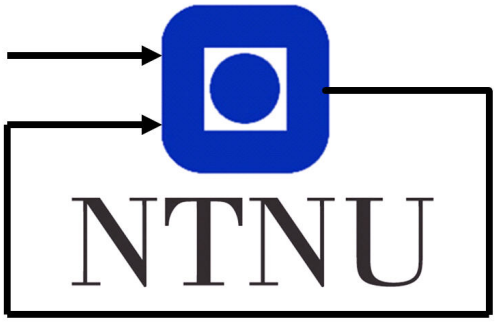
TTK4550 - Specialization Project
Long-range path planning for fixed-wing UAVs using A* search and OctoMap

Bendik Stuevold Eger



Norwegian University of Science and Technology
Department of Engineering Cybernetics

Trondheim, January 15, 2020

Abstract

Fixed-wing UAVs are suitable for long-range tasks such as power grid inspection, due to their long flight time. This paper aims to contribute to the development of an autonomous system using such UAVs to perform inspection. This is done by evaluating how use of the OctoMap framework can be combined with a weighted A* search algorithm to perform long-range path planning.

Implementation of a simple planner is presented and tested in an environment created from real world terrain data. The testing evaluates the use of two different heuristic functions: the diagonal distance and the Euclidean distance. By using the weighted A* algorithm we allow the planner to find bounded sub-optimal paths in exchange for better search times. Results show that the diagonal distance performs better when only allowing the planner to find optimal paths. However, when the environment resolution is adequately fine to safely be used for UAV path planning, finding optimal results is, computationally, very resource demanding. When allowing for sub-optimal paths, searching with the diagonal distance heuristic resulted in longer paths than with the Euclidean distance at similar search depths.

Acknowledgement

I would like to thank Helge-André Langåker at KVS Technologies for his role as co-supervisor. He suggested evaluating the OctoMap-framework and helped forming the boundaries of the project.

Also, I would like to thank Lars Imsland for supervising the project and for giving solid feedback during the writing of this paper.

Lastly, I would like to thank Mia Olea Vettestad for proof reading this paper and being my linguistic sparring partner.

Table of Contents

List of Figures

List of Tables

1 Introduction

Nowadays, focusing on green energy plants, the need for reliable electrical transport is growing all over Europe. With the Norwegian power grid spanning thousands of kilometers, damage to grid towers and power lines is not uncommon. Power outages can be prevented by regular inspection and maintenance of the grid.

Damage inspection has for a long time been done manually by lineworkers, whose job is time consuming and dangerous. In the past years, inspection has also been performed using camera equipment and helicopters. With the advancement of robot technology, state of the art solutions makes it possible to create totally automated systems for power grid inspection (Pagnano, Höpf, and Teti, 2013).

Rotorcraft UAVs are frequently used for inspection of structures, such as windmills, buildings and even electrical towers. It is possible to get high quality inspection images due to the rotorcrafts payload capacity and stabilization properties (Boon, Drijfhout, and Tesfamichael, 2017). However, they are limited in flight time and area coverage.

Fixed-wing UAVs have greater flight time, are more cost efficient and are easier to maintain (Boon, Drijfhout, and Tesfamichael, 2017). This make them suitable for long-range tasks such as power grid inspection. A challenge arises, however, due to the fixed-wing UAVs inability to hover stationary. Where the rotorcraft has the ability to maneuver freely to get inspection images at any angle, the fixed-wing UAV needs extensive path planning.

The project summarized in this paper aim to contribute to a solution for this problem by assessing the performance and qualities of common robotic path planning methods when applied to long-range trajectory planning. By evaluating these methods we aim to present findings relevant for further development of a long-range path planner for fixed-wing UAVs, to be used in a system for automatic power grid inspection.

The evaluated methods are: the use of OctoMap as a framework for terrain representation, combined with weighted A* search for path planning. Evaluating the OctoMap framework in long-range setting was suggested by the Norwegian robotics company KVS Technologies.

This paper will present a simple implementation of a path planner, using weighted A* search in combination with the OctoMap framework. We will examine how different search heuristics affect the resulting path length and search depth. The heuristic functions that are evaluated are the diagonal distance and the Euclidean distance.

1.1 Related work

The Octomap framework is a frequently used environment mapping tool in the field of robotics. Anastasios and Zompas, 2016 presents a system using OctoMap paired with the A* algorithm to plan paths for Aerial Robotic Workers. They discuss several methods of structuring the map environment for path planning, among using a 3D grid and a visibility graph. Their thesis focuses on improving the performance of the system through optimizing the map environment.

In the field of fixed-wing UAV path planning Xia et al., 2009 presents a system for military threat avoidance by low penetration path planning. Their research shows that implementing the UAV dynamics via an improved A* algorithm gives better search results.

Section 6 further discuss both these papers, and suggest how their finding can be relevant for out system.

1.2 Outline

The remainder of this paper is divided into the following parts:

- **Theory**: Documents the theoretical background needed in this paper. This includes: heuristic estimates, best-first search algorithms and the OctoMap framework.
- **Implementation**: Describes challenges met during implementation of the path planner and how these challenges were solved.
- **Simulations and results**: Contains a description of how simulations were run and presents results from these simulations.
- **Discussion**: Discusses the results presented in the earlier chapter, and gives an intuitive explanation for the planner behavior.
- **Further work**: Presents topics relevant for further development of the path planner.
- **Conclusion**: Concludes the report.

2 Theory

The following sections depict a basic theoretical background necessary for the remainder of the paper. The part about informed search algorithms is mainly gathered from the book *Artificial Intelligence - A Modern Approach* by Russell and Norvig, 2010 and the publication *Weighted A* search - unifying view and application* by Ebendt and Drechsler, 2009. The figures used to illustrate the search algorithms are inspired by *Amit's A* Pages* by Patel, 1997.

The part containing an introduction to the OctoMap framework is developed from the publication *OctoMap: An efficient probabilistic 3D mapping framework based on octrees* by Hornung et al., 2013.

2.1 Manhattan, diagonal and Euclidean distance

The terms Manhattan, diagonal and Euclidean distance will be used throughout the paper. All three can be used as distance measurements for movement in a grid.

The Manhattan distance is a measurement of movement restricted to be parallel to the grid axes. The diagonal distance allows for movement along the diagonal axis as well. The Euclidean distance is not concerned with movement restrictions. Examples of the distances can be seen in Figure 1.

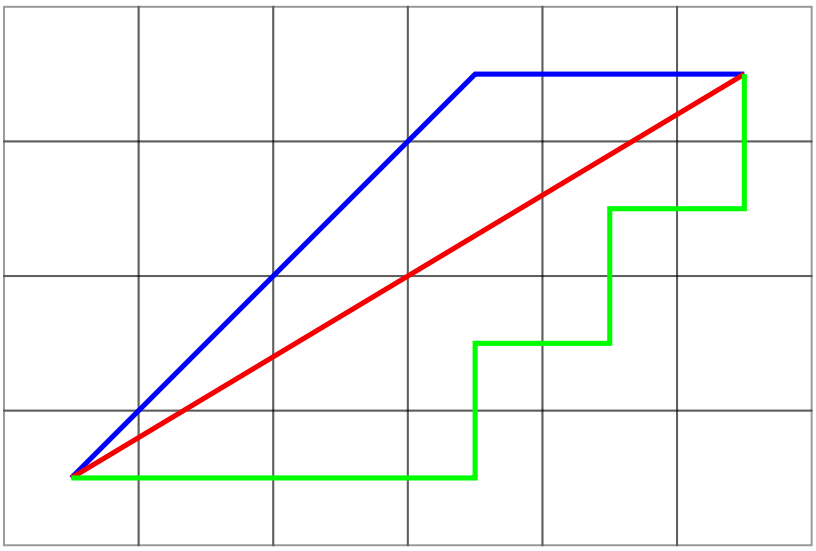


Figure 1: Comparison of different distance measurements. Green line: Manhattan distance. Red line: Euclidean distance. Blue line: diagonal distance.

2.2 Informed search algorithms

By using problem-specific knowledge beyond the definition of the problem itself it is possible to preform general informed searching yielding better results than any uninformed search strategy (Russell and Norvig, 2010). The so called best-first search is an instance of the general *tree-search* or *graph-search* that selects nodes for expansion based on an evaluation function, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation function is expanded first. The algorithm can be implemented as shown in Figure 2.

How $f(n)$ is chosen determines search strategy. Best-first algorithms often include a heuristic function as a component of $f(n)$, denoted $h(n)$. The heuristic function estimates the cost of the cheapest path from a state at node n to the goal state.

2.2.1 Greedy best-first search

The greedy best-first search algorithm tries to expand the node that is closest to the goal state. It therefore has the evaluation function

$$f(n) = h(n). \tag{1}$$

By always expanding the node closest to the goal greedy best-first is

---

function BEST-FIRST-SEARCH($start, goal$) **returns** a solution, or failure
  $node \leftarrow start$
  $explored \leftarrow$ an empty set
  $frontier \leftarrow$ an empty priority queue ordered by EVALUATION-FUNCTION
  $frontier \leftarrow$ INSERT($frontier, node$)
  loop do
    if EMPTY?($frontier$) **then return** failure
    $node \leftarrow$ POP($frontier$)
    if $node = goal$ **then return** SOLUTION($node$)
    add $node$ to $explored$
    for each $neighbour$ of $node$ **do**
      if $neighbour$ is not in $explored$ or $frontier$ **then**
        $frontier \leftarrow$ INSERT($frontier, neighbour$)
      else if $neighbour$ is in frontier with higher PATH-COST **then**
        $neighbour$.CAME-FROM $\leftarrow node$
        replace $frontier$ node with $neighbour$

Figure 2: Algorithm describing a best-first search for a graph environment. Search behavior is determined by choice of evaluation function.

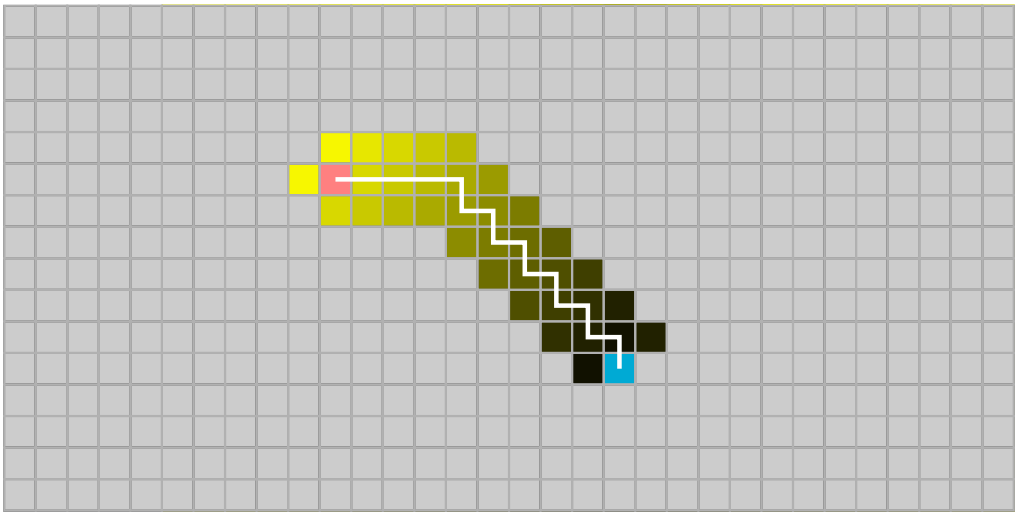


Figure 3: Path found by greedy best-first search using the Manhattan distance heuristic. The start node is in red and the goal in blue. Yellow tiles have a large heuristic value while black tiles have a low heuristic.

capable of finding a solution with minimal search cost (Russell and Norvig, 2010). See Figure 3 for an example of how a path is found in a two dimentional grid using the Manhattan distance as heuristic function.

