RAM ROBOTICS AND MECHATRONICS

Development of a Three Dimensional Path Planner for Aerial Robotic Workers

A. (Anastasios) Zompas

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
Dr.ir. J.F. Broenink
Dr.ir. M. Fumagalli
H.W. Wopereis, MSc
Dr.ir. R.G.K.M. Aarts

November 2016

050RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Abstract

With the development of industry the need of a safe and low cost method for maintenance and repair raised. A proposal to this need are the Aerial Robotic Workers (ARWs). These workers have to operate autonomously and conduct tasks in a variety of locations. To achieve this autonomous behavior, it is required to generate collision free paths between the position of the ARWs and the location where they have to operate in runtime.

Within this report a solution is documented which allows the robotic workers to generate three dimensional paths on runtime avoiding obstacle collision. The proposed method is constructed by a visibility graph and the A* path planner. The visibility graph is based on the octomap representation from which the nodes which construct the graph are generated. Finally, tools, such as the unordered map structure, were integrated in to the solution to lower the processing time. It was shown that with the use of this method three dimensional path generation is feasible in runtime.

Contents

1 Introduction

With the development of countries, industry has expanded in high rates over the last years. Maintenance of big scale constructions are costly and dangerous. Thus, projects develop an automated solution which will not only be more economical efficient but also safer and more precise. One of these projects is AEROWORKS.

The AEROWORKS project focuses at the development of a team of Aerial Robotic Workers (ARWs). The ARWS are flying multirotor robots equipped with manipulators, capable to perform cooperative maintenance tasks in industrial environments. To move around autonomously in this kind of environment, the ARWs require to feasible paths to follow.

The purpose of this thesis, as part of this project, is to develop, implement and test algorithms capable of autonomously navigate our ARWs in any desired three dimensional space. The main task here is for the flying robots to be able to find an optimal path, with given constraints, avoiding obstacle collision.

Industrial environments often contain dynamic features. Thus, it is not always possible to know the exact map a priori. Autonomous navigation in a (partially) unknown environment in runtime is a challenging aspect. In three-dimensional space path planning is computationally inefficient due to the increased dimensionality. In addition, it is also memory inefficient due to possibly large spaces in which the ARWs have to operate. Moreover, the planner has to run in parallel with a mapping process when the ARWs fly in such environments.

There has been extensive research in this field. Several solutions have been proposed to solve the shortest path problem. These solutions use different approaches. A part of the existing literature uses different map representations such as octomap, probabilistic roadmap, potential field and navigation mesh [1] [2] [3] [4] [5]. Also, different path planning algorithms have been developed which couple with one or more of the map representations mentioned. However, specific algorithms have been developed to give a specific solution to this problem [6] [7] [8]. Finally, there are general algorithms which perform well in higher dimensions such as rapidly-exploring random trees (RRTs) [9]. However, we decided to use the A* algorithm because it has parameters which can be tuned for an optimal resulted path given constraints. Furthermore, this algorithm can be easily patched with additional algorithms which provide specific behavior to the final path (less turns, shortest path, specific facing direction when reaching the goal etc). Moreover, we propose a combination of method which which lower the processing time. Thus the generation of the final path is feasible in runtime. Our solution results to a general solution and not to a solution targeted to a specific scenario.

Taking into consideration the problem mentioned above, In his thesis, the problem was approached from a map representation perspective. This approach regards the comparison of possible map representations from which the more efficient combination was chosen. Thus, the octomap structure was selected, from which a visibility graph was constructed. With the use of this graph it is possible for the ARWs to fly longer distances by taking into consideration less way points. The map representation technique was combined with the A* path planner[1] algorithm. Visibility graph was selected as it is more memory and computationally efficient than a normal grid. Finally, several methods have been introduced in order to increase the runtime performance to find the path.

[1] Path planner is an essential part of autonomous robots which provides optimal paths between two points, given constraints (e.g. shortest path, faster search, less turns etc)

The remainder of this document is divided in the following parts.

Background Documents the background knowledge needed for this project. Namely, the A* path planner, the use of the heuristic and the min heap structure.

Map Representation Presents the different map representations which were considered. Furthermore, techniques which coupled with each map representation are explained. These techniques result in a better optimized solution with respect to time .

Implementation and Results Contains implementation notes, results and evaluation of the compared methods.

Conclusion, Discussion and Future Work Concludes this report and suggests future work such that the proposed technique to improve the robustness in (partially) unknown environments.

2 Background

2.1 Path Planning Algorithm

There are a lot of algorithms which provide optimal paths under specific constrains given a map. However, an algorithm cannot understand a real life environment containing doors and walls. Thus, the environment has to be translated into a graph which contains traversable points, the nodes, and the connections between them, the edges. There are several kinds of graphs which can be used, of which later in this document, some of them will be analyzed.

Given a graph as an input, the path planning algorithm will try to find an optimal path between a starting node and a goal node. Several algorithms have been developed, but we decided to use the A* algorithm which is the evolution of two other algorithms, the Breadth First Search [10] and the Dijkstra's Algorithm.

Breadth First Search uses a method called *frontier* in order to search for a path. Frontier, works like an expandable ring around the starting point. It expands by keeping track of which neighbors have been visited and then it finds the neighbors which will be visited later. Thus, it explores equally in all directions without prioritizing lower movement cost (figure 2.1 (a)). Dijkstra's algorithm, uses the moving cost to search for a path. The moving cost is the cost we have to "pay" to go from one node to another. In a graph this cost can vary due to many reasons such as the distance between two nodes and the cost assigned to traversing specific kind of terrain (water, mountain etc). Using the moving costs, Dijkstra's algorithm prioritize the search exploring for paths with lower total cost (figure 2.1 (b)). Although, Dijkstra's algorithm guarantees to find the shortest path, it also searches in areas which are not promising. To solve this problem, the A* algorithm also implements a heuristic search on Dijkstra's algorithm. The use of the heuristic helps the A* to focus the exploration to the nodes towards to the goal faster (figure 2.1 (c)). Thus, it finds the same optimal path as Dijksta's algorithm does, but it will explore less nodes.

(a) **(b)** **(c)**

Figure 2.1: The search behavior of: (a) Breadth First Search, (b) Dijkstra's Algorithm, (c) A* Algorithm (http://www.redblobgames.com/pathfinding/a-star/introduction.html)

2.1.1 A* Path Planner

The A* path planner was selected because it is an optimal searching algorithm, can handle varying cost terrains easily and it is easy to implement. The later, does not seem to be a good argument. However, an easy to implement algorithm allow to invest more time to an other very important aspect of the path finding solution, the map representation. Map design really affects the running time of the algorithm as we will show later in the document. To continue with, the A* algorithm contains tunable variables which result to an optimal path when constraints are given. Finally, the path planner algorithm can be easily patched with with aditional algo-

rithms to adapt specific behavior to the final path (less turns, desired moving direction around an obstacle, specific facing direction when reaching the goal etc).

Let's explain now briefly how the algorithm works, with an example on a square grid for simplicity. A* is a heuristic based path finding algorithm. This means that it uses a heuristic function to conduct a more guided search towards the goal point. This method results in a limited searching area, reducing the calculation time. To continue with, for the construction of the path planner it is necessary to compute the movement cost from the current square to the neighbor square (g(n)) and to estimate the movement cost from the neighbor square till the goal node using the heuristic function (h(n)). Finally, two lists are required. One open list which contains all the possible nodes to search and a closed list which contains all parent-child nodes connections. Thus, it will eventually contain the desired path once the algorithm is finished.

To begin with, the algorithm starts by inserting the starting node to the closed list. Then, it calculates the $F(n) = g(n) + h(n)$ value for all its neighbors, ignoring any node which contains obstacles, and it inserts them to the open list. After that, the algorithm expands the node with the lowest $F(n)$ value and inserts it to the closed list.

To continue with, the process explained above is repeated by choosing the point with lowest F value from the open list and insert it to the closed list. Then, the algorithm performs the following checks for each one of its acceptable neighbors (non-obstacle etc). If the block is not already in the open list, or it exists with a higher g(n) value, then the path planner assigns this block to the new parent node and recalculate the node's F value. Else, if the node already exists in the open list with a lower g(n) value, then it does nothing. This process is repeated till the goal point is included in the closed list. Finally, the path can be constructed by following back the path from the assigned points in the closed list till the starting point.

To compute the g(n) value for the next selected node, we add the g(n) value of the parent (the point from which we want to calculate the next step) to the moving costs for the selected node. In general, the movement cost can vary in a graph with respect to time or the different kind of terrain (water, forest, mountain etc). In our example, for simplicity, for the square grid will be used a unit cost (1) movement for each neighbor conjoint at edge and a square root of two ($\sqrt{2}$) cost for each neighbor conjoint at the corner.

Once the g(n) values for the neighbor nodes have been computed, the h(n) values have to be computed also. These values can be computed by several ways regarding the selected formula for distance computation. For this example, the diagonal distance will be used which is given by the formula:

$$d = D \cdot (\Delta x + \Delta y) + (D2 - 2 \cdot D) \cdot min(\Delta x, \Delta y) \tag{2.1}$$

Where, Δx and Δy are the absolute distances between the centers of the squares on the x and y axis respectively given by:

$$\begin{aligned} \Delta x &= abs(x_{neighbor} - x_{goal}) \\ \Delta y &= abs(y_{neighbor} - y_{goal}) \end{aligned} \tag{2.2}$$

Also D and D2 represent the moving costs for the neighbors at the edge and at the corner respectively and they are given by:

$$D = 1,\ D2 = \sqrt{2} \tag{2.3}$$

In the case where the goal point is not reachable, the process is followed till the open list is empty. In this case there is no path, but if it is desired it is possible to follow the path till the closest point to the goal (lowest h(n) value).

A pseudocode for the implentation of the A* path planner can be found in Appendix A.

Note that, if the map is big, the open list can contain a huge amount of nodes. Thus, it is crucial to find an efficient way to sort it and access it without having to go through the whole list to find the element we want (node with the lowest F value) or to sort the list inefficiently. The solution to this problem is the implementation of a heap structure, which is more efficient than normal sorting. The heap sorting process is analyzed later in this document.

2.2 Heuristic

The heuristic is the essential part of the A* algorithm, and it is analyzed in this section. The heuristic (h(n)) is a function that estimates the distance between a query point and the goal. This function, should be chosen wisely in order to have the desired result.

