides the research objectives. Chapter 2 provides the background and related work to this research. Chapter 3 explains the methodology used in this research. Chapter 4 provides the implementation of the methodology. Chapter 5 presents the main results of the research. And finally, Chapter 6 concludes the research by answering the research questions and providing a future work section.

2 THEORETICAL BACKGROUND AND RELATED WORK

The related work in this research is fourfold: Section 2.1 provides a detailed description about the method created during the synthesis project. Section 2.2 gives an overview of path finding method in quadtree and octree data structures. Section 2.3 provides related work about neighbour finding in quadtree and octree data structures. And finally, section 2.4 presents collision avoidance methods in octree structures.

2.1 PROJECT POINTLESS, OCTREE GENERATION

The background of this research is the work of Broersen et al. [2016]. During the project we created a work flow to segment a point cloud in an octree. Besides we implemented a rudimentary path finding method. We created a linear octree, which means the octree is stored in a linear array instead in a tree. To refer to the nodes in the octree each node has a spatial location code. The spatial location code describes both the location in the octree, in the geometrical model and the size of the octant as a string of numbers Van Oosterom and Vijlbrief [1996]. Although the method was created for indoor point clouds it can be used for outdoor point clouds.

2.1.1 Octree generation

The octree generation method can roughly be divided into three steps. First the point cloud is pre processed, secondly the non-empty (black) nodes are computed. And finally the empty (white) nodes are derived from the non-empty nodes.

The first step involves processing the point cloud. The point cloud is translated so its origin has coordinate $(0, 0, 0)$. Next, the point cloud is scaled, so each node in the octree has a coordinates between integer 0 and integer 2^n.

Secondly, the non-empty nodes are computed. The non-empty nodes are generated by searching the location of each point in the octree. Because all nodes in the octree have a coordinate between integer 0 and 2^n each point can be snapped to a node by removing the decimals of the x,y and z coordinate. By removing the decimals of a point with coordinates $(1,1;1,3)$ it can be snapped to a quadrant where the lower left coordinates are $(1;1)$, see figure 2.1.

And finally the location code of each node needs to be computed. The octree is generated by recursive subdivision of cubes into eight children nodes. Therefore, each level in the three can be encoded with a single number between 0 and 7 (see figure 2.2). For each level in the octree a

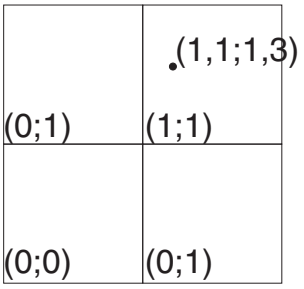

Figure 2.1: A point can be snapped to a quadrant by removing the decimals

number is concatenated to the string of the node. The final string of numbers is called the location code. The location code of a node indicates the path through the octree to reach that node, figure 2.3 shows the process of numbering nodes in an quadtree. To have the highest possible resolution of the empty space, all points are forced to split until the maximum octree depth. The location code is obtained by a technique called binary masking. The goal of the technique is to acquire the location of a node in each octree level. To do this a binary mask is used to check on which side of a split line a node lays in a certain octree level. This is done in the X, Y and Z direction (see figure 2.4). For example, a node lays: right or left of the split line (x direction), above or below (z direction) and front or back (y direction). With this information a number can be assigned to the location of the node. The number of this octant is concatenated to the location code. This is repeated until the maximum split level is reached. When computed, each point has a spatial location code consisting of a string of numbers. All the location codes of the non empty nodes are used to derive the spatial location codes of the empty nodes.

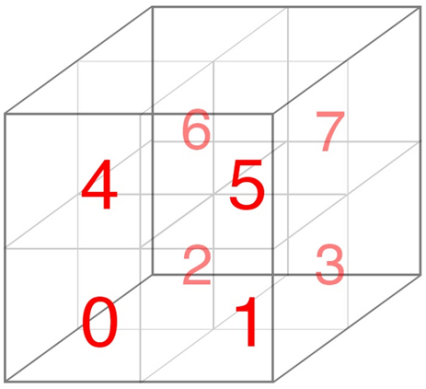

Figure 2.2: Z numbering of the octants, source: [Broersen et al., 2016]

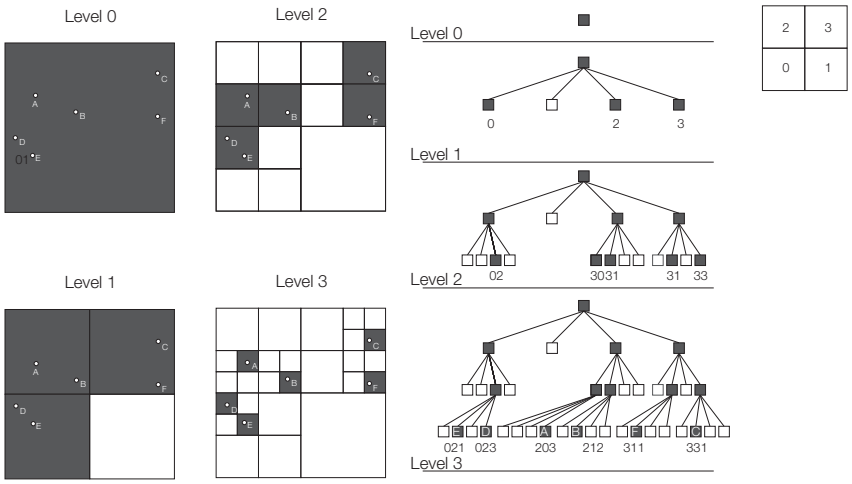

Figure 2.3: Numbering of the non-empty nodes in a quadtree

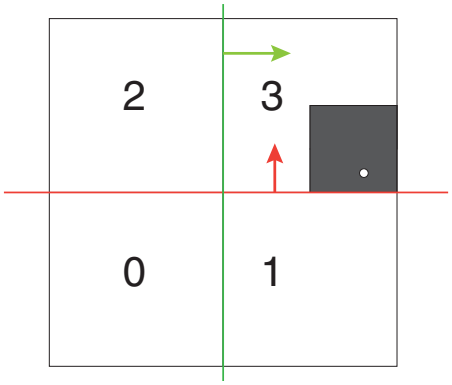

Figure 2.4: *(For visualization purposes a quadtree is used instead of an octree to explain the concept.)* The non-empty (black) node lays right of the green split line and above of the red split line, thus is lays in quadrant 3.

The final step is to derive the empty nodes from the non-empty nodes. This is done by taking the inverse of the non empty nodes. Important to note is that this process is performed per octree level, starting with the highest level where nodes are of maximum size. Figure 2.5 shows the process for a quadtree for two levels. For the first level it is checked if there are any non empty nodes intersecting with a mask with location code: 0, 1, 2 and/or 3. If the mask does not intersect with any black nodes the node is empty. For the next level the nodes which intersected in the previous step are split into four new masks. And they are checked for intersections.

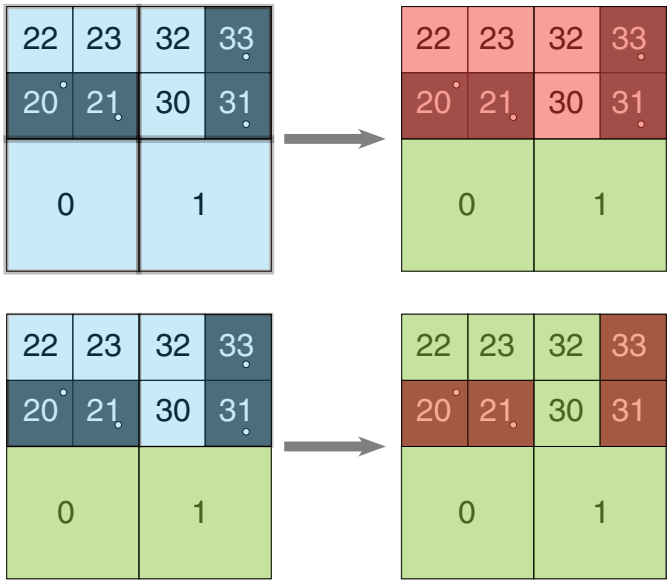

Figure 2.5: Binary masking in a quadtree.

2.1.2 Database management system

Both the location codes of the non-empty and empty nodes are dd in a database. Two tables in a single database were created, one for the non empty nodes and one for the empty nodes. There was no single table for the point cloud as the points were stored in the non empty nodes. Because most non empty nodes consisted out of multiple points it is possible that there are multiple rows with an identical spatial location code. In hindsight it would make more sense to cluster the points in a single row.

One could suffice by storing only the location codes for each empty node, ss the location and size of a node can be derived from this location code. Since the octree would be used for other applications like path finding we also stored the x, y and z coordinate and the leaf size of each node.

2.1.3 Path finding

The path finding method searched a shortest path based on an A* algorithm, section 2.2 describes the A* algorithm in detail. The method was able to move between nodes sharing a face, which were computed on the fly. The method had no collision detection. Another problem of the method is that there was no distinction between interior empty space and exterior empty space. This is a problem because the exterior empty space has been created during the octree generation without knowing if this really is empty space.

2.2 PATH FINDING ALGORITHMS IN OCTREES AND QUADTREES

This section describes the related work about path finding in octree and quadtree structures.

Path finding is computing an optimal path between a start and goal node. An optimal path does not imply this should be the shortest. Many parameters can define the term 'optimal'.

1. the Dijkstra algorithm

   The Dijkstra algorithm computes an optimal path between a start and a goal node by searching for the minimal travel cost. The algorithm starts with a start node, of which the movement cost to all adjacent nodes are calculated and are placed in an open list. The node with the lowest cost is marked. The movement cost from the start node to the neighbours of the marked node is calculated and are append to the open list, the node in the open list with the lowest cost is marked. The last two steps are repeated until the goal node is marked.

   Because the algorithm expands the node with the lowest cost it will expand concentric around the start node. Therefore, the speed of the algorithm depends on the size of the network and the distance between the start and goal node. This makes the Dijkstra algorithm unsuitable for real-time applications [Noto and Sato, 2000].

2. the A* algorithm

   The A* algorithm is an extension of the Dijkstra algorithm. The A* algorithm introduces a heuristic cost, which is an approximation of the cost from the marked node to the goal node Hart et al. [1968]. The total cost ($f(c)$) for node (c) is now:

   $$f(c) = g(c) + h(c) \tag{2.1}$$

   Where $g(c)$ is the cost from the start node to node c. And $h(c)$ is the heuristic cost from node c to the goal node [Nosrati et al., 2012] [Kambhampati and Davis, 1985].

   Due to the heuristic cost all nodes which are not between the start node and the goal node have a higher heuristic cost compared to the nodes which are not. Therefore, the speed of the algorithm is not related to the network size, but only to the length of the route. Figure 2.6 shows the difference between the algorithm of Dijkstra and A*.

Herman [1986] presents a combination of two path finding methods in an octree representation. It combines a hill climbing method in combination with an A* algorithm. Hill climbing is able to quickly compute a path because it only uses the distances to adjacent nodes to decide the next node. However, this method has a tendency to get stuck in 'U' shaped obstacles. When this happens an A* algorithm is used until the obstacle is avoided. This method is fast, although it trends to move away from the shortest path.

Vörös [2001] uses a hill climbing method in combination with a distance map to compute a path in a quadtree and octree representation. By using an octree representation instead of a voxel approach the memory demand and path finding time is reduced. The disadvantage of a distance map is the need for a specific distance map for each distinct goal node.

Xu et al. [2015] describes an A* path finding method in an octree representation. Neighbours of a current node are found by checking what

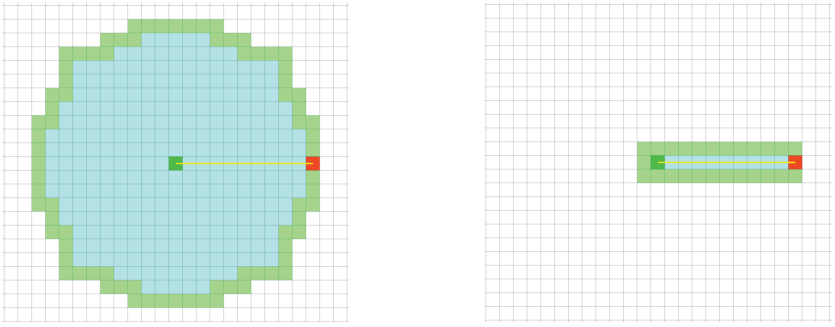

Figure 2.6: The left image shows a a route computed between the start(green node) and a goal (red node) using the Dijkstra algorithm. Al blue nodes have been marked as lowest in the open list. The right image shows a route compute by the A* algorithm. Due to the Heuristic cost in the A* algorithm the amount of nodes processed is much smaller compared to the Dijkstra algorithm. source: Xu [2012]

nodes have a distance not exceeding half the length of the sum of the lengths of the two octants. This is a computational expensive operation. The research did not include a method to avoid collisions in the algorithm.

Hwang and Ahuja [1992] use a potential field to navigate in an octree. The potential field indicates a heuristic potential of each node, in such a field there are local minima (nodes with a low potential). First a graph is computed between these local minima. A global planner searches a path in the graph. Subsequently, a local planner checks if the path is collision free. One of the advantages of an octree is efficient structuring of space, large empty space can be represented by a large node high in the octree, as shown in Figure 1 by a quadtree, the 2D counterpart of an octree.

Besides identifying the empty space Broersen et al. [2016] created a simple path finding algorithm based on an A* algorithm. The route was computed through the empty space in the octree. The path finding method considered two nodes as neighbours if they share a common face. Neighbours could be smaller, larger and of equal size. The neighbours are computed on the fly. Object avoidance was not implemented and no distinction between interior and exterior empty nodes was make.

Kambhampati and Davis [1985] propose a multi resolution path finding method in a quadtree. To reduce the amount of leaf nodes a pruned quadtree is created. In a pruned quadtree grey nodes containing empty (white) and non-empty (black) are possible neighbours.

2.3 NEIGHBOUR FINDING AND CONNECTIVITY GENERATION

All path finding algorithms rely on the exploration of adjacent nodes. In an octree structure, two nodes can be each others neighbour if they share a common: face, edge or vertex. Figure 2.7 illustrates the types of connectivities. In total there are 26 possible neighbours for each node. This section describes the related work of neighbour finding.

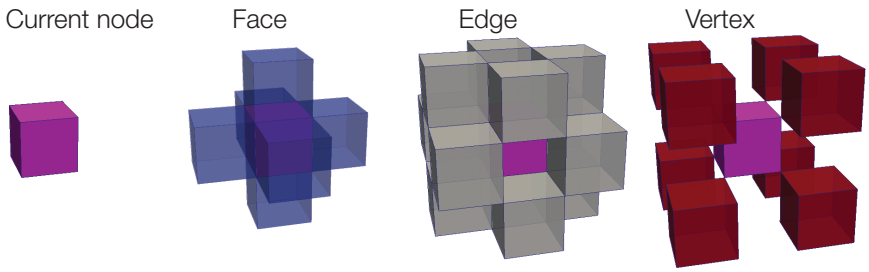

Figure 2.7: From left to right; neighbours sharing a common: face (6), edge (12), vertex (8)

In the neighbour finding method of Gargantini [1982] motion is possible in the face direction. Each octant has a unique location code. This location code describes the location of a node inside the octree. A distinction is made between nodes sharing a common parent and nodes having a different parent node. Adjacent nodes are computed using a separate function for each direction. The main drawback of this method is that the algorithm does not know if a neighbours is part of the octree.

Samet [1989] proposes a method to compute neighbours the direction of a face, edge and vertex of all possible sizes (smaller, equal, larger). Neighbours are found by ascending the octree in search of a common ancestor. The neighbouring node is found by descending the tree with mirrored moves. The common ancestor of a node is different for each direction. The method does not know if a neighbour is part of the octree (when the current node lays on the boundary of the octree) [Vörös, 1997]. Besides the method does not identify the octants with a hierarchy location codes. As instead it enumerates the octants.

The method of Samet [1982b] was later improved by Vörös [2000] on three areas. He used the work of Gargantini [1982] to implement the difference between inner (having the same parent node) and outer (having a different parent node) neighbours. Using location codes the octree could be stored as linear area instead of a tree structure. The inner neighbour is found by changing the directional relevant bit of the most right digit of the location code. This is done with an exclusive or exclusive OR (XOR) operation on the appropriate number of the location code and a bit mask. A bit mask is used to access the (directional) relevant bits of the location codes. Figure 2.9 illustrates the bit masks for the X,Y and Z direction. For example, the location code of the neighbour of node 000(numeric =0) in direction Dx, having the bit mask 001, is computed with the following XOR operation: 000 or $001 = 001$.

The computation of an outer neighbour in direction Dx is more complex. To know how many digits of the location code need to change, the closest common ancestor of the node needs to be computed in direction Dx. The steps to the closest common ancestor are the amount of digits which need to be changed in the location code. The method of changing these digits is the same as the inner neighbour. If the steps to the closest common ancestor exceed the steps to the root node, the outer neighbour in that direction does not exist [Vörös, 2000].

The closest common ancestor of node c is the parent node of the first ancestor node of node c which is located on the opposite side on axis

compared to the location of node c related to their individual parent nodes, see figure 2.8.

The closest common ancestor is found by testing the directional relevant bits of the appropriate number in the location code. Figure 2.9 illustrates the relevant bits for the X, Y and Z direction. Staring at the most right number in the location code, the numeric number is converted to an bit number. Next, the directional relevant bits of the next left numbers is checked with the most right number until the first complement bit is found.

The number of steps to ascend to the common ancestor refers to the amount of digits in the location code which need to be changed. Besides, this number is used to make sure a common ancestor is not higher than the root node. Thus, neighbours are not found when they are situated outside the boundary of the octree.

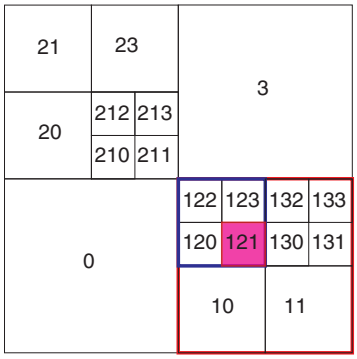

Figure 2.8: Node 121 is situated right (on the x axis) in its parent node. The closest common ancestor is node with location code 1 (red) because the blue node with location code 12 is located on the opposite side related to node 121;

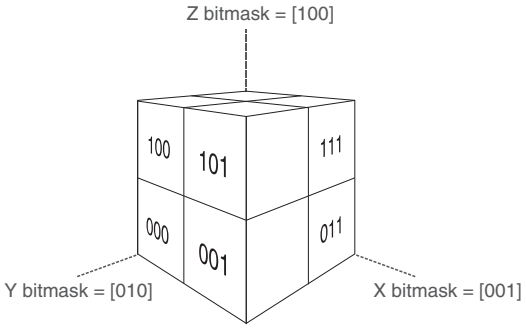

Figure 2.9: Bitmask for the X Y and Z direction.

Xu et al. [2015] checks for face neighbours based on their geometrical location. Two nodes are neighbours if their centre points have a distance half the length of the sum of the octant sizes.

The method of Kim and Lee [2009] uses a lookup table to find face neighbours in a direction. The table is used to change the location code of the current node to compute the neighbour in a certain direction with simple arithmetic operations. Further, the paper describes a method to find neighbours within a radius r. This is done with the geometrical operation

within. This is an expansive computation as they do not make a selection using the octree structure.

Payeur [2006] provides a method to search neighbours in the face, edge and vertex directions and for all possible sizes. A lookup table is used to compute neighbours. Each type of neighbour has its own lookup table. The octants in the octree have a location code similar to that of Major et al. [1989] and Vörös [2000]. If the neighbour of a node in direction Dd share the parent node, only the last digit of the node needs to be changed. If not, the lookup table defines per digit what its transformation is and when the last digit is changed using simple arithmetic operations. Neighbours of node x which are larger are found by deleting the last digit in the neighbours of node x of equal size. Neighbours smaller than node x are found by recursively concatenating digits to the location code to equal sized neighbours of the current node x until the maximum octree level is reached. The method does not explicitly provide a method to exclude neighbours which are outside the boundary of the octree. Is does give two options which could be researched.

In the method used in this thesis this is not a major problem. This is because the found neighbours are compared to the white nodes. So neighbours which do not exist would be invalid and not used in path finding.

Namdari et al. [2015] describes a method neighbour finding during the octree construction. The method is used to find all 26 neighbours in all sizes. The method is based on a bread-first search octree generation. Meaning: the nodes of the octree are created in a top down approach, so from the root node to the leaf nodes. The computational effort is minimized by searching equal and larger neighbours. To make sure also smaller neighbours are stored, a neighbouring connection is stored in both directions. If node X has a neighbour node Y, this neighbour information should be stored in both node X and Y. This last method ensures that the smaller neighbours of a node are found and stored in a later stage of the octree construction. The actual approach in which neighbours are found is quite basic and does not take advantage of the locational codes. It basically checks if two nodes share x, y and/or z coordinates.

2.4 OBJECT AVOIDANCE

Collision avoidance can be pre processed or done during path finding. This related work in this section is limited to the pre-processed methods.

Jung and Gupta [1996] uses a distance map for collision detection. For each node the distance to the closest object is calculated. In the method, motion is not limited between centre points of nodes, as instead motion is possible from any point in the octree. For this reason one value indicating the minimal distance to an object is not sufficient. Therefore, for each node there are two values stored, a minimal value and a maximal value to the closest object. In search for the closest node the algorithm searches outward from the node for obstacle nodes. If one is found, the Euclidean distance is calculated. The actual collision detection occurs during path planning, for each target node it is computed if it is collision free considering the two distance values.

Samet [1982a] describes an efficient method, which can be used, to compute the closest boundary with a black node for each white node in a quadtree. The method is based on the theorem: not all equal sized neighbouring nodes of a white node can be white. Since merging between

the nodes will take place and the white node would not exist, see figure 2.10. If the closest boundary is computed with a chessboard distance the boundary must be, or a descendant of, one of the eight equal neighbours, see figure 2.11. The amount of neighbours increase if an Euclidean or Manhattan distance is used, see figure 2.12.