However, greedy best-first search does not always provide optimal results. This can be seen in Figure 4 where an obstacle is placed between the start and goal tiles, resulting a path consisting of a detour.

2.2.2 A* search

The A* search algorithm is one of the more widely known forms of best-first search. The algorithm combines the path cost to reach a node, $g(n)$, with the heuristic estimate to reach the goal from the same node, $h(n)$, giving the evaluation function:

$$f(n) = g(n) + h(n). \tag{2}$$

Since $g(n)$ gives the shortest path from the start node to node n, and $h(n)$ gives an estimate of the cost from n to the goal node, $f(n)$ estimates the cost of the cheapest solution through n.

A* can give both optimal and complete results if $h(n)$ is chosen according to the right conditions (Russell and Norvig, 2010).

The first condition required is for $h(n)$ to be admissible. An admissible heuristic is one that never overestimates the minimum cost to reach the

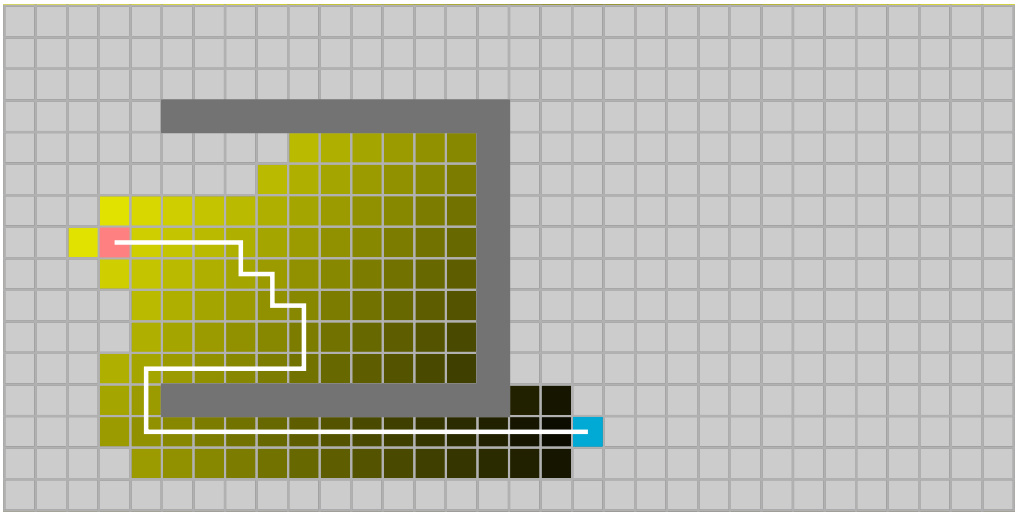


Figure 4: Non-optimal path found by using greedy best-first search. Yellow tiles have a large heuristic value while black tiles have a low heuristic.

goal.

The second condition is for $h(n)$ to be consistent. Consistent means that, for every node n and every successor n' available from n, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.

The right choice of heuristic function is dependent on the problem formulation. Figure 5 illustrates how using the Manhattan distance in an environment where the actions are restricted to horizontal and vertical movement finds an optimal path. If we relax the problem in Figure 5, i.e. by allowing diagonal movement, the Manhattan heuristic will overestimate the shortest path to the goal and we might not get an optimal result. Using the diagonal distance would be better in this case.

Using the diagonal distance as a heuristic for the original, un-relaxed, problem would however yield an optimal path. In fact, any heuristic yielding an optimal result for a relaxed problem will be admissible for the original problem (Russell and Norvig, 2010).

2.2.3 Weighted A*

By rewriting (2) as

$$f(n) = g(n) + (1 + \epsilon)h(n), \tag{3}$$

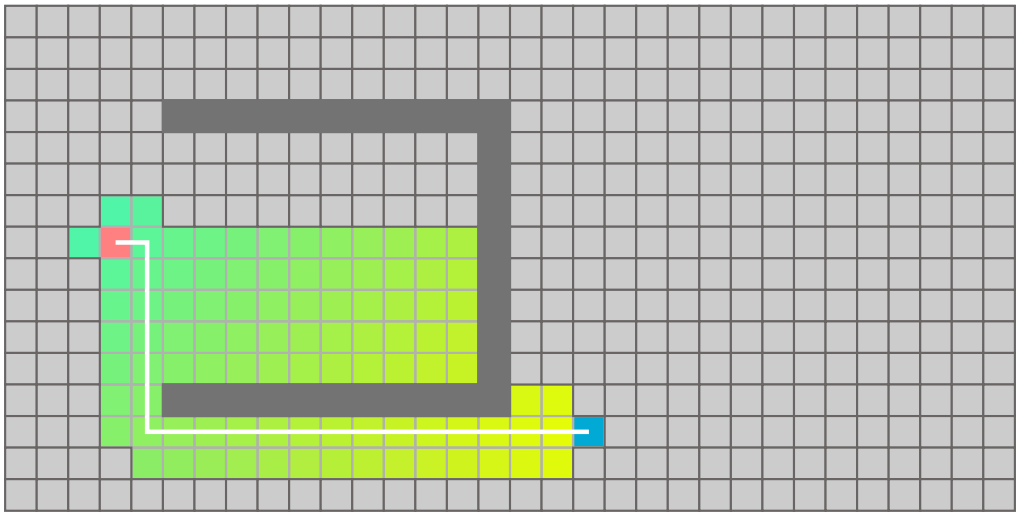


Figure 5: Optimal path found by using A* and the Manhattan distance heuristic. Teal tiles have a large heuritic value and yellow tiles have a large travel cost.

where $\epsilon \geq 0$, we get an algorithm called weighted A*. Increasing ϵ relaxes the admissibility condition on the heuristic to direct the search in a more promising direction. When searching, this will break ties in the evaluation function in favor of points with lower $h(n)$ value, i.e. points closer to the goal.

Given that $h(n)$ is admissible it can be shown that weighted A* is ϵ-admissible, meaning that is always finds a solution whose cost does not exceed the optimal cost by more than a factor of $1 + \epsilon$ (Ebendt and Drechsler, 2009). Giving a bound to how sub-optimal the solution can be.

If we choose a large ϵ the $h(n)$ term will dominate $g(n)$ and the algorithm will behave like a greedy best-first search.

2.3 OctoMap

Octomap is an integrated framework for representation of three-dimensional environments (Hornung et al., 2013). It uses octrees to efficiently store information regarding structure and occupancy. An octree is a hierarchical data structure for subdivision in 3D.

Each node is an octree represents the space contained in a cubic volume, called a voxel. This volume is recursively subdivided into eighth smaller voxels until a given minimum voxel size is reached. This subdivision is visualized in Figure 6. The minimum voxel size is determined by the resolution

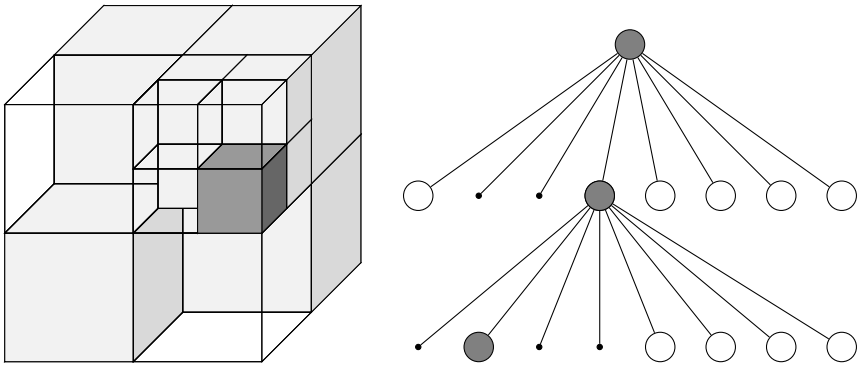


Figure 6: Example of how an octree contains its voxels. The hierarchical volumetric model is shown to the right and the corresponding voxels to the left. The occupied voxels are colored with dark grey and free voxels are white.

of the octree.

Voxels in the OctoMap framework are indexed using unique keys embedding their xyz-coordinates within the tree boundaries. A voxel's key consists of a triplet of sub-indexes, each representing its placement along the coordinate axes. Conversion between xyz-coordinate and voxel key is available through the framework.

Each voxel in the octree holds a field containing a pointer to an array of eighth pointers, one for each child. This array is only allocated if the voxel has any children, meaning it is not a leaf node. If a field in this child pointer array is uninitialized the volume contained by the respective voxel is unexplored or unknown.

Each voxel also holds a field containing the probability for the it being occupied. This probabilistic representation of occupancy makes OctoMap a framework well suited for dynamic environments.

In the OctoMap framework the maximum tree depth is 16, meaning that there are 16 possible subdivisions between the largest root node and the smallest leaf nodes. This gives a maximum number of 8^{16} leaf nodes. As mentioned earlier the pointer to the array of child pointers is only allocated if there indeed exists any child voxel nodes within it. This has a beneficial impact on the memory performance of the framework as it does not allocate memory for nodes without explicitly inserting information about its occupancy (Hornung et al., 2013).

As for data insertion there are three different methods available through the framework. Individual range measurements can be integrated using ray-casting by calling the method `insertRay`. This updates the end point of the measurement as occupied while all other voxels along a ray to the

sensor origin are updated as free. Point clouds, e.g., from 3D laser scans or stereo cameras can be integrated using `insertScan`. This batch operation is optimized to be more efficient than tracing each single ray from the origin. Finally, a single node in the octree can be updated with a point measurement by calling `updateNode`.

3 Implementation

The following chapter will explain how the OctoMap framework is integrated with a best-first search algorithm, as the one in Figure 2, to create a path planner capable of planning long distance trajectories in real world terrain data. The evaluation function used in the planner is implemented as (3), where $h(n)$ can be both the diagonal and Euclidean distance.

System implementation is done using Robot Operating System (ROS) and the chosen programming language is C++. The actual code implementation can be seen in Listing 1 in Appendix A.1.

The movement of the search algorithm is dependent on how neighbours are defined in the environment. The OctoMap, being a hierarchical tree structure, has no direct relation between neighbouring voxels of the same resolution. A node only stores pointers to its children, unless it is a leaf and has no children. Therefore, when creating an algorithm to search through an OctoMap environment, is was necessary to implement a method that made traversing to neighbouring voxels possible.

Once an efficient way to retrieve neighbouring voxels was available the next challenge was to implement the search algorithm. How nodes should be maintained as the search progresses, yielding new results for the evaluation function, and how these nodes should be represented programmatically.

This chapter will also explain how the two different heuristic functions are implemented and lastly, describe how the planner finds a path that keeps a safe distance from nearby terrain by making sure a volume surrounding each point in the path is unoccupied.

3.1 Neighbour finding

A simple way to solve the neighbour finding problem would be to iterate through the entire octree and check if a node's coordinate placed it as a neighbour to the current node. Anastasios and Zompas, 2016 discuss several methods to retrieve neighbouring voxels. They suggest more efficient ways of doing so than the naive full tree iteration approach.

