EXAMENSARBETE

# Path Planning with Weighted Wall Regions using OctoMap

Jerker Bergström

**Civilingenjör, Rymdteknik**
**2018**

Luleå tekniska universitet
Institutionen för system- och rymdteknik

LULEÅ TEKNISKA UNIVERSITET

Luleå University of Technology

# *Abstract*

Faculty of science and technology
Department of Computer Science, Electrical and Space Engineering, Signals and Systems.

Master of Aerospace Engineering

**Path Planning with Weighted Wall Regions using OctoMap**

by Jerker Bergström

In the work of the Control Engineering research group of the Department of Computer Science, Electrical and Space Engineering, Signals and systems at Luleå University of Technology a need had arisen for a path planning algorithm. The ongoing research with Unmanned Aerial Vehicles(UAVs) had so far been done with any complicated paths being created manually with waypoints set by the uses. To remove this labourious part of the experimental process a path should be generated automatically by simply providing a program with the position of the UAV, the goal to which the user wants it to move, as well as information about the UAV's surroundings in the form of a 3D map.

In addition to simply finding an available path through a 3D environment the path should also be adapted to the risks that the physical environment poses to a flying robot. This was achieved by adapting a previously developed algorithm, which did the simple path planning task well, by adding a penalty weight to areas near obstacles, pushing the generated path away from them. The planner was developed working with the OctoMap map system which represents the physical world by segmenting it into cubes of either open or occupied space. The open segments of these maps could then be used as vertices of a graph that the planning algorithm could traverse. The algorithm itself was written in C++ as a node of the Robot Operating System(ROS) software framework to allow it to smoothly interact with previously developed software used by the Control Engineering Robotics Group.

The program was tested by simulations where the path planner ROS node was sent maps as well as UAV position and intended goal. These simulations provided valid paths, with the performance of the algorithm as well as the quality of the paths being evaluated for varying configurations of the planners parameters. The planner works well in simulation and is deemed ready for use in practical experiments.

# *Acknowledgements*

I would first like to thank my supervisors Emil and George for their help and support throughout the thesis work. Emil especially for his work as the C++ guru of the office, solving any problem that would thwart my progress in no time, and teaching me some of the finer points of the language along the way. I would also like to thank Christoforos Kanellakis who provided lots of help with the map system. Without him wrapping my head around the workings of that less than optimally documented piece of software would have been even more of a pain.

# Contents

**Abstract** iii

**Acknowledgements** v

**1 Goals** 1
- 1.1 Introduction . . . 1
  - 1.1.1 What is a path? . . . 1
  - 1.1.2 What is a path planner? . . . 1
  - 1.1.3 The lay of the land . . . 2
- 1.2 Goal . . . 2
  - 1.2.1 Requirements on the planner . . . 2

**2 Software used** 5
- 2.1 ROS . . . 5
- 2.2 OctoMap . . . 7

**3 Path Planner Algorithms** 9
- 3.1 Path planning in general . . . 9
- 3.2 Algorithm alternatives . . . 9
  - 3.2.1 Simple planners and their flaws. . . . 9
  - 3.2.2 A* basics . . . 10
  - 3.2.3 A* Efficiency and Optimality Issues . . . 10
  - 3.2.4 LPA* . . . 10
  - 3.2.5 D* . . . 11
  - 3.2.6 D* lite . . . 11
  - 3.2.7 Random Sample based solutions . . . 11
- 3.3 Choice of algorithm . . . 12
- 3.4 Differences between WAPP and other A*-derivatives . . . 12
  - 3.4.1 Penalties for moving close to walls . . . 12
  - 3.4.2 Reduction of the Path to Waypoints . . . 13

**4 How the Planner Operates** 15
- 4.1 Overview . . . 15
- 4.2 Initialization . . . 16
  - 4.2.1 Inputs and basic workings . . . 16
  - 4.2.2 Parameters . . . 17
  - 4.2.3 Loading . . . 17
  - 4.2.4 Verification of Start and Goal . . . 18
- 4.3 The Search . . . 20
  - 4.3.1 The Vertex Costs . . . 20
  - 4.3.2 The Priority List . . . 20
  - 4.3.3 The Expand Function . . . 21
  - 4.3.4 The Search . . . 22

4.4 Way-points . . . . . 23

**5 Results** **25**
5.1 Tests . . . . . 25
5.1.1 Building Interior Map(pseudo-2D) . . . . . 25
5.1.2 Cityscape Map(full 3D) . . . . . 27
5.1.3 Analysis of the Cost Parameters Impact . . . . . 29

**6 Future Work** **31**
6.1 Easily adjustable wall costs . . . . . 31
6.2 Re-planning . . . . . 31
6.3 Trajectory Planning . . . . . 31

**Bibliography** **33**

# List of Figures

1.1 An example of a tree being represented as a cubic honeycomb in the OctoMap format . . . . . 2

2.1 Example of what the network of nodes running a robot able to move autonomously to a user specified goal could look like. . . . . 6
2.2 The structure of an octree and the cube shaped space it represents. . . 7
2.3 OctoMap storing one occupied vertex(black) within a larger free(white) space. . . . . 8
2.4 The same object represented as OctoMaps at three different resolutions. 8

3.1 Comparison of a path generated by A* on a 2D grid with 8 directions of movement and the optimal path. . . . . 10
3.2 RRT space filling tree. . . . . 12
3.3 A path(red) though a 2D map, reduced to waypoints(black). Blue vertices are walls, orange free explored and white are free unexplored . 13

4.1 An example of a map corresponding to the real coordinates $x_{min} = y_{max} = 5$, $x_{min} = y_{max} = 5$, with real coordinates shown in red, and the planner coordinates shown in black. . . . . 17
4.2 How penalty is distributed around a wall vertex. . . . . 18
4.3 The Initialization process. . . . . 19
4.4 The vertex updating process for every vertex examined by the expand function . . . . . 21
4.5 The Search process. . . . . 22
4.6 The Waypoint generation process. . . . . 23
4.7 A path(red) though a 2D map, reduced to waypoints(black). Blue vertices are walls, orange free explored and white are free unexplored . 23

5.3 An example of how a path generated with too high GCL(red) could be suboptimal, as well as a better alternative(black), generated with a lower GCL . . . . . 30

# List of Abbreviations

| | |
|---|---|
| **WAPP** | **W**all **A**voiding **P**ath **P**lanner |
| **UAV** | **U**nmanned **A**erial **V**ehicle |
| **WCL** | **W**all **C**ost **L**evel |
| **MCL** | **M**ove **C**ost **L**evel |
| **GCL** | **G**oal **C**ost **L**evel |
| **ROS** | **R**obot **O**perating **S**ystem |

# Chapter 1

# Goals

## 1.1 Introduction

When a robot moves through an environment many different components need to interact. The robot will need to know its position in space, it will need either a map of its surroundings or a way to generate a map through sensory input, and then there is the actual motion. Electrical engines will need to run at appropriate speeds, servos controlling steering or control surfaces will need to move. Then there is keeping track of the direction, speed and acceleration of the robot, all which is needed to get the robot to move in the direction in which its operator wants it to move.