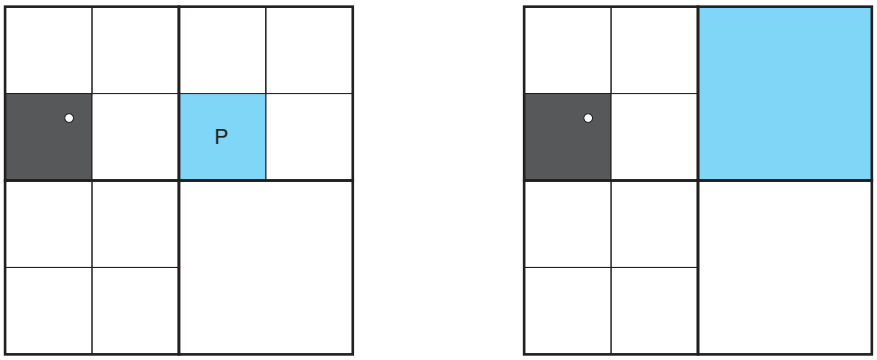

Figure 2.10: Not all neighbours of node p can be white otherwise merging will take place

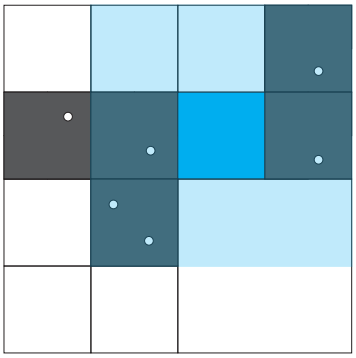

Figure 2.11: The closest border with a black node and the dark blue node must lie in the light blue area

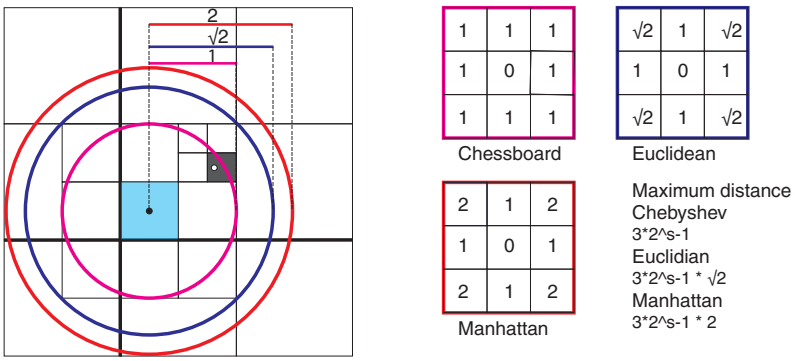

Figure 2.12: The coloured circles indicate in which area the closest non empty node can be

In a potential field approach an artificial field is generated in which both the target node and all obstacles direct a force on each white node in a field. The target node has an attractive force and the obstacles have a repulsive force. These forces are strong at the source and gradually decrease as the distance to the source increase, see figure 2.13. The sum of these forces are the potential value of a node. Together all nodes create a potential field. The maximal potential field is in the obstacles, a low potential field is thus favourable in a path finding applicationsHou and Zheng [1994] Hwang and Ahuja [1992].

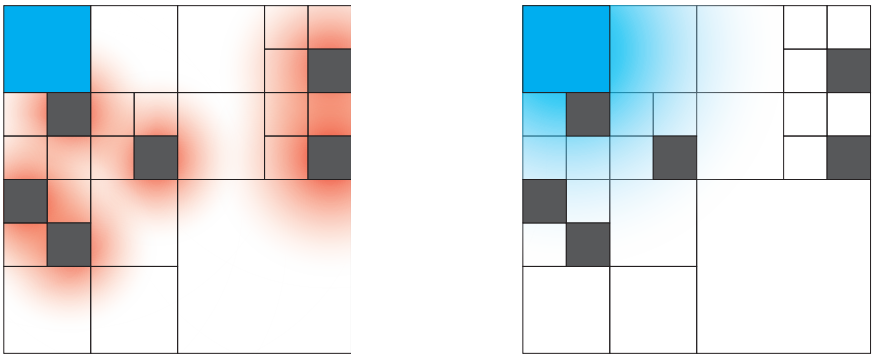

Figure 2.13: Black nodes have a repulsive force and the target node(blue) has a attractive force on the white nodes

Wu and Hori [2006] uses a potential field to avoid obstacles. Obstacles have a repelling force and the target an attracting force. A path is computed in the potential field. The potential in each node is calculated. A high value indicates objects to be close. This method is developed for robot arms. The potential field needs to be computed for each octree level. The disadvantage of the method is the need for a specific potential field for each distinct goal node.

Hamada and Hori [1996] combines a global path planner with a local path planner. Collision detection is performed in the local path planner. They basically check if the nodes of an object intersects with the nodes of an obstacle. The process starts at the highest level and checks if it intersects with a empty node (no collision) of with a non empty node (collision). This process continuous by ascending the tree until it reaches a empty node or a non empty node. The main drawback of this method is the high computational effort.

Kambhampati and Davis [1985] creates a buffer around object to prevent collisions. This buffer must have a minimal distance of half the object size, see figure 2.14. The drawback of this method is that the buffer works only for objects of the same size. Otherwise an other buffer needs to be computed.

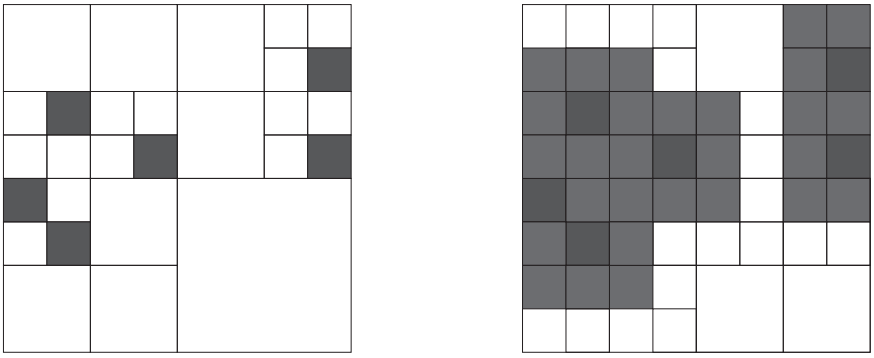

Figure 2.14: A buffer is created around each black node ensuring no collisions are possible

3
METHODOLOGY

The goal of this research is to identify the effects of geometrical point cloud operation, octree operators and the A* operations on A* path finding. For this, an octree needs to be constructed from a point cloud which can be used as a kind of catalyst for indoor path finding. An overview of the methodology is illustrated in Figure 3.1.

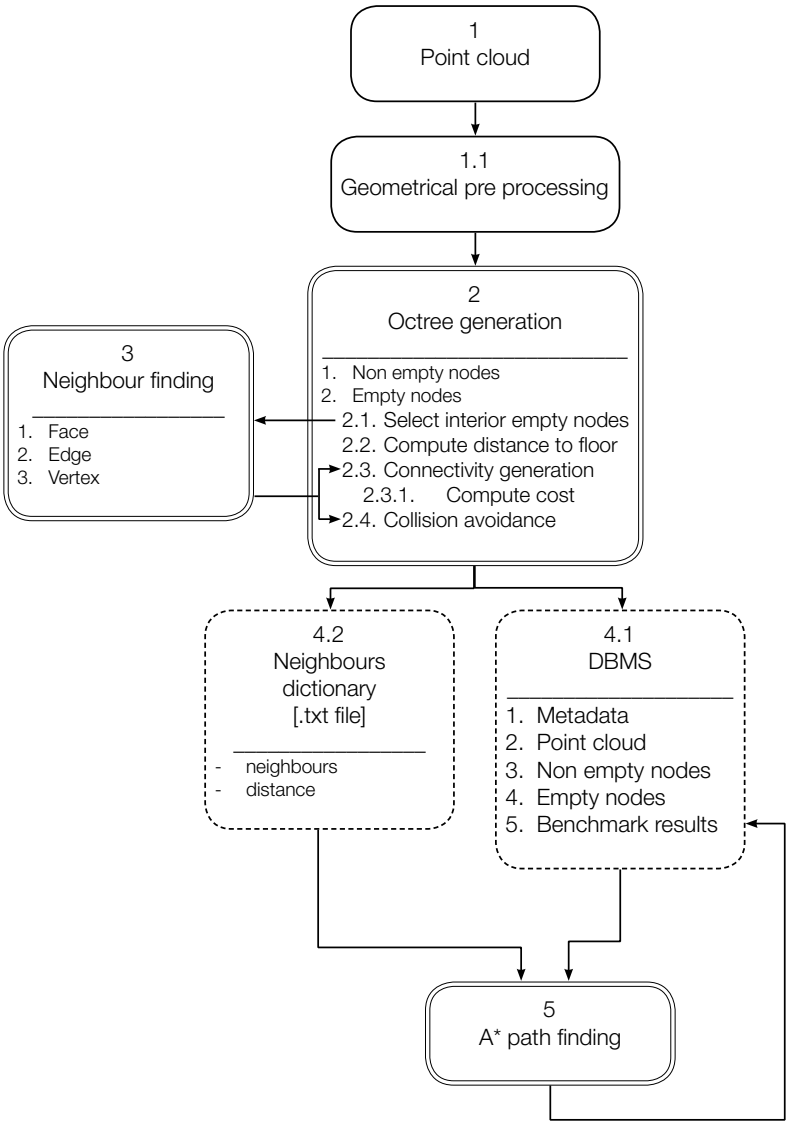

Figure 3.1: Overview of the method, boxes with double lines are scripts, boxes with dotted lines are data files

The basic idea of the method is to use an octree to classify a point cloud. The octree is used as a kind of catalyst for A* path finding. During the octree generation the non empty and empty nodes are generated as described in section 2.1. However, the empty nodes processing method is extended with a number of components. Firstly, interior empty nodes are identified. Only for the interior empty nodes a connectivity is generated. For the generation of the connectivity all possible neighbours are computed. Using the possible neighbours, a pre-processed collision avoidance system is computed.

The output of the octree generation script is: the connectivity of each node, a table of non empty nodes, a table of empty nodes and metadata. All but the connectivities are stored in a postgreSQL database. The connectivity or network graph is stored as a .json file.

The A* path finding algorithm takes as input the connectivities and the empty nodes. All results of A* path finding are stored in the same postgreSQL database.

The remainder of this chapter is structured as follows: Section 3.1.1 describes the point clouds used in this research and geometrical point cloud operation. Section 3.2 describes how to identify interior empty nodes. Section 3.3 describes how to construct an interior node connectivity. Section 3.4 presents the method to pre process a collision avoidance system. Section 3.5 provides a description of the different distance types used in this research. And finally, Section 3.6 explains how the effect of octree operators and A* operation on A* path finding is researched.

3.1 POINT CLOUD

This section describes the point cloud datasets used in this research. The next subsection describes a work flow to geometrical pre process a point cloud for optimal classification.

3.1.1 Point cloud datasets

In this research three point cloud datasets are used. Two datasets of existing buildings and one dataset created from a selection of random points.

point cloud	points	bounding box[m]
Test point cloud	3000	64*64*64
Bouwpub	2.196.903	15,94*9,20*6,18
Fire department	2.266.067	10,74*13,87*12,71

Table 3.1: Metadata about the point cloud datasets

The first point cloud is of the pub of the Faculty of Architecture and the Built Environment at DUT. This dataset will be called 'Bouwpub' in this research. Table 3.1 describes the metadata of the point cloud and Figure 3.2 presents the point cloud of the Bouwpub.

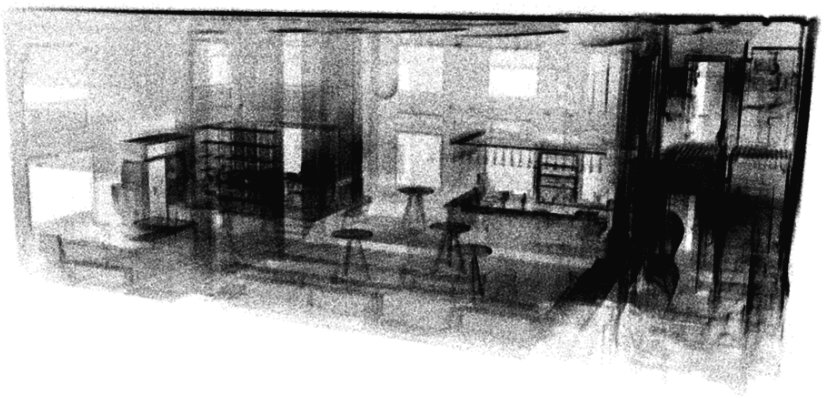

Figure 3.2: Point cloud of the Bouwpub.

The second point cloud is a section of the fire department in Berkel en Rodenrijs. This dataset will be called 'fire department' in this thesis. Table 3.1 describes the metadata of the point cloud and Figure 3.3 illustrates the point cloud.

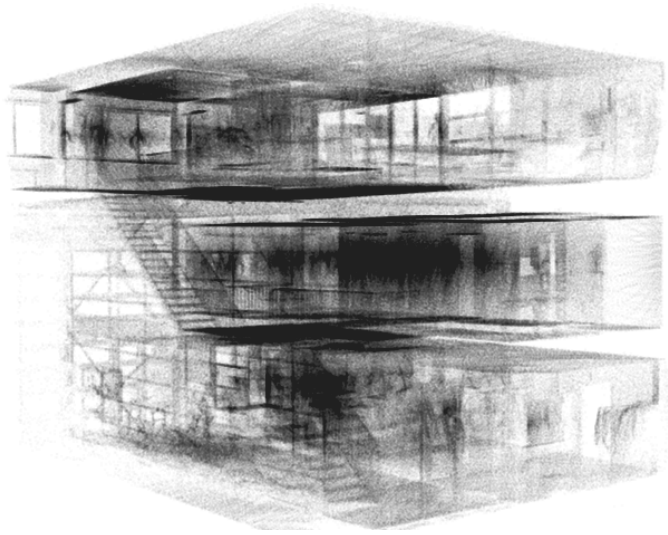

Figure 3.3: A section of the point cloud of the fire department.

The last point cloud is a selection of random points, which was created with a simple python script. After a few tests with a different amount of points, a point cloud consisting of 3000 point proved suitable for this research. The points were generated in a grid of $64 * 64 * 64$. Table 3.1 describes the metadata of the point cloud and Figure 3.4 illustrates the point cloud.

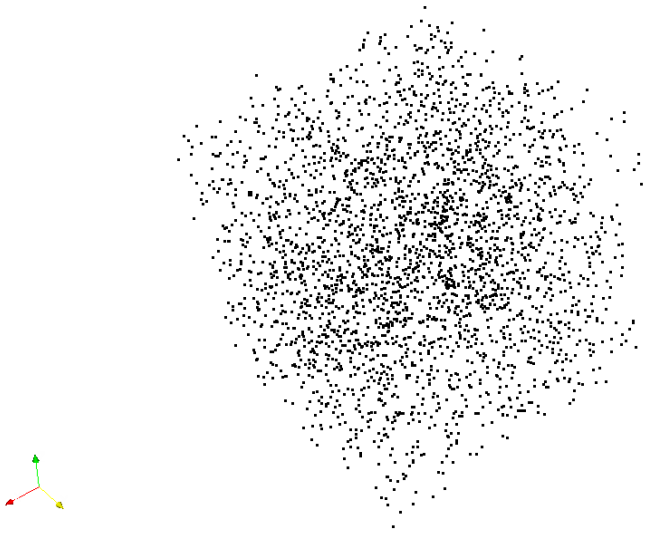

Figure 3.4: Test point cloud with 3000 random points in a grid of $64 * 64 * 64$

3.1.2 Geometrical point cloud processing

As the octree depth increases, it gets progressively more demanding to generate the octree. For (indoor) path finding the smallest octants should represent a minimal size in a point cloud. The length which is represented in an octree by the smallest octants of an octree is called the spatial resolution Lemmens [2015]. With a high spatial resolution small objects can be detected and thus is beneficial for path finding. Maximizing the spatial resolution by geometrical processing, a point cloud can be used to minimize the necessary octree depth. In turn reducing the octree generation time, storage space, amount of empty nodes and connectivity of the interior empty nodes.

The goal is to create a work flow to geometrical pre-process a point cloud so the empty space can be classified most efficiently in an octree. This means that the point cloud is processed in a way that generates the highest possible spatial resolution of the smallest octants for an octree with an arbitrary depth.

There are three geometrical operations which make sense on an point cloud: 1) rotation; 2) translation; 3) scaling. where scaling is the only operation which can influence the spatial resolution of an octree. There are two types of scaling: positive and negative. In negative scaling a point cloud is to big too fit in the octree space of an arbitrary depth. So unless the point cloud is negatively scaled a part of the octree will miss in the octree. In positive scaling a point cloud is smaller than the octree space. By positively scaling the point cloud would fit better in an octree with an arbitrary depth. The best spatial resolution is achieved when a point cloud is scaled so it axis-aligned bounding box fits in an octree grid of $2^n * 2^n * 2^n$, where n refers to the octree depth.

Both rotation and translation can effect the maximal possible scale of a point cloud. By rotating a point cloud so its minimum bounding box is aligned with the octree axis, a point cloud covers the smallest volume in an octree. This means a point cloud can have a larger scale to fit best in the octree space. Although, for visualization purposes it is recommended to

keep the floor surface horizontal. In that case, a building with a footprint of $100 * 100$ m has maximal diagonal distance of $\sqrt{100^2 + 100^2} = 141$. If the minimum bounding box of the building has a 45 degrees angle with the octree grid the axis-aligned bounding box of the point cloud will have a footprint of $141 * 141$. Whereas the minimum bounding box has a footprint of only $100 * 100$. Therefore, the scale of the point cloud can be increased with a factor of $\sqrt{2}$ by aligned the minimum bounding box to the octree axis. Subsequently, the spatial resolution of the smallest octants will be a factor $\sqrt{2}$ higher. Finally, the point cloud should be translated so the origin of the bounding box is located in coordinates $(0, 0, 0)$, as this is the origin point for the scaling operation. In short, both rotation and translation prepare the point cloud so it can be scaled to its full potential.
The work flow of pre-processing a point cloud for optimal classification consist of three steps: 1) align the minimum bounding box to the octree axis; 2) translate the point cloud so the origin is in coordinates $(0, 0, 0)$ and finally 3) scale the point cloud so it fits in a grid of $2^n * 2^n * 2^n$. These steps are visualized in Figure 3.6.

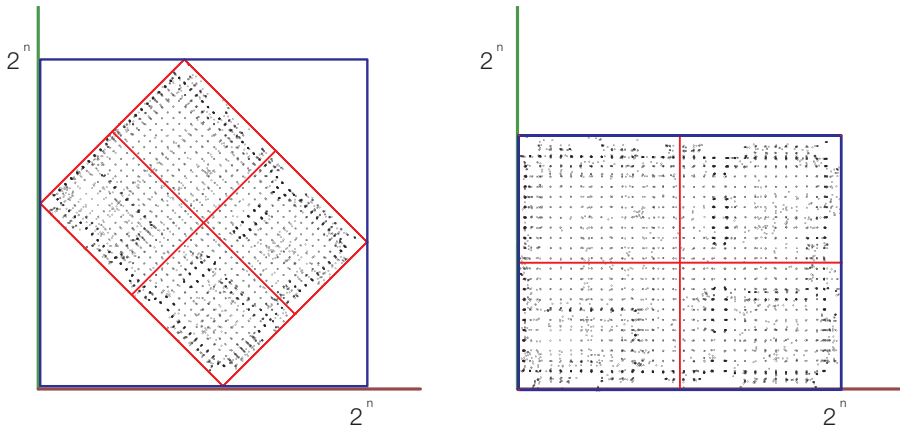

Figure 3.5: By aligning the minimum bounding box (red) of a point cloud with the axis minimum bounding box becomes the axis-aligned bounding box (blue) and is scaled more efficiently.

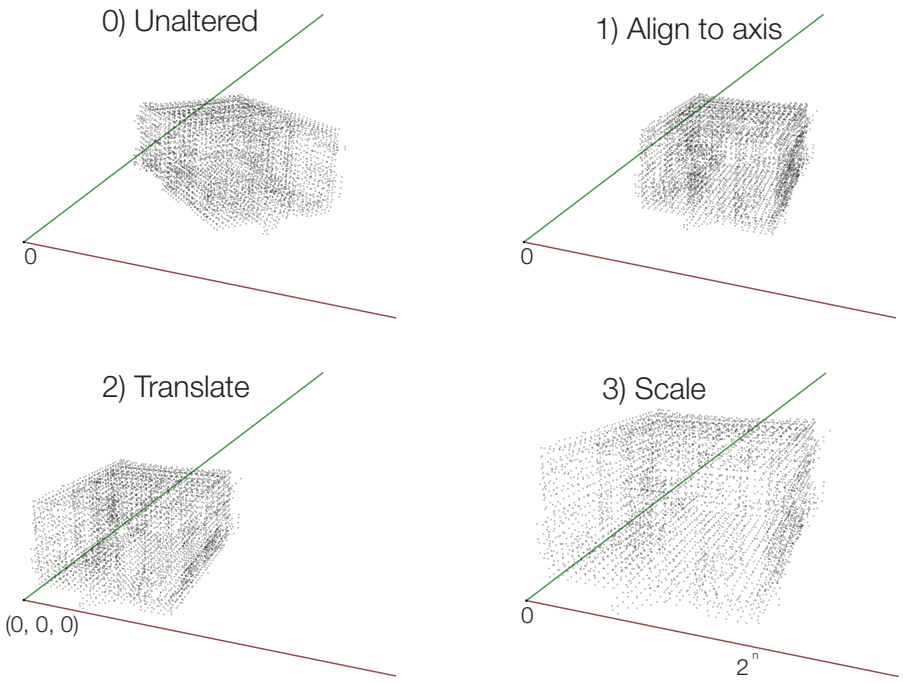

Figure 3.6: Work flow to geometrical pre-process a point cloud for maximal octree resolution

3.2 INTERIOR EMPTY NODES

In this research only interior point clouds are used. These point clouds are scanned from the inside of a building, therefore only the space between a point and the location of the scanner can be classified with certainty as empty. Therefore it is key to identify the interior empty space for cases like path finding and volume calculations.

The goal of identifying interior empty nodes is twofold. Firstly, by identifying the interior empty nodes the exterior empty nodes do not have to be further processed and stored. This reduces the amount of computations and storage. Secondly, only the interior empty nodes are used in path finding. This excludes the path to exit the interior space via glass or other open areas.

An empty nodes are only interior if it has both a non empty node straight above an beneath it. Figure 3.7 illustrates a section of a building represented by a quadtree, all nodes which do not have a non empty node above and under it, are exterior empty nodes. To identify interior empty nodes neighbours on the z direction are recursively computed outward. This is first done for the node above a current empty node and next for the node under the current node. The neighbours are computed until a non empty node is reached or until the border of the octree is reached (the node is exterior). An empty node is only further processed if it has both a non empty node above and beneath it. This method works fine for rooms with closed roofs but will generate errors for any glass or open roofs/ floors.