The method for neighbour finding implemented in this project uses a voxel's unique key and the fact that this key directly relates to its xyz-coordinate. A voxel's key consists of a triplet of indexes, each representing its placement along the coordinate axes. By incrementing one of these indexes its possible to move along the axis to the neighbouring voxel. The function `getNeighbourKeys` retrieves the keys belonging to the 26 neighbouring voxels of a node. The function implementation can be seen in Listing 4 in A.4.

3.2 Search algorithm

The implementation of the search algorithm combines two data structures to manage the relevant information. One being an OctoMap, modelling the terrain to be navigated. The OctoMap is the source of information for if a point is pathable and for calculating distances between points.

The other data structure is a map storing pointers to a custom node struct, called `PointNode`. This structure contains information about a point that relevant to the search algorithm. The `PointNode` struct keeps track of values used in the evaluation function, along with what node was previously visited to reach the current point. It also stores a flag marking a point as pathable or not. Its implementation can be seen in Listing 5 in Appendix A.5.

Whenever a new OctoMap voxel is chosen for expansion a corresponding `PointNode` object is instantiated and a pointer to it is stored in the map. The access key used in the map is the same as the OctoMap's voxel key. If the expanded voxel has been traversed before, i.e. there already exists a node in the map with the same key as the voxel, the corresponding `PointNode` is retrieved from the map instead of being re-instantiated.

Pointers to the instantiated node objects are also stored in the search algorithm's frontier and explored sets. Where the frontier is sorted by the evaluation function.

If the goal is located, the `PointNode` corresponding to that point is returned. By using the `parrentKey` member of the `PointNode` struct it is possible to iterate through all points in the path leading to the goal. By converting the keys of all points in this path to the coordinate of the corresponding voxels, the optimal path can be generated.

3.3 Heuristic calculation

The Euclidean distance heuristic is calculated using an implementation of the Pythagorean theorem. It's code can be seen in Listing 3 in Appendix A.3.

The implementation of the diagonal distance heuristic can be seen in Listing 2 in Appendix A.2. Here, the distance D_1 is defined as the resolution of the OctoMap. This is the distance to a neighbouring voxel found by incrementing the key in one direction, moving along only one axis. The distance $D_2 = \sqrt{2}D_1$ is the distance to a diagonal neighbour found by incrementing the key in two directions. $D_3 = \sqrt{3}D_1$ is the distance to a neighbour found by incrementing the key in all directions.

3.4 Collision detection

When creating a long-range path planner for a fixed-wing UAVs it was important to make sure the path kept a safe distance from any surrounding terrain. This is done by making sure none of the voxels contained in a volume around the expanded point are marked as occupied. By using the OctoMap framework's bounding box iterators it was possible to loop through all voxels contained in a specified volume around a point. How this is implemented can be seen in Listing 6 in Appendix A.6.

The bounding box is set to have a 50 meter radius around the selected point.

This check is preformed whenever a new voxel is expanded and a new `PointNode` is instantiated. A traversability flag in the `PointNode` object is set depending on the result of this check. This way, if the point is expanded again, the check does not need to be re-evaluated.

If a `PointNode` is marked as not traversable the corresponding point is disregarded by the search algrorithm.

4 Simulations and Results

Searching for an optimal solution to a problem where the number of states is massive can be very hard. So is the case for optimal path planning using an OctoMap to precisely model a vast terrain. By using weighted A* it is possible to sacrifice some path optimality and in return relax the problem, as described in Section 2.2.3, making the search less demanding.

All mentions of optimality in this paper refer to shortest distance possible.

This chapter will present results of how optimal a path it was possible to achieve using the path planner, while still not demanding immense computing resources. This is done by testing the planner in a relevant simulation environment. The simulation environment is described in the following section.

4.1 Simulation environment

To test the capabilities of the path planner it was desirable to run simulations in a real world environment with diverse terrain. The Norwegian west coast, consisting of fjords and mountains, could serve this purpose. More specifically the municipality of Sykkylven and the surrounding area was chosen as a simulation environment.

Elevation data for the Sykkylven area was collected from `hodedata.no` and converted to a point cloud modelling the terrain. An elevation map of the selected area can be seen in Figure 7. An octree was created by ray-casting from a ceiling, set to the highest Z value in the data set plus 500m, down to the each point in the cloud. Several OctoMap environments were created with different resolutions. A part of the Sykkylven environment modelled with an OctoMap with a resolution of 100 meters can be seen in Figure 8.

The start and goal points were the same for all simulations. The start point was at (1250m, 2250m, 750m) and the goal at (49750m, 11750m, 750m). These points are relative to the bounding box of the octree. The distance between these points is 49421.7m.

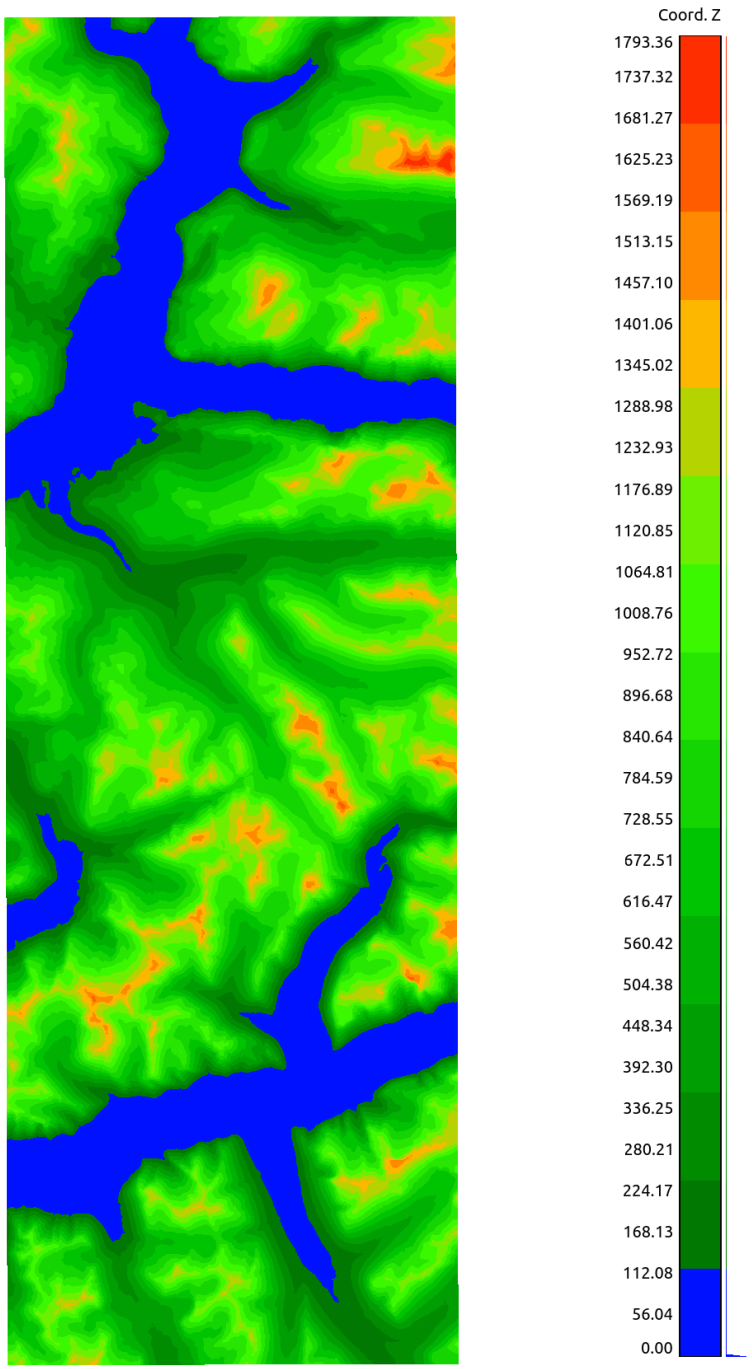


Figure 7: Elevation map of the Sykkylven environment. The area is roughly 50000 × 17000 meters with the lowest point at sea level and the highest peak at around 1793 meters. Visualization is done using CloudCompare.

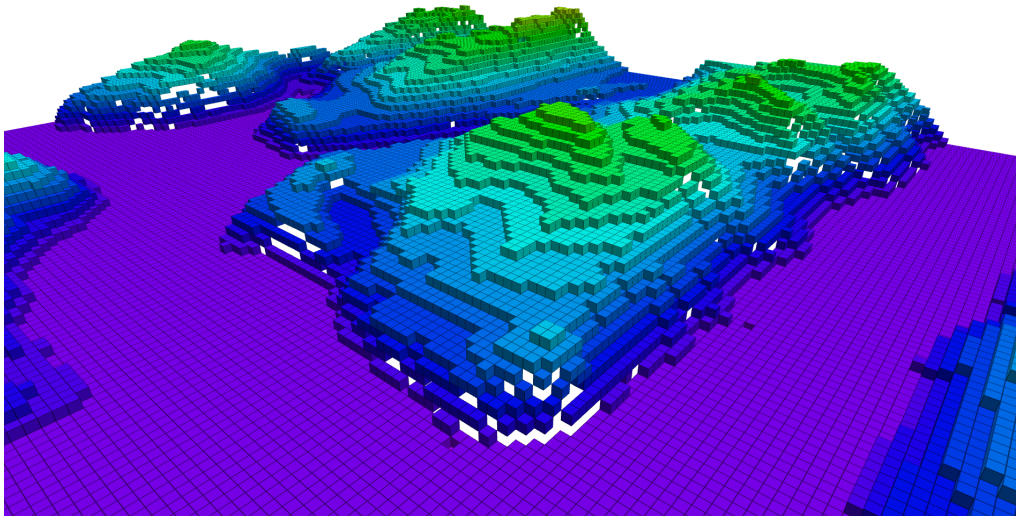


Figure 8: Example of how the Sykkylven environment is modelled using an OctoMap with minimum voxel size of 100 meters. Visualization is done using RViz.

4.2 Reduction of heuristic admissibility

To evaluate the performance of weighted A* search, a series of tests were run using the two different heuristic functions, the diagonal distance and the Euclidean distance. This was done at four different resolutions: 25, 50, 100 and 300 meters, varying the admissibility constraint parameter ϵ.

The evaluation criteria for the simulations were the number of nodes explored by the search and the total distance of the resulting path.

Search time is also included to get an idea of how the admissibility constraint affects execution timings. However, these simulations were not run on a dedicated computer, but on a computer with a general purpose operating system, leading to inconsistent run times if the program execution was affected by system overhead.

For the simulations run at finer resolutions the search time grew to large and the tests were not able to finish within a reasonable time frame. These results will therefore try to indicate how good a path we can achieve, given limited computing resources, using the high resolution environments.

All results will be presented as raw data in tables, one for each heuristic, and a combining graph representation to better compare their performance. Each subsection will also include a small summary of what is noteworthy before further discussing the results in the next chapter.