But even if all these things are functioning, where should the robot move to? Suppose it has been given a goal, a point on its map to where it wants to move. Does it just aim for that point and move straight towards it? What if something is in the way? What if the robot is situated in a building with rooms and corridors, or even in a maze?

For the robot to autonomously move towards its goal it needs more detailed instructions, it needs a series of coordinates all in line of sight of each other, starting at the robots position and ending at the goal.

The robot needs a path.

### 1.1.1 What is a path?

What path means in this case is a path though 3D space which a UAV can follow without colliding with obstacles. This could be a smooth, continuous curve generated by taking into the dynamics of flying or, in its most basic form, a simple series of coordinates which a robot could follow sequentially in a series of straight lines. This simple type of path is what is generated by the path planner developed during this thesis work.

The problem that needs addressing is thus how to take information about the physical environment, the position of a robot and it's desired destination and extract a viable path using this data.

### 1.1.2 What is a path planner?

A path planner is a program designed to solve this problem, to figure out a path through its environment without colliding with walls or other obstacles whilst doing so. How this is achieved varies and depends largely on how the map containing the walls and obstacles work, but generally it boils down to representing the map as a weighted graph that can be searched for a path between two of its vertices. This can either be an inherent feature of the map, such as on maps that represents the world as a cubic honeycomb of either free or occupied cubes of space. The free vertices of such maps essentially make up one or more searchable graphs, one if all free vertices

are interconnected, more if there are groups of them that are blocked off from each other.

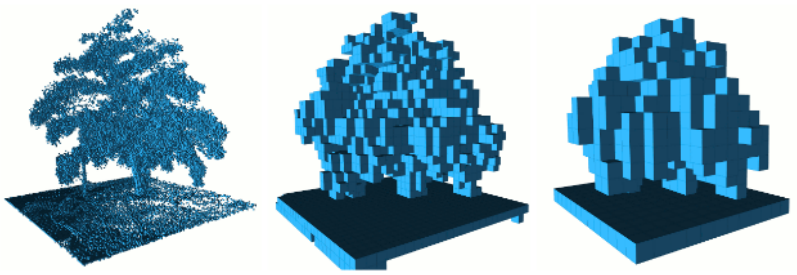


FIGURE 1.1: An example of a tree being represented as a cubic honeycomb in the OctoMap format. [1]

Another method of generating a searchable graph is to use randomly sampled points on the map and see which of them can be connected to neighbouring points without causing a collision. The points that can be connected forms a grid around the various obstacles, and gradually the resolution can be increased by adding new points and connecting them to nearby previously added points. As the point density increases the chances that there is a connection around obstacles between any two points on the map eventually approaches one.

When the graph is established there is a multitude of options for search algorithms, a selection of which will be presented in chapter 3.

### 1.1.3 The lay of the land

In the robotics research group, for which the path planner created during this thesis work was made, there was already a map system in use. This map system is called OctoMapHornung et al., 2013 and is of the previously mentioned type that abstracts the 3D environment into a cubic honeycomb of free or occupied spaces. To be able to integrate the planner with software already in use and to be able to draw on previous knowledge and experience available in the research group, it was deemed that using the OctoMap format as the map input to the planner would be best.

Both the map software and all other software components which the path planner will need to interact with runs under the Robot Operating System (ROS)"ROS". ROS is a software framework for robotics which lets the various programs, called nodes, communicate and interact This means the path planner will need to be a ROS node as well.

## 1.2 Goal

The goal of this thesis work was to implement a functioning path planner that could be of practical use in research and experiments. The path planner should generate effective paths that can be used both as a simple trajectory for UAVs and as a basis for further development.

### 1.2.1 Requirements on the planner

The following requirements for the planner were given:

- The planner should be deterministic(i.e. if there is a path to the goal, the planner should find it and not get stuck in local minima)
- The planner should read 3D grid maps in the OctoMap format, using ROS node(s) to do so to facilitate integration with systems in use.
- The Planner should avoid planning paths near walls, but when no other path is available plan paths through more narrow passages. This is both to reduce the risks that comes with trying to fly too close to a wall, as well as facilitating an easier later implementation of a trajectory planner.

# Chapter 2

# Software used

## 2.1 ROS

The Robot Operating System is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

Even though ROS stands the Robot Operating System, it is not an operating system per se, but rather a software framework, or so called middleware, for use in robotics. The programs running under ROS are called nodes and what ROS mainly does is providing easy data communication between them. The nodes can publish data to topics or subscribe to topics and receive data when it is published to them. The data sent between nodes can be sensor data, coordinates or any info that various nodes might need to share to facilitate the use of a robot and the performance of its various functions.

The following image shows a visualization of how the path planner constructed during this thesis work could fit into a network of ROS nodes.

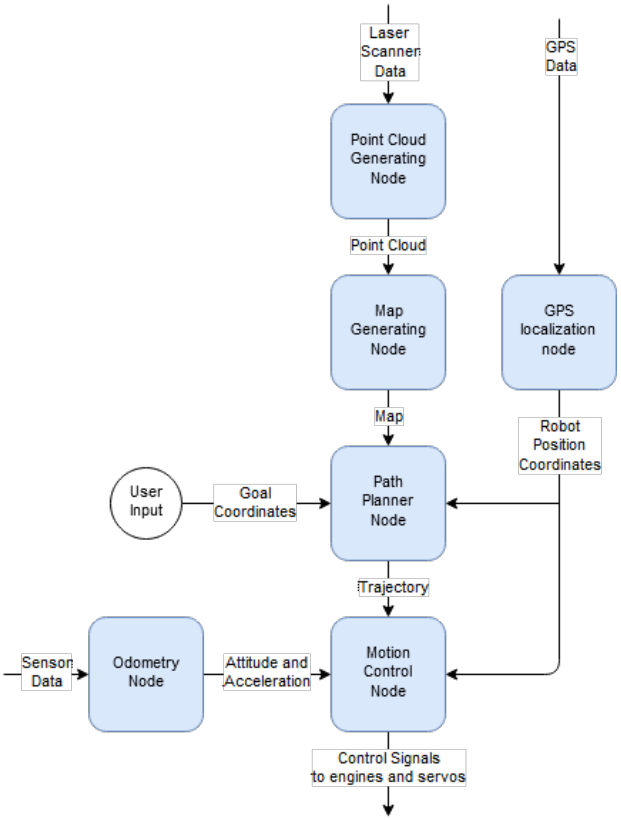



FIGURE 2.1: Example of what the network of nodes running a robot able to move autonomously to a user specified goal could look like.