To choose the heuristic it is essential to bare in mind several rules which control the behavior of the algorithm [11]. First, if a heuristic is not considered at all, $h(n) = 0, \forall n$, then the only part which play a role in the path finding search is the g(n). In this occasion, the A* algorithm reduces to Dijkstra algorithm which guarantee to find the optimal path. On the other hand, if g(n) can be neglected with respect to h(n), then the A* algorithm transforms into Greedy Best-First-Search [12]. Next, if the heuristic function is consisted, meaning that it is lower than or equal to the real distance between a vertex and the goal, then it is guaranteed that the A* will find the the shortest path. On the other hand, this means that the algorithm has to expand more vertexes leading to a slower search. However, if h(n) is exactly equal to the real distance, then the algorithm will follow the best optimized path and will not search any other vertex. Finally, If the heuristic is grater than the real distance, then the path planner will not necessarily find the shortest path but it will search faster by expanding less vertexes.

As could be understood from above, that the heuristic can be chosen and modified to select a desired behavior between time efficiency and shortest path. For instance, in a three dimensional map, sometimes it might be better to have a good and fast search than an excellent and slow one. Furthermore, this varying behavior does not have to be global but it can change in several areas of the map. For instance, in specific scenario, it might be more efficient to search for a path that goes around an obstacle than one that goes above it or vice versa.

Regarding the types of the heuristic function, they differ with respect to the terrain (map representation). Several heuristic functions can be used with more common ones the Manhattan distance, the Diagonal distance and the euclidean distance. For instance in a square grid which allows four-directional move $(+x, -x, +y, -y)$ the best option is the Manhattan distance given by:

$$D \cdot (\Delta x + \Delta y), \tag{2.4}$$

where D is the minimum cost to travel from a square to its neighbor, $\Delta x = abs(x_{node} - x_{goal})$ and $\Delta y = abs(y_{node} - y_{goal})$. By increasing or decreasing the D value, we can switch between speed and accuracy of the path planner, as explained earlier.

The diagonal distance is used in a square grid that allows eight-directional move and it is represented by:

$$D \cdot (\Delta x + \Delta y) + (D2 - 2 \cdot D) \cdot min(\Delta x, \Delta y), \tag{2.5}$$

where $D2 = D \cdot \sqrt{2}$.

Finally, euclidean distance is usually used when the map representation allows movement at any angle and it is given by:

$$D \cdot \sqrt{\Delta x^2 + \Delta y^2}. \tag{2.6}$$

However, if this heuristic is used in a square or cube grid, it will make the path planner run slower as the decrease of the heuristic will not match the increase of the movement cost value g(n).

Another important aspect of the heuristics is the tie braking. In a map there might be several optimal paths with the same length. The A* path planner will explore all of them since they have the same $F(n) = g(n) + h(n)$ value. A common way to overcome this problem, is to overestimate the heuristic value by a really small amount. This action, will increase the $F(n)$ value which will make the A* algorithm to expand vertexes closer to the goal faster. This overestimate can be only 0.1% and it will result to a way less exploration area on the map.

2.3 Min Heap

A heap is a tree based data structure which can be used for shorting lists. In this structure each node has a maximum of two child nodes. Furthermore, each parent node must contain an element, in our case the $F(n)$ value, smaller than both of each child nodes.

To add an element in the heap tree, primarily, it will be added to the last available position. Then, a check is performed whether it is smaller than its parent or not. If the last argument is true, the nodes swap their content. This process continues till the argument is no longer true.

To remove an element from the heap, a similar procedure is followed. The desired element is removed, in our case will always be the first one as it contains the smallest $F(n)$ value. Then, this position is filled with the last element of our tree. Finally, a check is performed regarding the property of the smaller parent. If the parent node holds a higher value than one or both child nodes, the content is switched with the child node which contains the smallest value. This property continues till the property of the smaller parent is restored.

On the implementation point of view, the heap structure is implemented with an array. Thus, the children of node n exist in the nodes $2n+1$ and $2n+2$. The parent of node n exists in the node $^{(n-1)}/_{2}$. As can be deduced, if the child is in an even node, then the parent node is not an integer. Thus, only the quotient part of the division is taken into consideration. For instance, the parent of node 12 exists in node 5.

As explained, a heap is a more efficient way of sorting an array as way less elements have to be accessed.

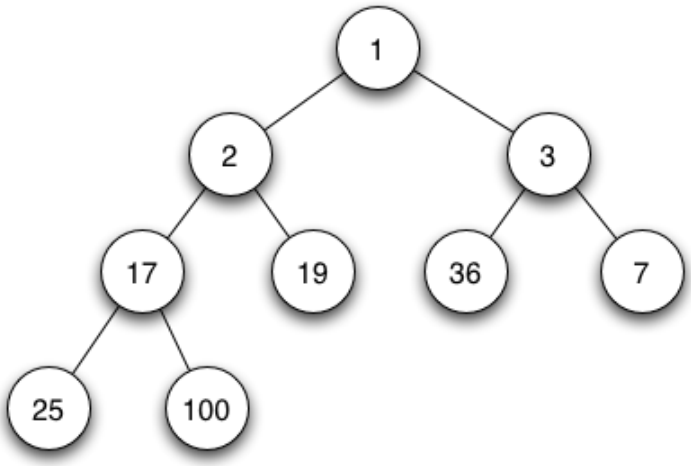


Figure 2.2: Min heap structure. (https://en.wikipedia.org/wiki/Binary_heap)

3 Map Representation

As it has already been mentioned, the map representation is one of the most important aspect in path planning. Especially in higher dimensions, map representation can make a noticeable difference in processing time of the path planner. To demonstrate this difference three different map representations were considered in three dimensional space.

The A* path planner was combined with several map representations in order to select the best method. To understand how the algorithm works and how the heuristic affects the searching a square grid representation was used. After that, a cube grid was considered for a three dimensional representation. To continue with, an octomap representation was used in order to make our map more memory efficient than the cube grid. Also, it was considered in order to make our path planner more computational efficient. Finally, a visibility graph was constructed out of the octomap and combined with the path planning technique.

3.1 Square Grid

As mentioned above, the A* path planner was first combined with a square grid representation. This decision was taken in order to gain a better understanding of the path planning algorithm and also to check how the variation of several parameters affects the algorithm.

Several maps environments were considered (random maps, maze like, etc). For this configuration we used under estimated heuristic by a factor of 0.9999 and the heuristic

$$dx + dy + (1.4 - 2) \cdot min(dx, dy), \tag{3.1}$$

where dx and dy are the difference from a query point to the goal on x and y direction respectively.

Furthermore, an eight-neighbor configuration was considered with travel cost of 1 for the edge neighbors and $\sqrt{2}$ (or roughly 1.4 for simplicity and faster calculations)[1] for the corner neighbors.

3.2 Cube Grid

The square grid had to be expanded to cube grid for real world representation. Furthermore, the algorithm was expanded to be combined with a cube grid. Thus, three dimensional path planning was possible. Moreover, several maps were considered also for this map representation.

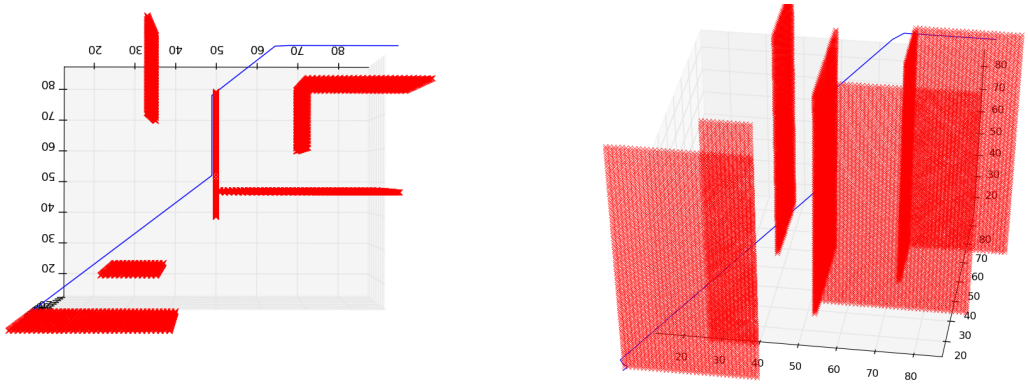
For this map representation, the twenty six possible neighbors configuration was considered for each query node. Face neighbors, have moving cost of 1, edge neighbors have moving cost of $\sqrt{2}$ (or roughly 1.4 for simplicity and faster calculations) and corner neighbors have moving cost of $\sqrt{3}$ (or roughly 1.7 for simplicity and faster calculations)[1]. For the cube grid the following heuristic was used:

$$\begin{aligned} 1.7 \cdot min(dx, dy, dz) + 1.4 \cdot (med(dx, dy, dz) - \ldots \\ min(dx, dy, dz)) + max(dx, dy, dz) - med(dx, dy, dz), \end{aligned} \tag{3.2}$$

where dx, dy and dz are the given differences from a query point to the goal on x, y and z axis respectively.

[1] Note that this is an underestimation and so the heuristic remains consistent.

The results are depicted in the figures below.



(a) **(b)**

Figure 3.1: Different view of three dimensional path planner combined with cube grid. obstacles are marked with red color and the final path with blue. The dimensions of the cube grid are 100x100x100. The starting node is (5,5,5) and goal node is (90,90,90)

3.3 Octomap

The cube grid has a draw back. It contains an excessive amount of nodes in big spaces. To make our map more computationally and memory efficient, and also to allow faster traversing of free space, an octomap structure was considered. This structure combines smaller voxels[2] together.

An octomap is a mapping technique for three-dimensional space. It uses a point cloud method which later is divided to octree data structure for spatial representation [13].

Octree is a tree structure where each node is a cube which can be divided in eight children, the octants (Figure 3.2). Each child, can be either a node or a leaf, meaning that it can either have eight children or it contains a value which indicates that the octant is free or occupied respectively. This procedure goes on till they reach the lowest resolution level.

In our setting the Octomap ROS package is used. This package uses a probabilistic approach in order to determine the occupancy level of each node/child. Thus each node/leaf contains the occupancy data and a pointer to the children if any exist.