This method can also be used to compute the minimal downward distance to a border with a non empty node. In the process of checking if an

empty node is interior, the closest downward non empty nodes is computed. The downward distance is the delta z between the border of the non empty node and the centre point of the empty node. If we have the downward distance to the closest non empty node, this distance can be used as constraint in A* path finding. For example, a maximum distance of 1 meter roughly represent the volume in which a person can reach the closest non empty node underneath him/her. By selecting a start and goal node maximal 1 meter above a non empty node, the path will likely be bound to the floor. Note that this method is very basic and there is no proof that it will work in all circumstances.

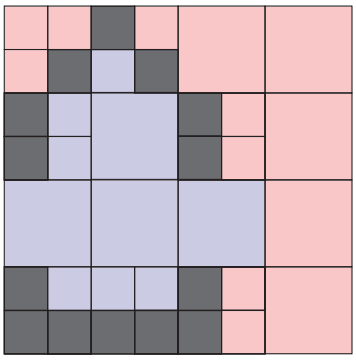

Figure 3.7: Section of a building represented by an octree: blue is interior and red exterior.

3.3 NEIGHBOUR FINDING AND CONNECTIVITY GENERATION

For path finding in an octree, the connectivity between the nodes must be known. This thesis proposes to pre-process the connectivity of each node and store this in a network graph. For this, two steps need to be performed: neighbour finding and connectivity generation. Neighbour finding focusses on computing all possible neighbours for an interior empty node given a certain connectivity. Connectivity generation aims on selecting neighbours which are interior empty nodes. The work flow of this is presented in Figure 3.8.

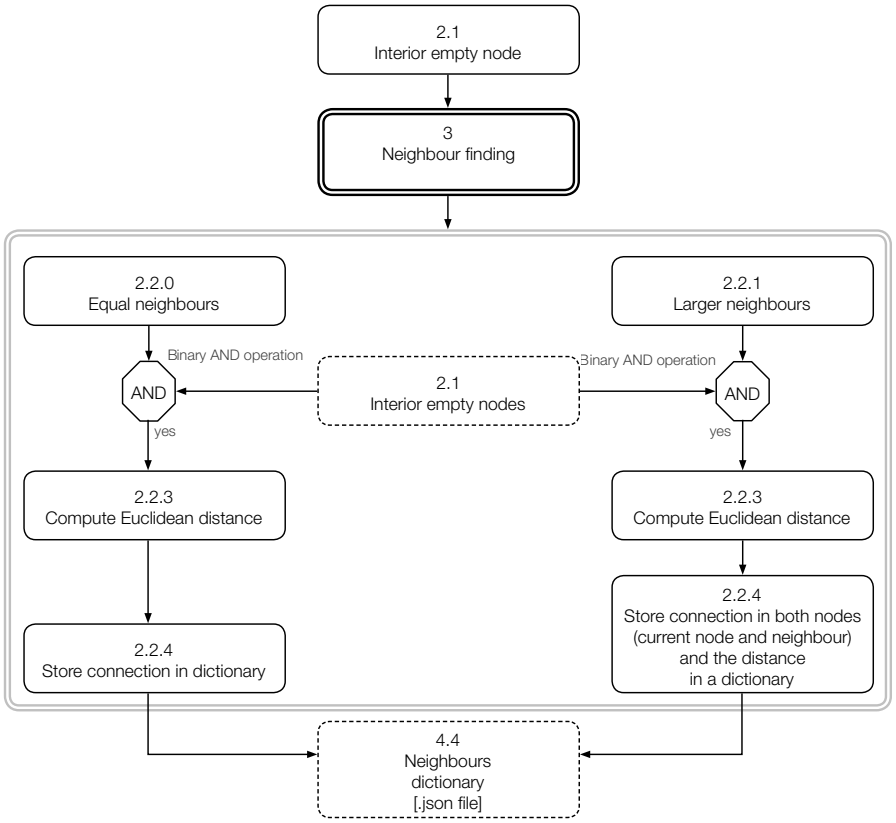

Figure 3.8: Work flow of connectivity generation

3.3.1 Neighbour finding

This subsection describes how all possible neighbours are computed. The method is based on the work of Vörös [2000]. He proposes to find neighbours based on their common ancestor. A distinction is made between inner neighbour (having the same parent) and outer neighbours (having a different parent node).

The method of Vörös [2000] differs on two front from this research. In this research only equal and larger neighbours are needed whereas Vörös computes smaller, equal and larger neighbours. Secondly, in this research two nodes can be neighbours if they share a common face, edge and vertex. The method of Vörös is restricted to face neighbours. Therefore the method of Vörös is extended to compute edge and vertex neighbours. The next two subsections describe the method for finding equal edge and vertex neighbours. Finally is describes how to compute larger neighbours.

Equal Edge Neighbours

The method of Vörös [2000] does not include edge neighbours. The binary operations used to find equal face neighbours, do not work for equal edge neighbours when multiple digits of the location code need to be changed. To solve this, I developed a method to find edge neighbours. The remainder of this subsection will describe the method.

The basic idea of the method is: the edge neighbours of node c have a face connection with a face neighbours of node c. Therefore the face neighbours of the face neighbours of node c have to be computed to get the

edge neighbours. The method is better explained with the use of Figure 3.9. The method is divided into four steps. In the first step the face neighbours of node c are computed. In the second step the face neighbour which are computed in the directions Dx and Dy are selected. In the third step the neighbours in Dz and Dx are computed of the face neighbours in direction Dx. In the final step the neighbours in Dz of the face neighbours in direction Dy are computed. The computed neighbours of the face neighbours form the complete set of equal edge neighbours.

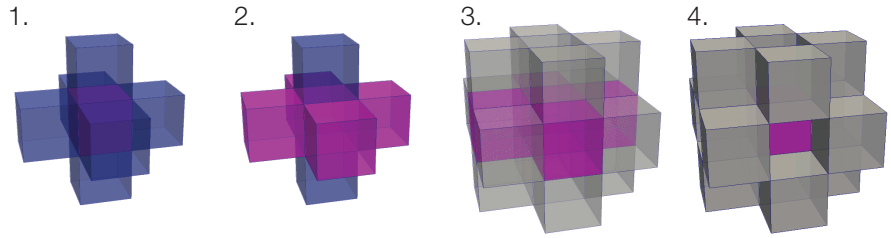

Figure 3.9: 1. face neighbours, 2. selected face neighbours, 3. compute face neighbours of face neighbours, 4. edge neighbours

Equal Vertex Neighbours

The method for computing equal vertex neighbours is similar to that of edge neighbours. The idea is: vertex neighbours of node c have a face connection with a edge neighbour of node c. So the face neighbours of the edge neighbours of node c have to be computed to get the edge neighbours. The method is better explained with the use of Figure 3.10. The method is divided into four steps. In the first step the edge neighbours are computed. Next the edge neighbours which were computed in direction Dx are selected. Of the selected nodes, the face neighbours in the direction Dz are computed. These nodes are the vertex neighbours.

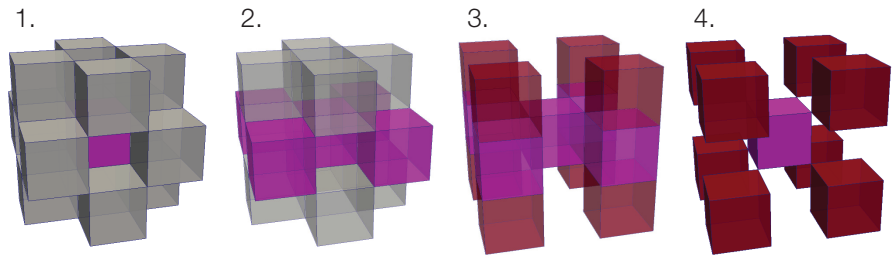

Figure 3.10: 1. edge neighbours, 2. selected edge neighbours, 3. compute face neighbours of edge neighbours, 4. vertex neighbours

Larger Neighbours

The method to compute larger neighbours is based on that of Vörös [2000] and is the same for each kind of neighbour (face, edge and vertex). The idea is that the ancestor nodes of a equal neighbour of node c are also neighbours of node c. Thus, the larger sized neighbours are computed by recursively deleting the most right digit from the location codes of the equal sized neighbours.

3.3.2 Connectivity generation

The previous section presented the method to compute equal and larger face, edge and veretx neighbours. Neighbours can be computed on the fly in A* path finding or they can be pre-processed. For both situations a suitable solution is developed. Both methods compute the neighbour the same way. This section will present a method to pre-process the node connectivities.

The method of Namdari et al. [2015] is used to compute neighbours during bread-first search octree generation. Only equal and larger neighbours are computed during the octree generation. By storing the connection of a larger neighbour in both connected nodes, also smaller neighbours are found in a later stage of the octree generation.

Like the method of Namdari et al. [2015] the octree used in this research is computed in a top down approach. For each interior empty node the equal and larger neighbours are computed based on the method described in section 3.3.1. Next it is checked if the neighbours are interior empty nodes. If so, the distance between the centre points of the intersecting interior nodes is computed. The connection of larger interior neighbours is stored in both the connected nodes. For example, if node x has a larger neighbour y the connection in node x and node y is stored. This step ensures that for the larger node y the smaller neighbours are stored. If all interior empty nodes are computed, a network graph is stored which can be used for indoor path finding.

3.4 COLLISION AVOIDANCE

The goal is to compute a collision free path for an object with an arbitrary size. A path is collision free if the object does not intersect with any non interior empty node along the path. For this, two things need to be known: can the object fit in a node, and can this object move between two nodes. Therefore this section consist of two parts: subsection 3.4.1 explains how to compute the distance from the centre of an empty node to the closest border with a non empty node. Subsection 3.4.2 explains how to compute the distance between a crossing point of two interior empt nodes and the closest non empty node.

3.4.1 Clearance map

The goal of the step is to compute for each interior empty node the minimal distance to a border with a non empty node. Together, these distances form a clearance map. On its own this clearance map can be used for object fitting. And it can be used for collision avoidance for path finding with a face connectivity. But this method will not suffice for an extended connectivity. The work flow to compute the clearance is presented in Figure 3.11.

The method of Samet [1982a] is used to compute the clearance for each empty node. The closest boundary with a non empty node is computed with a chessboard distance. Thus the closest non empty node must be, or a descendant of, one of the equal neighbours (see Figure 2.11). Since non empty nodes are always stored in the lowest level, a non empty node can never be a larger neighbour. So the 26 equal neighbours are needed to find the closest boundary with a non empty node. Since these neighbours were

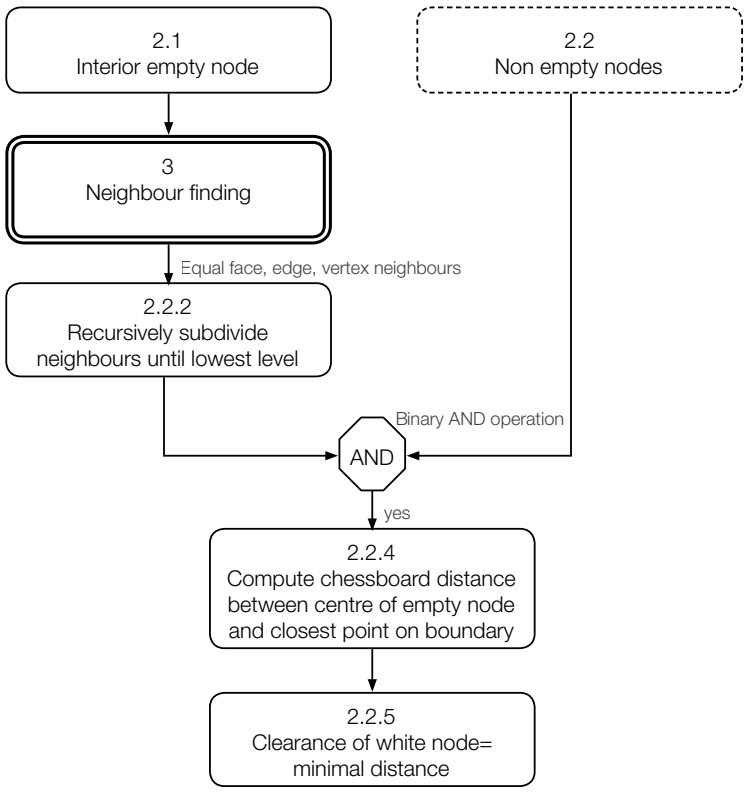

Figure 3.11: Work flow of Clearance map

computed and stored during the connectivity generation these are already available. To acquire the closest boundary, the closest non empty nodes has to be computed first. This is done by recursively subdividing the equal neighbours until the lowest octree level and storing it in a set.This is the level of the non empty nodes. This set can be compared with the set of non empty nodes. For each intersecting non empty node the distance between the border of the non-empty node and the centre point of the empty node is computed. As it is possible for multiple non empty nodes to intersect, it is the smallest distance that defines the clearance. This clearance is stored as attribute of the current interior empt node.

3.4.2 Maximal crossing value

Figure 3.12 shows two diagrams, in the left an object (blue circle) is in the centre of an empty node where the clearance is sufficient. Although the clearance is sufficient in the centre point of a node, the object collides with a non-empty node (red) when moving between two empty nodes. For this reason it is key to check the maximal crossing value for each connection between two interior empty nodes. This crossing value can never be bigger than the minimal clearance of the two connection empty nodes. Otherwise an object would be able to enter a node with a clearance smaller than the object size and a collision would occur.

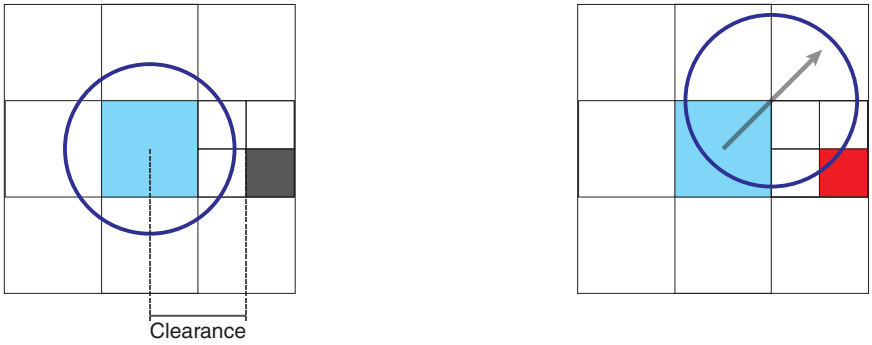

Figure 3.12: The clearance in the centre point is collision free, however point of movement collides with a non empty node

There are three types of connections between two empty nodes: face, edge and vertex.

In the case of an equal face connection the minimal clearance is equal to the maximal crossing value. Figure 3.13 shows two equal face neighbours. The image also shows the minimal chessboard distance (clearance) from point $c1$ and $c2$. The distance to the intersection point $i1$ cannot be smaller than one of the clearances. Thus the maximal crossing value is equal to the minimal clearance of node $c1$ and $c2$. In the case of a connection between node of arbitrary sizes the movement must go through the centre point of the smallest intersecting face.

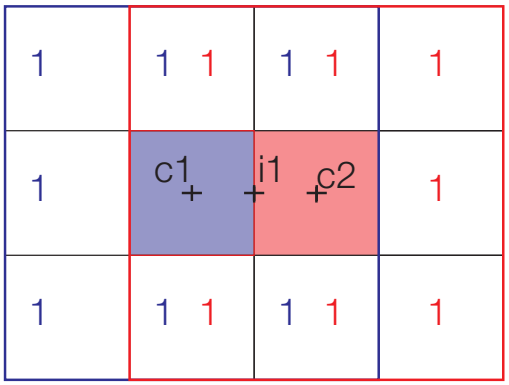

Figure 3.13: Only the common (purple) neighbouring nodes of the blue and red node can be closest to the intersection point

As mentioned: the maximal crossing value can never be larger than the minimal clearance of the two connected empty nodes. Therefore only common neighbours of two connected empty nodes can have a smaller distance to the crossing point than the minimal clearance. Figure 3.14 shows a vertex connection between a blue and red node in a quadtree. The area which defines the clearance for each node is coloured in a lighter tone of the node colour. The non empty nodes in those areas define the clearance of each node. So for a crossing value to be smaller than the minimal clearance, there must be a node in the common neighbours of the blue and red nodes. This means that only non empty nodes in the common neighbours of two adjacent empty node can create a smaller distance to the intersection point than the minimal clearance.

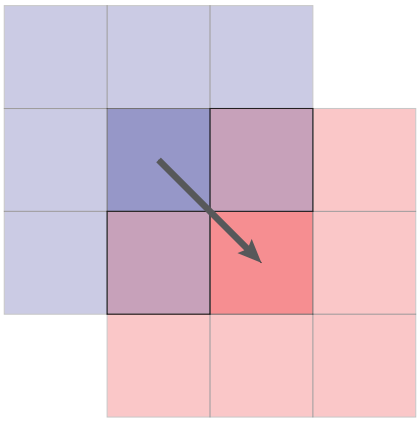

Figure 3.14: Only the common (purple) neighbouring nodes of the blue and red node can be closest to the intersection point

For an edge connection this means only two common face neighbours have to be explored. Figure 3.15 shows the common neighbours (blue) between two nodes (red and white) in an edge connection. The common neighbours can be found by doing an AND operation on the set of face neighbours of the two connected empty nodes. Next it is checked if the common neighbours are (ancestors of) a non empty node. If there are intersecting nodes, the distance between the intersection line and the closest point on the boundary of the non empty node is calculated. The maximal crossing value is now defined by the lowest value of the distances or the clearance of the connected empty nodes.

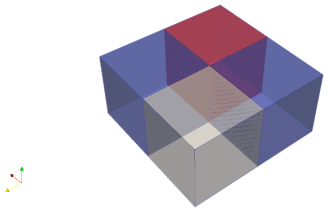

Figure 3.15: Only the common (purple) neighbouring nodes of the blue and red node can be closest to the intersection point

A vertex connection between two empty nodes has six common neighbours. Figure 3.16 illustrates a vertex connection between a red and white node. The six blue nodes are common neighbours of the red and white node. Instead of only face neighbours, also edge neighbours have to be explored to find the common neighbours. The method of finding the maximal crossing value works the same as the method described for the edge neighbours.

The cross value is computed during the construction of interior empty nodes. As an empty node is constructed, all the possible neighbours are found. The maximal crossing value is only computed for interior empty neighbouring nodes. This prevents computing the maximal crossing value for connections to non empty nodes or exterior empty nodes.

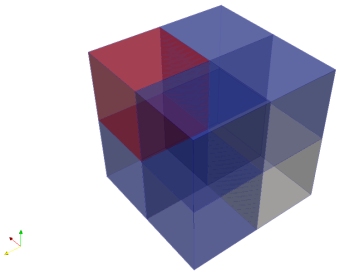

Figure 3.16: Only the common (purple) neighbouring nodes of the blue and red node can be closest to the intersection point in a quadtree

3.5 TYPES OF DISTANCE

A distance between two points can be calculated in different ways. In this research the effect on the computation time and path length of the following distances is tested:

1. **Euclidean**
   The Euclidean distance is the true distance between two points. It is calculated using theorem of Pythagoras ($a^2 + b^2 = c^2$).
2. **Manhattan**
   In a Manhattan distance only orthogonal motion is possible. The distance is the sum of all orthogonal components.
3. **Chessboard**
   Chessboard distance is the amount of steps needed for a king to move on a chessboard to the next node.

For each distance type a function is created, this functions are used to compute the distance between two adjacent nodes. Figure 2.12 illustrates the distance types in more detail.

3.6 BENCHMARK TESTS

The final goal of this research is to identify the effect of geometrical point cloud operations, octree operators and A* operations on A* path finding. To find these effects benchmark tests were conducted. The benchmark tests are divided into two stages: 1) The effects of octree operators on path length and computation time; 2) The effects of A* operations on path length and computation time.

3.6.1 Test system

All benchmark tests are calculated on a computer with the following specifications:

- **Processor** Intel(R) Core(TM) i5-4590 CPU @3.30GHz
- **Installed memory** (RAM) 8.00 GB
- **System Type** 64-bit Operation System, x64-based processor, Windows 10 Pro

3.6.2 Storage

The results of the benchmark test will be stored in a PostgreSQL database. There are two types of benchmark tests: 1) octree generation, 2) path finding. For each type of test a database table is used to store the results and metadata.

The table to store the octree generation results contains the following:
(ocree name, scale, rotation, translation, distance type, connectivity, bounding box of point cloud, minimal leaf node size)
And the metadata of the octree generation contains:
(number of points, number of non empty nodes, time to compute non empty nodes, number of interior empty nodes, time to compute empty nodes, total computation time)

The table to store the path finding results contains the following:
(path length, computation time, iterations, the path route)
And the metadata of the path finding contains:
(octree name, star node, goal node, clearance, distance to floor, connectivity, distance type)

3.6.3 Quality

To assure that time measurement is not influenced by outliers, all time measurements are conducted five times. Of these measurements, the median value is computed and used as result.

3.6.4 Benchmark test set up

As mentioned, the benchmark test consists of two parts. In all path finding tests the point cloud is aligned to the octree axis, translated and scaled. The octree operator and A* operations which are tested are: octree depth, path connectivity, pre-processing connectivity and distance type. Table 3.2 provides the test setting of the benchmark tests.

Parameters	**octree depth**	**Pre-processed connectivity**	**Connectivity**	**Distance type**
Octree depth	6-8	6	6	6
Pre-processed connectivity	True	True, False	True	True
Connectivity	6	6	6, 18, 26	6
Distance type	Euclidean	Euclidean	Euclidean	Euclidean chessboard, Manhattan

Table 3.2: Test settings for A* path finding benchmark tests

Octree Depth

The octree depth defines the spatial resolution of an octree. For path finding a minimal resolution is necessary to pass through all rooms. Therefore this test aims to identify the effect of different octree depths, or spatial resolutions, on the path length and computation time. Because the depth is closely related to the spatial resolution it does not make sense to perform the test on the test

point cloud, as it does not represent a 'real' building and cannot give a usable leaf size spatial resolution of the smallest octants in an octree. The octrees used in the test are generated with a minimal clearance and crossing value of $0,17$. Further, empty nodes can be maximal $1,25$ m above a border with the closest non empty node.

Pre-processing connectivity

During path finding, the connectivity of a neighbour can be derived from a network graph or computed on the fly. Pre-processing neighbours obviously makes the octree generation computational more demanding. However, a network graph eliminates a number of operations during path finding: 1) neighbour finding; 2) connectivity generation; 3) distance to neighbouring node. So instead of computing the latter, these are known, which saves time in path finding. Therefore, this benchmark test aims to identify the effect of pre-processing node connectivities on computation time. As the path finding algorithm is identical, there will be no difference in path length and is thus not tested.