The path planner node is being fed data though the ROS topic system from three sources. Firstly it receives the goal coordinates from a user input which is the only non-automatic part of this system. In addition to this input the robot receives its own position from a GPS localization node which in turn receives its data directly from the signals from a GPS receiver device on the robot.

The map data needed by the planner is provided by a map generating node which transforms a point cloud message into a map format that the planner can use. This point cloud is supplied by a node which receives measurement data from a rotating laser scanner which measures the distance between the scanner and objects and walls in proximity tot he robot. This measurement data is published as a ROS topic and made available to the map generating node.

When provided with its three needed inputs the planner node generates a path which is published as the desired trajectory for the robot. This trajectory is then received by a motion control node which uses the trajectory, the robot position and odometry data and computes how the robot shall move and when, and proceeds to send the required control signals to the various servos and motors that need to be activated to move the robot along the path.

This is just an example of how it could work, and all the nodes work in the same fashion no matter what nodes were publishing the input data or subscribing to the output data. This makes a system running ROS highly flexible allowing different configurations of nodes to smoothly interact.

The reason for using ROS is mainly that is provides easy integration with existing software as well as with future developments within the robotics group. There are several other similar types of robotics middleware that could replace ROS but since this work was done to provide a working piece of software to a research group where ROS was being used these alternatives were neither investigated nor seriously considered, due to practicality and time constraints.

The program in its entirety was written in C++ in the ROS environment.

## 2.2 OctoMap

The map that the path planner will navigate will be come in the form of an OctoMap, published in ROS by the OctoMap server plug-in. The OctoMap library uses an octree based cubic grid system to represent three dimensional space, with data structures and mapping algorithms provided in C++. An octree is a data structure that divides the space into cubes, subdivided into smaller octant cubes, which in turn are subdivided into octants and so on until a set minimum cube size is reached. Cubes whose constituent octants are all either occupied or open can be saved as a single element. This results in a great reduction in file size on maps with large areas where all vertices are either occupied or open.

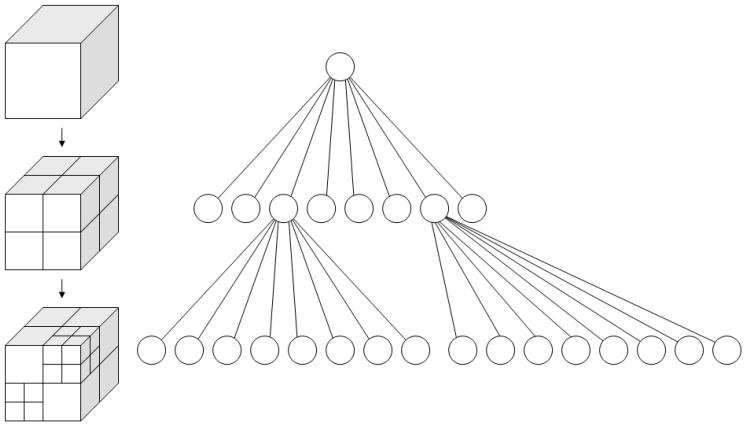


FIGURE 2.2: The structure of an octree and the cube shaped space it represents.[1]

This allows OctoMap to efficiently store a model of a 3D environment that can rapidly be sent between robots running ROS, or between nodes on a single ROS using device. This compression affects how the OctoMap messages are stored and sent, but have no real impact on the workings of the planner algorithm, to which the OctoMap seems to be made up of equal sized cubes of a size corresponding to the maps resolution.

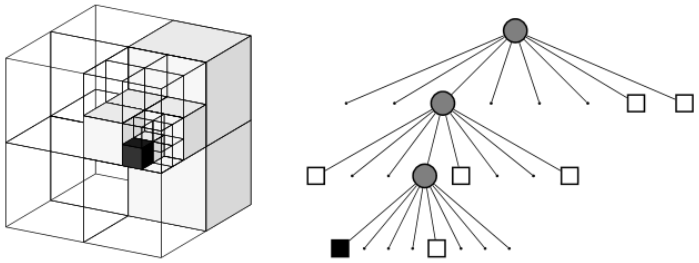


FIGURE 2.3: OctoMap storing one occupied vertex(black) within a larger free(white) space. [2]

OctoMap allows for three different states for each division of space: free, occupied and unknown. The unknown status was not used in this thesis work, but it could be useful at a later stage when planning the velocity of a UAV in different areas of the map.

The OctoMap plugin for ROS can either open an OctoMap from file or create one out of a point cloud. The point cloud is the list of coordinates where a detector such as a laser scanner has detected a surface. When generating an OctoMap from a point cloud one can set any resolution deemed suitable to the task at hand, depending on such factors as available computational power and the size of the robot that is to navigate the map. When loading a map from a file the minimum cube size can be set to that of the file or larger, allowing for quicker processing of maps of unnecessary high resolution.

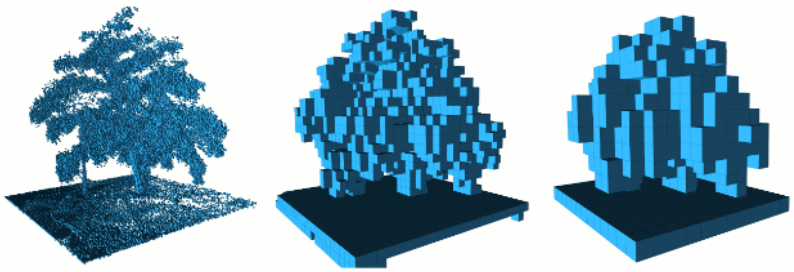


FIGURE 2.4: The same object represented as OctoMaps at three different resolutions. [3]

# Chapter 3

# Path Planner Algorithms

## 3.1 Path planning in general

The area of path planning came about as a subject within the area of Graph Theory, itself an area within discrete mathematics. More specifically the planning of paths between two points on a map is the solving of Graph Theories shortest path problem, how to find the cheapest path between two points on a weighted graph. There are several ways of doing this and countless variations of path planning algorithms has been developed over the years.