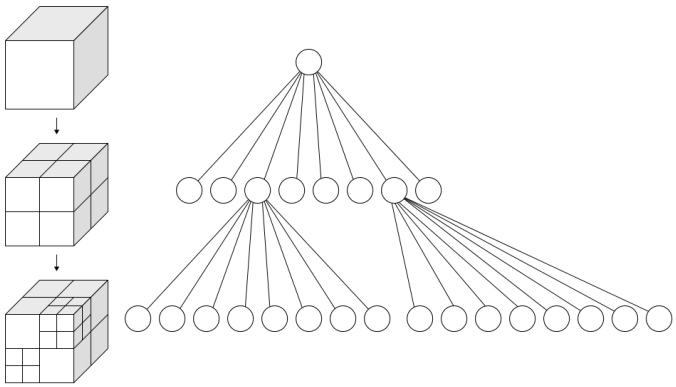


Figure 3.2: Left: Recursive subdivision of a cube into octants. Right: The corresponding octree. (https://en.wikipedia.org/wiki/Octree)

[2]A voxel is the three-dimensional representation of pixel. In other words it is a cube in space.

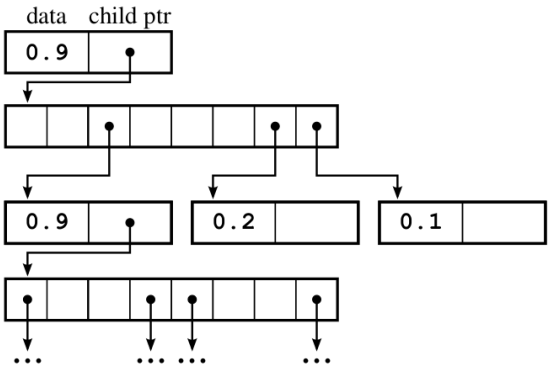


Figure 3.3: Example of octree structure in memory connected with pointers. Data is stored as one float denoting occupancy [13].

Moreover, every node/leaf is assigned a unique digital key which represents its coordinates and gives us information about the nodes size. Two keys which differ by a unit in one direction, means that they might be neighbors of highest resolution (lowest voxel size). However, it is unknown if the neighbor exists and so a check has to be performed to test whether the new key corresponds to a leaf or not.

Taking the above into consideration, since this approach reduces the number of nodes, it is easy to understand that the octomap structure is both more memory than a cube grid. Also, it is more computational efficient since the path planner will have to expand less nodes. However, a new problem rises regarding finding the neighbor of the octree voxels. A normal grid is a linear structure. This means that the coordinates of the neighbors can be found if the coordinates of the query point are increased or decreased by one unit. This does not apply to octomap as the exact size, and thus the coordinates, of the neighbor voxels are unknown.

3.3.1 Neighbor Finding

An easy way to overcome the problem is to iterate through the whole tree. While this procedure, a naive approach is used to search for the neighbors. This combination allow to get the coordinates and size of each point and finds the distance in between the query point and the possible neighbor point on each axis (x, y and z). All distances have to meet a criteria to be considered neighbors. This criteria is that the computed distance (d) has to be less or equal to the sum of the half size of each voxel in each direction.

$$d_i \leq \frac{s_{qp} + s_{pn}}{2},\ i = x,\ y,\ z, \tag{3.3}$$

where s_{qp} is the size of the query point voxel and s_{pn} is the size of the possible neighbor voxel.

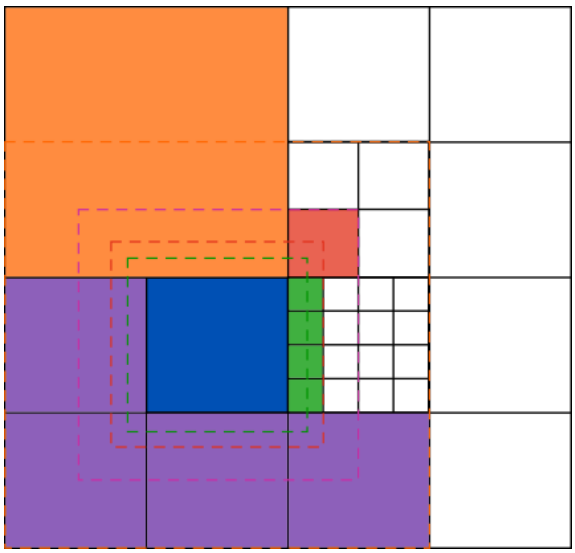


Figure 3.4: Example of naive neighbor finding search in two dimensional octree structure (quadtree) for better visual representation. The blue square is the one for which we want to search the neighbors. The centers of the neighbor squares must be within the dashed lines of the respective color.

This procedure is not efficient because it has to iterate through the whole tree leading to many unnecessary calculations. Thus, an improved solution was implemented to decrease the amount of voxels that need to be traversed.

The solution concerns the creation of a small bounding box around our query point with adaptive dimensionality, using its unique key and size and also the size of the biggest unoccupied voxel. Then, the iteration occurs only throughout the leafs in this bounding box in order to find the neighbors through their distance. This method speeds up the process, but it is inefficient to repeatedly search for the neighbors of the same query point. Thus, another technique which can reduce the processing time is to construct lookup tables. These tables store the points and their neighbors. The lookup tables are updated constantly, every time a voxel changes its occupancy.

This process in the future can run in parallel with the path planner, and possibly in a separate computer with parallel programming, in order not to delay the path planner. Thus, since the neighbors will be found before the search, the computation for the shortest path will be more time efficient.

Finally, after the neighbors are found, the A* path planner algorithm is performed and the results are illustrated in figures 3.5 and 3.6.

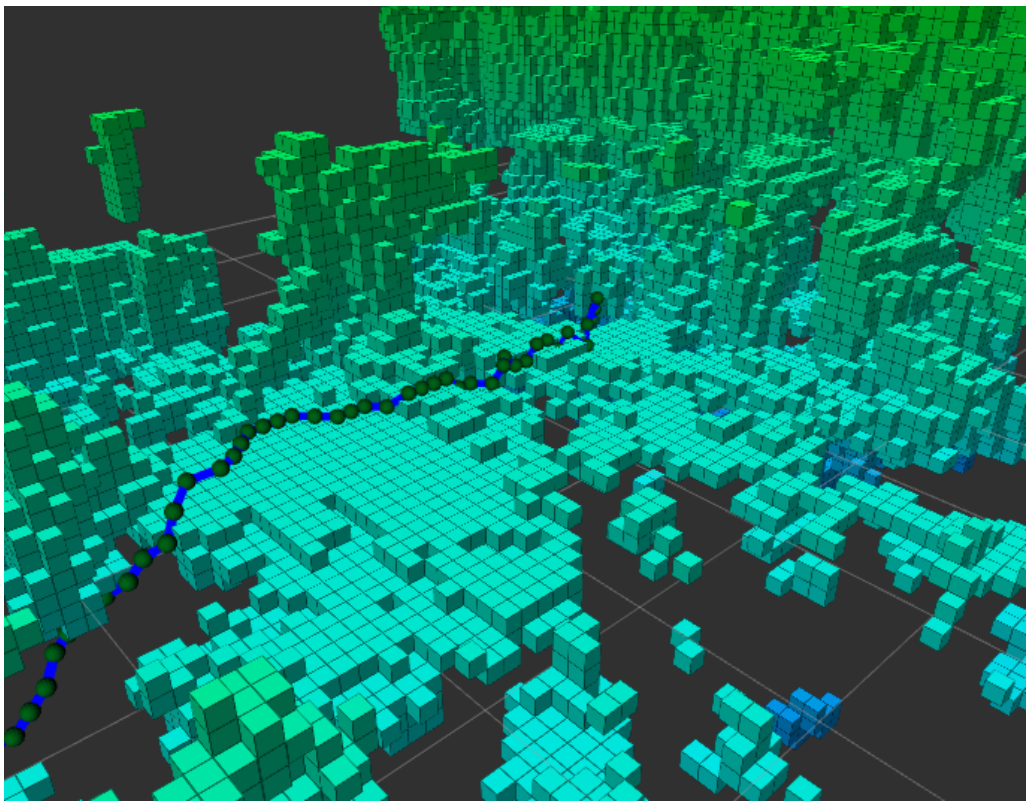


Figure 3.5: Generated path using octomap and A* path planner visualized in rviz. The green spheres are the nodes who construct the final path an the blue lines are the exported path from the A* algorithm.

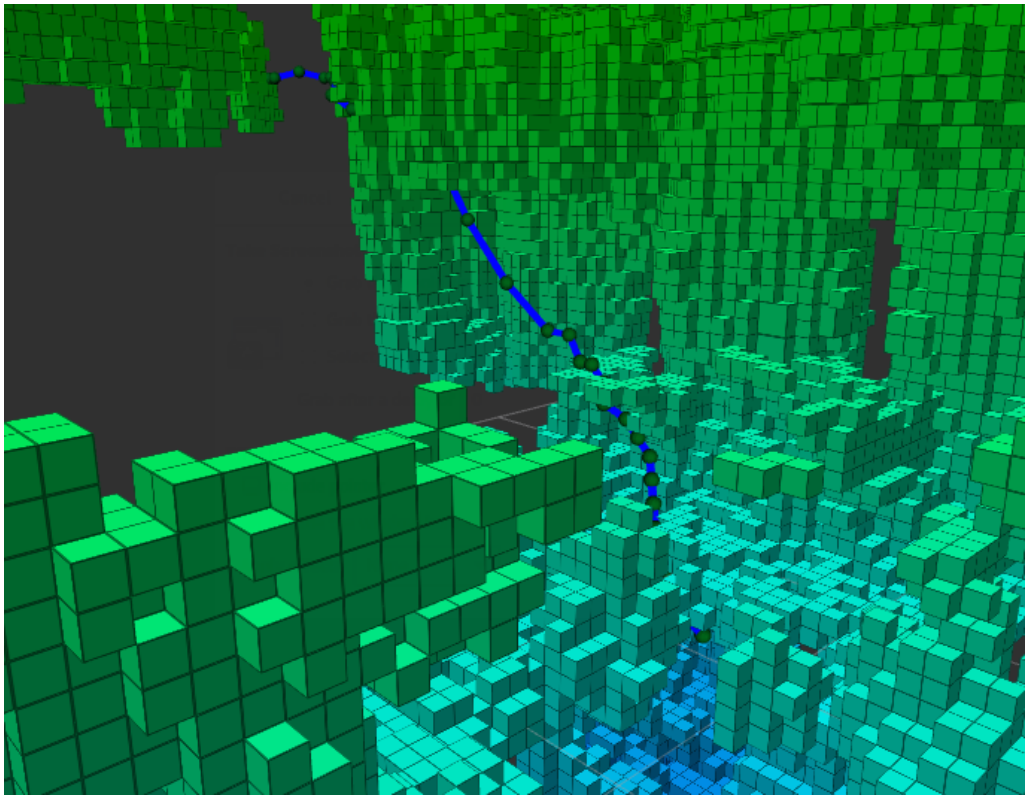


Figure 3.6: Generated path using octomap and A* path planner visualised in rviz. The green spheres are the nodes who construct the final path an the blue lines are the exported path from the A* algorithm.