All point clouds will be used for this benchmark test. All routes in the Bouwpub and Fire department are computed with a minimal clearance of $0,17$ and a maximal downward distance of $1,25$ m to the border of the closest non empty node. For the test point cloud the minimal clearance and distance to a non empty node is set to 0. For each dataset two octrees with 7 levels are computed: one with pre-processed face neighbours and one without a connectivity computed.

Path connectivity

Increasing the path connectivity extend the amount of possible directions in which movement is possible. This makes it possible to find a shorter an smoother path. Although due to the extended connectivity there are more possible nodes to discover. This benchmark test aims to identify the difference of a face (6); face and edge (18) or face, edge and vertex (26) connectivity on the path length and computation time. A face connectivity is pre-processed for these benchmark tests. The test will be performed on all point clouds. All octrees have a depth of 7 levels. All routes in the Bouwpub and Fire department are computed with a minimal clearance of $0,17$ and a maximal downward distance of $1,25$ m to the border of the closest non empty node. For the test point cloud the minimal clearance and distance to a non empty node is set to 0.

Path distance type

Three types of distance types were tested: Euclidean, Manhattan and chessboard. This test aims at identifying the effect of different distance types used in A* path finding on path length and computation time.

The test will be performed on all point clouds. All octrees have a depth of 7 levels. All octrees have a tree depth of 7 and a face neighbours is computed on the fly. All routes in the octree of the Bouwpub and fire department are computed with a minimal clearance of $0,17$ and empty nodes can be maximal $1,25$ m above a border with the closest non empty node. For the test point cloud the minimal clearance and downward distance to a non empty node is set to 0.

4 IMPLEMENTATION

This chapter presents the implementation of the methods. The methods are implemented in a software package containing three scripts: 1) octree generation; 2) neighbour finding and 3) A* path finding. The latter will be explained in sections 4.1 and 4.2. Section 4.3 explains the implementation of the point clouds. Section 4.4 presents the implementation of the benchmark test. And finally, section 4.5 presents the software and tools used in the implementation.

4.1 EMPTY NODE GENERATION

The main contribution to the octree generation process is the way empty nodes are processed. Each empty node follows a number of decision steps. These steps are designed to process only what is necessary. To explain these steps I will refer to lines in a python code. Most of the lines in the code are accompanied with a small descriptive text (in red). The script for the generation of empty nodes is provided below.

1. For each empty node in the octree, the following steps are taken:
    1.1. Interior empty nodes are identified *(lines 9 - 13)*, this process is explained in section 4.1.1. Only interior empty nodes are further processed, the exterior empty nodes are filtered out and will not be stored in the database of network graph. Each interior empty node returns the closest non empty node directly beneath it.
    1.2. Compute the distance to the closest non empty node under each interior empty node. *(lines 16 - 31)*.
    1.3. All possible neighbours of the interior empty node are computed with the neighbour finding scripts *(lines 36 - 37)*. This script is explained in section 4.1.2.
    1.4. The clearance of the interior empty node is computed *(lines 44 - 48)*. This process is explained in section 4.1.3.
    1.5. Create connectivity and maximal crossing value of the interior empty node *(lines 50 - 94)*. This process is explained in detail in section 4.1.3.
        1.5.1. Check for each neighbour if it is a processed interior empty node *(line 53)*. This is done by checking if the location code exists in the set of interior empty nodes. Only the interior empty neighbours are further processed.
        1.5.2. Compute the crossing value. For this the interior empty neighbours are used to compute the crossing value *(lines 57 & 58)*. This process is explained in section 4.1.3.

1.5.3. Create the connectivity in a temporary dictionary with location code as key and the distance and maximal crossing value as values *(line 60)*.

1.5.4. Store the connection of larger neighbours in both directions *(lines 66 - 76)*.

2. If all neighbours are computed, the temporary neighbour dictionary is stored in a connectivity dictionary *(line 79)*.

3. Clean the neighbours dictionary of all neighbours which are not interior empty nodes *(line 81 - 87)*.

4. Store the neighbours dictionary as .json file and return the clearance dictionary *(line 89 - 84)*

```
generation of interior empty nodes"""
for emptyNode in tree:
    currentNode = emptyNode + number
    """check if node is not an non empty node"""
    if currentNode not in nonempty[level+1]:

        """Compute the coords of the current coord"""
        currentNodeCoord = getCoord(depth, currentNode).split(' ')

        """Check if a emptyNode is interior or exterior"""
        if int(currentNode[-1])<3:
            closestNode = interiorEmptyNodes.inner(currentNode, nonempty[level+1], NonEmptyTable)
        else:
            closestNode = interiorEmptyNodes.outer(currentNode, nonempty[level+1], NonEmptyTable)

        """Select only the interior empty nodes"""
        if closestNode:
            """Check distance to nodes under empty node"""
            """In here the function 'interiorEmpty' nodes is called"""
            if int(currentNode[-1])>3:
                closestNode = interiorEmptyNodes.inner(currentNode, nonempty[level+1], NonEmptyTable)
            else:
                closestNode = interiorEmptyNodes.outer(currentNode, nonempty[level+1], NonEmptyTable)

            """check if there is a non empty node under the interior empty node comp distance"""
            if closestNode:
                """Only compute the coords of the node if there is a non empty node under it"""
                closestNode = getCoord(depth, closestNode).split(' ')
                """distance to floor is delta z"""
                distanceToFloor = (int(currentNodeCoord[3])+halfLeafSize)-(int(closestNode[3])+1)
                distanceToFloorDict[currentNode] = round((distanceToFloor/scale),2)


                """Compute all possible neighbours for each interior empty node"""
                """This calls the neighbour finding script"""
                equalFaceNeighbours, largerFaceNeighbours,\
                equalEdgeNeighbours, largerEdgeNeighbours,\
                equalVertexNeighbours, largerVertexNeighbours = \
                neighbourFinding.giveMeUpToEqualSizedNeighbours(currentNode, depth)

                """Compute the clearance for each node"""
                allEqual = equalFaceNeighbours.copy()
                allEqual.update(equalEdgeNeighbours)
                allEqual.update(equalVertexNeighbours)

                """In here the function 'clearance' is called"""
                clearanceDict[currentNode] = round((clearance(NonEmptyTable,\
                 currentNodeCoord, maximumLevels, allEqual, (1,2,3))/scale),2)

                """Store the clearance in a dictionary"""
                clearanceCurrent = clearanceDict[currentNode]

                """The connectivity and collision avoidance is generated"""
            if connectivity == 'face' or connectivity == 'faceEdge' or connectivity == 'faceEdgeVertex':
                """Cross value for equal face neighbours"""
                for neighbour in equalFaceNeighbours:
                    """only compute the maximal crossing value if the neighbours is an interior empty node"""
                    if neighbour in emptyFaceNeighbourDict:
```

```
59                      """The maximal crossing value can never be larger then the minimal clearance"""
60                      maxCrossValue =  min(clearanceCurrent,
61                      emptyFaceNeighbourDict[neighbour][currentNode][1])
62                      """Store the connectivity and the maximal crossing"""
63                      equalFaceNeighbours[neighbour] = (equalFaceNeighbours[neighbour], maxCrossValue)
64
65                  else:
66                      equalFaceNeighbours[neighbour] = (equalFaceNeighbours[neighbour], clearanceCurrent)
67
68              """Compute the larger face neighbours"""
69              for largerNeighbour in largerFaceNeighbours:
70                  """compute only larger neighbours which are interior empty nodes"""
71                  if largerNeighbour in emptyFaceNeighbourDict:
72                      """Compute Euclidean distance between centre points of two neighbouring nodes"""
73                      euclideanDistance = euclidean(currentNodeCoord, largerNeighbour)
74
75                   """Store the connection in the neighbour dictionary of the current node"""
76                   equalFaceNeighbours[largerNeighbour] = (euclideanDistance, clearanceCurrent)
77
78                   """All larger nodes are stored in both directions, this
79                   way the smaller neighbours get stored for each node"""
80                   emptyFaceNeighbourDict[largerNeighbour].update({currentNode:equalFaceNeighbours[largerNeighbour]})
81
82      """Store the neighbours in the master dictionaries"""
83      emptyFaceNeighbourDict[currentNode] = equalFaceNeighbours
84
85  if connectivity == 'face' or connectivity == 'faceEdge' or connectivity == 'faceEdgeVertex':
86  """Delete all non empty neighbour nodes"""
87  emptyfaceNeighbourDict =  dict()
88
89  for emptyNode in emptyFaceNeighbourDict:
90      """Filter out all non empty neighbours using a dictionary comprehension"""
91      emptyfaceNeighbourDict[emptyNode] = {k: v for k, v in \
92              emptyFaceNeighbourDict[emptyNode].iteritems() if k in emptyFaceNeighbourDict}
93
94  """Store the neighbours dictionary in a .json file"""
95  json.dump(neighboursDict, open("neighourDicts/"+name+"_neighboursDict1.txt",'w'))
96
97  return clearanceDict, distanceToFloorDict
```

4.1.1 Interior empty nodes

This section explains the implementation of computing interior empty nodes. In the following example it is checked if a node has a non empty node above it. This is done by recursively computing a neighbours upward until a non empty node is reached or until there are no more upward neighbours (the node is exterior). The following enumeration describes the script. The python code is provided below.

1. Check if the interior empty node is located in the top or bottom in its parent node. If it is located in the top there is no inner neighbour above the empty node. This means the neighbour above the empty node is an outer neighbour *(lines 3-8)*.

2. The inner or outer neighbour is computed *(lines 11, 19)*.

3. It is checked if the neighbours is, a parent node of, a non empty node *(lines 12, 20)*. The neighbour is compared with a set of nodes which would have been non empty at the depth of the current interior empty node.

    3.1. If this is true, the script returns the non empty node closest to the empty node in question *(lines 12, 24)*.

    3.2. If this is not true, the next neighbour is computed and step 3 is repeated *(lines 15-16, 27-28)*.

3.3. If there are no more neighbours (this happens when the border of the octree is reached) the node is exterior *(lines 24-26)*.

```
1  for empty node in octree:
2      """Check if a node is interior or exterior"
3   """check if the last digit of the location code is smaller or large than 3."""
4   """" This decides if the first neighbours must be inner or outer"""
5      if int(currentNode[-1])<3:
6          closestNode = innerNeighbour(currentNode, nonempty[level+1], NonEmptyTable)
7      else:
8          closestNode = outerNeighbour(currentNode, nonempty[level+1], NonEmptyTable)
9
10 def innerZNeighbour(spatialCode, nonEmptyNodesCut, nonemptyNodes):
11     zNeighbour = getEqualInnerNeighbours(spatialCode)
12     if zNeighbour in nonEmptyNodesCut:
13         """The node is interior"""
14         return max([x for x in nonemptyNodes if x[0:len(spatialCode)] == zNeighbour])
15     else:
16         return outerNeighbour(zNeighbour, nonEmptyNodesCut, nonemptyNodes)
17
18 def outerZNeighbour(spatialCode, nonEmptyNodesCut, nonemptyNodes):
19     zNeighbour = getEqualOuterNeighbours(spatialCode)
20     if zNeighbour in nonEmptyNodesCut:
21         # print max([x for x in nonemptyNodes if x[0:len(node)] == zNeighbour])
22         """The node is interior"""
23         return max([x for x in nonemptyNodes if x[0:len(spatialCode)] == zNeighbour])
24     elif len(zNeighbour) == 0:
25         """The node is exterior"""
26         return None
27     else:
28         return inner(zNeighbour, nonEmptyNodesCut, nonemptyNodes)
```

4.1.2 Neighbour finding and connectivity generation

The goal of this step is to compute the connectivity for each interior empty node. First, all possible neighbours are computed. For each neighbour the Euclidean distance is computed between the centre points of the interior empty neighbours *(line 20)*. Besides, for each connection to an interior empty neighbours, the maximal cross value is computed. Note that the maximal crossing value is only computed if a neighbour is an interior empty node, see line 18. In the case of face neighbours the maximal crossing value is the minimal clearance of the two connected empty nodes. When all interior empty nodes are computed, all non interior empty neighbours are filtered out for each interior empty node.

The python script below shows the implementation for face neighbours. The implementation works similar for edge and vertex neighbours.

```
1  for interiorEmptyNode in octree:
2      """Compute all possible neighbours for each interior empty node"""
3      equalFaceNeighbours, largerFaceNeighbours = \
4      neighbourFinding.giveMeUpToEqualNeighbours(currentNode, maximumLevels)
5
6      if connectivity == 'face':
7          """Cross value for equal face neighbours"""
8          for neighbour in equalFaceNeighbours:
9              if neighbour in emptyFaceNeighbourDict:
10                 maxCrossValue =  min(clearanceCurrent, emptyFaceNeighbourDict[neighbour][currentNode][1])
11                 equalFaceNeighbours[neighbour] = (equalFaceNeighbours[neighbour], maxCrossValue)
12                 emptyFaceNeighbourDict[neighbour][currentNode] = \
13                  (emptyFaceNeighbourDict[neighbour][currentNode][0],maxCrossValue)
14
15             else:
16                 equalFaceNeighbours[neighbour] = (equalFaceNeighbours[neighbour], clearanceCurrent)
17
18         """Compute the larger face neighbours"""
19         for largerNeighbour in largerFaceNeighbours:
20             if largerNeighbour in emptyFaceNeighbourDict:
21                 """Compute Euclidean distance between centre points of two neighbouring nodes"""
```

```
22                  euclideanDistance = euclidean(currentNodeCoord, largerNeighbour)
23
24                  """Store the connection in the neighbour dictionary of the current node"""
25                  equalFaceNeighbours[largerNeighbour] = (euclideanDistance, clearanceCurrent)
26
27                  """All larger nodes are stored in both directions, this\
28                   way the smaller neighbours get stored for each node"""
29                  emptyFaceNeighbourDict[largerNeighbour].update({currentNode:equalFaceNeighbours[largerNeighbour]})
30
31      """Store the neighbours in the master dictionaries"""
32      emptyFaceNeighbourDict[currentNode] = equalFaceNeighbours
33
34  """Delete all non interior empty nodes from the neighbours dictionary"""
35  emptyfaceNeighbourDict[node] = {k: v for k, v in emptyFaceNeighbourDict[node].iteritems() if k in emptyFaceNeighbourDict}
```

4.1.3 Collision avoidance

The goal is to compute a collision free path for an object with an arbitrary size. A path is collision free if the object does not intersect with any non interior empty node along the path. For this, two things need to be known: can the object fit in a node, and can this object move between two nodes.

Clearance map

The goal of this step is to create a clearance (minimal distance to a boundary with a non empty node) for an interior empty node. For each current interior empty node the face, edge and vertex neighbours are computed. Of each neighbour all possible children nodes at the lowest octree level are computed, see line 18 to 29. These nodes are compared to the non empty nodes with a binary AND operation. All non empty neighbours are stored in a list. For each of these non empty neighbours, the chessboard distance between the centre of the current interior empty node and the closest point on the boundary of the non empty neighbour is calculated. The smallest distance is the clearance of the current interior empty node. The relevant python code for the clearance is presented below.

```
1   def clearance(nonEmptyNodes, currentNode, depth, neighbourSet, numbers):
2   """Compute the clearance of a interior empty node"""
3       level = depth - len(currentNode[0])
4       leafSize_currentNode = (2 ** level)
5
6       """Compute all possible leafnodes"""
7       possibleLeafNodes = findChildrenNeighbours(set(neighbourSet), level)
8
9       """Compute the non empty leaf nodes which descend from the non empty neighbours"""
10      nonEmptyLeafNode = possibleLeafNodes & nonEmptyNodes
11
12      BorderDistanceList = []
13      for node in nonEmptyLeafNode:
14          """Compute the distance with distance function"""
15          BorderDistanceList.append(chessboardDistance(currentNode, node, depth, numbers))
16      """Return the clearance"""
17      return min(BorderDistanceList)
18
19  def findChildrenNeighbours(nodeSet, depth):
20      """Recursively subdivede not until finest resolution is reached"""
21      newNodeSet = set()
22      if depth ==0:
23          return nodeSet
24
25      elif depth>0:
26          for node in nodeSet:
27              for number in range(0,8):
28                  newNodeSet.add(node + str(number))
29      depth -=1
30      """return all possible leaf nodes"""
31      return findChildrenNeighbours(newNodeSet, depth)
```

```
32
33  def chessboardDistance(current, neighbour, depth, directions):
34   """Compute the chessboard distance"""
35      neighbour = getCoord(depth, neighbour).split(' ')
36      halfLeafSize = 0.5*int(current[4])
37
38      lengthList = []
39      for direction in directions:
40          """check if the node is legft or right of the current interior empty node"""
41          if int(current[direction])<int(neighbour[direction]):
42              lengthList.append(abs((int(current[direction])+halfLeafSize)-int(neighbour[direction])))
43          else:
44              lengthList.append(abs((int(current[direction])+halfLeafSize)-(int(neighbour[direction])+1)))
45      """Return the minimal distance"""
46      return max(lengthList)
```

Maximal crossing value

The python code below computes the maximal crossing value for equal edge and vertex neighbours. First, it is checked if the neighbour is an interior empty node. Only the interior empty neighbours are further processed. Next the intersecting line of point between the current interior and the interior empty neighbour node is computed. This is used to compute the distance to a non empty node. Next, the common neighbours of the interior current and neighbour node are computed. All possible descendants of the common neighbours are computed. If a descendant is a non empty node the distance to the intersection line/ point is computed. The smallest distance defines the maximal crossing value. The maximal crossing value is stored in both directions. The relevant python code is provided below.

```
1   def crossValue(equalNeighbours):
2       """Check if the neighbour exist in the dictionary"""
3       for neighbour in equalNeighbours:
4           if neighbour in checkEmptyNeighbourDict[0]:
5
6               """Compute the intersection line/point between the current  interior
7               empty node and empty interior neighbour node"""
8               neighbourCoord = getCoord(depth, neighbour).split(' ')
9               intersectLineCoords = [0]
10              numbers = []
11              for number in (1,2,3):
12                  if int(neighbourCoord[number]) == int(currentNodeCoord[number]):
13                      intersectLineCoords.append(999)
14                  else:
15                      intersectLineCoords.append((int(neighbourCoord[number])+int(currentNodeCoord[number]))/2)
16                      numbers.append(number)
17              intersectLineCoords.append(0)
18
19              """compute the common neighbours"""
20              testset = set()
21              for dictionary in checkEmptyNeighbourDict:
22                  testset = testset | set(dictionary[neighbour])
23
24              commonNeighbours = set(equalCheckNeighbours) & testset
25
26              """Compute all non empty nodes in the intersecting neighbours"""
27              """Select all non empty nodes starting with the possible neighbours"""
28              nonEmptyLeafNode = [x for x in NonEmptyTable if x[0:len(currentNode)] in commonNeighbours]
29
30              """Compute the distance between the intersection
31              line and the closest boundary on the black node"""
32              edgeDistance = []
33
34              if nonEmptyLeafNode:
35               """If there are any non empty nodes that intersect"""
36                  for nonEmpty in nonEmptyLeafNode:
37                      edgeDistance.append(distanceType(intersectLineCoords, nonEmpty, depth, numbers))
38
39                  """The maximal cross value is the minimal distance of the clearance
40                  of the current and neighbour node or minimal cross value"""
41                  maxCrossValue =round(min(((edgeDistance)/scale), clearanceDict[currentNode], clearanceDict[neighbo
```

```
                equalNeighbours[neighbour] = (equalNeighbours[neighbour], maxCrossValue)
                """Store the maximal cross value in both directions"""
                emptyNeighbourDict[neighbour][currentNode] = (emptyNeighbourDict[neighbour][currentNode],maxCrossValue)
            else:
             """If there are no non empty nodes in the common neighbours"""
             """The maximal crossing value is the minimal clearance """
                maxCrossValue =min(clearanceDict[currentNode], clearanceDict[neighbour])
                equalNeighbours[neighbour] = (equalNeighbours[neighbour], maxCrossValue)
                """Store the maximal cross value in both directions"""
                emptyNeighbourDict[neighbour][currentNode] = (emptyNeighbourDict[neighbour][currentNode],maxCrossValue)

    return equalNeighbours
```

4.2 A* PATH FINDING

The first step in path finding is loading and constructing a network graph and loading empty nodes from the database.

```
def get_nodesDict():
    """In this function we extract the non empty nodes from the database"""
    conn = psycopg2.connect("host='localhost' dbname='"+dbms_name+"' user='"+ user+ "'")
    cur = conn.cursor()
    cur.execute("select * from "+name+"_emptySpace where clearance >=" +str(clearance) +" \
    and distancetofloor < " +str(distancetofloor) +";")
    nodes = cur.fetchall()
    nodesDict = dict()

    for node in nodes:
       nodesDict[node[0]] = (node[1], node[2], node[3], .5* node[4])

    """Create neighboursDict"""
    neighboursDict= json.load(open(name+'_neighboursDict.txt', 'r'))

    """filter out all interior empty nodes where the clearance and/ or
     distance to floor is not sufficient"""
    neighboursDict = {k: v for k, v in neighboursDict.iteritems() if k in nodesDict}

    """Filter out all connections where the clearance and distance to floor is to little"""
    for node in neighboursDict:
       neighboursDict[node] = {k: v for k, v in neighboursDict[node].iteritems() \
                if v[1] > clearance and k in nodesDict}
     """Filter out all interior empty nodes having no connections"""
    neighboursDict = {k: v for k, v in neighboursDict.iteritems() if v}

    return neighboursDict, nodesDict
```

The next step is the actual A* path finding. If no path can be found, an error message returns 'no path found'.

```
def a_star_search(start, goal):
"In this function we run the A* algorithm"""
en_list = Priority()
en_list.put(start, 0)
me_from = {}
st_so_far = {}
st_so_far[start] = 0
" while the empty_list is not empty """
ile not open_list.empty():
""With this line we get the smallest total cost,
otal cost is the cost_so_far + cost to neighbour  """
urrent = open_list.get()
""If the current is the goal than the path is found
nd return the came_from dict and the cost_so_far  """
f current == goal:
"""The path is found"""
return came_from, cost_so_far
"" Check for all neighbors in current """
or neighbour in neighboursDict[current]:

```

```
21  """Check if the cross value is higher than the minimal clearance"""
22  """ Calculate the cost to get to a neighbor from the star """
23  new_cost = cost_so_far[current] + neighboursDict[current][neighbour][0]
24  """If neighbor cost_so_far or new_cost to get to the neighbor
25  is smaller than getting their via a different path  """
26          if neighbour not in cost_so_far or new_cost < cost_so_far[neighbour]:
27          """ Update or create new cost to get to neighbor """
28          cost_so_far[neighbour] = new_cost
29          """ Sum the cost to get to cost(start, neighbor) + cost(neighbor, goal) = total_cost"""
30          total_cost = new_cost + euclideon_dist(nodesDict[goal], nodesDict[neighbour])
31          """ Put neighbor in open_list with the total cost  """
32          open_list.put(neighbour, total_cost)
33          """ add current and neighbor to cam_from dict """
34          came_from[neighbour] = current
35  ise ValueError('No Path Found')
```

4.3 POINT CLOUD DATASETS

The point clouds of the Bouwpub and the fire department are both existing datasets. But the datasets still needed to be cleaned. The outliers were removed using CloudCompare. Further, a section was cut out of the fire department dataset to be used in the benchmark tests.