## 3.2 Algorithm alternatives

There are several methods of finding a path. The problem comes in two parts, firstly the generation of a graph that contains a path around any obstacles to the goal, and secondly the search algorithm which tries to find the best path through the grid once it has been established. The algorithms discussed in this chapter are mainly solutions to the second problem, that of efficiently finding a good path through a large graphs with many vertices. However some other solutions which focus more on the graph generation will receive a short mention as well.

### 3.2.1 Simple planners and their flaws.

One of the first solutions to the shortest path problems was Dijkstra's Algorithm (Dijkstra, 1959), conceived by Edsger Dijkstra in the late 1950s. Dijkstra's Algorithm finds the shortest path by simply searching all paths in order of their cost, starting at one end of the path. This means that whilst the search will find the shortest path in the graph, it will search aimlessly for it and it will end up searching vast areas of the map unnecessarily. This means that for any applications other than searching for paths through narrow, non branching corridors with the start at a dead end it is highly inefficient.

One example of a simple method of path planning that goes a completely different way is finding a path by simulating a potential field. Here the robot is made to act as a charged particle in a potential field with the goal attracting it and obstacles repulsing it. This solution can create a very quick planner that is clearly aimed at the goal and does not spend unnecessary computational power on searching irrelevant areas of the map. However, this approach is not certain to find a path to the goal, since it can lead to the robot getting stuck in local minima.

Both the very basic algorithms that go straight for the goal and the ones that explore the map methodically has serious flaws, and a combination of the two approaches is needed.

### 3.2.2 A* basics

One of the first solutions that markedly improved Dijkstra's algorithm was A*Hart, Nilsson, and Raphael, 1968, first described in 1968. This algorithm introduced a heuristic to Dijkstra's algorithm that made the planner give priority in the order of vertex exploration based on the distance to the goal in addition to the length of the path. Basically A* works like a variant of Dijkstra's that tries to explore only vertices relevant to the search for a path between two points. This algorithm was highly successful, and variants of it are in common use to this day.

### 3.2.3 A* Efficiency and Optimality Issues

A* and it's descendants will always generate a path if there is one on the grid, however there are a few things that determines how close this path will be to the optimal path. Firstly there is the resolution of the map. A higher resolution will lead to paths closer to the optimum, at the cost of longer computation times. Since the number of vertices in the map increase cubically to the increase in resolution this is a major factor. A lower resolution map will generate a path much quicker, but this comes at the cost of producing a path further from the optimum path, and risks closing off narrow passages that cannot be seen at too low a resolution.

Another obstacle that makes the path suboptimal is the fact that A* operating in a square or cube grid is limited to moving in a set number of angles. This is solved in a variety of ways, usually by either checking for optimal parts in a small area at a time while the planner is running, or by post processing of the generated path.

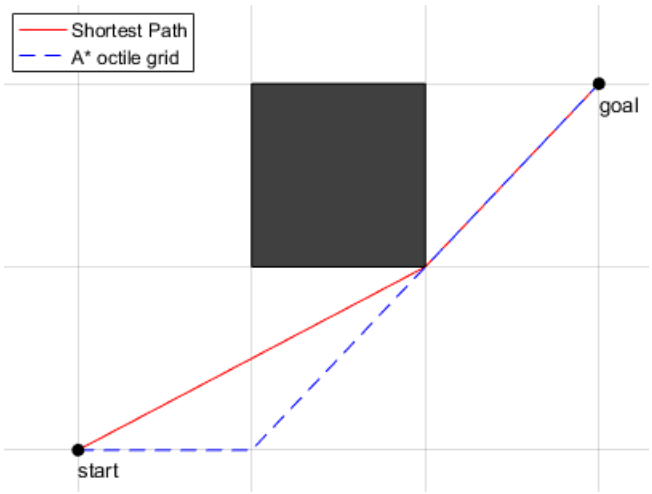



FIGURE 3.1: Comparison of a path generated by A* on a 2D grid with 8 directions of movement and the optimal path.[1]

When it comes to performance in terms of the computational weight of the algorithm the main issue with A* however is the fact that the basic algorithm is designed to solve the problem once only, and to get an updating path one has to run the algorithm over and over. This is one of the main reasons plain A* has to a large extent been replaced by updated variants of the algorithm.

### 3.2.4 LPA*

The lack of support for map changes in A* is dealt with in a very efficient manner by the LPA*Koenig, Likhachev, and Furcy, 2005, or lifelong planning A* algorithm.

LPA* deals with this in the following way: When the occupation status of a vertex changes, that is, a wall vertex is added or removed, the surrounding vertices are checked to see if the move distance from the start is consistent with those of the vertices that surround them. This means that vertices look at their neighbours to check if their data is up to date, or if they have been blocked off. For each vertex that was found to have changed the neighbours of that vertex is then checked, forming a wave that identifies all vertices whose data is now outdated, add them as potential targets for exploration again. This leads to only the information that is outdated is discarded, and the segments of the map which were explored stays that way for the next search, making the re-planning more efficient.

When this method is applied one has to take into consideration the direction of the search as well. If the map is generated or updated by a scanner positioned on the robot the planning should generally be done from the goal in the direction of the robot position. The reason for this is that when a vertex occupation status changes near the robot the direction of the wave of vertex recalculations will move toward the robot and only a mall part of the map will need to be recalculated at a single time. Thus the rule is generally that the direction of the search should go from the end of the path where vertices are more likely to change towards the end where this is less likely.

### 3.2.5 D*

D*Stentz, 1994, or Dynamic A* is a variant of A* made to be used with a moving end of the path, with the algorithm handling the re-planning this requires. D* works better than multiple runs of A*, but uses a fairly complex and inefficient algorithm using pointers at every vertex pointing to a parent vertex.

### 3.2.6 D* lite

D* liteKoenig and Likhachev, 2005 is effectively a combination of D* and LPA*, implementing the behavior and support for a moving end of the path from the former, and the efficient re-planning system of the latter. The efficiency of D* lite has led it to become somewhat of an industry standard over the last decade.

### 3.2.7 Random Sample based solutions

One way to solve path planning problems used by algorithms such as RRTLavalle, 1998, or Rapidly-exploring Random Tree, is to make use of random samples to build a graph on which the path later is found. These methods are not restricted to a grid in the same manner as A* and can produce highly efficient paths if left to generate a complex enough map. However, these methods in their basic state are direction-less and wastes time on probing every area as frequently.

There are improvements to these methods, which fall into two categories. The first involves speeding up the convergence to optimality by restricting samples to an area nearby the path when it has been found. One example of this is Informed RRTGammell, Srinivasa, and Barfoot, 2014, which greatly improves the speed of which an optimal path is found by forcing the algorithm to sample in an area restricted by the outer surface of an ellipsoid based on the shape of the best known path.