4.2.1 Simulations with 300 meter resolution

ϵ	Nodes explored	Path length (m)	Search time (s)
0	28 004	52 657.8	443.78
0.01	26 304	52 657.9	353.76
0.03	22 124	52 657.9	268.26
0.05	15 443	52 657.9	147.08
0.06	11 642	52 715.7	94.06
0.07	8 731	52 773.5	70.53
0.08	5 416	52 906.4	32.83
0.09	2 344	52 906.4	9.07
0.1	755	52 906.4	1.46
0.2	175	52 906.4	0.15
0.5	161	52 935.3	0.15
1	161	52 848.6	0.12
20	161	53 461.3	0.18
50	161	54 426.5	0.14
100	161	56 050.5	0.14

Table 1: Results for simulations using the Euclidean distance in an environment with minimum voxel size 300 meters.

ϵ	Nodes explored	Path length (m)	Search time (s)
0	5 548	52 657.9	29.68
0.01	1 873	52 686.8	3.33
0.03	551	52 935.3	0.52
0.05	372	52 964.2	0.28
0.06	379	52 964.2	0.29
0.07	341	53 594.1	0.26
0.08	309	53 594.1	0.22
0.09	256	53 594.1	0.18
0.1	234	53 594.1	0.17
0.2	183	54 120.1	0.13
0.5	163	54 923.5	0.12
1	161	55 172	0.12
20	161	56 050.5	0.12
50	161	56 050.5	0.12
100	161	56 050.5	0.12

Table 2: Results for simulations using the diagonal distance in an environment with minimum voxel size 300 meters.

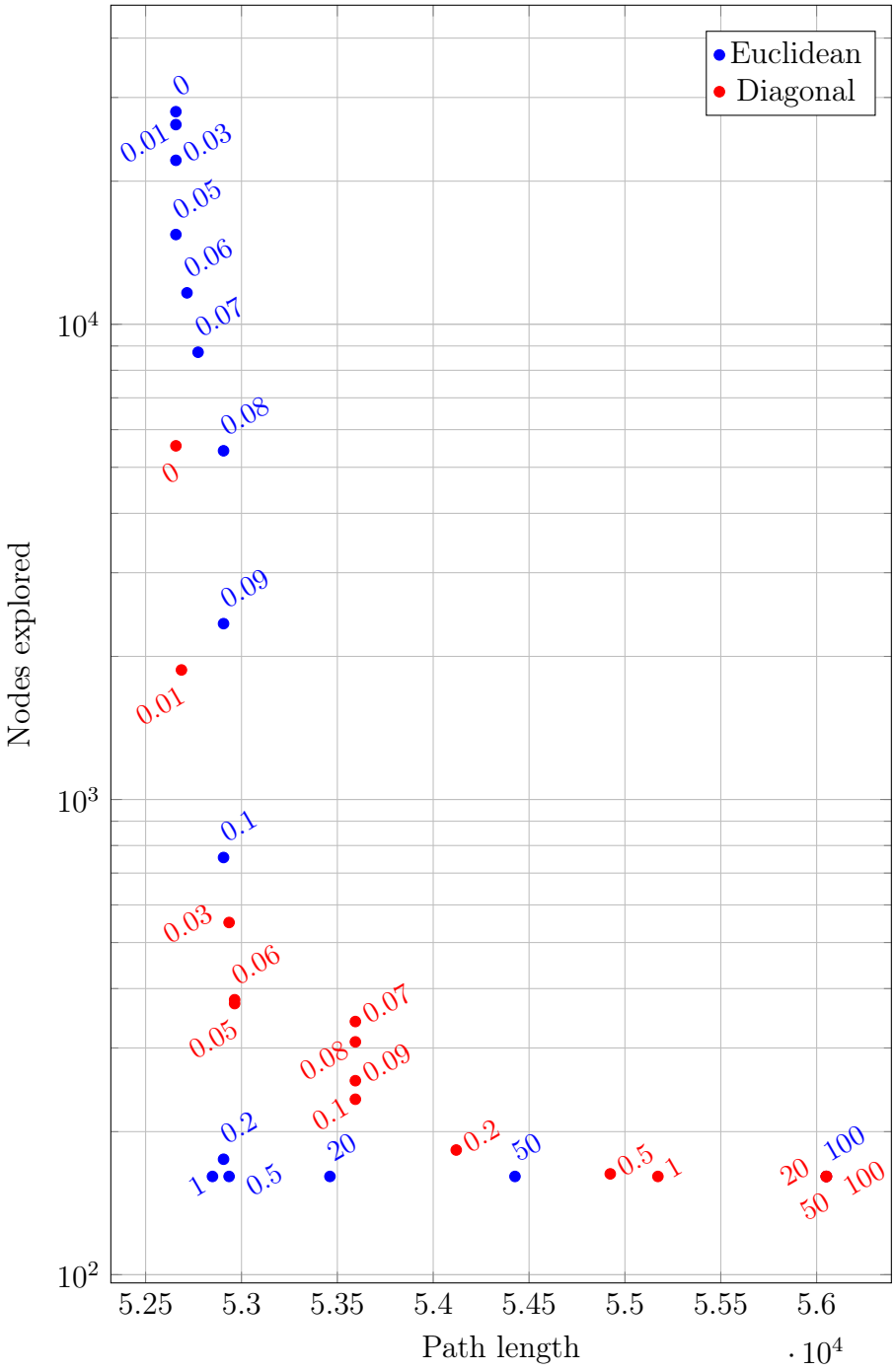
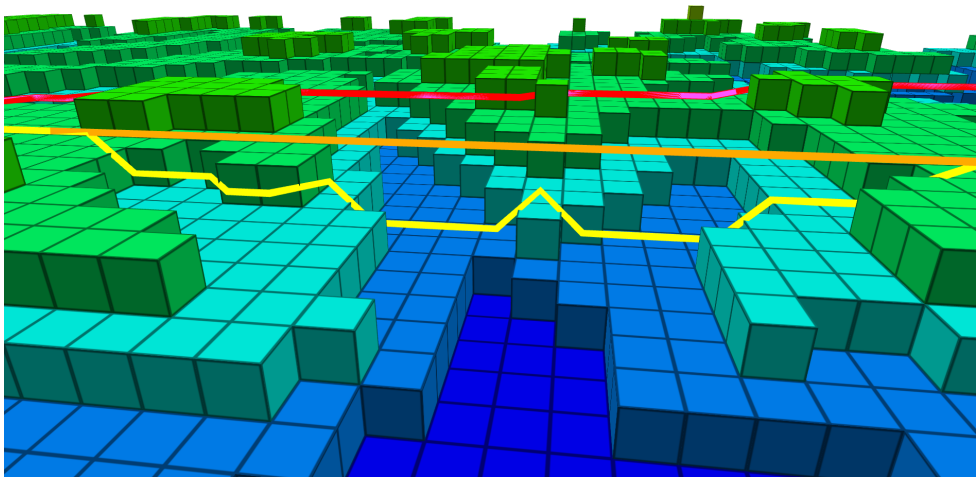
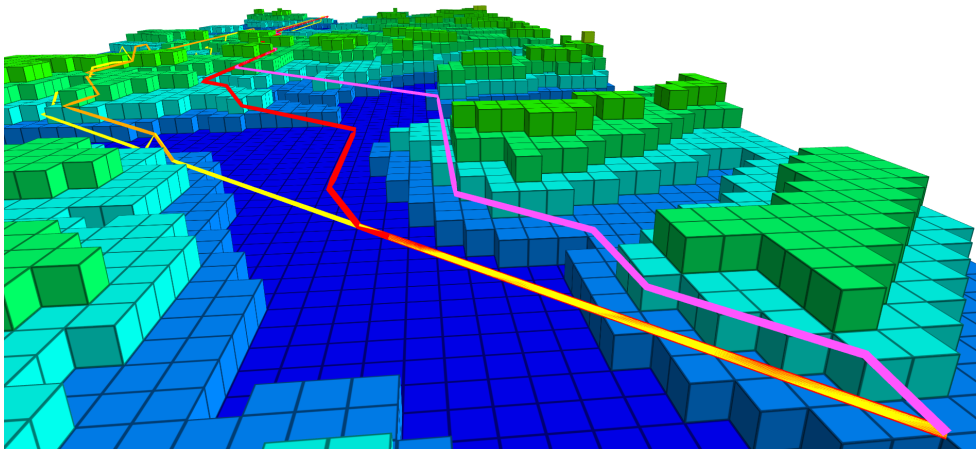


Figure 9: A comparison of the results from Table 1 and 2. Euclidean distance is in blue and diagonal distance is red. Each point is marked with the ϵ-value used for that simulation.



(a) The yellow path is an illustrative example of a path found by a search with greedy characteristics. The path drops in elevation quickly between terrain peaks due to the goal being at a lower position on the Z axis.



(b) Paths around the start point illustrating the divergence of the optimal paths. Due to how movement is implemeted in the 3D grid several paths can be optimal.

Figure 10: Illustrations of paths found in the environment with 300 meter resolution. Red and pink paths are the optimal paths found using diagonal and Euclidean distance, respectively. Yellow and orange paths are found with $\epsilon = 20$ using diagonal and Euclidean distance, respectively. Visualization is done in Rviz.

Summary

Using an environment resolution of 300 meters it was possible to produce optimal results ($\epsilon = 0$) using both heuristic functions. All the results found at this resolution can be seen in Figure 9 and Table 1 and 2.

The optimal path found with each heuristic are of roughly the same length, and the small deviation in distance between them is probably due to a rounding error in floating point arithmetic. The paths are illustrated in Figure 10. The optimal paths diverge in the beginning, but merge together as they encounter the first major terrain.

Comparing the performance of the heuristics it is obvious that using diagonal distance will yield optimal results while exploring fewer nodes. It does, on the other hand, have a quicker drop off in path optimality as ϵ increases.

With $\epsilon = 1$ both search variants explore the same number of nodes, but the one using diagonal distance is several hundred meters longer than the one using Euclidean distance. Increasing ϵ further it becomes clear that using the diagonal distance makes the search behave more greedy faster than using the Euclidean distance. This can be seen in Figure 10(a), where the search using the diagonal distance looses elevation quickly between terrain peaks.

4.2.2 Simulations with 100 meter resolution

ϵ	Nodes explored	Path length (m)	Search time (s)
0.07	160 129	52 864.6	30 528.1
0.08	92 863	52 928.2	11 002.5
0.09	26 823	52 928.2	993.51
0.1	6 119	52 928.2	63.08
0.2	547	52 947.5	1.35
1	485	52 880	1.28
10	485	53 064.9	1.43
20	485	53 406	1.54
50	485	54 658.2	1.45
100	485	55 371	1.28

Table 3: Results for simulations using the Euclidean distance heuristic in an environment with minimum voxel size 100 meters. Producing results for $\epsilon > 0.07$ was not possible within a reasonable time frame. Some results are not included due to little to no variation.

ϵ	Nodes explored	Path length (m)	Search time (s)
0	87 765	52 689.3	11 968.7
0.01	30 708	52 874.3	956.02
0.02	17 768	52 893.6	273.54
0.03	5 586	53 011	34.23
0.04	7 174	53 040	48.04
0.05	3 649	53 167.1	15.87
0.06	2 719	53 259.6	10.29
0.07	2 145	53 240.3	7.34
0.09	1 972	53 488.8	6.14
0.1	1 655	53 610.2	4.85
0.2	831	54 053.3	2.06
0.3	698	54 301.9	1.85
0.4	626	54 477.2	1.57
0.5	572	54 633.2	1.44
1	503	54 999.3	1.26
5	486	55 199.6	1.21
10	485	56 224.5	1.23

Table 4: Results for simulations using the diagonal distance heuristic in an environment with minimum voxel size 100 meters.