However, as discussed earlier, map representation is a more important component in path planning techniques than efficient neighbor finding techniques.

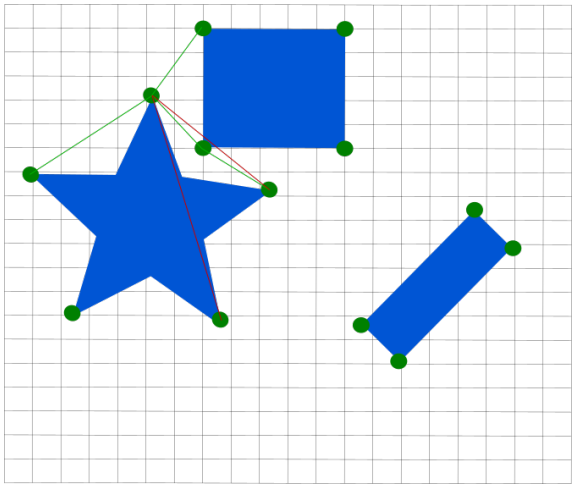
The ARWs have to fly in relative big, and mostly, free spaces (windmill field etc). Thus, a visibility graph could came in handy as each query point is not connected only to each neighbors but also to any other point which is in its visible area. This means that the computations conducted by the A* algorithm, can be reduced and so the total running time of the algorithm. The computational time is reduced due to the lower number of expandable nodes.

3.4 Three Dimensional Visibility Graph

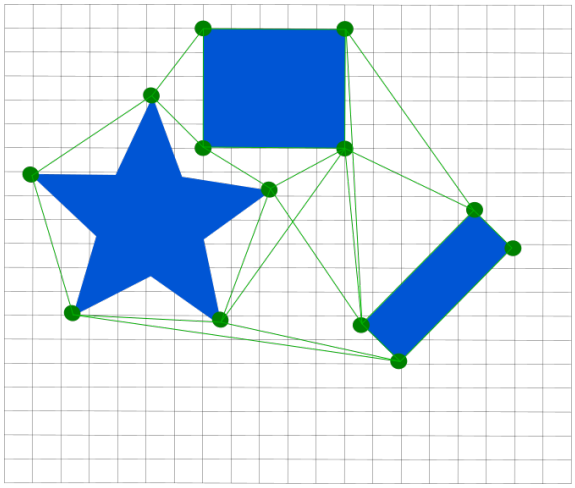
Industrial environments have big free spaces between obstacles where the aerial workers can move. Thus, instead of calculating a path from one voxel to the next one, the idea is to travel from one obstacle to the next one, for instance from one windmill to the next.

Visibility graphs have a wide range of applications such as radio antennas placement and architecture. Path finding is one of these. Visibility is defined as the ability of two points to be connected directly with a straight line without any obstacle being between them.

This kind of graph is very efficient for big map traversing. Instead of considering connected neighbors, just a few points in the map, which will constitute the nodes to be considered by the path planner, can be considered, for instance around the obstacles. An example is illustrated in figure 3.7.



(a)



(b)

Figure 3.7: (a) From each point all other visible points (non-obstructed) are found (green lines represent acceptable paths and red lines represent non acceptable), (b) an example of visibility graph.

The octomap representation was used as base to construct the visibility graph. The reason for this decision is because less points have to be considered around obstacles than a cube grid as is illustrated in figure 3.8. This allows to reduce processing time for the graph construction. This procedure is fast and can be operated on the runtime of the ARWs. Furthermore, the path planner will have way less point to process resulting to a more efficient search. This is expected to make the algorithm more applicable in dynamic environments.

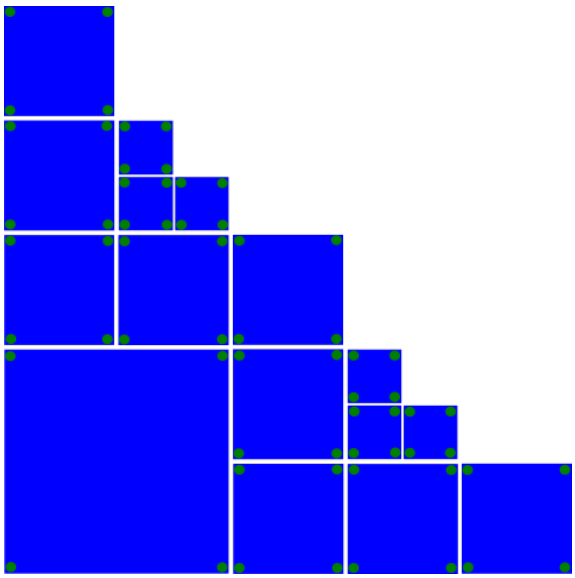


Figure 3.8: Two dimensional representation of octomap base for visibility graph construction. The green circles are the point who will be used to generate the graph. The spaces between the squares are used for better visualization.

To select the coordinates of the desired points, the coordinates of the corners of each occupied voxel in the octomap have to be computed. Then, the point which appear more than once are rejected. Following this procedure, the coordinates of the exterior corners of the occupied cell are considered and the interior ones are rejected. The interior corners are not considered because any point in these corners is directly visibly from the exterior corners. Figure 3.9 illustrates this procedure in two dimensions.

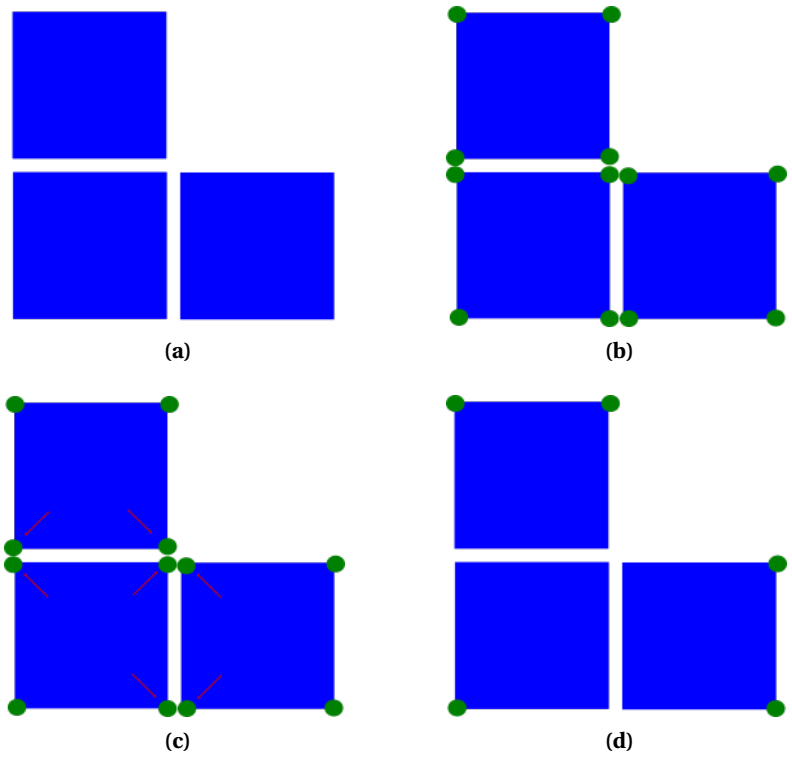


Figure 3.9: In this figure is illustrated the procedure of point selection to construct the visibility graph: (a) in this situation the obstacle is constructed from three voxels (spaces between them are for better visual representation). (b) The coordinates of the corners of each voxel are found and (c) all points who appear more than once are rejected. Finally, (d) the remaining points compose the visibility graph.

Once all the desired points have found, all visible pairs have to be found as well. To do this in octomap, the *raycast* function was used. This means, that a ray was sent from a point to the direction of each other point. This ray traverses the octree and returns true if it found an occupied cell. Initially, this function was developed for finding obstacles and thus constructing a map. However, with the reversion of the logic of raycasting, non-obstructed paths can be found between the points.

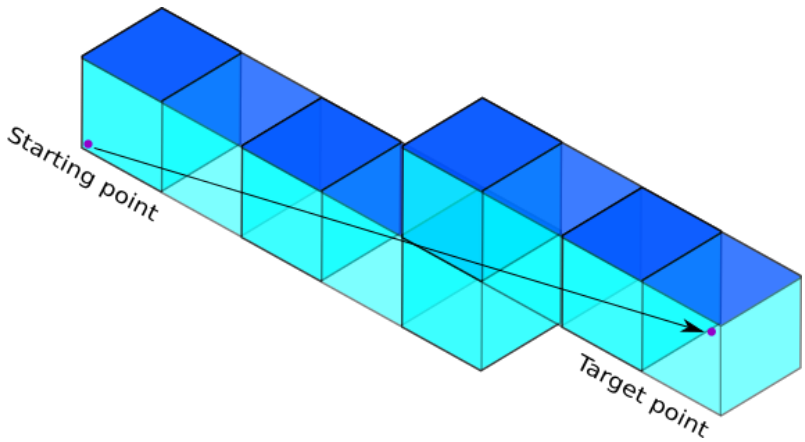


Figure 3.10: Raycasting method.

To continue with, two more points are added in the visibility graph, the starting point and the goal. When these points have been added, a raycasting is performed between these two points.

If there is a non-obstructed path between them the rest of the search breaks and the shortest path have been found. Else, the visibility graph algorithm is performed for these two points as well and all the acceptable (i.e. non-obstructed) connections are inserted to the graph. Finally, any graph based path planner can be performed. In addition, the start and goal points stay in and enrich the graph for any further path search.

3.4.1 Sampling

As mentioned above, the visibility graph was based on the octomap representation. A key aspect all voxel based maps is that every obstacle shape is constructed with cubes. This is a problem because, for instance, a sphere will end up having too many corners on its surface and thus too many possible points on the visibility graph (see figure 3.11(a)). Furthermore, when a wall is detected, then probably both sides of the wall are not needed. However, nodes are generated in both sides. Finally, octomap is a probabilistic representation and so even a straight wall will not be straight in the map but voxels will stand out.