The test point cloud was created using a simple python script. The basic idea of the algorithm is to create a point cloud with a set of x points having a random x, y and z values within a square bounding box with dimensions of 0 to 2^n, where n refers to the octree depth. The range of 0 to 2^n is chosen so the point cloud fits in a $2^n * 2^n * 2^n$ grid having a scale of an integer number.

4.4 BENCHMARK TESTS

The benchmark test are separated in two steps: the octree generation and path finding. Both steps start by filling in the input and finish with storing all results in a database. All time sensitive processes are computed multiple times to avoid outliers in the results.

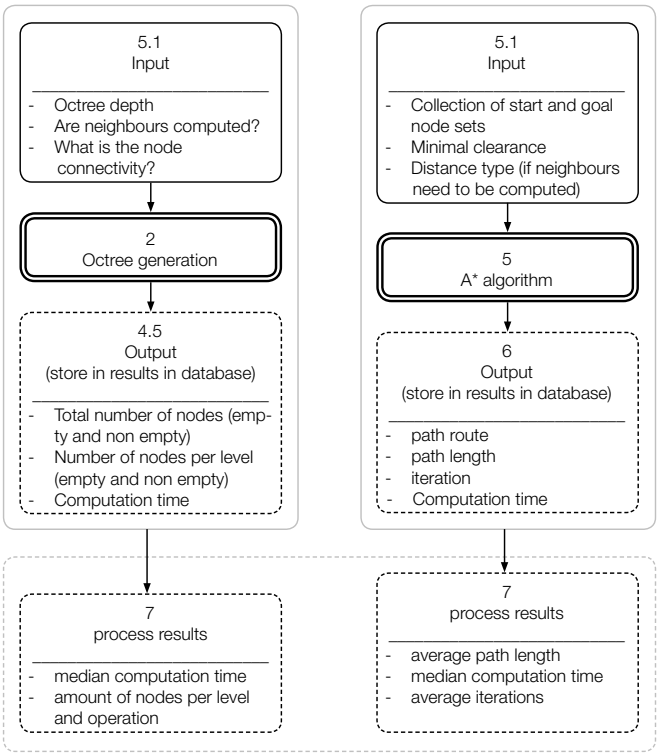

Figure 4.1: Work flow for the octree and path finding benchmark tests

4.5 SOFTWARE/TOOLS

All scripts are written in the programming language 'python'. For the database management postgreSQL was used. To connect Python to postgreSQL, the package psycopg2 is used. The package LibLas is used to work with .las files (point clouds) in python.

For visualizations, the open source software Paraview is used. This software is used to visualize the data. The program supports Python to program source files and filters. Besides, with python it is possible to load data from a postgreSQL database using the psycopg2 package.

The last tool used is CloudCompare, this is mainly used to clean and rotate point clouds.

5 RESULTS AND ANALYSIS

This chapter presents the results of this research. This chapter is structured as follows: section 5.1 describes the results of geometrical point cloud processing. Section 5.2 presents the results of components needed for the generation of the empty nodes and network graph. Section 5.3 shows some example routes in the datasets. And finally, section 5.4 presents the results of the benchmark tests.

5.1 GEOMETRICAL POINT CLOUD PROCESSING

This section describes the results of geometrical point cloud processing. Section 3.1.2 describes how to geometrical process a point cloud so the empty space can be classified most efficiently in an octree.

The effect of geometrical point cloud processing was tested by comparing the difference between a point cloud which is aligned to the axis and that is not. This was done with the Bouwpub and Fire department point cloud. The results are presented in Table 5.1. The size of the bounding box decreased with a factor of $0,91$ for the Bouwpub and $0,81$ for the Fire Department. This increased the spatial resolution of the smallest octant with $0,03$ meters. Keep in mind that the minimum bounding box of the Bouwpub is only $15,94$ meters. Doing this for much larger building can lead to much larger reductions of the spatial resolution.

point cloud	octree depth	aligned to axis	bounding box [m]	scale	spatial resolution of smallest octant [m]
Bouwpub	7	yes	15,94	8,03	0,12
Bouwpub	7	no	17,59	7,27	0,14
Fire department	7	yes	13,86	9,24	0,11
Fire department	7	no	17,02	7,52	0,13

Table 5.1

5.2 EMPTY NODE GENERATION

This section provides the results of the creating of empty nodes and a network graph. The goal of the octree generation is to identify the interior empty space, and create a network graph of this interior empty space. At the same time a collision avoidance system is computed. Figure 5.1 shows the process in which the interior empty space is derived from a point cloud.

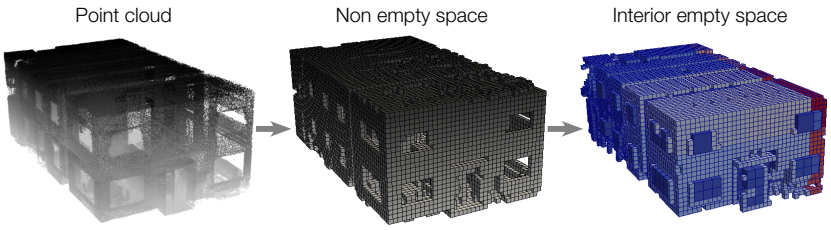

Figure 5.1: Steps in octree generation

5.2.1 Interior empty nodes

During octree construction interior empty nodes are selected and further processed. All exterior empty nodes are filtered out and excluded from further processing and storage. Figure 5.2 shows the exterior (magenta) and interior (blue) empty space of the Bouwpub dataset. Only nodes which have both a non empty node above and beneath them are interior. In this example 5699 exterior empty space nodes are filtered out on 11348 interior empty nodes for an octree with 6 levels. This means about one third of the empty nodes are not processed. In the octree of the Fire department there were 7399 exterior empty nodes filtered out of in total 44506 empty nodes.

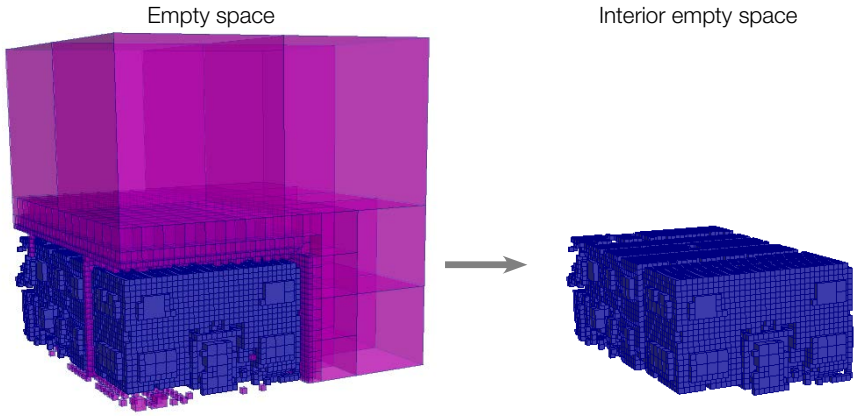

Figure 5.2: Interior and exterior empty nodes.

This method was also used to compute the downward distance between an interior empty node and the closest non empty node. Figure 5.3 shows the interior empty space of the Bouwpub in which the colour represents the downward distance to a non empty node. Figure 5.4 illustrates the effect on path finding if the downward distance to the closest non empty nodes is used in path finding. The line in magenta is computed with a maximal downward distance of 1.2 and the purple line is not constraint. The path of the magenta line remains bound to the ground, even when it has to ascent a stair (where the purple path goes straight to the target).

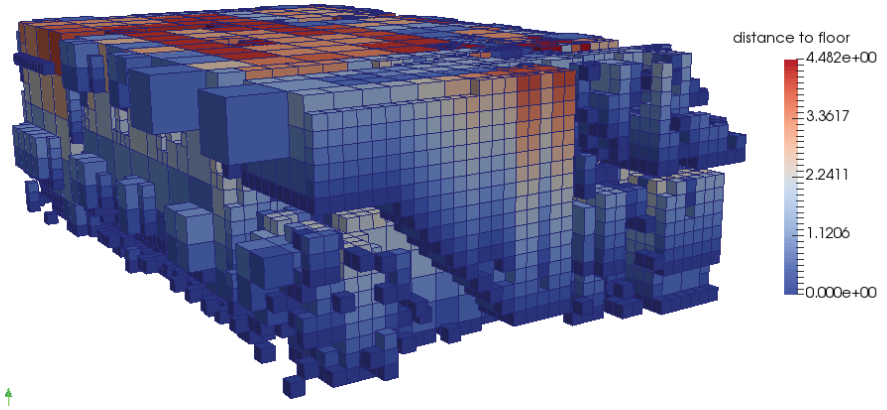

Figure 5.3: Interior empty nodes. The colour of each interior empty node indicates the downward distance to a non empty node.

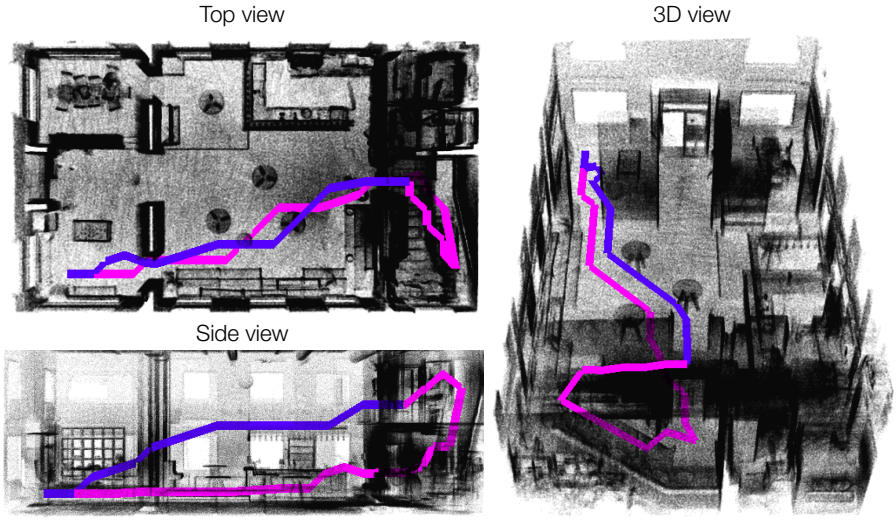

Figure 5.4: Effect of path finding constraint by the downward distance to a non empty node. The magenta path has to stay maximal 1.2 m above a non empty node. The purple line is free, which means it has no maximal downward distance to a non empty node.

5.2.2 Neighbour finding and connectivity generation

Two steps are required to create a connectivity for each interior empty node. Firstly, all possible neighbours are computed. And finally, only the interior empty neighbours are stored.

Neighbour finding

Neighbour finding is done via a python script. During the octree generation all equal and larger neighbours are computed. Figure 5.5 visualizes all possible neighbours of a node with location code '1072' in an octree with six levels. The results are computed in the neighbour finding script and visualized in Paraview.

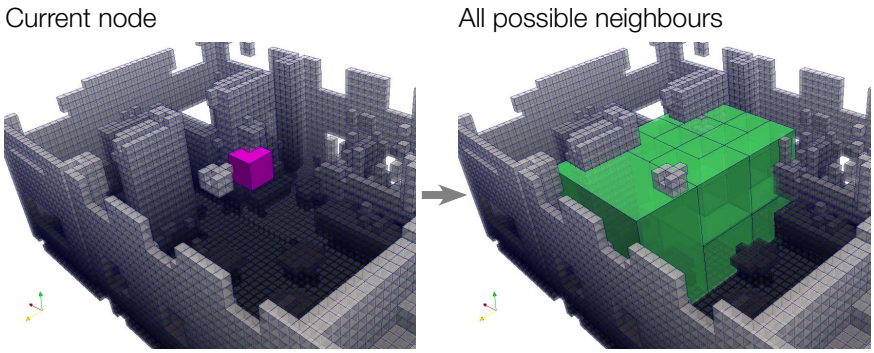

Figure 5.5: All possible equal and larger neighbours of interior empty node '1072' in an octree with six levels of the Bouwpub dataset.

Connectivity generation

During the octree construction, all possible neighbours are computed. If a larger neighbour is an interior empty node, the connectivity is stored in both neighbouring nodes. The complete set of possible equal neighbours is stored in a network graph during the octree construction. When all interior empty nodes are computed, all non interior empty nodes need to be filtered from the network graph. This is done by comparing the possible neighbours with the set of interior empty nodes. Figure 5.6 shows on the left all possible neighbours of node '1072'. On the right of the connectivity of the node, only the interior empty neighbours remain.

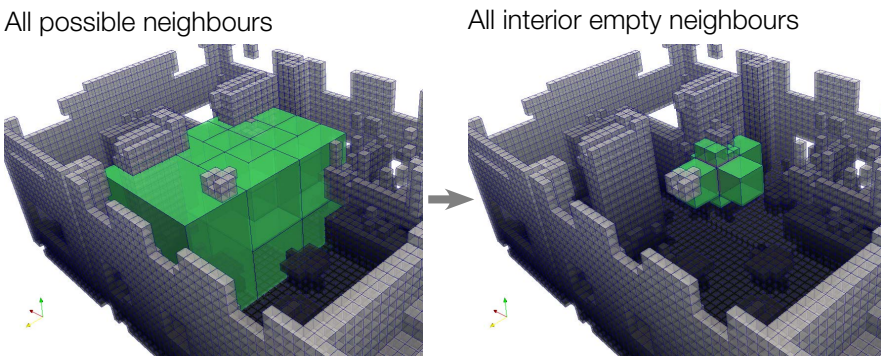

Figure 5.6: The connectivity of interior empty node '1072' in an octree with six levels of the Bouwpub dataset.

5.2.3 Collision avoidance

Collision avoidance is managed by a clearance of each empty node and a maximal crossing value for each connection. The clearance of an empty node expresses the minimal distance between the centre point and the closest border with an non empty node. The maximal crossing value is the minimal distance between an intersection point of a connection and the closest boundary with a non empty node. The following subsections provide the results of the clearance and maximal crossing value.

Clearance map

The clearance of each interior empty node is computed during the octree generation. Figure 5.7 illustrates a clearance map of the interior empty space of the Bouwpub. A horizontal section is created of the non empty nodes opening the roof. Three horizontal sections are made of the interior empty space to give an overview of the clearance throughout the building. A colour scale indicates the clearance of each empty node. Along surface there is little clearance and in open space there is a large clearance.

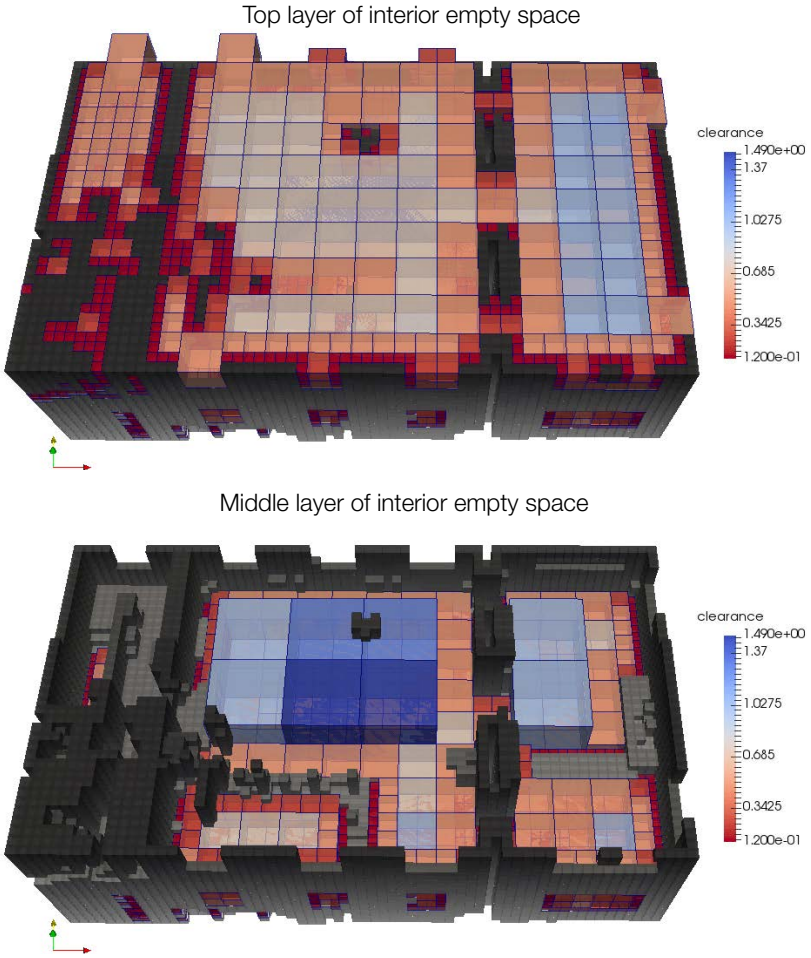

Figure 5.7: Three layers of empty nodes are visualized in a horizontal section of the non empty nodes. The colour of each node indicates the clearance.

Maximal crossing value

The maximal crossing value of each connection is stored in a network graph. For path finding in an octree of the Bouwpub with six levels and a minimal crossing value of 0.17, the average amount of connections of each interior empty node are reduced from $11,15$ to $10,85$.

Figure 5.8 illustrates the working of the collision avoidance system. The purple path is computed with a clearance of $0,17$ and the magenta path has a clearance of $0,4$. The figures show that the paths with a clearance of $0,4$ avoid smaller passages. Moreover, the blue path can move much closer to a

a surface compared to the magenta path. The clearance and crossing value can thus be used to compute a path for objects of different sizes.

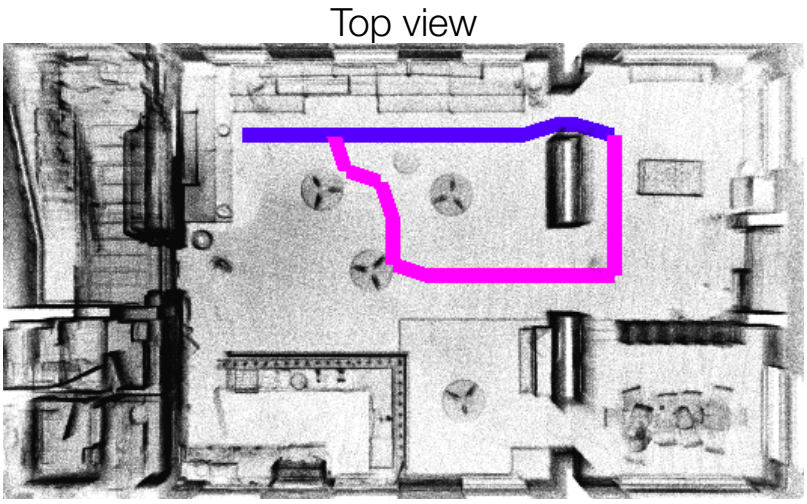

Figure 5.8: The effect of a larger minimal clearance and crossing value.

5.3 A* PATH FINDING ROUTES

Figure 5.9 shows two example paths through the Fire department. These are some of the routes which are used in the benchmark tests.

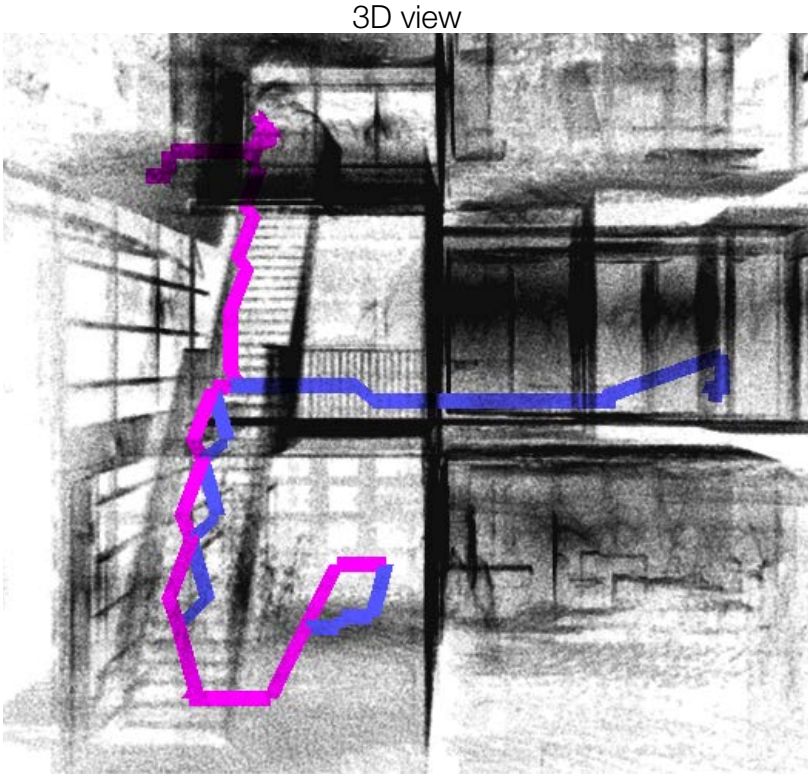

Figure 5.9: The two paths through the point cloud of the Fire department.

5.4 BENCHMARK TESTS

This section describes the benchmark results.

point cloud	depth	connectivity	scale	bounding box	spatial resolution
Test	7	non	2	64	0,5
Test	7	Face	2	64	0,5
Test	7	Face,edge	2	64	0,5
Test	7	Face,edge,vertex	2	64	0,5
Fire department	6	Face	4,62	13,86	0,22
Fire department	7	None	9,24	13,86	0,11
Fire department	7	Face	9,24	13,86	0,11
Fire department	7	Face,edge	9,24	13,86	0,11
Fire department	7	Face,edge,vertex	9,24	13,86	0,11
Fire department	8	Face	18,48	13,86	0,05
Bouwpub	6	Face	4,02	15,94	0,25
Bouwpub	7	None	8,03	15,94	0,12
Bouwpub	7	Face	8,03	15,94	0,12
Bouwpub	7	Face,edge	8,03	15,94	0,12
Bouwpub	7	Face,edge,vertex	8,03	15,94	0,12
Bouwpub	8	Face	16,06	15,94	0,06

Table 5.2: Metadata regarding the octrees used for the benchmark tests

5.4.1 Octree depth

All octrees in this test have a pre-processed face connectivity. All routes in the octree of the Bouwpub are computed with a minimal clearance of $0,17$ and empty nodes can be maximal $1,25$ m above a border with the closest non empty node.