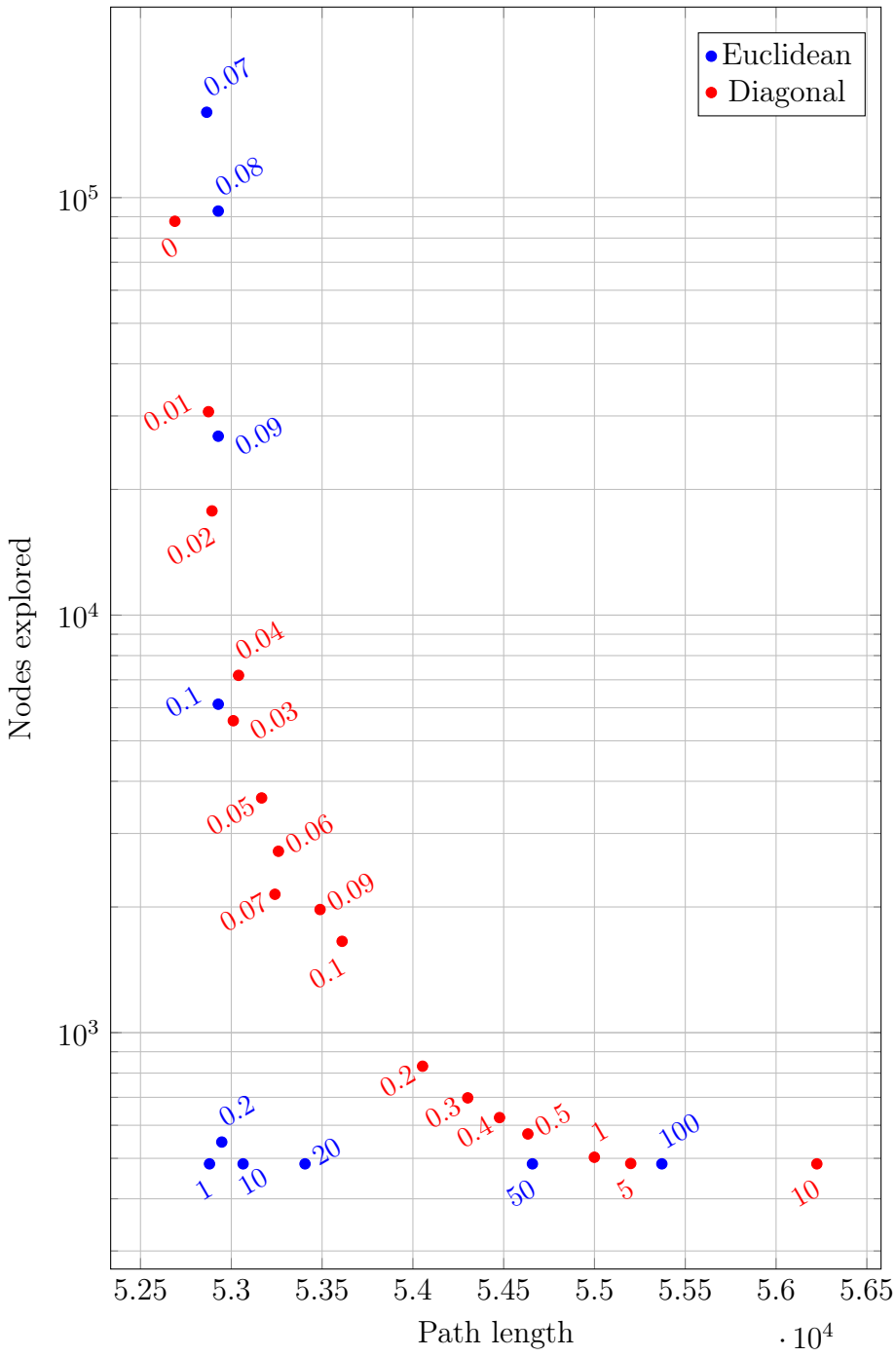


Figure 11: Comparison of the results from Table 3 and 4. Euclidean distance is in blue and diagonal distance is red. Each point is marked with the ϵ-value used for that simulation.

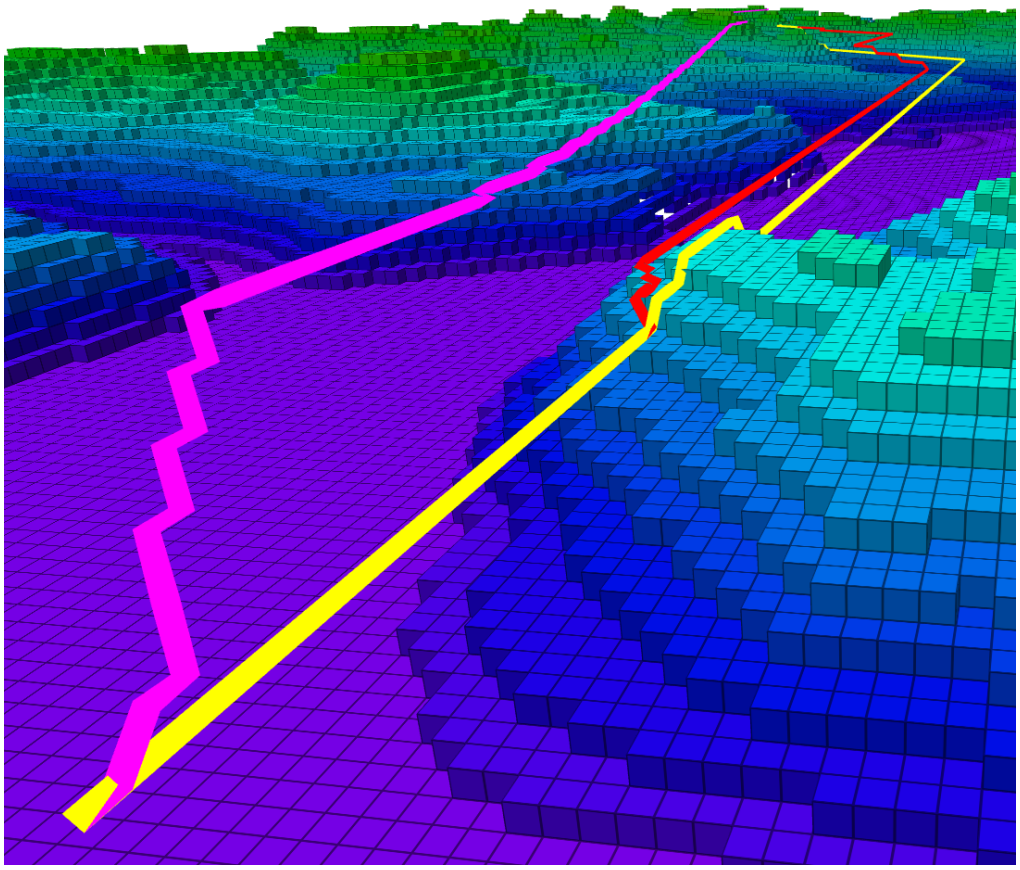


Figure 12: Illustration comparing paths found using both diagonal and Euclidean distance. The pink path is found using the Euclidean heuristic with $\epsilon = 0.07$. The red path is an optimal path found using the diagonal distance. The yellow path is found using the diagonal distance with $\epsilon = 0.01$.

Summary

The results from simulations done in an environment with a minimum voxel size of 100 meters can be seen in Figure 11 and Table 3 and 4.

The search variants behave similar to what is described in the previous section. The difference now, however, is that is was not possible to generate results using the Euclidean distance heuristic with $\epsilon < 0.07$ within a reasonable time frame. The simulation ran with $\epsilon = 0.07$ used almost 8.5 hours to finish.

Using the diagonal distance it was possible to find an optimal path. Comparing it to the best path found using Euclidean distance we see that

the optimal path is about 200 meters shorter.

Comparing sub-optimal results we can see from Figure 12 that using the Euclidean distance gives a path that maintains altitude towards the goal. Using the diagonal distance the path quickly converges to a straight line towards the goal. However, this straight line is obstructed so the path needs to regain elevation to reach the goal.

4.2.3 Simulations with 50 meter resolution

ϵ	Nodes explored	Path length (m)	Search time (s)
0.09	161 697	52 896.4	35 966.6
0.1	34 487	52 896.4	1 797.08
0.11	11 738	52 891.6	191.35
0.12	4 772	52 896.4	49.34
0.13	1 940	52 896.4	10.48
0.15	1 322	52 901.3	5.81
0.2	1 097	52 874.3	6.11
0.5	970	52 928.2	5.78
1	970	52 848.2	5.38
20	970	53 393.5	6.24
50	976	54 746.3	6.16
100	975	55 348.7	6.06

Table 5: Results for simulations using the Euclidean distance heuristic in an environment with minimum voxel size 50 meters. Results for $\epsilon < 0.09$ was not possible to produce within a reasonable time frame.

ϵ	Nodes explored	Path length (m)	Search time (s)
0.01	224 199	52 859.8	62 758.9
0.02	127 512	52 888.7	19 278.6
0.03	39 295	53 025.5	2 134.81
0.04	49 498	53 066.9	3 684.09
0.05	26 429	53 301	1 002.67
0.06	16 658	53 342.4	330.58
0.08	13 626	53 425.2	250.95
0.09	12 473	53 517.7	200.52
0.1	11 148	53 637.2	185.49
0.2	3 698	54 058.2	27.53
0.3	2 492	54 270.1	15.68
0.4	2 097	54 550.4	12.13
0.5	1 666	54 699.2	9.66
0.6	1 455	54 740.6	8.34
0.8	1 289	54 996.8	7.53
1	1 127	55 096.9	6.33
5	971	55 353.2	5.54
10	970	56 229.3	5.47

Table 6: Results for simulations using the diagonal distance heuristic in an environment with minimum voxel size 50 meters. Results for $\epsilon = 0$ was not possible to produce within a reasonable time frame.