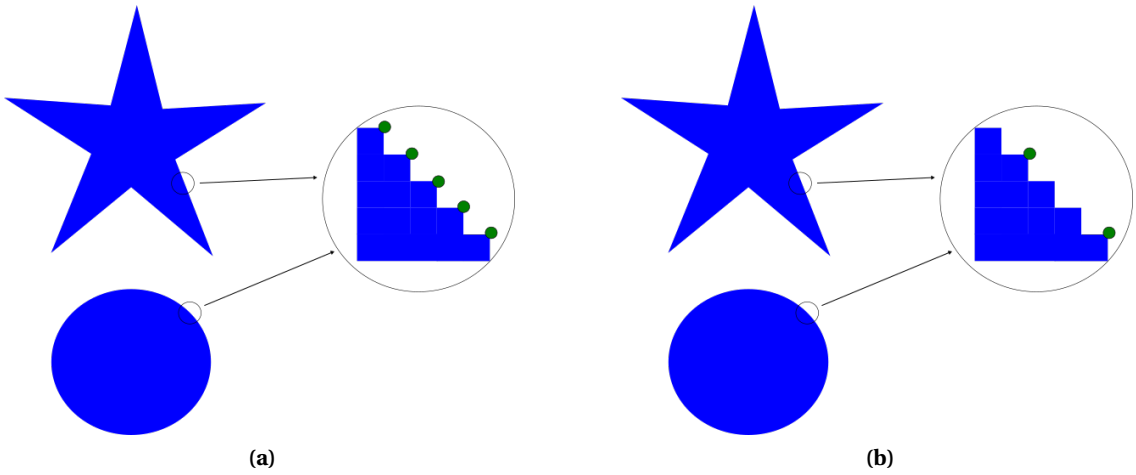


Figure 3.11: (a) Every obstacle shape, in octomap, is made out of cubes and so it contains a lot of points for the construction of the visibility graph, (b) after sampling we eliminate most of the points

To overcome these problems, and eventually reduce the visibility graph size, a solutions have been introduced. After the procedure explained, the visibility graph will contain too many points. Thus, in the case where not all map contains useful information, such as the outer side of the wall or the ground in windmill field, boundaries can be used to neglect specific regions of the map. Thereafter, the rest of the points are sampled to minimize their multitude. This is done by random sampling with a rate that will leave a big amount of points in the final graph such that it is computationally efficient to produce. This method ensures that the points are spread equally through out the desired region of the map.

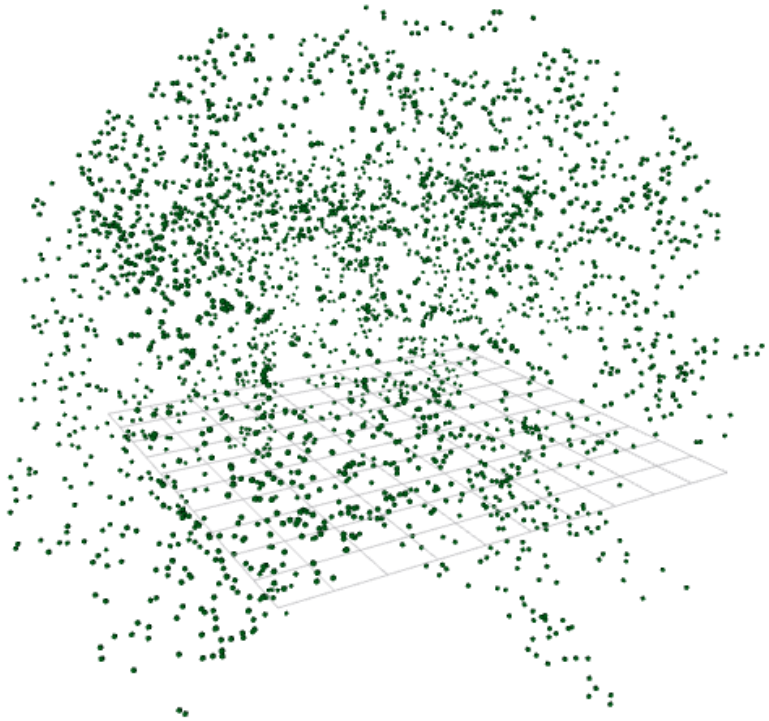
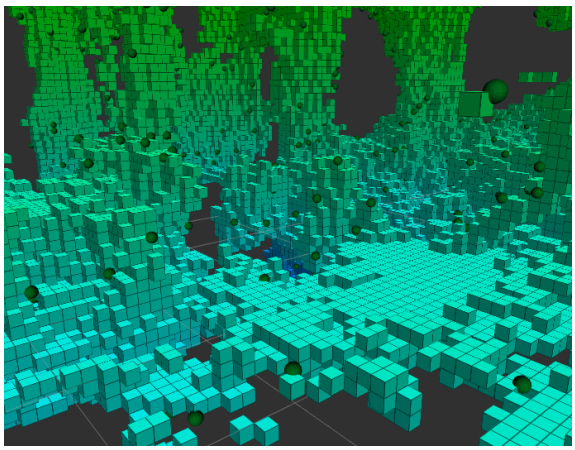
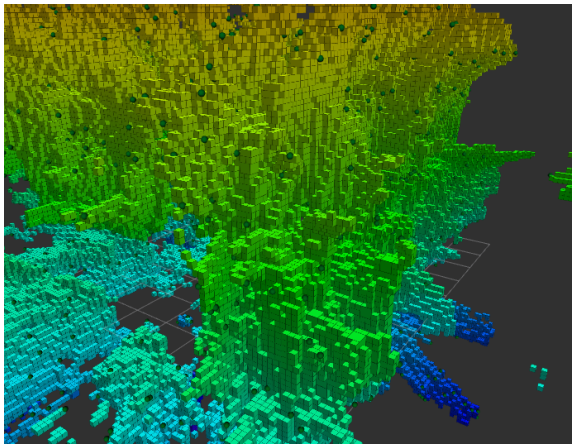


Figure 3.12: Generated points for visibility graph construction from SmartXP Lab, University of Twente, visualized in rviz.



(a)



(b)

Figure 3.13: Generated points (green spheres) for visibility graph construction after sampling and without selecting a specific region. This map was constructed at the SmartXP Lab, University of Twente, visualized in rviz. (a) inside view of the map, (b) outside view of the borders of the map.

3.4.2 Unordered-Map and Hashing

The visibility graph has to contain as many points as possible in such a way that the multitude will be computational efficient. The more rich the graph is the better it represents the real world. Therefore, the use of an array to store the nodes is computationally inefficient since an iteration has to occur in order to access a node.

An improved solution to this problem was introduced by storing the points in an unordered map structure. Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value. Thus, an element in an unordered map can be accessed in O(1). Key value is unique for each container and can be accessed directly by using the key value. To calculate the key values a hash function was implemented. This function gets as input the coordinates of a point and gives as output a unique value for each point.

Hash functions have a wide area of applications, from finding duplicates in large files to cryptography. Hash functions map any data, in our case the coordinates of a three dimensional point, to a value of given length which is called hash value. Thus, every time the same point is inserted in the hash function, it gives the same hash value. The combination of the hash function with the unordered map gives several advantages.

First of all, unordered maps do not have fixed size. Thus, their size can grow as new points are inserted the visibility graph. Next, due to the hash value, insertion, deletion and look up can be

performed in constant time. Finally, as the map might change, new points will be inserted or some existing points might be deleted. The mapped value of the unordered map can be another unordered map (see figure 3.14). This is useful because it allow us to access any existing pair in the visibility graph in constant time. Thus, also we can insert new visible pairs or delete existing ones, as the map changes, in O(1).

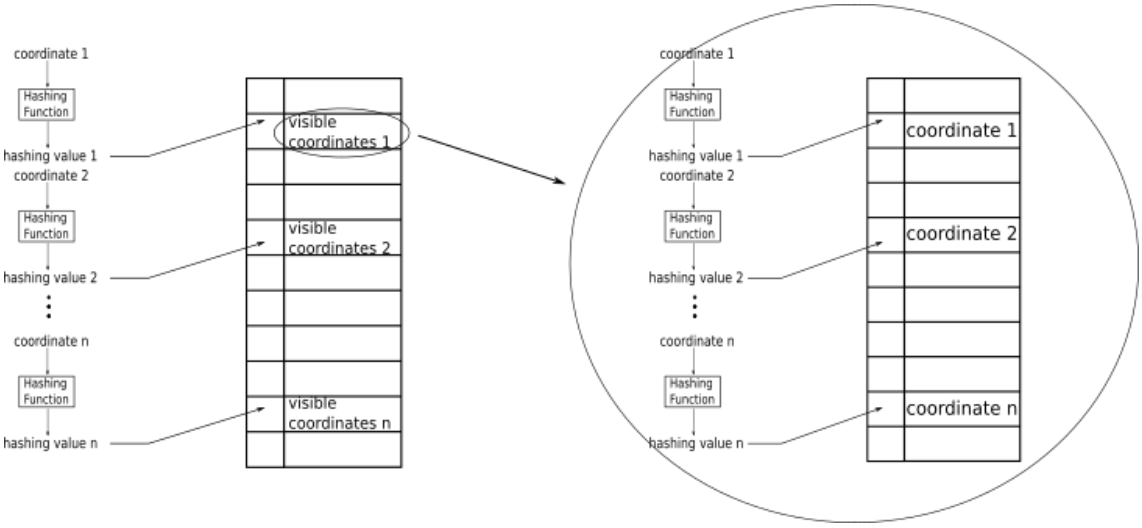


Figure 3.14: This figure illustrates how the coordinates are transformed to hash values through the hashing procedure and inserted in the unordered map structure representation used for the construction of the visibility graph.

To find all visible pairs a raycasting should be performed for each possible pair in the graph. This procedure would normally take n^2 searches.

A method was used for minimizing the required searches. This approach takes advantage of the unordered map's property to access directly any stored element. Therefor, an iteration occur through the elements of the unordered map. Within this iteration, the raycasting is performed from the inquiry node to only the nodes which are stored after it as it is illustrated in figure 3.15. Then, if a visible pair is found, the visible node is stored as neighbor of the inquiry node and vice versa as it is depicted in the pseudocode 3.1. This is a fast procedure because an iteration is not required to access any place in this data structure.

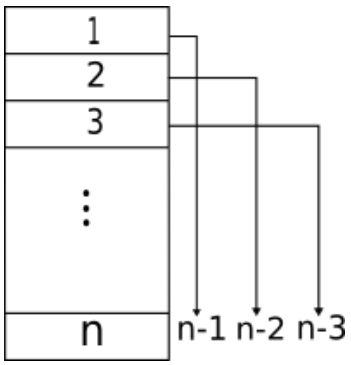


Figure 3.15: This figure illustrates how the coordinates are transformed to hash values through the hashing procedure and inserted in the unordered map structure representation used for the construction of the visibility graph.