Figure 5.10 shows the effect of octree depth on the path. As the octree depth increases the path goes through smaller octants. This makes is possible to get closer to obstacles. Moreover, the path appears more fluent as smaller neighbours are incorporated in the path.

Initially the effect on path finding would be tested on a range octree depths from 5 to 8. However, for both the point cloud of the Bouwpub and the fire department, the smallest leaf nodes would have a size of about $0,5$ m. The Dutch building regulations 2012 state that a door should have minimal width of $0,85$ meters. Therefore, for indoor path finding, the minimal size an octant represents, should be smaller than $0,85/2 = 0,425$. This ensures that openings in a point cloud are also open space in an octree representation.

Figure 5.11 illustrates the effect on path length. Generally the path length decreases with every increasing octree depth. But this decrease stagnates with an octree of eight levels. The smallest octants in an octree with eight levels represents a size of $0,12$ and $0,11$ for the Bouwpub and the Fire department. So if the smallest octant represents an area smaller than $0,12$ m, the effect on the path length is minimal.

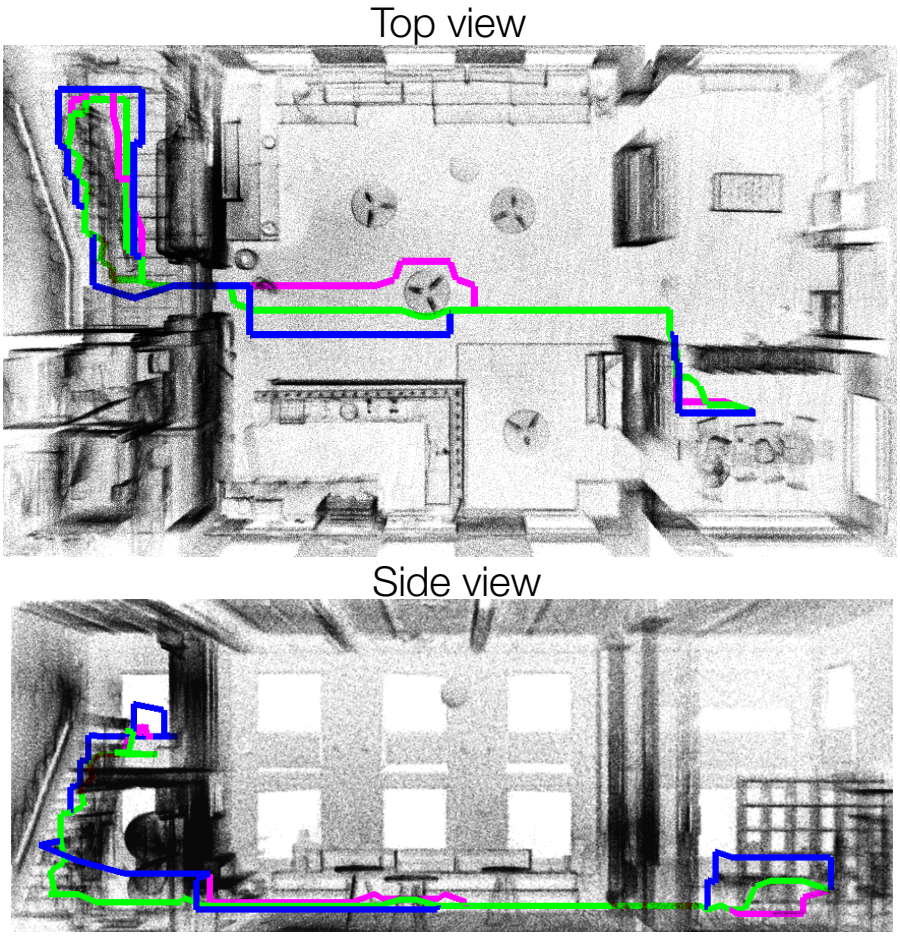

Figure 5.10: Effect of octree depth on the path in the Bouwpub. The colours of the line represent the following octree depths: blue = 6, magenta = 7 and green = 8

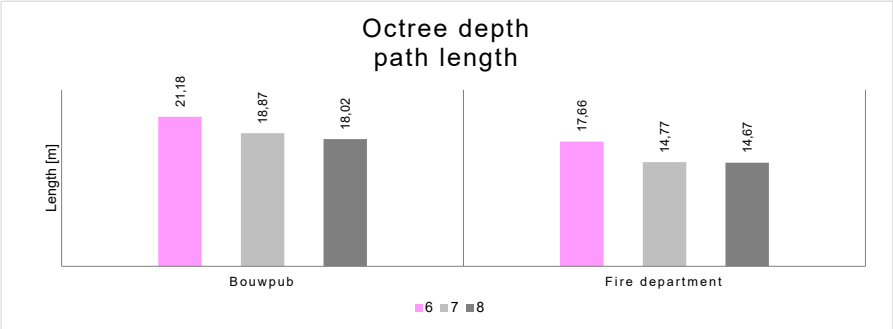

Figure 5.11: Effect of octree depth on path length

The computation time of path finding can be separated into two components: 1) loading and processing of a network graph (when the connectivity is pre-processed); 2) the time to compute path using A* algorithm. Figure 5.12 illustrates the effect on these components. The majority of the computation time is due to loading and processing the network graph into a python dictionary. Also, the increase in computation time can be explained by this, as the octree depth increases the .json file

increases in disk size, thus the loading time increases. The effect on the A* path finding time is minimal. In fact all A* computation times are between $0,206$ and $0,219$ seconds. Although there is a slight increase in computation time.

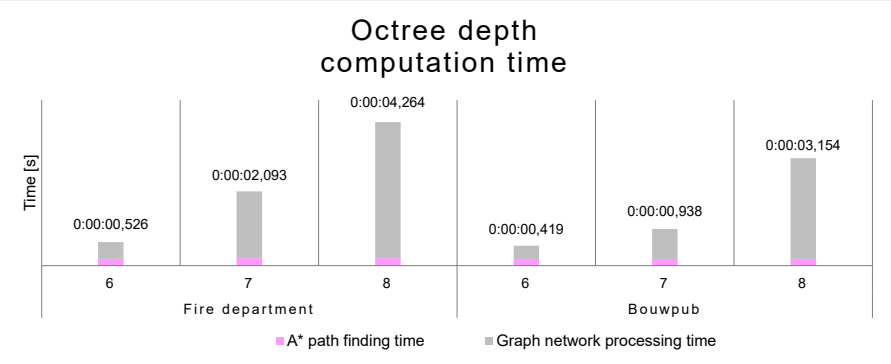

Figure 5.12: Effect of octree depth on computation time. The data labels present the total computation time.

Table 5.3 illustrates the effect on the octree generation time. As the octree depth increases, it gets progressively more demanding to compute an octree. This is partly due to the increasing amount of interior empty nodes and because this extends the connections of smaller interior nodes.

point cloud	octree depth	non empty nodes	interior empty nodes	computation time [s]	stdev [%]
Bouwpub	6	18756	11627	0:02:38	4,3
Bouwpub	7	74080	63899	0:29:56	1,1
Bouwpub	8	169332	191679	3:47:46	5,0
Fire department	6	27848	28606	0:05:46	3,2
Fire department	7	92960	121489	1:05:38	0,8
Fire department	8	186186	309716	6:39:58	0,9

Table 5.3: The effect of depth on octree generation. In all octrees a face connectivity is pre-processed.

5.4.2 Pre-processing connectivity

Like mentioned in section **??**, pre-processing connectivity does not influence the path length as the path finding algorithm is similar. The computation time of path finding can be separated into two components: 1) computation of possible empty nodes (when neighbours are computed on the fly) or loading and processing a network graph (when the connectivity is pre-processed); 2) the time to compute path using A* algorithm. These results are separately presented in Figures 5.13 and 5.14. All octrees have a tree depth of 7 and the connectivity is pre-processed. All routes in the octree of the Bouwpub are computed with a minimal clearance of $0,17$ and empty nodes can be maximal $1,25$ m above a border with the closest non empty node. For the test point cloud, the minimal clearance and distance to

a non empty node is set to 0.

Figure 5.13 shows that loading and processing a network graph takes much more time than loading interior empty nodes from a database. This makes sense as for constructing a network graph, first all interior nodes need to be loaded from the database. Next, the network graph is loaded from a .json file and finally, all non accessible interior nodes are filtered out.

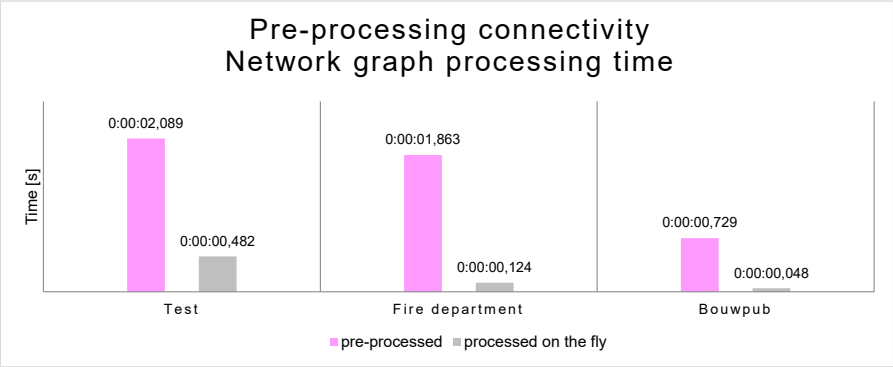

Figure 5.13: Effect of pre-processing neighbours on the path computation time. The results are computed with an octree with 7 levels.

Pre-processing the network graph shows a massive improvement in A* path finding computation time. The paths in the different datasets are on average computed in about $0,2$ second. The reason is that all the steps are pre-processed and do not have to be computed during path finding. Also, Figure 5.12 shows that there is little effect on the computation time due to an increasing octree depth.

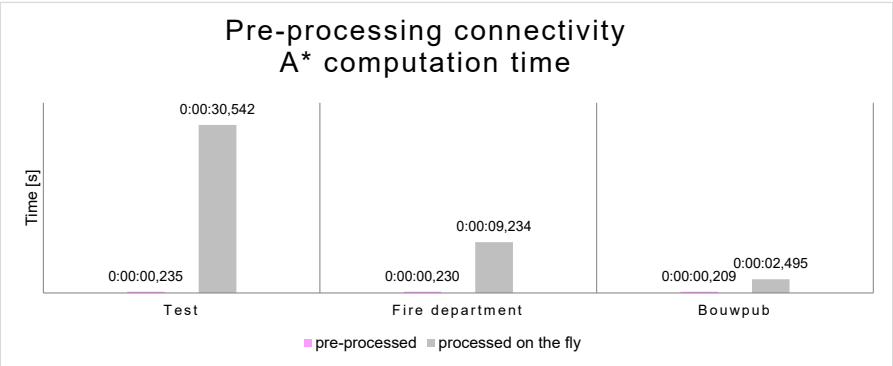

Figure 5.14: Effect of pre-processing neighbours on the path computation time. The results are computed with an octree with 7 levels.

Summing the two time components gives the total path finding computation time. Pre-processing a network graph is in all datasets faster than processing neighbours on the fly. But the bottle neck in all datasets is the speed of processing the network graph.

Table 5.4 shows the octree computation time for the Bouwpub and Fire department and test point cloud. Pre-processing a face connectivity has a negative effect on the computation time, however this effect is minimal. The only difference between pre-processing a face connection and no connectivity is the creation and storing of a network graph. In both octrees,

all neighbours are computed as they are needed to compute the clearance of each interior empty node.

point cloud	octree depth	pre-processed	computation time [s]	stdev [%]
Test	7	yes	0:04:58,067	0,55
Test	7	no	0:04:56,199	0,53
Fire department	7	yes	1:05:38,803	0,79
Fire department	7	no	1:05:17,381	0,61
Bouwpub	7	yes	0:29:56,306	1,08
Bouwpub	7	no	0:29:41,680	0,95

Table 5.4: The effect of pre processing a face connectivity on octree generation time.

5.4.3 Type of path connectivity

Three types of path connectivities were tested: 6 connectivity (face), 18 connectivity (face and edge) and 26 connectivity (face, edge and vertex). All octrees have a tree depth of 7 and the connectivity is pre-processed. All routes in the octree of the Bouwpub and the Fire department have a minimal clearance of $0,17$ and empty nodes can have a maximal downward distance to the closest border with a non empty node of $1,25$ m. For the test point cloud, the minimal clearance and downward distance to a non empty node is set to 0.

Figure 5.15 shows an example route with the three types of connectivities. Due to the edge and vertex connection, the 18 and 26 connectivity are able to connect to nodes which are not accessible for a 6 connectivity. This is way the 28 and 26 connectivity are able to find a path over a bar (L shaped object). The path of a 6 connectivity is more effected by obstacles due to the limited connections and therefore has a route with more detours. Finally, in the figure it is clearly visible that the 6 connectivity move orthogonal where the extended connectivities are able to move diagonal.

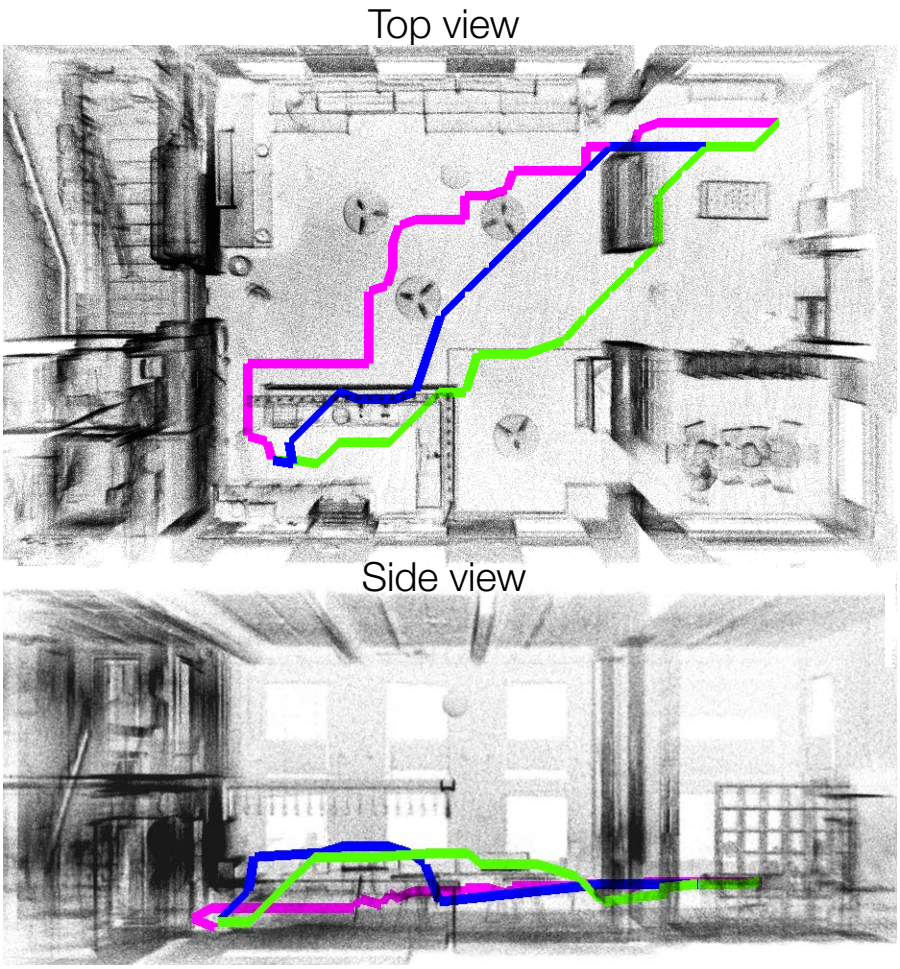

Figure 5.15: A path is computed between a start and goal node for each type of connectivity(face (magenta); face and edge(blue); face, edge and vertex(green))

Figure 5.16 presents the effects on the path length. In all test datasets, the path length decreases as the connectivity extends. Between 6 and 18 connectivity the path length decreases on average with 10%. Between a 6 and 26 connectivity the path decreases with 12%. The reduction in path length can be explained by the difference in directions in which movement is possible between the different connectivities. Between a 6 and 18 connectivity there are three times as much directions in which movement is possible for each node. This makes it more likely to find a path which moves straight to the target.

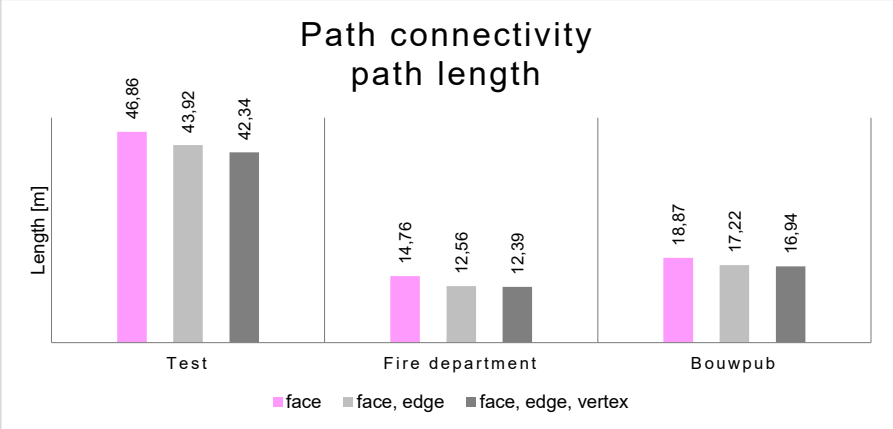

Figure 5.16: Effect of octree depth on octree generation on the path length

Figure 5.17 presents the path finding computation times. The computation time increases as the connectivity is extended. The computation time is mainly due to loading and processing a .json file into a python dictionary. Also, processing the .json file is the prime reason for the increase in computation time due to the extended connectivity. The reason for this is an increasing larger .json file size for the 18 and 26 connectivity.

The A* computation time is between $0,211$ and $0,232$ for the different datasets. There is no clear proof that an extended connectivity improves the A* computation time. The computation time is mainly related to the A* iterations, which are illustrated in Figure 5.22. The amount of A* iterations is effected by two things: 1) the amount of empty nodes which are explored; 2) the amount of possible neighbours for each empty node. A path with an extended connectivity is likely to have a more direct line toward the goal node and therefore visits less empty nodes. Accordingly it is expected to have less A* iterations. Although, due to an extended connectivity the amount of possible neighbours increase for each empty node. The average amount of connections for each interior empty node was measured at 10 for a 6 connectivity, 15 for an 18 connectivity and 19 for a 26 connectivity. All these neighbours are processed and measured as an A* iteration. Thus, due to an extended connectivity, there can be more A* iterations.

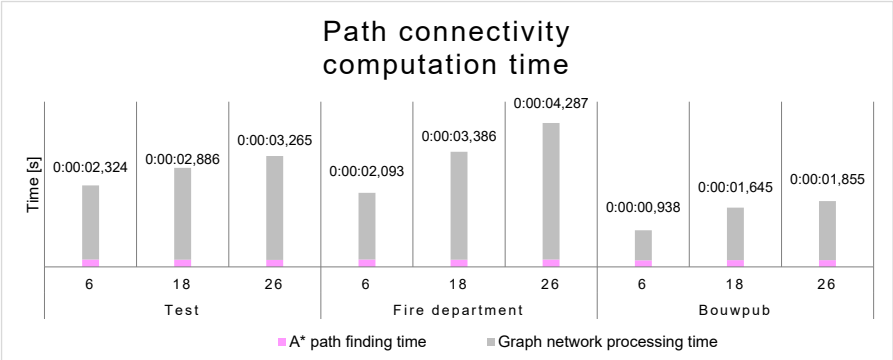

Figure 5.17: Effect of octree depth on octree generation on the path length

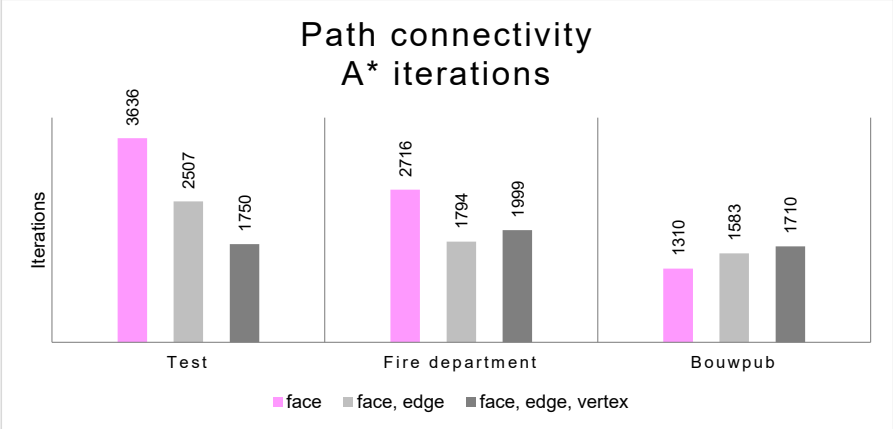

Figure 5.18: Effect of octree depth on octree generation on the path length

Table 5.5 shows the effect of an extended path connectivity on the octree generation time. The computation time increases on average with a factor of $2,2$ between a 6 (face) and 18 (face, edge) connectivity and on average with a factor of $1,2$ between an 18 (face, edge) and 26 (face, edge, vertex) connectivity. The amount of interior empty nodes remains the same for each octree, as only the connectivity is extended. This means the increase in computation time is due to processing of interior empty neighbours.

point cloud	octree depth	connectivity	computation time [s]	stdev [%]
Bouwpub	7	face	0:29:56,306	1,06
Bouwpub	7	face, edge	1:06:17,476	2,74
Bouwpub	7	face, edge, vertex	1:18:22,991	1,47
Fire department	7	face	1:05:38,803	0,79
Fire department	7	face, edge	2:19:32,189	1,29
Fire department	7	face, edge, vertex	2:52:27,485	0,80
Test	7	face	0:04:58,067	0,55
Test	7	face, edge	0:10:58,690	0,30
Test	7	face, edge, vertex	0:11:46,376	0,36

Table 5.5: Effect of path connectivity on octree generation time

5.4.4 Distance type between nodes

Three types of distance types were tested: Euclidean, Manhattan and chessboard. All octrees have a tree depth of 7 and face neighbours are computed on the fly. All routes in the octree of the Bouwpub and fire department are computed with a minimal clearance of $0,17$ and empty nodes can be maximal $1,25$ m above a border with the closest non empty node. For the test point cloud the minimal clearance and downward distance to a non empty node is set to 0.