FIGURE 3.2: RRT space filling tree.[2]

The other method for improving the use of Algorithms such as RRT is by first using a low-resolution run of A* or a similar planner, and then using RRT to improve upon the path by using a similar restriction as Informed RRT.

## 3.3 Choice of algorithm

Since the planner should work on the OctoMap square grid maps generated in various experiments A* and it's derivatives are especially suited for this due to them being made for square grid as well as them being deterministic, and always finding a path if there is one. In addition the heuristic-system for A*-derived planners are highly modifiable which makes the adaption of the wall avoidance objective fairly straight forward to implement. A variant of A*, similar to D*lite in its mechanism, was chosen due to this, and since it is a very well established and widely used it seems like an excellent options for an OctoMap reading path planner. This path planner algorithm will be referred to as WAPP, which stands for Wall Avoiding Path Planner

## 3.4 Differences between WAPP and other A*-derivatives

### 3.4.1 Penalties for moving close to walls

To add a penalty for going close to walls an additional cost is added to moving to the vertices closest to the wall. The penalties are added to all vertices that lies within the boundaries of a sphere, in this case with a diameter of seven vertices. The distance this represent in reality is dependent on map resolution, and with a high resolution map the radius of the sphere of added wall penalties would probably need to be increased. On the other hand, the use of high resolution maps is unmotivated unless ones robot was very small and nimble and able to pass though passages, which might need less free space, so perhaps this is less of an issue than it might initially appear.

### 3.4.2 Reduction of the Path to Waypoints

WAPP will be used to generate paths for UAVs in the form of a trajectory made up of waypoints, published in the MultiDOFJointTrajectory Message format. This reduction of an A* style path to a list of waypoints serves different purposes.

First and foremost it solves the problem of the planner being restricted when it comes to which directions it can plan in. The 45° minimum angle for turns allowed in a cube grid is removed by letting the path go between vertices that are not neighbours. This solution needs to examine fewer number of vertices to interpolate between them compared to some any-angle A* variants which continuously do interpolation during the expansion part of the algorithm.

Secondly the waypoint generation will only reduce series of path vertices to straight lines if said vertices are not in the proximity of a wall. This leads to a higher waypoint density in areas with a larger collision risk for a UAV, which can make it easier to implement a trajectory planner that converts the path into a continuous curve, leaving a curve fitting algorithm more freedom to go off the path where there are no walls nearby. Or if a UAV simply is set to pass by every single waypoint on the list, it will be forced to more more slowly and carefully in areas where there are walls to avoid. Thirdly the waypoint generation process will compensate for suboptimal paths which can be generated when setting the goal distance heuristic to be the dominant factor in the exploration prioritization.

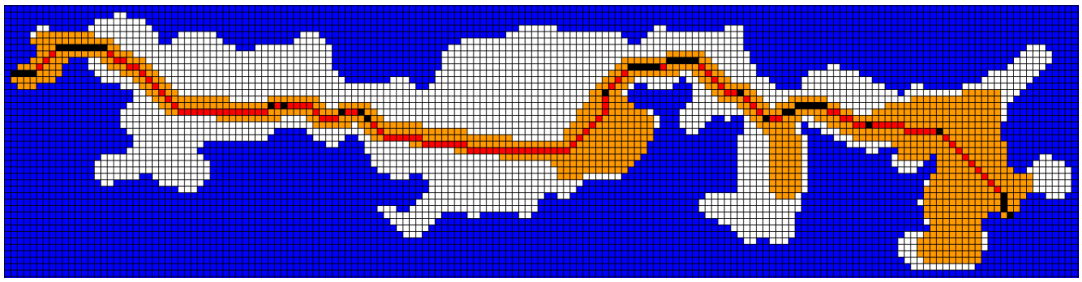


FIGURE 3.3: A path(red) though a 2D map, reduced to waypoints(black). Blue vertices are walls, orange free explored and white are free unexplored

Note that the example image is in 2D rather than 3D. This is merely to make the behaviour of the waypoint generation easier to understand for the viewer. The algorithm works in the same manner on a 3D map as when generating waypoints on a flat map only one OctoMap block deep.

# Chapter 4

# How the Planner Operates

## 4.1 Overview

The path planner can be divided into three parts, The Initialization, the Search and the waypoint generation.

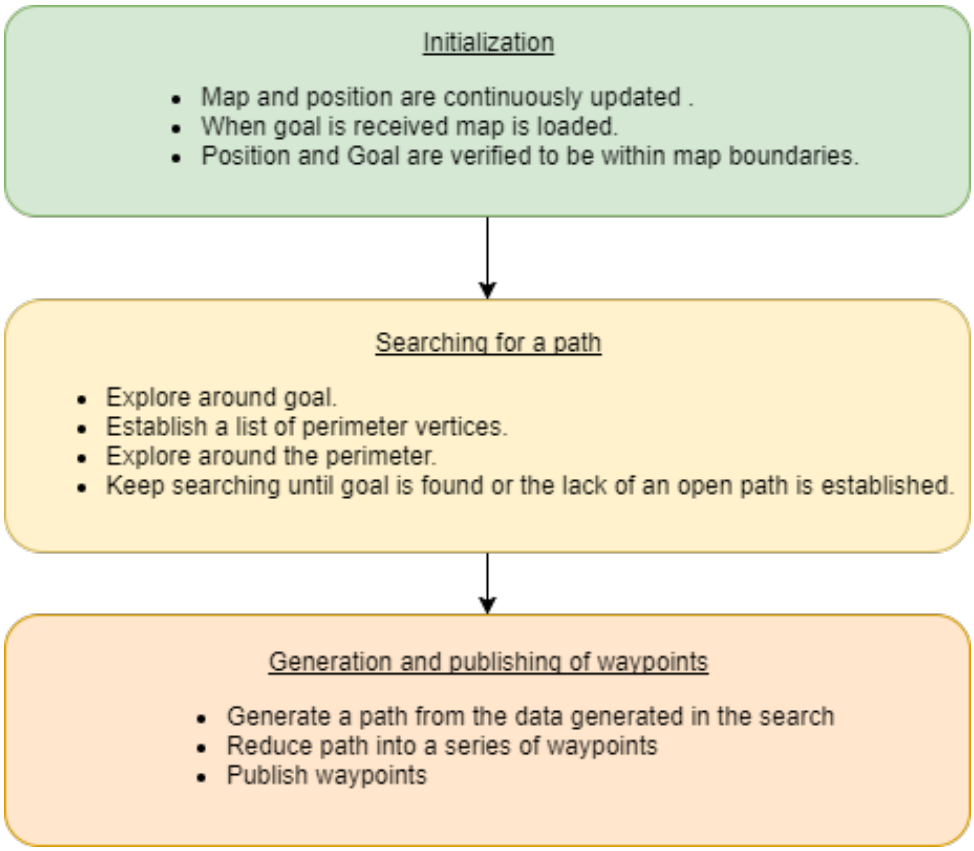
- The initialization contains all the things that need to take place before the actual path planning can take place. This includes the node receiving the coordinates for the current robot position and goal, as well as the loading of the map. This part makes sure the requirements for the path planner has been fulfilled, and if this is not the case the program will wait until that is the case.