Using this approach, the amount of searches are reduced to:

$$(n-1)+(n-2)+\cdots+1=\frac{n^2}{2} \tag{3.4}$$

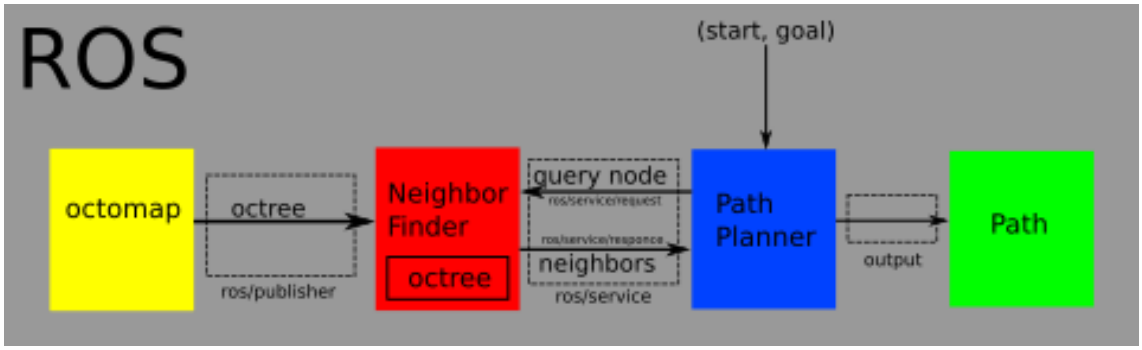
```
unordered_map sampledMap //Contains the sampled nodes
unordered_map finalMap   //Will contain the nodes of the
                         //final Visibility Graph

for ( it = sampledMap.begin(); it != sampledMap.end(); ++it ) {

        get coordA from it
        it2 = it
        it2++

        for ( it2; it2 != sampledMap.end(); ++it2 ) {

                get coordB from it2
                visible = raycast(coordA, coordB)

                if (visible) {

                        Store coordB in finalMap[coordA]
                        Store coordA in finalMap[coordB]
                }
        }
}
```

Listing 3.1: Pseudo code for visible neighbor search reduction

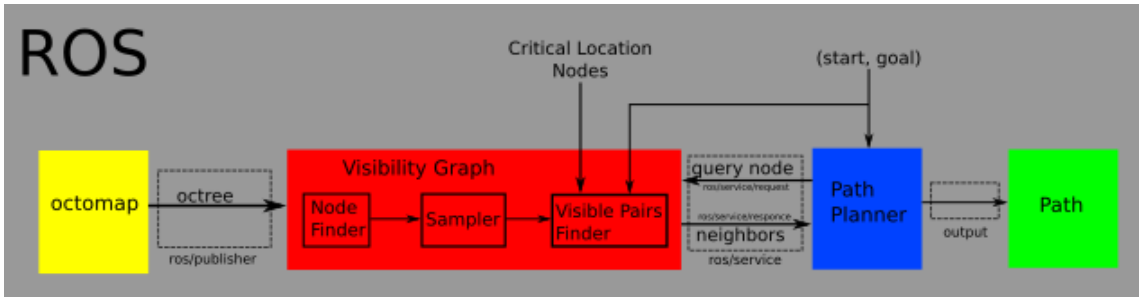
4 Implementation and Results

4.1 Implementation

The implementation was done in ROS (Robot Operating System). Several methods which result to low processing time were considered. The most important were the visibility graph representation, the unordered map with hashing and the min heap structure.



(a)



(b)

Figure 4.1: Implementation representation. (a) Octomap and Cube Grid implementation in ROS. (b) Visibility Graph implementation in ROS.

For the octomap representation the octomap ROS package was used, which allows to publish the octree map through the octomap server node. Separate C++ programs were developed which subscribe to the octomap server node. These programs include the implementation of neighbor finding for the cube grid method and the octomap. An other C++ program was developed for the implementation of the visibility graph. Furthermore, a python script was developed for the A* path planner with the techniques which have already been discussed. The path planner, communicates with the C++ programs via ROS service. Thus, when the planner algorithm is about to expand a node, it requests the node's traversable or visible neighbors for the cube and octomap or the visibility graph representations respectively. When the request has been made, then the path planner waits for the response.

In figure 4.3 below the rviz representation of the neighbors in octomap is illustrated. This figure illustrates that all the neighbors are indeed connected via face, edge or corner with the query point. For the representation, a random query point was selected and the neighbor finding algorithm was performed for this point. This algorithm used the method mentioned above and can find all different sized voxel neighbors with no errors.

To find all connected points in the visibility graph, the raycasting function from the ROS octomap package was used. In this function the direction and the length of each desired ray was specified to verify the visibility between two nodes.

For the cube grid, the standard twenty six (26) neighbor model was used. This means that in order to find a neighbor in a desired direction, the specific coordinate(s) had to be increased by the highest resolution of octomap. In our examples the highest resolution (lower voxel size) was 0.1m. Finally, from all the possible neighbors the occupied ones were filtered out. The highest resolution of octomap was used for the neighbor finding because otherwise there would not make any sense to use the octomap structure.

In our implementation, for the two-dimensional map the Diagonal distance was used, for the three-dimensional cube grid and octomap the Diagonal distance adapted in the three dimensions ($1.7 \cdot min(\Delta x, \Delta y, \Delta z) + 1.4 \cdot (median(\Delta x, \Delta y, \Delta z) - min(\Delta x, \Delta y, \Delta z)) + max(\Delta x, \Delta y, \Delta z) - median(\Delta x, \Delta y, \Delta z)$) and for the visibility graph the euclidean distance adapted in the three dimensions was chosen ($\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$).

Finally in figure 4.2 are illustrated all the steps for the implementation and use of the visibility graph and the A* path planner.

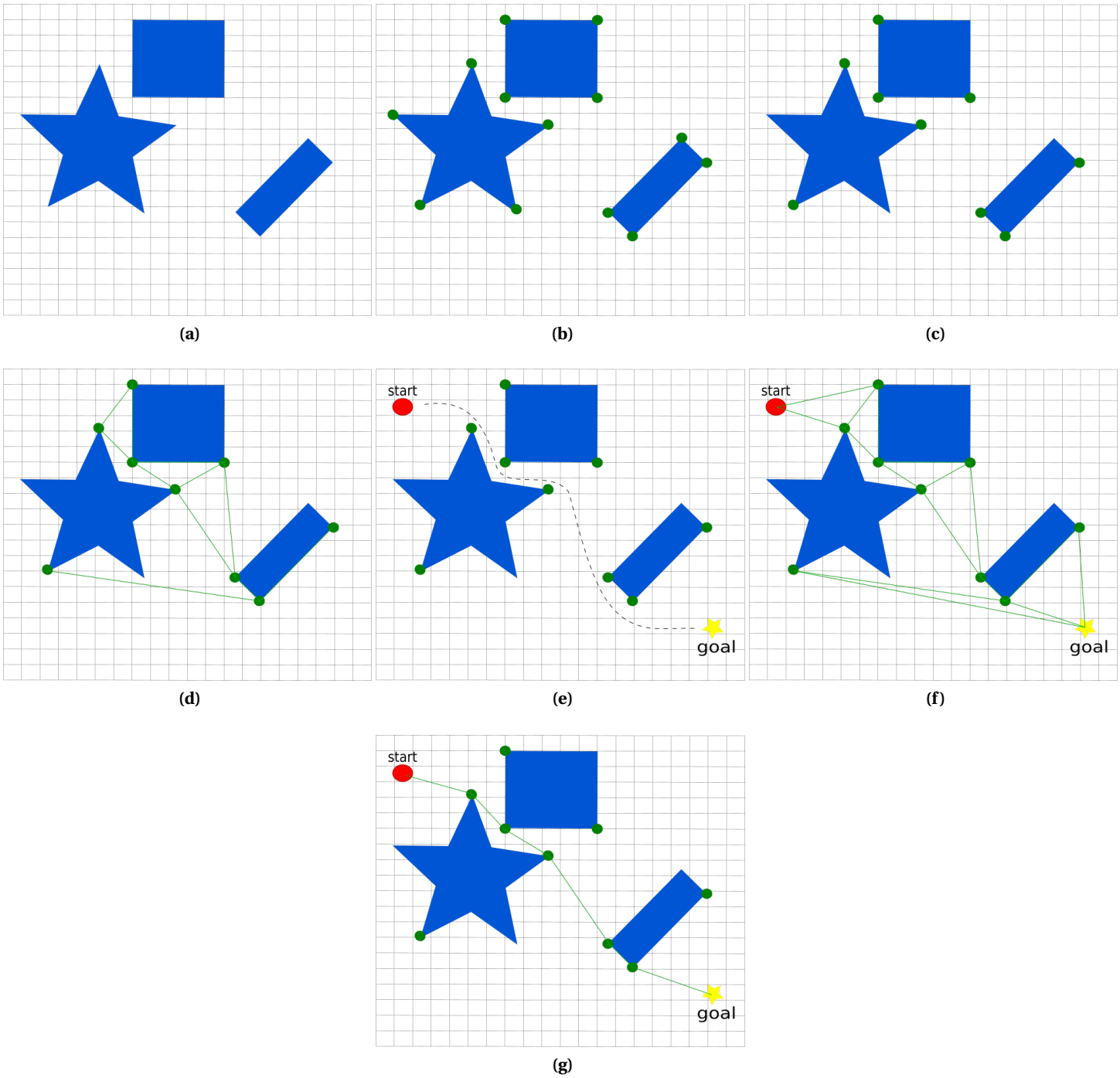


Figure 4.2: In this figure it is illustrated the procedure of the visibility graph: (a) a map is constructed. Then,(b) all points at the corners of the obstacles are found and (c) sampled. Hereafter, (d) the visibility graph is constructed. (e) The points for which we want to find the path are added and (f) inserted in the graph. Finally, (g) the A* algorithm generates the path from start to goal.

4.2 Results

Furthermore, figures 4.4 and 4.5 illustrates the exported paths from the A* path planner algorithm in octomap representation and visibility graph representation respectively for different starting and goal points. This was done to characterize the shapes of the paths of the different methods. The visualization was done in ROS RVIZ. The green spheres in the figures illustrate the searching nodes for the path planner. The blue lines are the generated paths. These paths

can be processed further in order to be smoother so the drone will have a more realistic path to follow and also reaches the final point in a specific orientation.