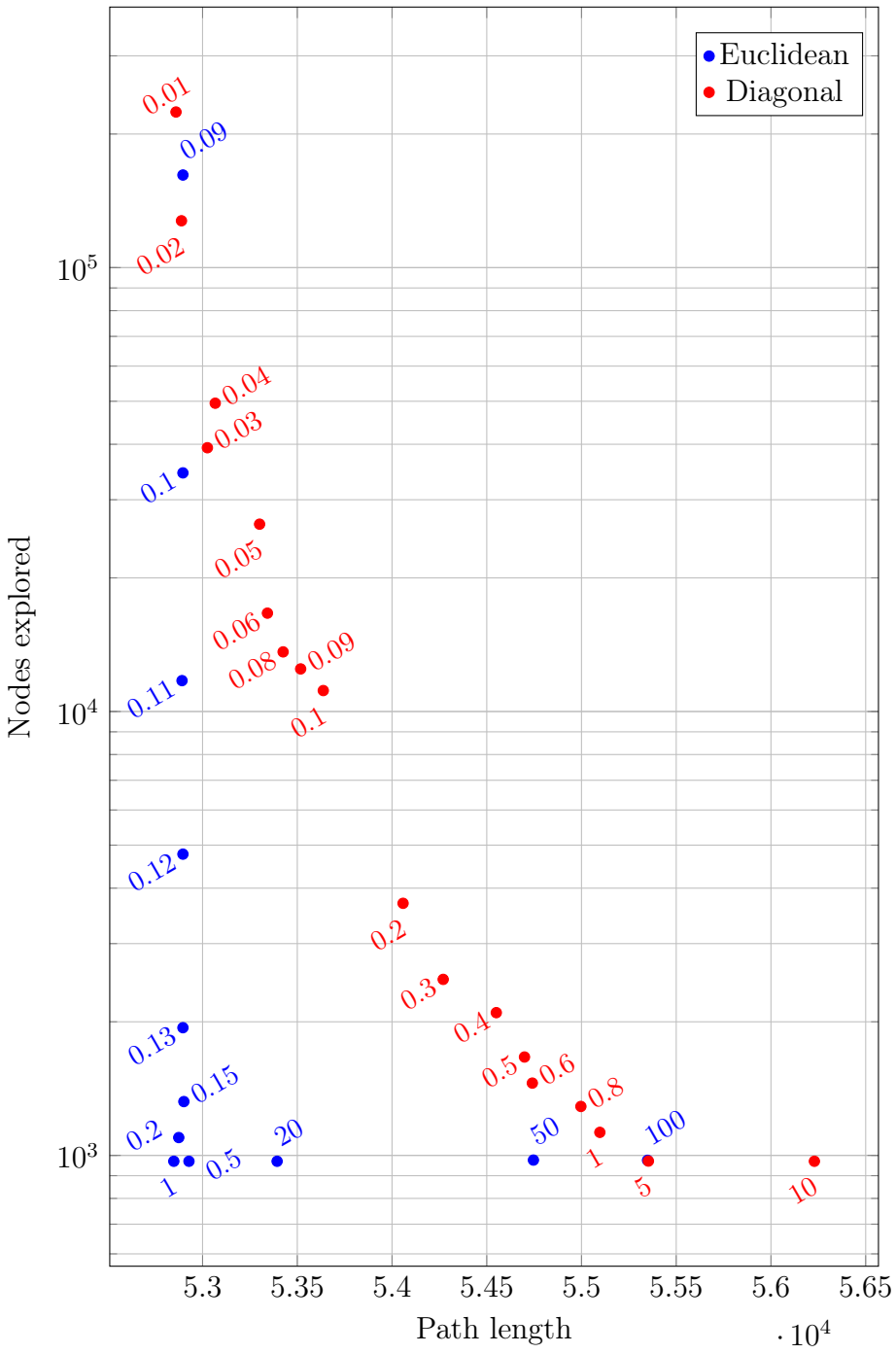


Figure 13: Comparison of the results from Table 5 and 6. Euclidean distance is in blue and diagonal distance is red. Each point is marked with the ϵ-value used for that simulation.

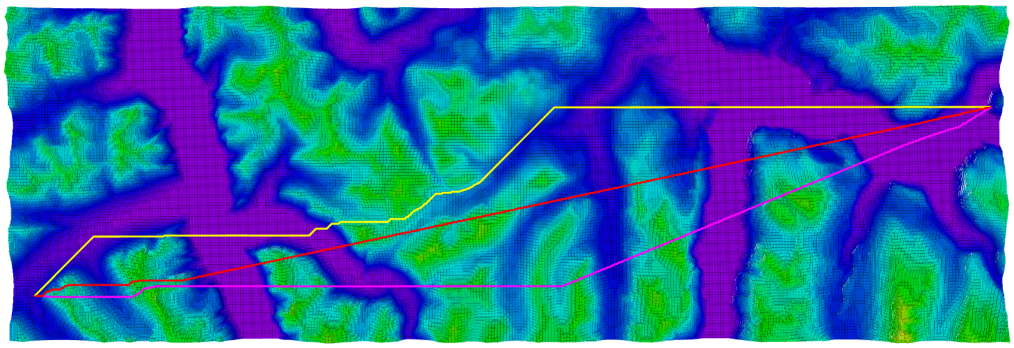


Figure 14: Aerial view of the paths found in the environment with 50 meter resolution. The red path has $\epsilon = 1$ and uses the Euclidean distance heuristic. The pink path has $\epsilon = 0.09$ and uses the Euclidean distance heuristic. Both paths found using the Euclidean heuristic maintains a continuous altitude until close to goal. The yellow path has $\epsilon = 0.01$ and uses the diagonal distance heuristic. This path lays close to the terrain, if the terrain has a higher elevation than the goal.

Summary

The results from simulations done in an environment with a minimum voxel size of 50 meters can be seen in Figure 13 and Tables 5 and 6.

When running simulations in this environment neither of the search variants were able to find an optimal solution within a generous time frame. The shortest path found using diagonal distance heuristic explored over 200000 nodes, resulting in a search time of almost 17.5 hours!

The shortest path found in this resolution was discovered using the Euclidean distance heuristic with $\epsilon = 1$. Looking at the results from simulations at other resolutions this combination of heuristic and ϵ-value has given a short path length while exploring a small set of nodes. In Figure 14 we see that this path moves straight towards goal with minimal change of heading.

4.2.4 Simulations with 25 meter resolution

ϵ	Nodes explored	Path length (m)	Search time (s)
0.11	82 954	52 896.4	9 303.83
0.12	29 997	52 896.4	1 191.06
0.13	8 989	52 901.3	94.11
0.14	7 204	53 002.4	62.91
0.15	4 485	52 903.7	34.02
0.16	3 885	52 908.5	30.52
0.17	3 249	52 908.5	27.11
0.2	2 535	52 910.9	24.02
1	1 940	52 834.8	22.03
5	1 940	52 897.8	25.58
10	1 940	53 000.4	25.75
20	1 940	53 337.6	30.82
50	1 955	54 807.4	29.19
100	1 961	55 597.8	27.92

Table 7: Results for simulations using the Euclidean distance heuristic in an environment with minimum voxel size 25 meters.

ϵ	Nodes explored	Path length (m)	Search time (s)
0.06	118 774	53 340	16 168
0.07	93 448	53 404.5	10 024.2
0.09	86 171	53 593.4	8 374.54
0.1	81 603	53 681	11 307.1
0.2	23 150	54 103.2	939.36
0.3	13 751	54 339.7	294.76
0.4	9 325	54 585.8	148.71
0.5	6 665	54 718.7	116.56
0.6	5 144	54 698	88.14
0.7	4 241	54 727.4	59.27
0.8	4 089	55 043.3	55.93
0.9	3 510	55 114.1	45.23
1	3 092	55 161.8	44.13
5	1 978	55 648.6	29.06
10	1 958	56 586.8	26.59

Table 8: Results for simulations using the diagonal distance heuristic in an environment with minimum voxel size 25 meters.

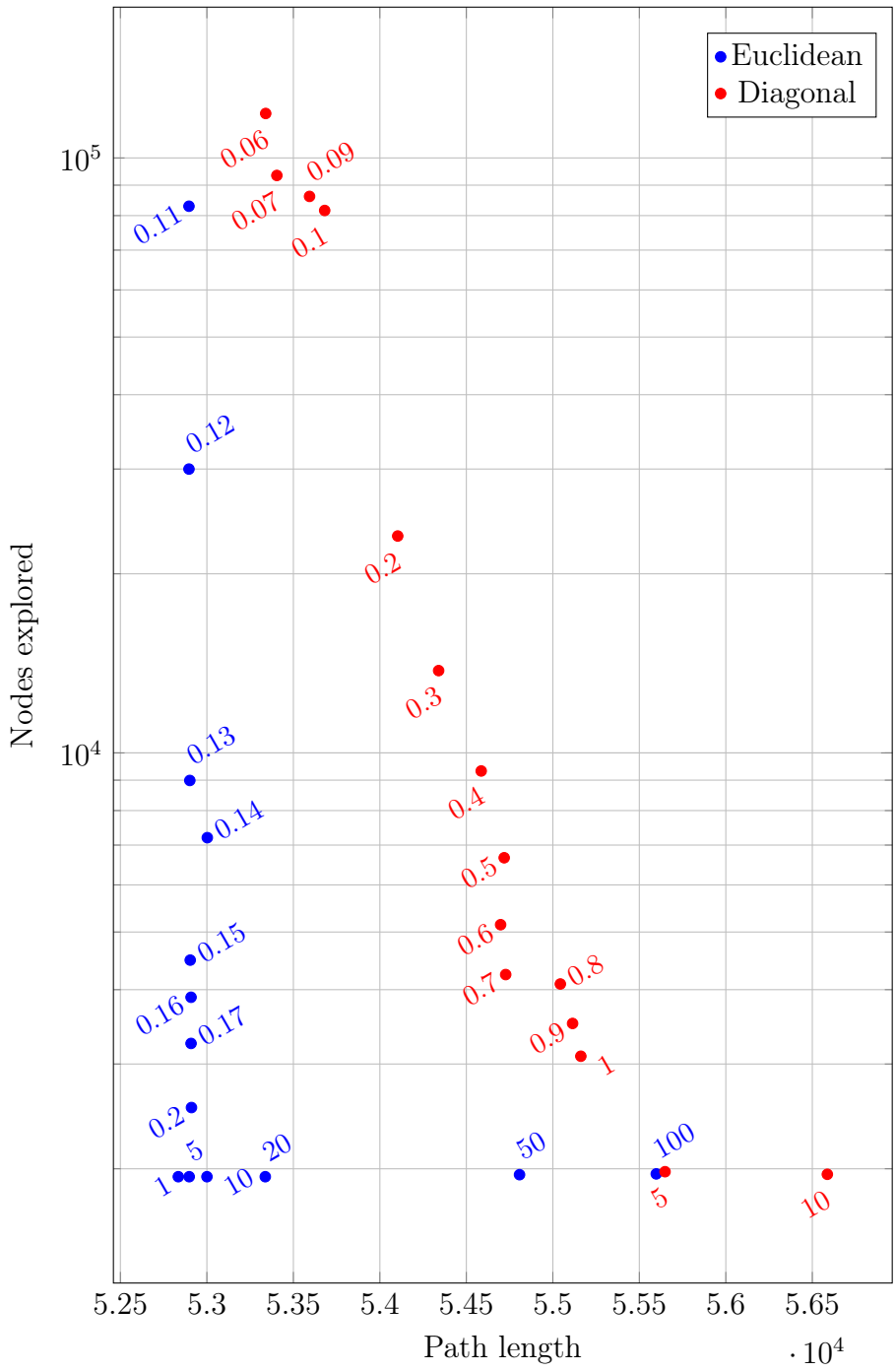


Figure 15: Comparison of the results from Table 7 and 8. Euclidean distance is in blue and diagonal distance is red. Each point is marked with the ϵ-value used for that simulation.