- The Search is the core of the program, which searches the map, exploring until a path to the goal has been found. It does this by searching the map, first expanding around the start, assigning values to the vertices based on their distance to the goal(referred to as the g-cost) and the length of path needed to move to them from the start(referred to as the m-cost). The program keeps track of the vertices who received these values and picks the next centre of expansion from them based on the two costs. The program repeats this until the goal has been reached or it has been established that there is no path that leads to the goal.

- The waypoint generation is done if a path to the goal was found during the search process in order to make the path smoother and better for both direct UAV use as well as for use as a basis for more advanced UAV motion planning.

  It finds the path by picking from amongst its neighbours the one with the lowest m-cost, with represents the path length, then does the same with that neighbour, and continues as such until the robot position has been reached.

  It then uses the list of these vertices and removes any vertices that can be bypassed by moving in a straight line between the previous vertex and the next on the list. After this is done the waypoints are published to a ROS topic to be used by other nodes.




## 4.2 Initialization

### 4.2.1 Inputs and basic workings

WAPP has three inputs which it receives by subscribing to three ROS topics to which other nodes can publish them. These are the map, the position of the robot and the goal to which the robot's operator wants the robot to move.

The map is an OctoMap which is sent in a compact serialized format to ensure quick transfer rates. The two coordinates come in the format geometry_msgs:: TransformStamped which is a ROS format used by nodes to transmit both the position and the orientation of objects. In this case though only the position coordinates contained within these messages are read.

The map and the position are continually updated and saved while the program waits for the operator to send it a request for a path to be generated. This request is the act of sending the planner a goal coordinate by publishing it to the topic which WAPP node is subscribed to. Every time a new goal is sent to the WAPP node a path will be generated.

When the node receives the goal coordinates it first checks that an OctoMap as well as the position coordinates have already been received and saved. If this is confirmed the program first goes ahead and loads the planner parameters.

### 4.2.2 Parameters

The parameters are the settings that dictate the behavior of the planner, and during the initialization they are loaded. These parameters are loaded from a .launch file in the WAPP nodes ROS directory and are set by the operator beforehand. The search process is guided by three motivations: Moving towards the goal, keeping the path of minimal length and keeping the path away from walls and obstacles. The parameters are the weight values which guide this behavior, the goal-cost level(GCL), the move-cost level(MCL) and the wall-cost level(WCL).

These parameters are simply integer values that set the behaviour of the planner algorithm, the ratio between them is what affects how WAPP prioritizes between the three aforementioned motivations.

If the launch file containing the parameters is missing the parameters are set to hard-coded default values of GCL=1, MCL=1, WCL=10.

### 4.2.3 Loading

When the parameters have been loaded, the map can be deserialized and properly loaded. First of all the planner begins with identifying the dimensions of the map and it's resolution using built in functions of the OctoMap package for C++. The planner map is created as a three dimensional data structure with the same amount of vertices as the OctoMap, but with the three coordinates all starting at 0 and the resolution being 1. This is in contrast to the OctoMap which allows for negative coordinates and resolutions smaller or larger than 1. This coordinate transform is done in the following manner:

$$x_{planner} = (x_{OctoMap} + x_{min,OctoMap}) * s + 0.5 \tag{4.1}$$

$$y_{planner} = (y_{OctoMap} + y_{min,OctoMap}) * s + 0.5 \tag{4.2}$$

$$z_{planner} = (z_{OctoMap} + z_{min,OctoMap}) * s + 0.5 \tag{4.3}$$

Where s is the size of the OctoMap cubes

The 0.5 comes from the fact that WAPP works with the centers of the blocks in OctoMap, and without it the grid would be off one half vertex.

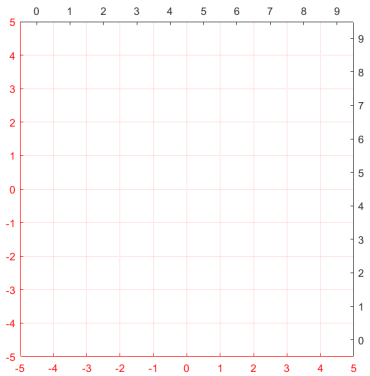



FIGURE 4.1: An example of a map corresponding to the real coordinates $x_{min} = y_{max} = 5$, $x_{min} = y_{max} = 5$, with real coordinates shown in red, and the planner coordinates shown in black.

The reason for creating an internal map in this manner is that the vertices of the map needs to be able to store several values in addition to their occupation status, these are previously mentioned g-cost and m-cost, a tag to keep track of previously expanded vertices as well as the penalty a vertex receives for being in near proximity of a wall vertex.

This wall penalty is added to the vertices during loading in the following manner: As the map is loaded into this data structure, every time a wall vertex is added a wall proximity penalty is added to vertices in a sphere surrounding it. The diameter of the sphere is seven vertices, so that vertices whose distance to the wall vertex is less than 3.5 times the OctoMap cube size will receive a penalty. The sphere in which the costs are added contains two smaller spheres that adds higher penalties, to the effect that the penalty at the edge of the largest sphere is simply the WCL, whilst vertices nearer to the wall vertex receive as penalty the WCL multiplied by higher numbers, as shown below.

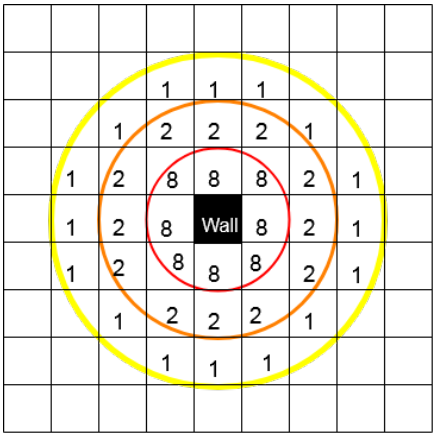



FIGURE 4.2: How penalty is distributed around a wall vertex.