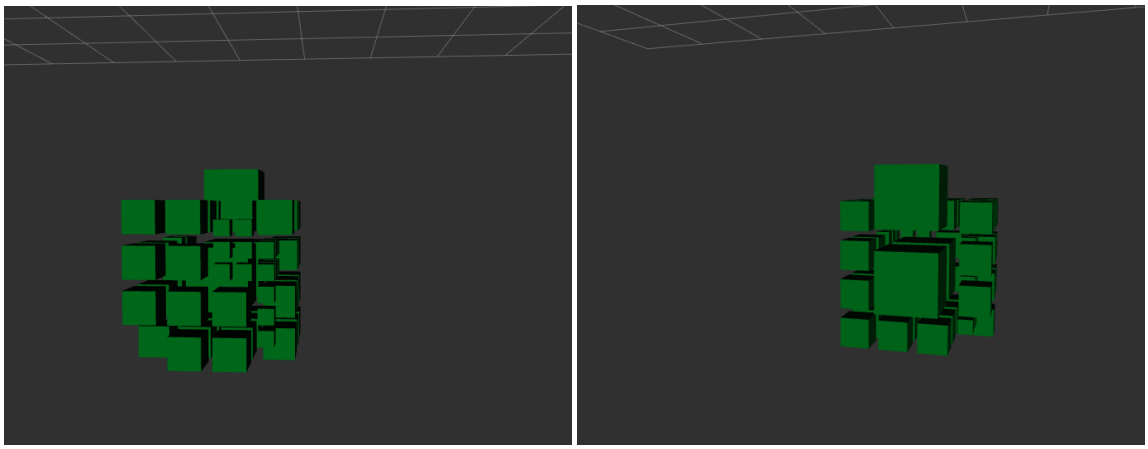


Figure 4.3: Different view angle of ROS RVIZ representation of neighbors in octomap using the naive approach. In these pictures is illustrated a random query point and all of its different sized voxel neighbors as they were found from the algorithm.

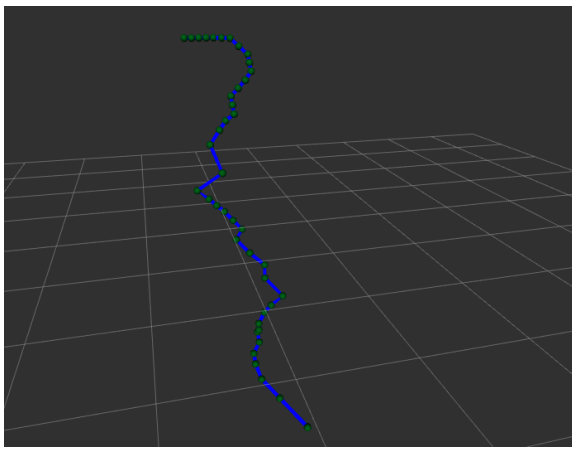


Figure 4.4: ROS RVIZ illustration of the exported path using A* path planner algorithm and octomap representation.

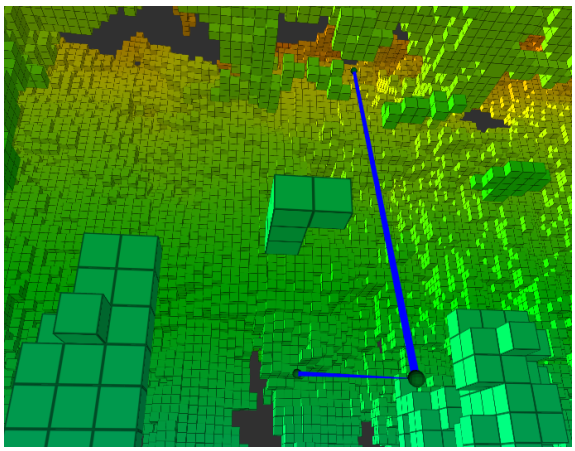
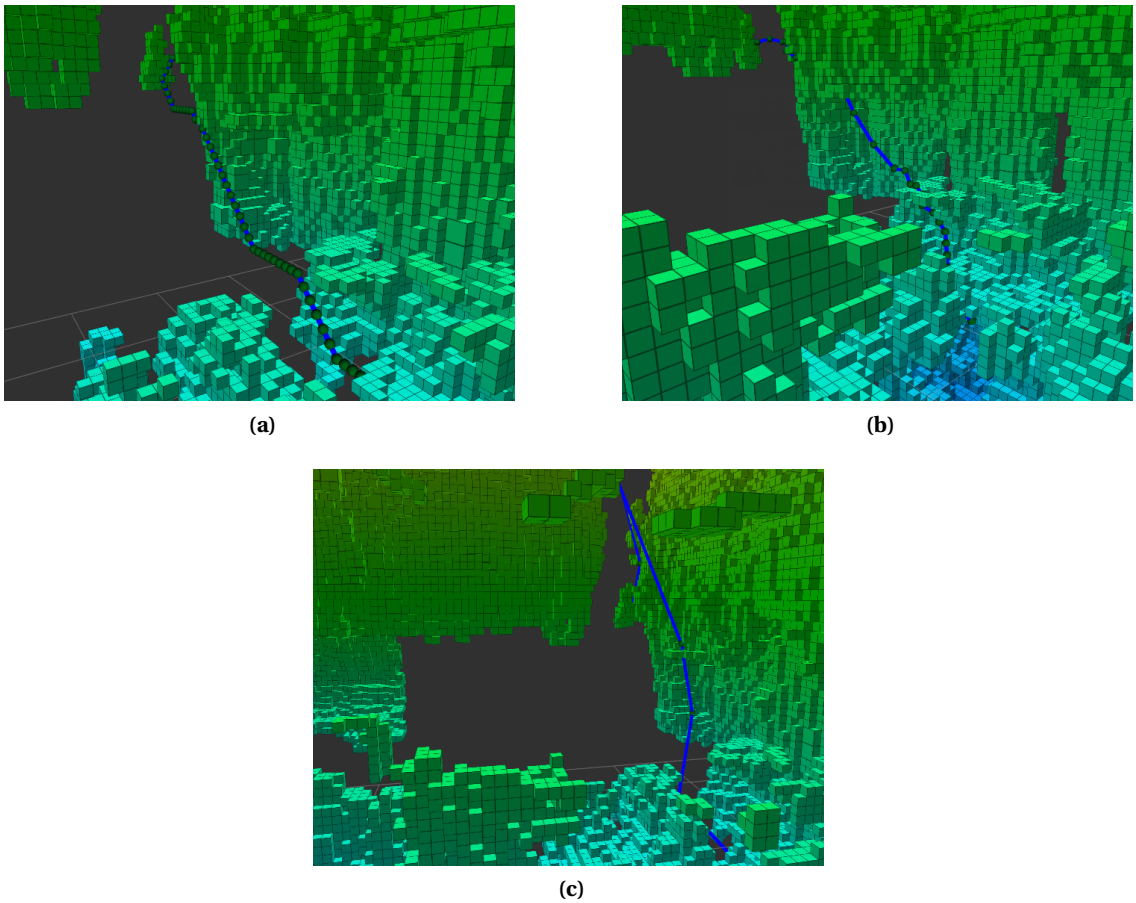


Figure 4.5: ROS RVIZ illustration of the exported path using A* path planner algorithm and visibility graph representation.



(a) **(b)** **(c)**

Figure 4.6: In this figure are illustrated the generated paths for the same starting point and goal (a) in cube grid, (b) in octomap representation and (c) in visibility graph representation.

Simulations have been conducted to test different map representations with respect to processing time and path length. Below, a table depicts the results which were given by the paths in figure 4.6. Note that these paths were generated with the same start and goal location.

	time (s)	expanded nodes	path nodes	path length (m)
cube grid	548.03	207535	55	6.85
octomap	2.14	404	45	7.63
visibility graph	0.24	40	7	9.56

Table 4.1: Results for the path illustrated in figure 4.6 for all map representations.

In our system, the construction of the visibility graph took in total 7.94 seconds. This time regards the computation of visible paths between nodes. The total amount of nodes computed in this time was three thousand (3000). For the octomap, a non perfect, real situational, octomap representation was constructed from the SmartXP Lab in the University of Twente. Finally, in this example we did not have any prior search (starting and goal points of prior searches stay in the graph to enrich it) and we had not insert any other points in the visibility graph than the ones generated after sampling.

The results interpreted show that the path from the visibility graph was generated faster than the paths from the octomap and the cube grid. However, the total path length is longer. This is not necessarily a drawback as no extra points were inserted in critical locations. By critical locations are meant all the place that are of interest can be of potential help by resulting to a better

optimized final path, with given constraints. Such locations could be the center of both sides of a door (or an opening in general) or random locations around the center of big free spaces. For instance, in the windmill field, nodes could be inserted in the space between the wind-power generators. This procedure right now is not automated but it can be fully automated in the future.

Furthermore, if the sampling frequency raise, then also the total path length will reduce. Moreover, if the path was generated in free space (e.g. returning from a windmill to base), then the visibility graph can find shorter paths in way less amount of time.

As noted, the visibility graph might not seem the best option, depending on the physical map, if we want to generate just one path. However, in industrial maintenance many ARWs have to operate together and change starting and goal locations more than once. Consequently, visibility graph has a bigger advantage regarding the total processing time in this scenario than the octomap does. Regarding the path length, it can be comparative to the octomap generated path if critical points are inserted or if a high sampling rate is chosen.

5 Conclusion, Discussion and Future Work

5.1 Conclusion

To conclude, we propose the use of the visibility graph map representation coupled with the A* path planner to achieve path planning in a three dimensional environment on runtime. In addition to the procedure presented, it is also advised to use additional nodes in the visibility graph in critical places. The later term implies all these nodes which are not automatically generated and have to be added by hand in places which would help the planner to find a better optimized solution given constraints. Such places are the sides of doors or points randomly distributed in large free spaces. This technique will contribute to the generation of a shorter path.

The visibility graph representation reduces the traversing time of a big areas with respect to the octomap or the cube grid representations. The octomap package's build in function raycast was used to find all the visible pairs for the construction of the graph. Furthermore, as it is documented, advanced techniques have been implemented to our solution. The hash in combination with the unordered map structure was used to store the graph for insertion, deletion and lookup in constant time. The later, helps to have faster access to the nodes. When the graph is constructed, the A* path planner requests the neighbors of an expanded node. This saves time so the path planner can compute the path faster. Finally, a heap was constructed for faster list sorting in the path planner.