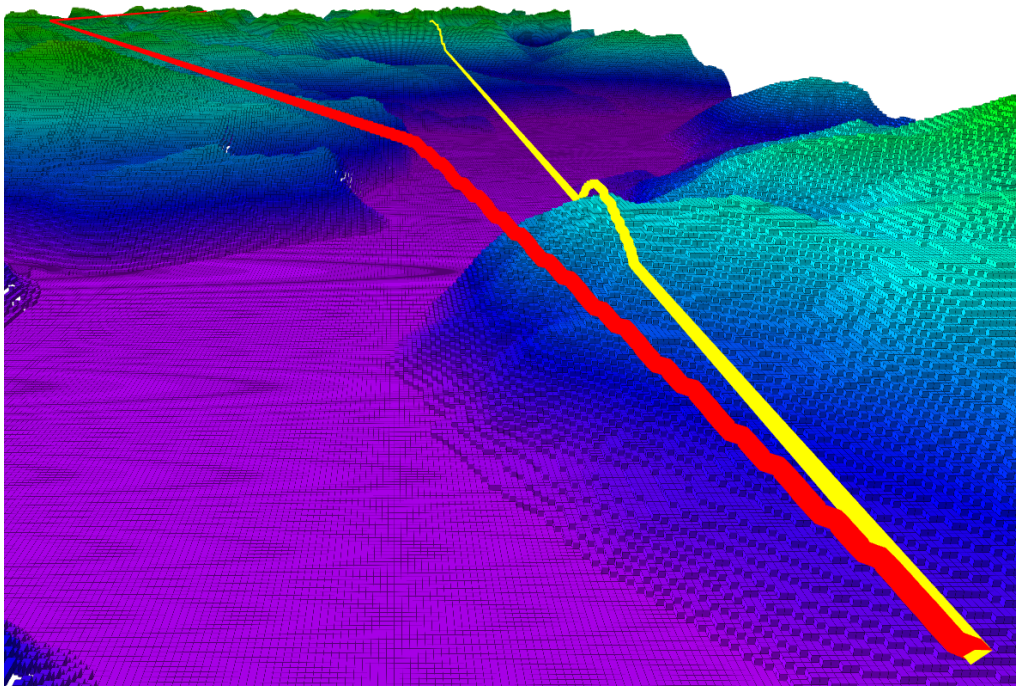


Figure 16: Illustration comparing sub-optimal paths found using the two heuristics in an environment with 25 meter resolution. The red path uses the Euclidean distance with $\epsilon = 0.11$, while the yellow path uses the diagonal distance with $\epsilon = 0.06$.

Summary

The results from simulations done in an environment with a minimum voxel size of 25 meters can be seen in Figure 15 and Tables 7 and 8.

When using the environment with 25 meter resolution neither got close to finding an optimal solution. These results does highlight how using the Euclidean distance outperforms the diagonal distance when the results can be sub-optimal.

Figure 16 illustrates a similar scenario to the one in Figure 12: the path found using the Euclidean distance maintains its altitude after traversing high terrain before descending when closing in on the goal point. The diagonal distance converges to a straight line towards the goal, but is hindered by obstructiong terrain.

5 Discussion

All results presented in the previous chapter were found by simulating in only one scenario. The aim of these simulations were merely to get an idea of how the different heuristics affect the behavior of the planner.

With this in mind the following sections will further discuss the results presented in the previous chapter. We will compare the performance of the two heuristic functions and give an intuitive explanation of their behavior. This can give information that could prove useful for further development of the path planner.

5.1 Search behaviour for optimal results

Looking at the results from the simulations presented in Sections 4.2.1 and 4.2.2 (coarse resolution simulations) we see that using the diagonal distance heuristic decreases the search size when only allowing optimal results. This is due to the diagonal distance being a more accurate estimate of the actual movement cost between two nodes in our environment. In fact, if the path between to nodes is unhindered the diagonal distance between them is an exact estimate of the movement cost.

The movement cost is dependent on how neighbour finding and movement is implemented in the search algorithm. In our case: allowing only movement to each neighbouring voxel in the 3D grid, matches the diagonal distance behavior.

The Euclidean distance, on the other hand, will underestimate the movement cost between two nodes as long as movement in more than one direction is necessary to traverse between them. This underestimation of $h(n)$ will cause the search to explore more nodes.

As an example on how an underestimating heuristic increases the search size imagine $h(n) = 0$, a heuristic that always will underestimate, giving $f(n) = g(n)$ in (3). This will make the search behave like Dijkstra's algorithm. This gives an optimal result, but does so in an uninformed manner, increasing the number of nodes explored drastically.

So if the path needs to be optimal, with regards to distance, using the diagonal distance heuristic might be a good choice. However, if the path is to be used as a basis for a UAV trajectory the need for high-resolution terrain models is apparent. Finding an optimal solution in such environments has proven to be hard, using either heuristic, due to the size of the environment.

5.2 Search behaviour when allowing sub-optimality

Even though the diagonal distance performs better for optimal results, the case is not the same when allowing sub-optimality. From Table 4 we can see that $\epsilon = 0.01$ increases the path length by over 200 meters. Increasing the bound even further, at $\epsilon = 1$, the diagonal distance heuristic gives a sub optimal path that is 2300 meters longer than the optimal result. Comparing results with the same ϵ from Table 3 (Euclidean distance) the path is only about 200 meters longer than than the optimal path.

Investigating the sub-optimal paths, illustrated in Figure 12, 14 and 16, the different behavior of the two search variants is apparent. Using the Euclidean distance the path prefers to maintain it's heading until the direction towards the goal has changed significantly enough. While the trajectories found using the diagonal distance quickly converges to a direction in a straight line towards goal.

This behaviour can again be explained by how movement is defined in the 3D grid and how the heuristics are influenced by relaxation of admissibility. When the heuristic is relaxed the search will prioritize moving to nodes that minimizes $h(n)$.

For the diagonal distance heuristic this will always be movement to a diagonal neighbour if the current node is not in a straight line towards goal. This is intensified when the admissibility of the heuristic is relaxed, leading to the greedy behavior shown in Figure 10(a).

The behavior of the Euclidean distance is due to how, when far from goal, the estimate is dominated by one axis. Imagine a vector $[x, y, z]$, where $x \gg y$ and $x \gg z$, the length of the vector (Euclidean distance) will be $\sqrt{x^2 + y^2 + z^2} \approx x$. So as long as a move decreases the dominating coordinate the decisive factor for what neighbour to explore next is determined by the movement cost. Moving to non-diagonal neighbours have the smallest movement cost and is therefore preferred.

By relaxing the admissibility of the heuristic the evaluation function becomes more sensitive to nuances in Euclidean distance between neighbours. Looking at Figure 14, this explains why the path found using $\epsilon = 1$ changes its heading faster than $\epsilon = 0.09$.

5.3 Using OctoMap as a terrain model

The aim of this paper is to investigate the possibilities for using an A* path planner in conjunction with the OctoMap framework for long-range UAV trajectory planning. The following section will therefore discuss some experiences made while implementing the presented system.

The OctoMap framework has proven to be an intuitive representation of a 3D environment. It models both free and occupied volumes in a grid, making the implementation of an A* search algorithm not too difficult.

However, using the framework as presented in this project does not fully utilize all it's functionalities. OctoMap is capable of online terrain modelling, and is often used for collision avoidance in dynamic environments. Using it to represent a static environment is of course not wrong, but a simpler terrain model might have been just as good.

One possibility might have been to formulate the problem so it could be solved using mixed integer linear programming. For this purpose a simple elevation model can be suitable, modelling the terrain as a constraint.

Alternative environment models will be discussed further in the following chapter.

Another point to mention is how the resolution of an OctoMap not only affects search performance, but also has an impact on the pre-processing pipeline to generate said OctoMap. Although, this is not a major concern as the pre-processing is done offline beforehand.

6 Further Work

. The following chapter will present some ideas for what topics to explore for further development of the path planner.

6.1 Implementing a UAV model

The current solution of neighbour finding, by expanding all 26 surrounding neighbours to a center voxel, does not reflect the behavior of a fixed-wing UAV. By implementing a model for the UAV dynamics, the neighbour finding functionality could instead return voxels in a conical shape in front of the aircraft, dependent on it's heading and turning radius.

The method presented as an improved A* algorithm by Xia et al., 2009 suggest an implementation. Their results show a reduction of both path distance and number of explored nodes. Their implementation does not use the OctoMap-framework, but they do model terrain in a grid structure to be searched in by A*.

An implementation such as this would require choosing the OctoMap resolution with the UAV's turning radius in mind. A fine resolution might be necessary for this functionality to make sense, but it could result in a path well suited as a UAV trajectory basis.

6.2 Alternative environment models

The following section will present some alternative methods for modelling terrain to be used for path planning.

Visibility graph

Anastasios and Zompas, 2016 presents a method of using the OctoMap framework to create a visibility graph of the modelled environment. This reduces the set of nodes in the search, by instead of planning a path through adjacent voxels, the idea is to travel from one obstacle to another one. See Figure 17 for an illustrative image of what a visibility graph represents. This makes traversing large open areas trivial as they are modelled as only one edge in the graph.

The implementation presented in Anastasios and Zompas, 2016 calculates what voxels in the octree are obstacle corners, and sub-samples all the corner candidates to further minimize the node set. They then use the ray casting feature of the framework to create a visibility graph of the environment.

Their results show that the visibility graph implementation reduces the search time and the number of explored nodes.

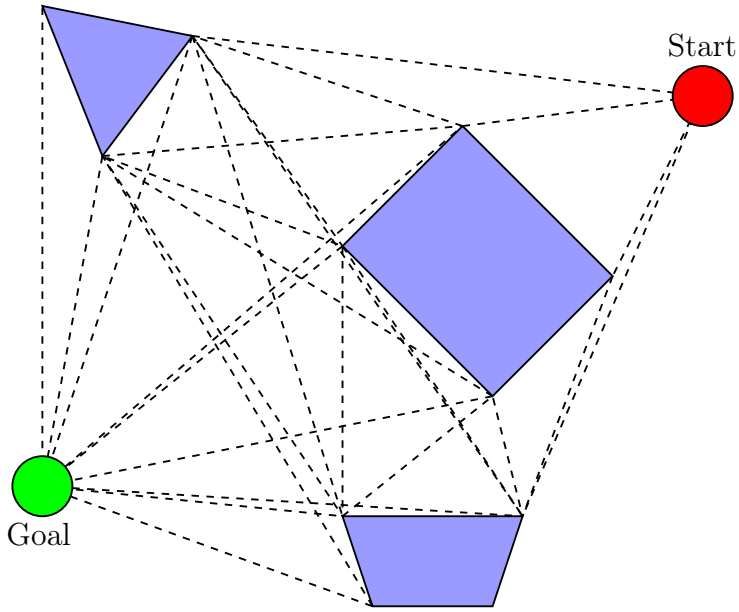


Figure 17: Example of a visibility graph. Each node in the graph represents a corner of an obstacle. En edge is inserted between the nodes if they are in clear view of each other.

Smoothed safe surface

Xia et al., 2009 creates a search environment by running terrain data through a smoothing algorithm $h(x, y)$ and adding a safe height h_c. A synthetic terrain is then given as:

$$H(x, y) = h(x, y) + h_c.$$

This smoothing method is based on a Chinese paper called *Smoothing Approach to Digital Terrain for Low Altitude Penetration*. The paper is unfortunately not available without access to the Chinese National Knowledge Infrastructure (CNKI).

7 Conclusion

The OctoMap-framework is a viable method to model terrain, and implementing it with the A* algorithm is feasible. The resolution of the octree has a severe impact on performance, both for the search algorithm and for data pre-processing. What resolution is needed for UAV path planning may be of importance for further development of the planner.

Choosing the heuristic function in the A* algorithm is also of great importance for further development of the path planner. The simulation results presented in this paper give reason to believe that a weighted A* algorithm implemented with the Euclidean distance heuristic is a valid choice.

Results also show that searching for an optimal path using the diagonal distance yields a more effective search, but the drop-off in path optimality when allowing sub-optimal results is not wanted.