These penalties are cumulative, so that vertices in the proximity of several wall vertices will receive large penalties. The effect this penalty has on the behavior of the path planner will be discussed in the subsection dedicated to the Search part of the program.

### 4.2.4 Verification of Start and Goal

When the map has been loaded the start and the goal are verified to be within the map boundaries by making sure the following conditions are true:

$$x_{min,OctoMap} \leq x_{position} \leq x_{max,OctoMap} \tag{4.4}$$

$$y_{min,OctoMap} \leq y_{position} \leq y_{max,OctoMap} \tag{4.5}$$

$$z_{min,OctoMap} \leq z_{position} \leq z_{max,OctoMap} \tag{4.6}$$

$$x_{min,OctoMap} \leq x_{goal} \leq x_{max,OctoMap} \tag{4.7}$$

$$y_{min,OctoMap} \leq y_{goal} \leq y_{max,OctoMap} \tag{4.8}$$

$$z_{min,OctoMap} \leq z_{goal} \leq z_{max,OctoMap} \tag{4.9}$$

This simply means that if the user feeds the program a goal or position coordinate that lies outside the physical area represented in the OctoMap the program will wait until a proper one has been sent.

When this is done the start and goal coordinates are converted in the same fashion as the map coordinates:

$$x_{start,planner} = (x_{position} + x_{min,OctoMap}) * s + 0.5 \tag{4.10}$$

$$y_{start,planner} = (y_{position} + y_{min,OctoMap}) * s + 0.5 \tag{4.11}$$

$$z_{start,planner} = (z_{position} + z_{min,OctoMap}) * s + 0.5 \tag{4.12}$$

$$x_{goal,planner} = (x_{goal} + x_{min,OctoMap}) * s + 0.5 \tag{4.13}$$

$$y_{goal,planner} = (y_{goal} + y_{min,OctoMap}) * s + 0.5 \tag{4.14}$$

$$z_{goal,planner} = (z_{goal} + z_{min,OctoMap}) * s + 0.5 \tag{4.15}$$

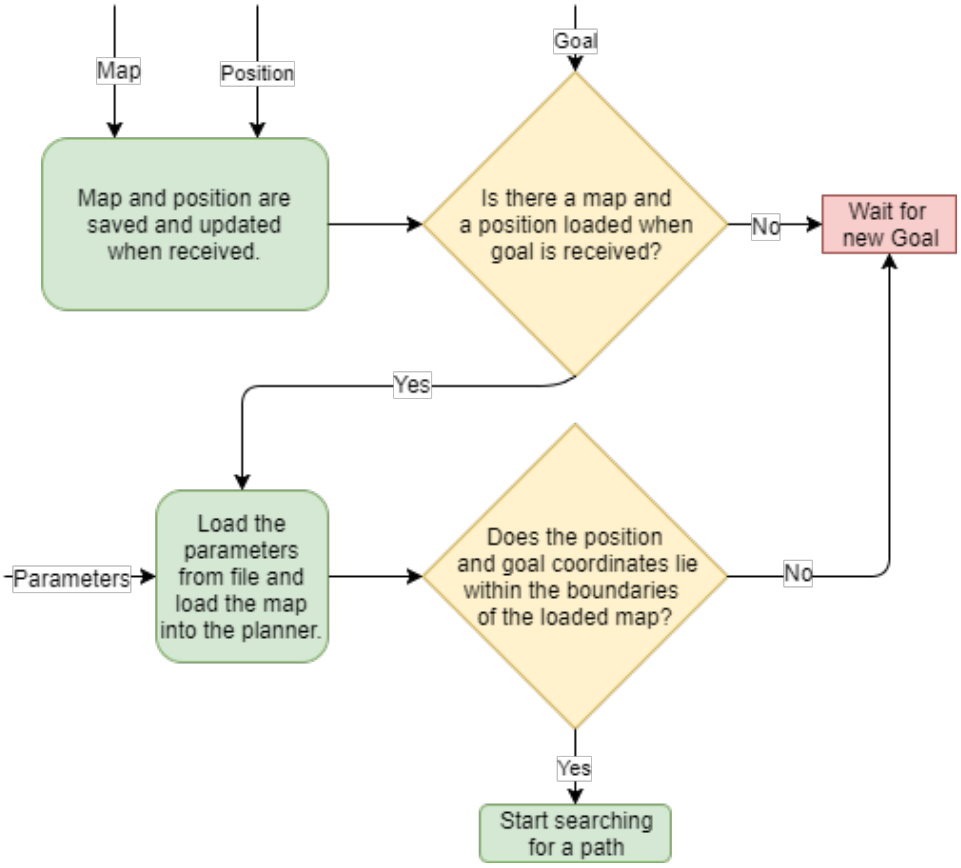



FIGURE 4.3: The Initialization process.

## 4.3 The Search

The search is done by exploring around a starting point, and repeatedly expanding the explored area by searching around its perimeter. To explain how this expansion works a few fundamental parts of the mechanism has to be explained first:the Vertex cost values, the priority list and the expand function.

### 4.3.1 The Vertex Costs

To differentiate between vertices in order to chose where in the map to explore next two costs are associated with the vertices, the g-cost and the m-cost. These values are assigned to a vertex by the expand function, and thus is only set for the explored map vertices The g-cost is for a vertex, v, is computed using the following formula:

$$gcost_v = 10\sqrt{(x_{goal} - x_v)^2 + (y_{goal} - y_v)^2 + (z_{goal} - z_v)^2} \tag{4.16}$$

The m-cost is for a vertex v computed using the following formula:

$$mcost_v = 10\sqrt{(x_{ec} - x_v)^2 + (y_{ec} - y_v)^2 + (z_{ec} - z_v)^2} + mcost_{ec} + pen_v \tag{4.17}$$

Where the index ec denotes values related to the expansion centre vertex from which v was explored, and pen is the wall proximity penalty added to the vertex during loading.

The multiplication by ten in both cases is to allow the number to be converted to an integer without losing too much accuracy. This is an efficiency measure, allowing the program to run almost exclusively integer operations which are faster to compute.

### 4.3.2 The Priority List

The priority list is basically a list of all the vertices that lies on the outer surface of the explored part of the map. This area, and the list with it, grows when the expand function picks one vertex at the surface as an expansion centre and adds its non-wall, unexplored neighbours to the list.

The list is used to sort out which vertex is to be the next centre of expansion, henceforth to be referred to as the expansion centre. In the list entries the coordinates of the vertices are saved, along with the sum of their g-cost and their m-cost, as well as the g-cost alone. An example of one of an entry of the vertex v to the priority list:

$$((mcost_v + gcost_v), gcost_v, x_v, y_v, z_v) \tag{4.18}$$

Everytime a new vertex is added to the list the list is sorted, first by their first values, the sum of g-cost and m-cost, and in case of a tie it is broken by g-cost.