Even though, the visibility graph takes a small amount of time to be produced, the path itself can be generated in a noticeable smaller amount of time than the other map representations do. This is useful because our configuration could also perform in partially unknown environments. Thus, if a previously unmapped obstacle appears in the map after the generation of the paths for the ARWs then they just have to wait until the new pairs will generate. Then, the generation of the path is done fast.

Finally, int the next section some suggestions for future work is proposed. These include the usage of a dynamic path planner for even faster path generation in (partially) unknown environments. However, it is considered that the most influence in the path planning process will be the use of parallel programming (CUDA). CUDA will allow the construction of a more dense visibility graph in less time. These modifications will allow the drones to operate in dynamic environments smoothly and in runtime.

5.2 Discussion and Future Work

Through this document, methods which result in rapid path generation have been considered and used. However, more can be considered for future work of this project. In this part some of them are presented, for which details are given in the following sections.

To begin with, as already has been mentioned, the map representation is a crucial component for path planning. Results depict that the A* algorithm performs better with the visibility graph. However, the visibility graph could be potentially better optimized with respect to generation time. Research has been conducted on the fast implementation of the visibility graph [14] [15] [16]. Moreover, a better sampling technique could be introduced which will sample points more efficient instead of the random sampling which is used now.

To continue with, parallel programming is very useful in testing big datasets. Thus, the implementation of the construction of the visibility graph in parallel programming (CUDA), could speed up the process noticeable. The advantage of this method is that it could handle way

more points in less time. Thus, our configuration could also perform as a dynamic path planner.

Furthermore, the A* algorithm does not perform well in dynamic environments as it is not designed to do so. Even though it is a really fast algorithm, its limitations are that it must have a good knowledge of the map. It may need a short amount of time to recalculate a path on a two dimensional terrain, but this is not the case when the dimensions become higher or when there are dynamic obstacles in the map. Several solutions could assign to this problem.

In the scenario in which there is a good knowledge of the map, but there are dynamic obstacles, for instance flying outdoors and a dynamic obstacle appears in front of the drone (another drone etc), the path could be computed with the A* algorithm, before the flying process, and patch with an obstacle avoidance algorithm. This algorithm could either wait for the moving obstacle to pass or to try to find a way around it and then follow again the precomputed path. However, this algorithm will work with small obstacles but not if for instance an unmapped wall appears in front of the ARWs as it will need to recalculate a big path and it might take a relatively big amount of time.

In the scenario that the map is not well known, different path finding algorithms are proposed which can handle this situations easier. Such algorithms are the D*-Lite [17] and the 3D Field D* [18].

Moreover, a path smoothing algorithm could be applied to the final path such that the drone can turn smoother and also arrive at the goal with a specific orientation. There are also other path planning algorithms which return a smoother path than the A*. One of these algorithms is the Lazy Theta* [19].

Finally, due to sampling, the goal or any other intermediate point could not be visible directly for the starting point but from an intermediate point between two graph nodes. Thus, it is suggested to implement a feature which will act after the generation of the path. This feature, will search for a visible connection between the current position of the ARWs and their goal. If there is no visible connection with the goal, then they will search for visible connection for the point which construct their path and have lower h(n) value (points closer to the end) recursively.

In the following section some of the aspects above are discussed and explained in more detail.

5.2.1 Parallel Programming (CUDA)

A good Central Processing Unit (CPU) may have up to 32 cores and thus manage to perform 32 processes in parallel. These CPUs are, generally, expensive. A cheap substitute, is the graphics processing unit (GPU). However, GPUs are not just cheap substitutes as they can outperform the CPUs. For instance, a high performance nVIDIA GPU can have up to a few thousands cores which can perform multiple calculations simultaneously. Furthermore, they have faster accessible RAM than those used from the computers (DDR5 in nVIDIA GPU and DDR3 in computer). However, GPUs have a draw back. They have small sized cache memory. To exploit the full potentials of the parallel programming computing, the programmer has to load his program in the cache memory in order to make it as fast as possible. On the other hand, this limitation can disappear when buying more expensive graphics cards.

We propose specifically nVIDIA graphics processing units as the company has developed CUDA. CUDA is powerful application programming interface (API) for parallel computing, which works only in nVIDIA architecture graphics cards. An other open source solution is OpenGL, but it needs advanced programming skills in parallel computing. Furthermore, CUDA community is bigger than OpenGL's making the development of an application easier.

5.2.2 Alternative Path Planning Algorithms

For dynamic environments it is preferred to use a path planner which is designed for such environments. A dynamic environment could be very challenging and so a path planner designed for static environments would not perform well.

All proposed path planners (D*-Lite and 3D Field D*) are designed for dynamic environments. They have fast replanning ability and can perform well with graphs. They use locally replannig techniques which means that when they figure out that their path is suddenly blocked, they remember the rest of the path after the obstacle and they try to find a new path to go there unless they find a shorter path to the goal. They repeat this process for every dynamic obstacle they meet and they remember all previous paths in case they need to use them again, minimizing thus the replanning time.

A Appendix 1

In this appendix is presented the A* path planner algorithm pseudocode.

```
Create a node containing the goal state node_goal
        Create a node containing the start state node_start
        Put node_start on the open list
        while the OPEN list is not empty
        {

        Get the node off the open list with the lowest f and
    call it node_current

        if node_current is the same state as node_goal we have
    found the solution; break from the while loop

            Generate each state node_successor that can come
        after node_current

            for each node_successor of node_current
            {
                Set the cost of node_successor to be the cost
            of node_current plus the cost to get to
            node_successor from node_current

                find node_successor on the OPEN list

                if node_successor is on the OPEN list but the
            existing one is as good or better then discard
            this successor and continue

                if node_successor is on the CLOSED list but the
            existing one is as good or better then discard
            this successor and continue

                Remove occurences of node_successor from OPEN
            and CLOSED

                Set the parent of node_successor to node_current

                Set h to be the estimated distance to node_goal
            (Using the heuristic function)

                Add node_successor to the OPEN list
            }
            Add node_current to the CLOSED list
        }
```

Listing A.1: Pseudo code the A* path planner (http://heyes-jones.com/pseudocode.php)

B Appendix 2

A big map could take a big amount of time to be processed in order to find the neighbors of all points for the lookup table. Thus, more advanced techniques need to be considered. Examples of such techniques are hashing tables [20] [21] [22] [23], directional lookup tables [24], probabilistic approaches [25] [26], or efficient search techniques [27] [28].

B.1 Efficient Neighbor Finding Techniques For Octomap

Efficient neighbor finding techniques can reduce the time needed to detect the neighbors of a query point and thus the total time to construct the look up table for our path planning. This method is useful when the drones operate in a dynamic environment. In this section are documented briefly the most important of the methods presented above.

To find neighbors with the hash method, first of all hash tables have to be constructed. These hash tables, apply specific digital keys to each voxel. Voxels with the same key means that they are neighbors. This process is repeated several times with different hash tables. The voxels which had the same digital key most of the times are neighbors. However, with this method does not ensure that all adjacent neighbors have been found. Moreover, the hash tables could be complex and hard to construct.

Directional look up tables are easier to implement. Through these tables the coordinates of the neighbors in each direction can be found, taking into consideration the size of the query point. However, they have to be accessed for each specific query point and a specific direction.

On the other hand, as presented in [27] it is possible to assign specific digital addresses to each voxel, making the neighbor finding method easier. This method, avoids to use complex hash tables as also the excessive search for each query point.

Bibliography

[1] Shengdong Xu, Dominik Honegger, Marc Pollefeys, and Lionel Heng. Real-time 3d navigation for autonomous vision-guided mavs. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 53–59. IEEE, 2015.

[2] Stefan Hrabar. 3d path planning and stereo-based obstacle avoidance for rotorcraft uavs. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 807–814. IEEE, 2008.

[3] Jean-Claude Latombe. *Robot motion planning*, volume 124. Springer Science & Business Media, 2012.

[4] Howie M Choset. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[5] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[6] J Foo, Jared Knutzon, James Oliver, and Eliot Winer. Three-dimensional path planning of unmanned aerial vehicles using particle swarm optimization. In *11th AIAA/ISSMO multidisciplinary analysis and optimization conference, Portsmouth, Virginia*, 2006.

[7] Weiwei Zhan, Wei Wang, Nengcheng Chen, and Chao Wang. Efficient uav path planning with multiconstraints in a 3d large battlefield environment. *Mathematical Problems in Engineering*, 2014, 2014.

[8] Halit Ergezer and M Kemal Leblebicioğlu. 3d path planning for uavs for maximum information collection. In *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*, pages 79–88. IEEE, 2013.

[9] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[10] Rong Zhou and Eric A Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4):385–408, 2006.

[11] Amit. A*'s use of the heuristic, http://theory.stanford.edu/ amitp/gameprogramming/heuristics.html.

[12] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[13] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[14] Dave Coleman. Lee's o(n2 log n) visibility graph algorithm implementation and analysis, 2012.

[15] Mojtaba Nouri Bygi and Mohammad Ghodsi. 3d visibility graph. *Computational Science and its Applications, Kuala Lampur*, 2007.

[16] Frédo Durand, George Drettakis, and Claude Puech. The 3d visibility complex. *ACM Transactions on Graphics (TOG)*, 21(2):176–206, 2002.

[17] Sven Koenig and Maxim Likhachev. D* lite. In *AAAI/IAAI*, pages 476–483, 2002.

[18] Joseph Carsten, Dave Ferguson, and Anthony Stentz. 3d field d: Improved path planning and replanning in three dimensions. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3381–3386. IEEE, 2006.

[19] Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *Third Annual Symposium on Combinatorial Search*, 2010.

[20] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006.

[21] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 563–576. ACM, 2009.

[22] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007.

[23] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *Signal Processing Magazine, IEEE*, 25(2):128–131, 2008.

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

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

[26] Rener Castro, Thomas Lewiner, Hélio Lopes, Geovan Tavares, and Alex Bordignon. Statistical optimization of octree searches. In *Computer Graphics Forum*, volume 27, pages 1557–1566. Wiley Online Library, 2008.

[27] Jaewoong Kim and Sukhan Lee. Fast neighbor cells finding method for multiple octree representation. In *Computational Intelligence in Robotics and Automation (CIRA), 2009 IEEE International Symposium on*, pages 540–545. IEEE, 2009.

[28] Bertram Drost and Slobodan Ilic. A hierarchical voxel hash for fast 3d nearest neighbor lookup. In *Pattern Recognition*, pages 302–312. Springer, 2013.