A Listings

A.1 Planner

Listing 1: Implementation of the search algorithm used in the simulations.

```
struct key_compare
{
  bool operator()(
     const OcTreeKey &k1,
     const OcTreeKey &k2) const
  {
    if (k1[0] == k2[0])
    {
      if (k1[1] == k2[1])
      {
        return k1[2] < k2[2];
      }
      else
      {
        return k1[1] < k2[1];
      }
    }
    else
    {
      return k1[0] < k2[0];
    }
  }
};

bool operator==(
    const shared_ptr<PointNode> &lhs,
    const shared_ptr<PointNode> &rhs)
{
  return *lhs == *rhs;
}

bool operator<(
    const shared_ptr<PointNode> &lhs,
    const shared_ptr<PointNode> &rhs)
{
  return *lhs < *rhs;
}

int main(int argc, char **argv)
{
  string tree_path;
  float epsilon = stof(argv[2]);
  point3d startp;
  point3d goalp;

  utils::readConf(argv[1], tree_path, startp, goalp);

  OcTree tree(tree_path);

  stringstream nodename;
  nodename << "planner_" << tree.getResolution();

  ros::init(argc, argv, nodename.str());
  ros::NodeHandle node_handle;
  ros::Rate rate(1);

  // cast goal and start points into their containing voxel centers
  OcTreeKey startk = tree.coordToKey(startp);
```

```
59    startp = tree.keyToCoord(startk);
60    OcTreeKey goalk = tree.coordToKey(goalp);
61    goalp = tree.keyToCoord(goalk);
62
63    // diagonal distances
64    float d1 = tree.getResolution();
65    float d2 = sqrtf(2.0) * d1;
66    float d3 = sqrtf(3.0) * d1;
67
68    // Astar begin
69    map<OcTreeKey, shared_ptr<PointNode>, key_compare> lookuptable;
70    set<OcTreeKey, key_compare> closed_list;
71    vector<shared_ptr<PointNode>> open_list;
72
73    shared_ptr<PointNode> startn =
74      make_shared<PointNode>(PointNode(startk));
75
76    startn->gval = 0.0;
77    // startn->hval = (1 + epsilon)
78    //   * utils::distance(startp, goalp);
79    startn->hval = (1 + epsilon)
80      * utils::diagonaldistance(startp, goalp, d1,d2,d3);
81    startn->fval = startn->hval + startn->gval;
82
83    open_list.push_back(startn);
84    lookuptable[startn->key] = startn;
85
86    int count = 0;
87
88
89    clock_t start;
90    double duration;
91
92    start = clock();
93
94    while (!open_list.empty() && ros::ok())
95    {
96      sort(open_list.rbegin(), open_list.rend());
97      shared_ptr<PointNode> currentn = open_list.back();
98      open_list.pop_back();
99      point3d currentp = tree.keyToCoord(currentn->key);
100
101     if (currentn->key == tree.coordToKey(goalp))
102     {
103       break;
104     }
105
106     closed_list.insert(currentn->key);
107
108     vector<OcTreeKey> neighbourkeys =
109       utils::getneighbourkeys(currentn->key);
110
111     for (auto neighbourk_it = neighbourkeys.begin();
112        neighbourk_it != neighbourkeys.end(); ++neighbourk_it)
113     {
114       OcTreeKey neighbourk = *neighbourk_it;
115
116       // is the node in the closed list
117       if (find(
118         closed_list.begin(),
119         closed_list.end(),
120         neighbourk) != closed_list.end())
121       {
122         continue;
123       }
```

```
124
125         point3d neighbourp = tree.keyToCoord(neighbourk);
126         shared_ptr<PointNode> neighbourn;
127
128         // if the node has not been expanded before
129         if (lookuptable.count(*neighbourk_it) == 0)
130         {
131           neighbourn =
132             make_shared<PointNode>(PointNode(neighbourk));
133
134           lookuptable[neighbourk] = neighbourn;
135
136           neighbourn->traversable =
137             !collisiondetect(tree, neighbourp);
138         }
139         // if the node has been expanded before
140         else
141         {
142           neighbourn = lookuptable[neighbourk];
143         }
144
145         OcTreeNode* n = tree.search(neighbourp);
146         if(n == NULL){
147           continue;
148         }
149
150         if (!neighbourn->traversable)
151         {
152           continue;
153         }
154
155         float tentative_gval = currentn->gval
156           + distance(currentp, neighbourp);
157
158         if (find(
159           open_list.begin(),
160           open_list.end(),
161           neighbourn) == open_list.end())
162         {
163           open_list.push_back(neighbourn);
164         }
165         else if (tentative_gval > neighbourn->gval
166           && neighbourn->gval != -1)
167         {
168           continue;
169         }
170
171         neighbourn->parrentkey = currentn->key;
172         neighbourn->gval = tentative_gval;
173
174         // neighbourn->hval = (1 + epsilon)
175         //   * distance(neighbourp, goalp);
176         neighbourn->hval = (1 + epsilon)
177           * diagonaldistance(neighbourp, goalp, d1,d2,d3);
178
179         neighbourn->fval = tentative_gval + neighbourn->hval;
180       }
181     }
182
183     duration = ( clock() - start ) / (double) CLOCKS_PER_SEC;
184
185
186     if (!ros::ok()) {
187       // ctrl+c signal given to ROS
188       return 0;
```

```
189     }
190
191     if (open_list.size() == 0) {
192       // no solution found
193       return 0;
194     }
195
196     // The following code appends the data
197     // about the found path to a output file.
198     ofstream outfile;
199     stringstream outname;
200     outname << "all_paths_res_" << tree.getResolution() << ".txt";
201
202     outfile.open(outname.str(), ios_base::app);
203     outfile
204     << epsilon << "  "
205     << closed_list.size() << "  "
206     << lookuptable[goalk]->gval << "  "
207     << duration
208     << endl;
209
210     OcTreeKey key = goalk;
211     outfile.close();
212
213
214     // The following code write the path found to a file
215     ofstream pathfile;
216     stringstream ss;
217     ss << "path_res_" << tree.getResolution()
218     << "_eps_" << epsilon << ".txt";
219     pathfile.open(ss.str());
220
221     while (key != startk)
222     {
223       point3d p = tree.keyToCoord(key);
224       pathfile << p.x() << " " << p.y() << " " << p.z() << endl;
225       key = lookuptable[key]->parrentkey;
226     }
227
228     // add the start coorinate also
229     point3d coord = tree.keyToCoord(key);
230     pathfile
231     << coord.x() << " "
232     << coord.y() << " "
233     << coord.z()
234     << endl;
235
236     pathfile.close();
237
238     return 1;
239 }
```

A.2 Diagonal Distance

Listing 2: Implementation of how the diagonal distance heuristic is calculated.

```
1  float diagonaldistance(
2      point3d start,
3      point3d end,
4      float d1,
5      float d2,
6      float d3) {
7    float dx = abs(start.x() - end.x()) / d1;
```

```
  float dy = abs(start.y() - end.y()) / d1;
  float dz = abs(start.z() - end.z()) / d1;
  float dmin = std::min({dx, dy, dz});
  float dmax = std::max({dx, dy, dz});
  float dmid = dx + dy + dz - dmin - dmax;

  return (d3 - d2) * dmin + (d2 - d1) * dmid + d1 * dmax;
}
```

A.3 Euclidean Distance

Listing 3: Implementation of how the euclidean distance heuristic is calculated.

```
float distance(point3d start, point3d end){
  float x = pow(end.x() - start.x(), 2.0);
  float y = pow(end.y() - start.y(), 2.0);
  float z = pow(end.z() - start.z(), 2.0);

  return sqrtf(x + y + z);
}
```

A.4 Get Neighbour Keys

Listing 4: Function for retrieving the keys of all neighbouring voxels to a current voxel.

```
vector<OcTreeKey> getneighbourkeys(OcTreeKey c_key){
  vector<OcTreeKey> neighbours;
  for(int xx = -1; xx < 2; ++xx) {
    for (int yy = -1; yy < 2; ++yy) {
      for (int zz = -1; zz < 2; ++zz) {
        if (xx == 0 && yy == 0 && zz == 0) {
          continue;
        }
        else {
          OcTreeKey new_key;
          new_key.k[0] = c_key.k[0]+xx;
          new_key.k[1] = c_key.k[1]+yy;
          new_key.k[2] = c_key.k[2]+zz;
          neighbours.push_back(new_key);
        }
      }
    }
  }
  return neighbours;
}
```

A.5 PointNode

Listing 5: The PointNode struct. The `gval`, `fval` and `hval` members are initialized to -1 functioning as an infinite value.

```
struct PointNode
{
  OcTreeKey key;
  OcTreeKey parrentkey;
  float gval, fval, hval = -1;
  bool traversable = true;
};
```

A.6 Collision Detection

Listing 6: Implementation of the collision detection check.

```
bool collisiondetect(OcTree &tree, point3d center)
{
  OcTreeNode* n = tree.search(center);
  if(n == NULL){
    return true;
  }
  if (tree.isNodeOccupied(n)) {
    return true;
  }

  point3d bbx_offset(50, 50, 50);
  for (OcTree::leaf_bbx_iterator
  it = tree.begin_leafs_bbx(center - bbx_offset, center + bbx_offset),
  end = tree.end_leafs_bbx();
     it != end; ++it)
  {
    if (tree.isNodeOccupied(*it))
    {
      return true;
    }
  }
  return false;
}
```

References

Anastasios, A (and) Zompas (2016). *Development of a Three Dimensional Path Planner for Aerial Robotic Workers*. Tech. rep. URL: https://pdfs.semanticscholar.org/900e/87b9c9228f755d8d37f9ee8f68532d0a0ffe.pdf.

Boon, M. A., A. P. Drijfhout, and S. Tesfamichael (2017). "COMPARISON OF A FIXED-WING AND MULTI-ROTOR UAV FOR ENVIRONMENTAL MAPPING APPLICATIONS: A CASE STUDY." In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-2/W6, pp. 47–54. ISSN: 2194-9034. URL: https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLII-2-W6/47/2017/.

Ebendt, Rüdiger and Rolf Drechsler (2009). "Weighted A* search – unifying view and application." In: *Artificial Intelligence* 173.14, pp. 1310–1342. ISSN: 0004-3702. URL: https://www.sciencedirect.com/science/article/pii/S000437020900068X.

Hornung, Armin et al. (2013). "OctoMap: An efficient probabilistic 3D mapping framework based on octrees." In: *Autonomous Robots* 34.3, pp. 189–206. ISSN: 09295593. URL: http://link.springer.com/10.1007/s10514-012-9321-0.

Pagnano, A., M. Höpf, and R. Teti (2013). "A Roadmap for Automated Power Line Inspection. Maintenance and Repair." In: *Procedia CIRP* 12, pp. 234–239. ISSN: 22128271. URL: https://linkinghub.elsevier.com/retrieve/pii/S2212827113006823.

Patel, Amit (1997). *Amit's A* Pages*. URL: http://theory.stanford.edu/~amitp/GameProgramming/.

Russell, Stuart and Peter Norvig (2010). *Artificial Intelligence - A Modern Approach (3rd ed)*. ISBN: 0136042597.

Xia, Li et al. (2009). "Path planning for UAV based on improved heuristic A* algorithm." In: *2009 9th International Conference on Electronic Measurement & Instruments*. IEEE, pp. 3–488. ISBN: 978-1-4244-3863-1. URL: http://ieeexplore.ieee.org/document/5274271/.