Figure 5.19 shows a path computed between a start and goal node for the Manhattan and Euclidean distance. The chessboard distance was not visualized as it had almost the same path as the Euclidean distance. Note

that the Manhattan distance has a tendency to follow an orthogonal path. The Euclidean distance is able to move more freely.

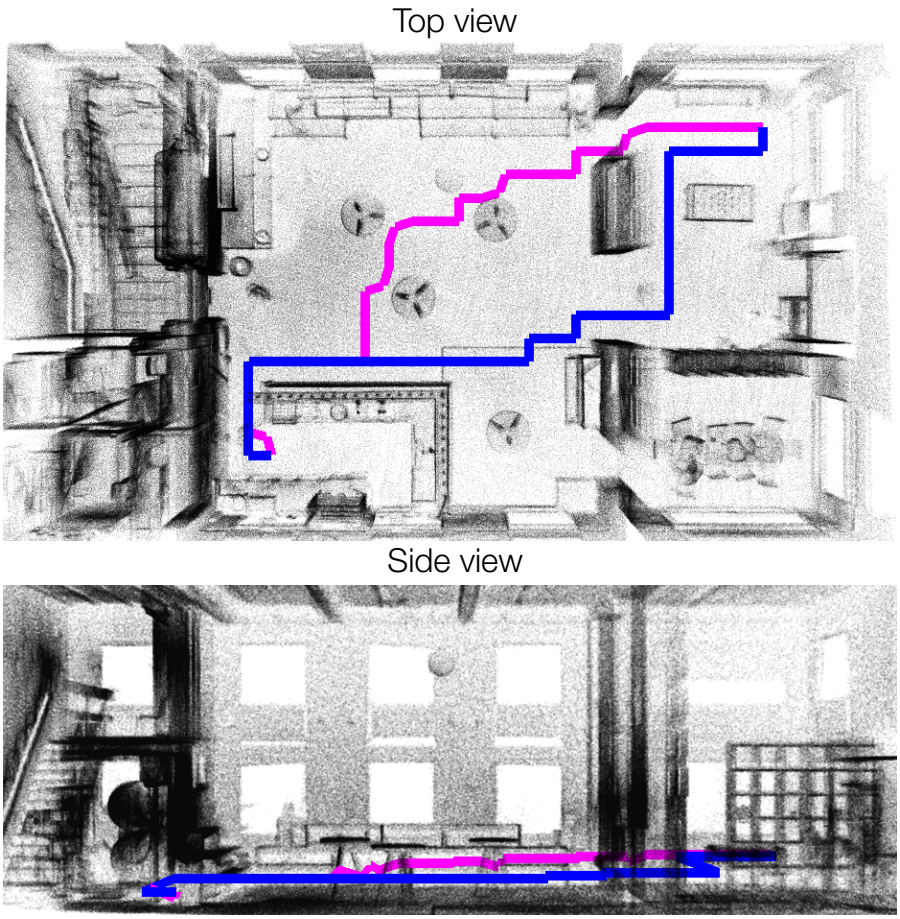

Figure 5.19: A path is computed between a start and goal node for the Manhattan (blue) and Euclidean (magenta) distance

Figure 5.20 shows the results of the path finding benchmark tests on path length. In all datasets, the Euclidean distance finds the shortest path and the Manhattan distance the longest path. Especially in the octree of the test point cloud there is a significant difference ($54,42 - 46,86 = 7,56$ m). The difference in the octree of the Bouwpub and fire department is less significant ($0,65$ m and $0,66$ m). In the Manhattan distance it is expensive to move diagonal as the distance is the sum of orthogonal components. It therefore prefers to move in straight lines, by doing so it drifts from the optimal path, whereas the Euclidean distance always follows the optimal path by computing the distance using the theorem of Pythagoras.

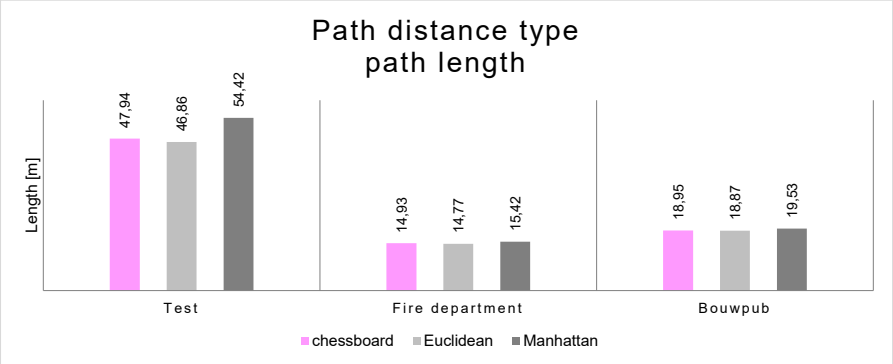

Figure 5.20: Effect of octree depth on octree generation on the path length

Figure 5.21 and 5.22 present the A* computation time and iterations. In contrast to the path length the Manhattan distance is most efficient in terms of computation time. Especially in the test and the fire department point cloud, there is significant improvement compared to the other distance types. The differences in computation time can be explained by the A* iterations. In all datasets, the Manhattan distance has the smallest amount of A* iterations. Meaning, less nodes need to be inspected in A* path finding.

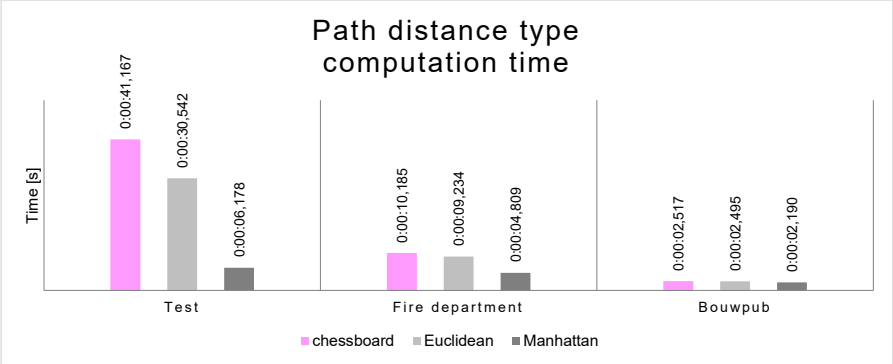

Figure 5.21: Effect of octree depth on octree generation on the path length

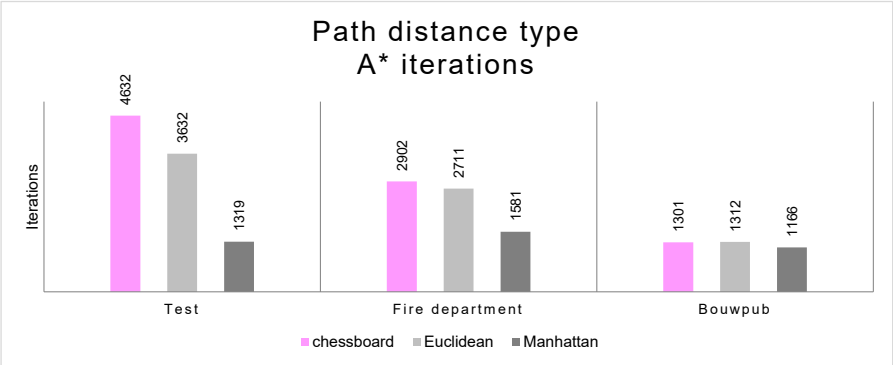

Figure 5.22: Effect of octree depth on octree generation on the path length

6 CONCLUSION AND DISCUSSION

This chapter will conclude this research with the following sections: 1) summary of the results; 2) answering of the research questions; 3) explain my own contribution; 4) a discussion about the research and finally 5) a future work section.

6.1 RESULTS

This section summarises the main results of this research. The research was constructed in two parts: the first covered the octree generation. The main contribution to the octree construction is the way empty nodes are processed. During the construction of these empty nodes, interior empty nodes are identified, a network graph is constructed and collision avoidance system is implemented. The second part of the research consist of benchmark test. These tests identified the effect of A* characteristics on path length and computation time.

6.1.1 Octree generation

The octree generation method used in this research is based on the design of Broersen et al. [2016]. It consist of three steps: 1) pre processing a point cloud; 2) identifying the non empty space from the point cloud; 3) deriving the empty space from the non empty space. I used this method to create a data structure for path finding in an indoor point cloud, in which the octree acts like a catalyst. The following list summarizes the results of all the components which are needed to construct this data structure.

1. **Point cloud datasets**
   Three point cloud datasets were used in this research. A point cloud of the pub of the Faculty of Architecture and the Built Environment at DUT called the 'Bouwpub'. A second point cloud of a Fire department in Berkel en Rodenrijs called the 'fire department'. And finally a point cloud created from 3000 random points called 'Test point cloud'.

2. **Geometrical point cloud processing**
   To segment a point cloud to its full potential it should be geometrical processed. The work flow contains three steps: 1) align the minimum bounding box of a point cloud to the axis of the octree grid, keeping the floor surface horizontal; 2) translate the point cloud so the origin has coordinate $(0,0,0)$; and 3) scale the point cloud so it fits perfectly in a grid of $2^n * 2^n * 2^n$, where n represents the octree depth.

   This work flow scales the point cloud to its full potential, and in turn minimizing the spatial resolution of the octants. A large spatial resolution is beneficial for the octree depth needed for (indoor) path

finding, as a minimal spatial resolution is needed to keep all open spaces connected.

The scale of a point cloud can increase up to a factor $\sqrt{2}$ by aligning it to the octree grid and keeping the floor horizontal. This means that the spatial resolution of the smallest octant decrease with a factor $\sqrt{2}$. In a building with a footprint of $100 * 100$ this means a maximal reduction of the spatial resolution of $0,41$ meter. This way a point cloud can be segmented in an octree with a smaller depth while keeping the minimal octant size needed for indoor path finding.

3. **Interior empty nodes**
   Identifying the interior empty nodes works well for the datasets used in this research. Filtering the exterior empty nodes reduces the amount of computations and memory needed for the octree construction. Besides there are major benefits for indoor path finding. By filtering all exterior empty nodes path finding is solely possible through the interior empty space. It is no longer possible to exit a building via windows or doors. This method works only for closed roofs and floors.

   By computing the downward distance to the closest non empty node a first step is taken towards indoor path finding over the floor. However there are still two problems: firstly, the downward distance to the closest non empty node does not mean it is the distance to a floor. It can be the distance to any horizontal surface, like furniture or even lights. Secondly, for collision free path finding over the floor the distance to the floor might conflict with the clearance of a node. As the clearance is dependant on the closest non empty node. This method was successfully used in path finding.

4. **Connectivity generation**
   Computing the connectivity of an interior empty node consist of two steps: first all possible equal and larger neighbours are computed. IF a larger neighbours is an interior empty node the connection is stored in both directions to connect larger nodes to smaller neighbours. Between all connecting nodes the distance between the centre points is computed and stored. And second, all non interior empty neighbours are filtered out. Finally, the connectivity of all interior empty nodes is stored in a dictionary which can be used as network graph for path finding.

5. **Collision avoidance**
   Collision avoidance is managed with a clearance of each interior empty node and a maximal crossing value for each connection. The clearance indicates the distance to a border with a non empty node from the centre point of the interior empty node. The clearance can be used for object fitting, but not for moving between neighbouring interior empty nodes. For this, the distance from the crossing point to the closest border with non empty node is needed. Both the clearance and maximal crossing value are used to filter the network graph of all inaccessible interior empty nodes and connections. By increasing the minimal clearance and crossing value, the method can be used for path finding for different objects.

6. **A* path finding**
   A script was created for A* path finding through the interior empty

space of a point cloud. The script could be used in two ways: 1) with a network graph; 2) computing neighbours on the fly. In the first method, a network graph is loaded and filtered and used in A* path finding. In the second method the neighbours are computed and filtered on the fly during path finding.

6.1.2 Benchmark tests

1. **The effect of Octree depth**

   For path finding in an indoor point cloud the smallest octant in an octree should represent a space smaller than $0,425$* m $0,425$ m $*0,425$. This ensures that all open volumes in an octree are connected. The formula to compute the minimal octree depth is provided in equation 6.2.

   The spatial resolution of a point cloud can be calculated with the formula:

   $$bbox/2^n = resolution \tag{6.1}$$

   Where n is the octree depth, $bbox$ is the size of teh axis-aligned bounding box and $resolution$ refers to the spatial resolution.

   The equation to compute the minimal octree depth for indoor path finding:

   $$n = \log(bbox/resolution)/\log(2) \tag{6.2}$$

   The benchmark test showed that there is little improvement in path length with octants representing a size smaller than $0,15$ m. Non empty nodes are always split until the lowest octree level. By increasing the octree depth all non empty and interior empty nodes in the lowest octree level are created by splitting their non empty ancestor node. This means that an interior empty nodes in the lowest level must be connected to a non empty node. If an octree has a spatial resolution lower than $0,15$ the interior empty nodes in the lowest level can never be used in a path if the minimal clearance is larger than $0,15/2 = 0,08$ m. This means it makes sense that splitting the octree any more than a spatial resolution of $0,15$ does not improve the path length.

   Therefore, for indoor path finding the depth an octree should be defined by the size which is represented by the smallest octants.

   The path finding computation time has two components: 1) loading and processing a network graph; 2) the time to compute a path using the A* algorithm. The majority of the total computation time is due to loading and processing the network graph. For each increasing octree depth the network graph grows in size. This results in more time to load and process the network graph. Still, the loading and processing time for an octree with seven levels measured at $0,7$ and $1,9$ seconds for the Bouwpub and Fire department point cloud. The A* path finding computation time remains constant around $0,2$ seconds for all octree depths. Loading and processing of the network graph is the bottleneck

in the method. But even for an octree with eight levels the maximum computation time is about 4 seconds.

A solution where the network graph is not loaded in the Random-access memory (RAM) but stored and accessed from the main computer memory, could be beneficial for path finding. This way the network graph can be accessed without loading it in the local memory. Now this would not make up most of the total path finding time. With a A* path finding time remaining constant, this could be a solution for path finding in large buildings where a bigger octree depth is necessary.

Extending the octree depth progressively increases the octree computation time. This is due to the increasing amount of interior empty nodes, and because this can extend the amount of connections of interior nodes. The benchmark test showed computation times well over 6 hours for eight octree levels. The computation time could be problematic for octrees with more than eight levels. More research should be conducted to identify the use of the method for large buildings.

2. **The effect of pre-processing node connectivity**

   Pre processing neighbours has no effect on the path length as there is no difference in the path finding algorithm. However the computation time is effected. Loading and processing a network graph takes more time than loading interior empty nodes from a database.

   This makes sense as for constructing a network graph first all interior nodes need to be loaded from the database. Next the network graph is loaded from a .json file and all non accessible interior empty nodes are filtered out. The time to load and process the network graph is mainly dependant on the disk size of the network graph.

   Path finding with pre-processed connectivities proves to be beneficial for the computation time. All paths were computed in around $0,2$ seconds in all datasets. A* path finding without a network graph took between 2 to 30 seconds. Using a network graph eliminates a number of operations during path finding: 1) neighbour finding; 2) connectivity generation; 3) computing the distance to neighbouring node.

   Loading and processing a network graph is the bottle neck in this implementation. Loading the entire network graph in the temporary memory takes up almost all of the total computation time.

   Pre-processing a face connectivity has a minimal effect on the octree computation time. The only difference between pre-processing a face connection and no connectivity is the creation and storing of a network graph. In both octrees all neighbours are computed as they are needed to compute the clearance of each interior empty node.

3. **The effect of path connectivity**
   In all test datasets the path length decreases as the connectivity extends. Between a 6 and 18 connectivity the path length decreases with an average of 10%. And between a 6 and 26 connectivity the path decreases with 12%. The reduction in path length can be explained by the increase in directions in which movement is possible between two connected interior empty nodes.

The computation time increases as the connectivity is extended. The computation time is mainly due to loading and processing a .json file into a python dictionary. Also, processing the .json file is the prime reason for the increase in computation time due to the extended connectivity. The reason is an increasing larger .json file size for the 18 and 26 connectivity.

The octree computation time increases on average with a factor of $2,2$ between a 6 (face) and 18 (face, edge) connectivity. And with on average with a factor of $1,2$ between an 18 (face, edge) and 26 (face, edge, vertex) connectivity. The amount of interior empty nodes remain the same for each octree as only the connectivity is extended. This means, that the increase in computation time is due to processing of interior empty neighbours.

4. **The effect of path distance types**
   A path computed with a Manhattan distance tends to follow an orthogonal path. A path computed with a chessboard distance is more likely to follow a diagonal path.

   Tests showed that the Euclidean distance finds the shortest path and the Manhattan distance the longest path. In the Manhattan distance it is expensive to move diagonal as the distance is the sum of orthogonal components. It therefore prefers to move in straight lines, by doing so it drifts from the optimal path, whereas the Euclidean distance always follows the optimal path by computing the distance using the theorem of Pythagoras.

   In contrast to the path length, the Manhattan distance is most efficient in terms of computation time. The differences in computation time can be explained by the A* iterations. The Manhattan distance has the least amount of A* iterations. Meaning, less nodes need to be inspected in A* path finding. This also means that it is not the speed of the calculation that makes the difference. But the amount of nodes which need to be explored to find a goal.

6.2 RESEARCH QUESTIONS

This section answers the research questions. It begins with answering the research sub questions and end with an answer of the main research question.

1) What geometrical point cloud processing operations are important for the generation of an octree and what is their effect?

As the octree depth increases it gets progressively more demanding to generate the octree. For (indoor) path finding the smallest octants should have a minimal spatial resolution. Maximizing the spatial resolution by geometrically processing a point cloud can be used to minimize the necessary octree depth, in turn reducing the octree generation time, storage space, amount of empty nodes and connectivity of the interior empty nodes.

There are three geometrical operations which should be performed on a point cloud: 1) rotation; 2) translation; 3) scaling. Of which scaling is the only operation which can influence the spatial resolution of an octree.

To get the best octree resolution, a point cloud should be scaled so the axis-aligned bounding box fits in an octree grid of $2^n * 2^n * 2^n$, where n refers to the octree depth.

Both rotation and translation can effect the efficiency of scaling by changing the size and location of the axis-aligned bounding box.

By aligning the minimum bounding box to the octree axis, the point cloud covers the smallest volume in an octree space. This means a point cloud can have a larger scale to best fit in the octree space. Although, for visualization purposes it is recommended to keep the floor plane horizontal. The scale and spatial resolution of the point cloud can be increased with a factor of $\sqrt{2}$ by aligning the minimum bounding box to the octree axis.

Finally, the point cloud should be translated so the origin of the bounding box is located in coordinates $(0,0,0)$. As this is the origin point for the scaling operation. Both rotation and translation prepare the point cloud so it can be scaled to the full potential.

2) What octree operators influence the computational effort and path length and what is their effect?

1. **Octree depth**
   There is no universal octree depth that is best suited for (indoor) path finding. The octree depth depends on the minimal and maximal spatial resolution needed for path finding, which in turn depends on the building size.

   For path finding in an indoor point cloud the minimal spatial resolution of an octree should be $0,425$ meter. This ensures that all open volumes in an octree are connected.

   Benchmark tests show that there is little improvement in path length in an octree with a spatial resolution larger than $0,15$ m. The smallest interior empty nodes in this resolution are always connected to a non empty node. Therefore they have little clearance and are filtered out if the clearance is not sufficient. The maximal resolution should be defined by the expected largest objects which have to find a path.

   The path finding computation time has two components: 1) loading and processing a network graph; 2) the time to compute a path using the A* algorithm. The majority of the total computation time is due to loading and processing the network graph. For each increasing octree depth the network graph grows in size. This results in more time to load and process the network graph.

   Extending the octree depth progressively increases the octree computation time. This is due to the increasing amount of interior empty nodes, and because this can extend the amount of connections of interior nodes. The benchmark test showed computation times well over 6 hours for eight octree levels. The computation time could be problematic for octree with more than eight levels.

2. **Pre processing connectivity**

   Pre-processing neighbours has no effect on the path length as there is no difference in the path finding algorithm. Path finding with pre-processed connectivities proves beneficial for the total A* computation time. In all tests the path finding time with a

pre-processed network graph was significantly faster than its equivalent of computing a node connectivity on the fly. The majority of path finding time was due to loading and processing the network graph. An interesting direction for future work is therefore researching other ways of storing and accessing a network graph.

Pre-processing a face connectivity has a minimal effect on the octree computation time. The only difference between pre-processing a face connection and no connectivity is the creation and storing of a network graph. In both octrees all neighbours are computed as they are needed to compute the clearance of each interior empty node. Although the pre-processing time will increase with extended path connectivities.

3) What A algorithm operations influence the computational effort and path length and what is their effect?*

1. **Path connectivity**
   The path length decreases as the connectivity extends. Between a 6 and 18 connectivity the path length decreases on average with 10%. Between a 6 and 26 connectivity the path decreases with 12%. The reduction in path length can be explained by the increase directions in which movement is possible.

   The computation time increases as the connectivity is extended. The computation time is mainly due to loading and processing a network graph. Also, processing the network graph is the prime reason for the increase in computation time due to the extended connectivity. The reason is an increasingly larger file size for the 18 and 26 connectivity.

   The octree computation time increases on average with a factor of $2,2$ between a 6 (face) and 18 (face, edge) connectivity. And with on average with a factor of $1,2$ between an 18 (face, edge) and 26 (face, edge, vertex) connectivity. The increase in computation time is due to processing of interior empty neighbours.

2. **Distance type between nodes**
   Tests showed that the Euclidean distance finds the shortest path and the Manhattan distance the longest path. In the Manhattan distance it is expensive to move diagonal as the distance is the sum of orthogonal components. It therefore prefers to move in straight lines, by doing so it drifts from the optimal path, whereas the Euclidean distance always follows the optimal path by computing the distance using the theorem of Pythagoras.

   IThe Manhattan distance is most efficient in terms of computation time. The differences in computation time can be explained by the A* iterations. The Manhattan distance has the least amount of A* iterations. Thus needs to explore less node to find a goal node.

With the answers of the sub questions the main research question can be answered:

What is the effect of A path finding characteristics on the path length and performance in an octree representation of an indoor point cloud?*

The A* path finding computation time is positively effected by: firstly, pre-processing a network graph, because most information needed for path

finding is pre-processed. Secondly, using a Manhattan distance in A* path finding proved beneficial for the computation time, as it needs to explore less nodes to reach a goal node. And finally, keeping the octree depth to a minimal proved beneficial, as there are less interior empty nodes and path connections. Test results showed that loading and processing the network graph takes up most of the path finding computation time. An efficient storage and accessing method could make the method scalable, as the actual A* path finding time remained constant for all path finding tests.

From the answers of the research sub questions, it can be concluded that the path length is positively influenced by: firstly, extending the path connectivity, due to the extended connectivity each empty node has more directions in which movement is possible. Secondly, by using a Euclidean distance type the path is less likely to drift from the shortest path. And finally, by increasing the octree depth, this allows a path to select smaller empty nodes closer to obstacle, making it possible to find a path which is less influenced by obstacles. Of the methods, increasing the path connectivity reduces the path length the most. Even more than increasing the octree depth. And compared to extending the octree depth, an extended path connectivity has less effect on the octree computation time.

There is no universal optimal octree depth for indoor path finding. The octree depth depends on the minimal and maximal spatial resolution needed for path finding, which in turn depends on the minimum bounding box of the point cloud. By improving the spatial resolution, the octree depth can be minimized. The spatial resolution can be maximized by pre-processing the point cloud. This thesis proposes the following three step for a optimal spatial resolution:

1. The minimum bounding box should be aligned to the octree axis.
2. The origin of the aligned minimum bounding box should be translated to coordinate $(0,0,0)$.
3. The point cloud should be scaled to fit in a grid of $2^n * 2^n * 2^n$, where n refers to the octree depth.

6.3 OWN CONTRIBUTION

Beside answering the research questions, this thesis aimed to provide a method for path finding through the interior empty space of a point cloud. The following list summarizes my contributions to this research.

1. This research presents a new workflow for path finding through an octree representation of an indoor point cloud.
2. I created a new method to identify the interior empty nodes.
3. This method can also be used, little more computations, to compute the downward distance to a non empty border.
4. I implemented a new method for collision avoidance in the-pre processing stage. This method consist of two components: 1) a clearance of the centre of each interior empty node; 2) a maximal crossing value for each intersection between connected interior empty neighbours.

5. I implemented an efficient method to pre processing node connectivity.
6. I extended the neighbour finding method of Vörös [2000] by computing equal and larger edge and vertex neighbours.
7. I identified geometrical point cloud processing operations important for octree generation. And proposed a work flow for optimal point cloud classification.
8. I identified the effects of octree operators and A* operations on the computation time and path length of A* path finding.

6.4 DISCUSSION

This section discusses the methods and results in this research.

The method implemted in this research is not completely autocatic, aligning the minimum bounding box to the axis is done manual. Automatic alignment of the minimum bounding box could be an interesting direction for future work.

Currently, loading and processing the network graph for A* path finding is the bottle neck in the path finding implementation. A solution could be storing and accessing this in the computer memory. This eliminates the step of loading and processing the network graph.

A side product of identifying the interior empty space is the downward distance to non empty node for each interior empty node. This distance can be used to compute a path suited for walking. There are some problems, firstly the downward distance can conflict with the clearance. This is because the clearance is computed with all equal face, edge and vertex neighbours. This includes nodes directly under the node. Therefore the clearance of nodes directly above a non empty node is the same distance as the downward distance. To make the method suitable for a walk able path the clearance should be computed only with neighbours in the x and y direction. And finally, the downward distance in not per definition the distance to the floor level. as it should be possible to find a path over furniture or other obstacles.

In this method path finding is always through the centre points of interior empty nodes. In the case of a door opening where the open space is represented by two interior empty nodes side by side. Together these nodes can have a clearance large enough for movement but individually nodes have a clearance which is not sufficient. This can block certain paths which are actually passable. This could be solved by allowing movement between the borders of interior empty nodes.

The main criticism about the total implementation is that it does not yet work to the full potential, because the implementation is constructed of several scripts some processes are computed double. The total implementation could work more efficiently when more processes are merged.

The main criticisms regarding to the benchmark tests are twofold. Firstly, to get a more reliable test result, more datasets should be tested. Secondly, the datasets used were of relatively small buildings. It could be interesting how the method works with large building, e.g congress halls or the Faculty of Architecture and the Built Environment at DUT.

6.5 FUTURE WORK

In section I present a list of interesting ideas I had, but was not able to implement and possible future work.

1. **Improve the method to a web service**
   The next step of this research is to build a web service for processing a point cloud for path finding. The idea would be to load a point cloud in the web service. This point cloud would be geometrical pre processed and the interior empty space would be identified. Once this is done the point cloud with the interior empty space should be loaded to a screen and the user should be able to pick a start and goal node for indoor path finding. Subsequently, the web service should compute and visualize the path.

2. **Automatic alignment of minimum bounding box to octree axis**
   The method in this research is, besides the alignment of the minimum bounding box completely automatic. Therefore, a good future work subject could be automatic alignment of the minimum bounding box. This could possible be done using RANSAC plane fitting algorithm.

3. **Storing and accessing of a network graph from the computer memory**
   The bottleneck in the path finding algorithm was the loading and processing of the network graph. One way to avoid this could by storing and accessing the network graph from the computer memory instead of the Random Access Memory (RAM). This way the path finding time would only be defined by A* path finding. Tests in the research showed the A* path finding time to be relatively constant for different octree depths, making it useful for larger building where a higher octree depth is needed.

4. **Integrating indoor and outdoor path finding**
   Indoor and outdoor path finding are seen as two separate things. But it can be very useful to have a seaming less connection between the two. For this the interior and exterior environment should be connected somehow. This could either be done by placing them in the same spatial reference system, or by semantically labelled interior nodes in exits and using these semantically enriched nodes to enter and exit a building.

5. **Use Dijkstra path finding to find closest emergency exit**
   If the Dijkstra algorithm would be used in path finding it is possible to find all possible destinations from an interior node. This property could be used to find the closest emergency exit to a certain node (location). Also for this the interior empty nodes in or near doors should be semantically labelled.

6. **Path finding constraint to vertical and/or horizontal surface**
   Indoor path finding can be useful for multiple modalities, including a walk able path. This means that the path should be constraint to the floor non empty nodes.

   When a fire fighter enters a building where sight is blocked by smoke he always walks along a wall. It could be useful for a fire fighter to know how to navigate through a building while waling along a path. For this

a path should be constraint to nodes over the floor and with a certain distance to a wall.

7. **Movement through borders and centres of interior empty nodes.**
   A path where movement is possible through centre points and boundaries of interior empty nodes, expands the possible path in a building.

8. **Implement the method on really big buildings**
   For now the method is only tested on small buildings. But how does the method react to really big buildings? It could be interesting to research how the method works with large building, e.g congress halls or the Faculty of Architecture and the Built Environment at DUT. For this both the datasets should be available and the possibility of implementing the method should be researched.

9. **Update goal nodes**
   The method in this research assumed the start and goal node are fixed. But with moving targets the location of the goal node should be updated.

7 APPENDICES

7.1 BENCHMARK RESULTS OF OCTREE GENERATION

Benchmark results of Octree generation

Point cloud	Depth	Scale	connectivity	bbox	spatial resolution	non empty nodes	empty nodes	time points	points stdev %	time non empty	non empty stdev %	time empty	interior empty stddv %	time total	total stdev	count
Test	6	1,00	face	64	1	2976	10173	0:00:00,077	2,60	0:00:00,086	2,33	0:02:00,320	0,65	0:02:00,511	0,65	4
Test	6	1,00	face, edge	64	1	2976	10173	0:00:00,075	2,67	0:00:00,086	2,33	0:02:08,041	0,73	0:02:08,228	0,73	4
			face, edge,													
Test	6	1,00	vertex	64	1	2976	10173	0:00:00,075	2,67	0:00:00,085	2,35	0:02:15,290	0,96	0:02:15,476	0,96	4
Test	6	1,00	non	64	1	2976	10173	0:00:00,078	2,56	0:00:00,086	3,49	0:01:58,872	0,18	0:01:59,064	0,18	4
Test	7	2,00	face	64	0,5	2976	18424	0:00:00,080	2,50	0:00:00,120	1,67	0:04:57,747	0,55	0:04:58,067	0,55	4
Test	7	2,00	face, edge	64	0,5	2976	56658	0:00:00,075	1,33	0:00:00,088	1,14	0:10:58,473	0,30	0:10:58,690	0,30	4
			face, edge,													
Test	7	2,00	vertex	64	0,5	2976	56658	0:00:00,076	2,63	0:00:00,089	2,25	0:11:46,162	0,36	0:11:46,376	0,36	4
Test	7	2,00	non	64	0,5	2976	18424	0:00:00,082	4,88	0:00:00,106	14,15	0:04:55,955	0,51	0:04:56,199	0,53	4
Test	8	4,00	face	64	0,25	2976	11901	0:00:00,079	2,53	0:00:00,095	1,05	0:10:44,021	0,35	0:10:44,220	0,35	4
Fire																
department	6	4,62	face	13,86	0,22	27848	28606	0:00:38,683	2,63	0:00:00,997	21,36	0:05:04,161	3,31	0:05:46,794	3,18	4
Fire																
department	6	4,62	face, edge	13,86	0,22	27848	28606	0:00:38,784	2,34	0:00:01,001	21,88	0:09:43,371	5,76	0:10:26,296	5,47	4
Fire			face, edge,													
department	6	4,62	vertex	13,86	0,22	27848	28606	0:00:39,116	1,79	0:00:01,016	28,35	0:12:00,036	6,08	0:12:43,385	5,79	4
Fire																
department	6	4,62	non	13,86	0,22	27848	28606	0:00:38,933	2,56	0:00:01,008	26,88	0:05:10,155	4,30	0:05:50,461	4,61	4
Fire																
department	7	9,24	face	13,86	0,11	92960	121489	0:00:40,886	0,51	0:00:15,000	0,37	1:04:35,894	0,81	1:05:38,803	0,79	4
Fire																
department	7	9,24	face, edge	13,86	0,11	92960	121489	0:00:38,441	3,24	0:00:02,101	2,52	2:11:52,000	8,94	2:19:32,189	1,29	4
Fire			face, edge,													
department	7	9,24	vertex	13,86	0,11	92960	121489	0:00:38,709	3,40	0:00:07,833	102,90	2:44:39,000	7,01	2:52:27,485	0,80	4
Fire																
department	7	9,24	non	13,86	0,11	92960	121489	0:00:40,631	0,28	0:00:14,885	0,64	1:04:16,215	0,62	1:05:17,381	0,61	4
Fire																
department	8	18,48	face	13,86	0,05	186186	309716	0:00:40,040	1,18	0:00:04,000	0,63	6:39:13,980	0,90	6:39:58,140	0,90	4
Bouwpub	6	4,02	face	15,94	0,25	18756	11627	0:00:42,412	2,78	0:00:00,932	39,48	0:01:53,802	3,09	0:02:38,034	4,29	4
Bouwpub	6	4,02	face, edge	15,94	0,25	18756	11627	0:00:42,421	2,76	0:00:00,983	25,53	0:03:01,090	2,76	0:03:47,501	3,24	4
			face, edge,													
Bouwpub	6	4,02	vertex	15,94	0,25	18756	11627	0:00:42,520	2,19	0:00:00,920	29,24	0:03:35,219	3,04	0:04:19,352	3,60	4
Bouwpub	6	4,02	non	15,94	0,25	18756	11627	0:00:42,795	2,23	0:00:01,213	23,17	0:01:54,093	3,73	0:02:40,689	5,35	4
Bouwpub	7	8,03	face	15,94	0,12	74080	63899	0:00:44,946	0,61	0:00:17,162	0,60	0:28:47,611	1,10	0:29:56,306	1,06	4
Bouwpub	7	8,03	face, edge	15,94	0,12	74080	63899	0:00:44,139	4,20	0:00:02,000	3,65	1:05:29,601	11,91	1:06:17,476	2,74	4
			face, edge,													
Bouwpub	7	8,03	vertex	15,94	0,12	74080	63899	0:00:44,559	2,65	0:00:01,972	7,66	1:17:37,639	16,91	1:18:22,991	1,47	4
Bouwpub	7	8,03	non	15,94	0,12	74080	63899	0:00:44,321	0,74	0:00:16,681	1,51	0:28:34,641	1,01	0:29:41,680	0,95	4
Bouwpub	8	16,06	face	15,94	0,06	169332	191679	0:00:44,295	2,17	0:00:03,889	193,26	3:46:59,024	5,16	3:47:46,408	5,08	4

7.2 BENCHMARK RESULTS OF PATH FINDING OPERATIONS

Benchmark results of Pathfinding

Octree name	Depth	distance type	connectivity	clearance	Distance to floor	avg_length	neighbours time	stdev neighbours %	path finding time	stdev path finding time %	Total time	iterations	Cumputed
Test	7	Euclidean	face	0	0	46,8602	0:00:02,089	12,97	0:00:00,235	0,43	0:00:02,324	3636	50
Test	7	Euclidean	faceEdge	0	0	43,9274	0:00:02,661	9,28	0:00:00,225	0,89	0:00:02,886	2507	50
Test	7	Euclidean	faceEdgeVertex	0	0	42,3406	0:00:03,048	6,76	0:00:00,217	0,46	0:00:03,265	1750	50
Test	7	Manhattan	none	0	0	54,4238	0:00:00,483	3,52	0:00:06,178	4,86	0:00:06,661	1319	50
Test	7	Euclidean	none	0	0	46,86	0:00:00,482	3,53	0:00:30,542	4,05	0:00:31,024	3632	50
Test	7	chessboard	none	0	0	47,9404	0:00:00,483	2,48	0:00:41,167	3,43	0:00:41,650	4632	50
Fire department	6	Euclidean	face	0,17	1,2	17,596	0:00:00,315	11,43	0:00:00,211	0,47	0:00:00,526	1285	5
Fire department	7	Euclidean	face	0,17	1,2	12,828	0:00:01,863	1,88	0:00:00,230	0,43	0:00:02,093	3372	5
Fire department	7	Euclidean	faceEdge	0,17	1,2	11,62	0:00:03,161	2,31	0:00:00,225	0,44	0:00:03,386	2530	5
Fire department	7	Euclidean	faceEdgeVertex	0,17	1,2	11,512	0:00:04,056	4,02	0:00:00,231	0,87	0:00:04,287	2884	5
Fire department	7	chessboard	none	0,17	1,2	14,934	0:00:00,128	4,69	0:00:10,185	2,83	0:00:10,313	2902	5
Fire department	7	Euclidean	none	0,17	1,2	14,766	0:00:00,124	4,03	0:00:09,234	6,55	0:00:09,358	2711	5
Fire department	7	Manhattan	none	0,17	1,2	15,418	0:00:00,128	4,69	0:00:04,809	7,67	0:00:04,937	1581	5
Fire department	8	Euclidean	face	0,17	1,2	14,522	0:00:04,033	2,65	0:00:00,231	0,43	0:00:04,264	3379	5
Bouwpub	6	Euclidean	face	0,17	1,2	14,642	0:00:00,213	14,55	0:00:00,206	0,49	0:00:00,419	564	5
Bouwpub	7	Euclidean	face	0,17	1,2	13,812	0:00:00,729	6,86	0:00:00,209	0,48	0:00:00,938	1135	5
Bouwpub	7	Euclidean	faceEdge	0,17	1,2	13,136	0:00:01,433	4,47	0:00:00,212	0,47	0:00:01,645	1300	5
Bouwpub	7	Euclidean	faceEdgeVertex	0,17	1,2	12,91	0:00:01,643	4,63	0:00:00,212	0,47	0:00:01,855	1215	5
Bouwpub	7	chessboard	none	0,17	1,2	18,946	0:00:00,048	6,25	0:00:02,517	2,50	0:00:02,565	1301	5
Bouwpub	7	Euclidean	none	0,17	1,2	18,868	0:00:00,048	4,17	0:00:02,495	3,29	0:00:02,543	1312	5
Bouwpub	7	Manhattan	none	0,17	1,2	19,53	0:00:00,048	4,17	0:00:02,190	2,56	0:00:02,238	1166	5
Bouwpub	8	Euclidean	face	0,17	1,2	13,778	0:00:02,939	5,24	0:00:00,215	0,47	0:00:03,154	1712	5

8
REFLECTION P5

This presents a method for collision free indoor 3D path finding in an octree. Benchmark tests were performed to identify the effect of several component on the path length and computational time. The results show that by computation the time and path length can be reduced by pre processing neighbours, using a different distance type and path connectivity.

This research intersect with most of the core courses of the masters Geometics program. Especially knowledge learned in courses such as Python programming, Geo-dbms and sensing technologies is applied.

The scientific contribution of this research is twofold. Firstly an indoor path finding application through the empty space of a point cloud is developed. This application uses an octree data structure as a kind of catalyst for path finding.

The method consist of the following components: a work flow to pre process a point cloud for optimal classification into an octree structure. A novel method to identify interior empty nodes and at the same time the downward distance from each interior empty node to the closest non empty node is computed. A new method for pre-processing a collision avoidance system. A method to process a network graph during octree generation. And finally a path finding application, which uses the network graph and collision avoidance system.

The second contribution aims to identify the effects of octree operators and A* operations on A* path finding. It will research the impact on the computation time and path length. To my best knowledge no research has been conducted identifying the effects of these components in A* path finding through an octree data representation of an indoor point cloud.

Recently there is a growing interest and demand of 3D indoor path finding applications. Examples of this are:

In emergency situation an indoor path finding method can be used to direct persons to the closest emergency exit. In an hospital it can be used to direct patients to the room of their doctors appointment. In big buildings like airports or congress buildings indoor path finding can be used to navigate people to certain locations or compute a closest meeting point.

With an indoor path finding method outdoor and indoor navigation could be seemingly merged into one system. Think of situations where someone enters an unknown city via the train station and needs to go to a certain room in a building. Nowadays the navigation stops at the door step of the building. Especially in large buildings it might be useful to extend this navigation to the room.

BIBLIOGRAPHY

Broersen, T., Fichtner, F. W., Heeres, E. J., de Liefde, I., Rodenberg, O. B. P. M., Verbree, E., and Voûte, R. (2016). Project pointless. identifying, visualising and pathfinding through empty space in interior point clouds using an octree approach. In *AGILE 2016; 19th AGILE Conference on Geographic Information Science, 14-17 June, 2016; Authors version*.

Gargantini, I. (1982). Linear octtrees for fast processing of three-dimensional objects. *Computer graphics and Image processing*, 20(4):365–374.

Guang-lei, Z. and He-Ming, J. (2012). Global path planning of auv based on improved ant colony optimization algorithm. In *Automation and Logistics (ICAL), 2012 IEEE International Conference on*, pages 606–610. IEEE.

Hamada, K. and Hori, Y. (1996). Octree-based approach to real-time collision-free path planning for robot manipulator. In *Advanced Motion Control, 1996. AMC'96-MIE. Proceedings., 1996 4th International Workshop on*, volume 2, pages 705–710. IEEE.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107.

Herman, M. (1986). Fast, three-dimensional, collision-free motion planning. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, volume 3, pages 1056–1063. IEEE.

Hou, E. S. and Zheng, D. (1994). Mobile robot path planning based on hierarchical hexagonal decomposition and artificial potential fields. *Journal of Robotic Systems*, 11(7):605–614.

Hwang, Y. K. and Ahuja, N. (1992). A potential field approach to path planning. *Robotics and Automation, IEEE Transactions on*, 8(1):23–32.

Isikdag, U., Zlatanova, S., and Underwood, J. (2013). A bim-oriented model for supporting indoor navigation requirements. *Computers, Environment and Urban Systems*, 41:112–123.

Jung, D. and Gupta, K. K. (1996). Octree-based hierarchical distance maps for collision detection. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 454–459. IEEE.

Kambhampati, S. and Davis, L. S. (1985). Multiresolution path planning for mobile robots. Technical report, DTIC Document.

Kim, J. and Lee, S. (2009). Fast neighbor cells finding method for multiple octree representation. In *2009 IEEE International Symposium on Computational Intelligence in Robotics and Automation - (CIRA)*.

Lemmens, T. (2015). Geo-information from imagery.

Liu, L. and Zlatanova, S. (2011). A" door-to-door" path-finding approach for indoor navigation. In *Proceedings Gi4DM 2011: GeoInformation for Disaster Management, Antalya, Turkey, 3-8 May 2011*. International Society for Photogrammetry and Remote Sensing (ISPRS).

Major, F., Malenfant, J., and Stewart, N. F. (1989). Distance between objects represented by octtrees defined in different coordinate systems. *Computers & graphics*, 13(4):497–503.

Namdari, M. H., Hejazi, S. R., and Palhang, M. (2015). Mcpn, octree neighbor finding during tree model construction using parental neighboring rule. *3D Research*, 6(3):1–15.

Nosrati, M., Karimi, R., and Hasanvand, H. A. (2012). Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, 2(4):251–256.

Noto, M. and Sato, H. (2000). A method for the shortest path search by extended dijkstra algorithm. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2316–2320. IEEE.

Payeur, P. (2006). A computational technique for free space localization in 3-d multiresolution probabilistic environment models. *Instrumentation and Measurement, IEEE Transactions on*, 55(5):1734–1746.

Samet, H. (1982a). Distance transform for images represented by quadtrees. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (3):298–303.

Samet, H. (1982b). Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57.

Samet, H. (1988). An overview of quadtrees, octrees, and related hierarchical data structures. In *Theoretical Foundations of Computer Graphics and CAD*, pages 51–68. Springer.

Samet, H. (1989). Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, 46(3):367–386.

Van Oosterom, P. and Vijlbrief, T. (1996). The spatial location code. In *Proceedings of the 7th international symposium on spatial data handling, Delft, The Netherlands*.

Verbree, E. and Van Oosterom, P. (2014). Explorative point clouds maps for immediate use and analysis.

Vörös, J. (1997). A strategy for repetitive neighbor finding in images represented by quadtrees. *Pattern Recognition Letters*, 18(10):955–962.

Vörös, J. (2000). A strategy for repetitive neighbor finding in octree representations. *Image and Vision Computing*, 18(14):1085–1091.

Vörös, J. (2001). Low-cost implementation of distance maps for path planning using matrix quadtrees and octrees. *Robotics and Computer-Integrated Manufacturing*, 17(6):447–459.

Wang, M. and Tseng, Y. (2011). Incremental segmentation of lidar point clouds with an octree-structured voxel space. *The Photogrammetric Record*, 26(133):32–57.

Wu, L. and Hori, Y. (2006). Real-time collision-free path planning for robot manipulator based on octree model. In *Advanced Motion Control, 2006. 9th IEEE International Workshop on*, pages 284–288. IEEE.

Xu, S., Honegger, D., Pollefeys, M., and Heng, L. (2015). Real-time 3d navigation for autonomous vision-guided mavs. IROS.

Xu, X. (2012). Pathfinding.js.

Zhou, K., Gong, M., Huang, X., and Guo, B. (2011). Data-parallel octrees for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on*, 17(5):669–681.

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 André Miede.