### 4.3.3 The Expand Function

The expand function takes as its input the coordinates of a vertex on the outer surface of the explored area, the expansion centre. Around this vertex the surrounding box of vertices $v_{ijk}$ are investigated:

$v = (i, j, k)$
$(x_{ec}) - 1 \leq i \leq (x_{ec} + 1)$
$(y_{ec}) - 1 \leq j \leq (y_{ec} + 1)$
$(z_{ec}) - 1 \leq k \leq (z_{ec} + 1)$

If a vertex is occupied, or if it is the expansion centre itself it is ignored. If the vertex passes these tests and has not been expanded previously both the g-cost and m-cost of the vertex are set, and the vertex is added to the priority list.

If a vertex has been expanded previously it is only re-expanded only in the following case:

$$mcost_v > 10\sqrt{(x_{ec} - x_v)^2 + (y_{ec} - y_v)^2 + (z_{ec} - z_v)^2} + mcost_{ec} + pen_v \tag{4.19}$$

In other words, if it would result in the vertex getting a lower m-cost than it currently has. In this case it's m-cost is updated to the lower value, and it is re-added to the priority list. This is done to ensure that the final path is indeed the shortest one.

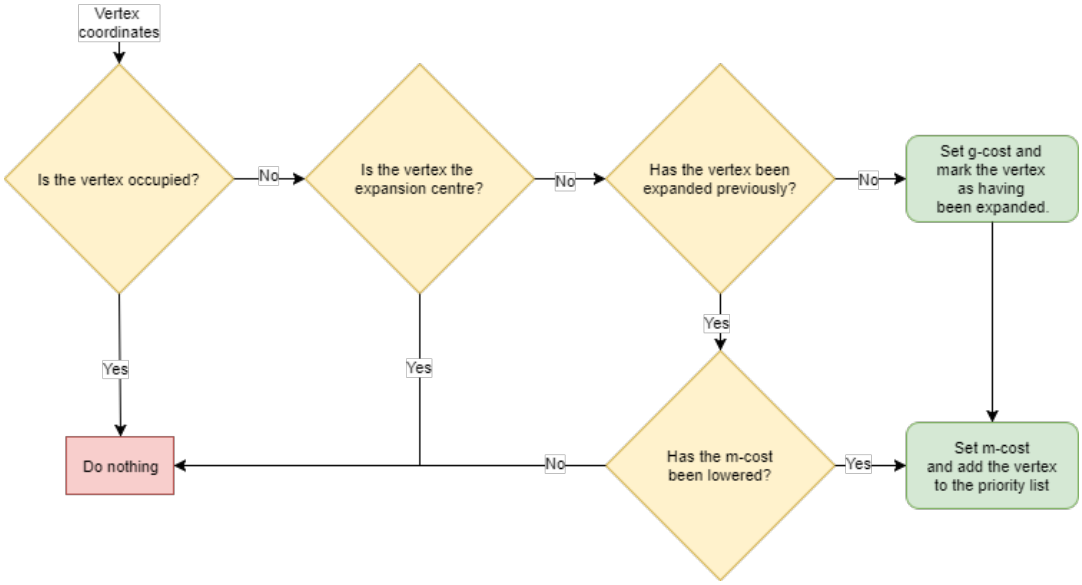


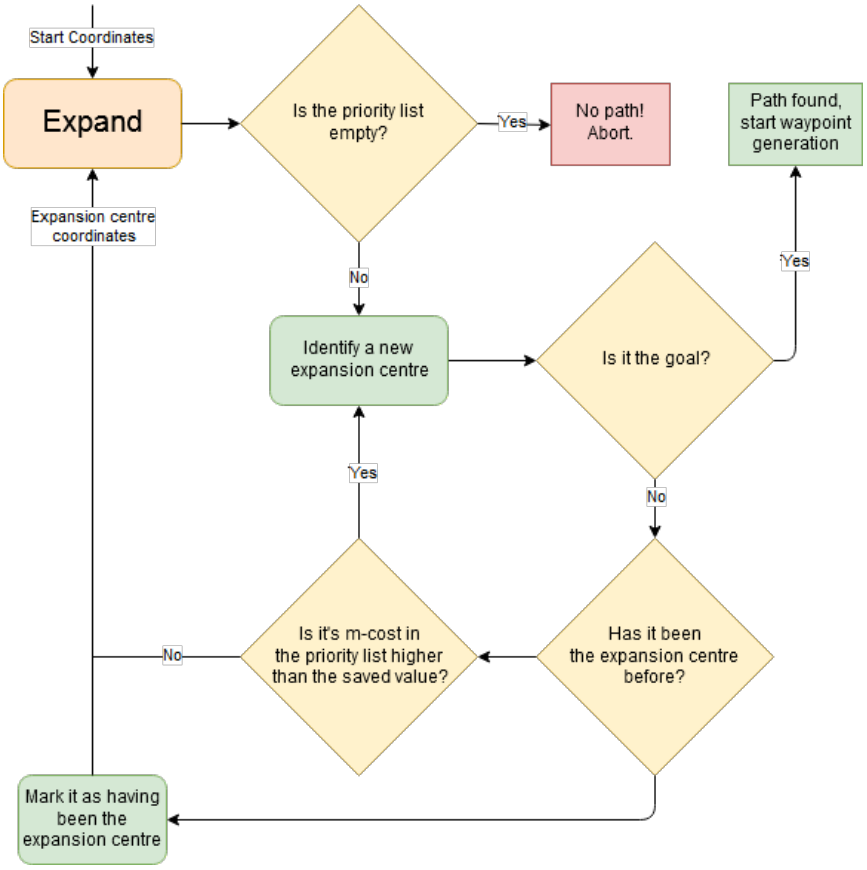

FIGURE 4.4: The vertex updating process for every vertex examined by the expand function

### 4.3.4 The Search

WAPP starts searching in one end of the path, here set to the position. The start coordinate is sent to the expand function as the first expansion centre, which adds the surrounding non-wall vertices to the priority list. The program then sorts the priority list, picks the vertex with the lowest sum of g-cost and m-cost, uses that as the next priority centre.

This is repeated until one of the following things happen: The list is empty when the program tries to extract a new expansion centre from it. This means there is no open, unblocked path between the start and the goal. The other thing is that the new expansion centre is the goal, which means a path has been found.

FIGURE 4.5: The Search process.

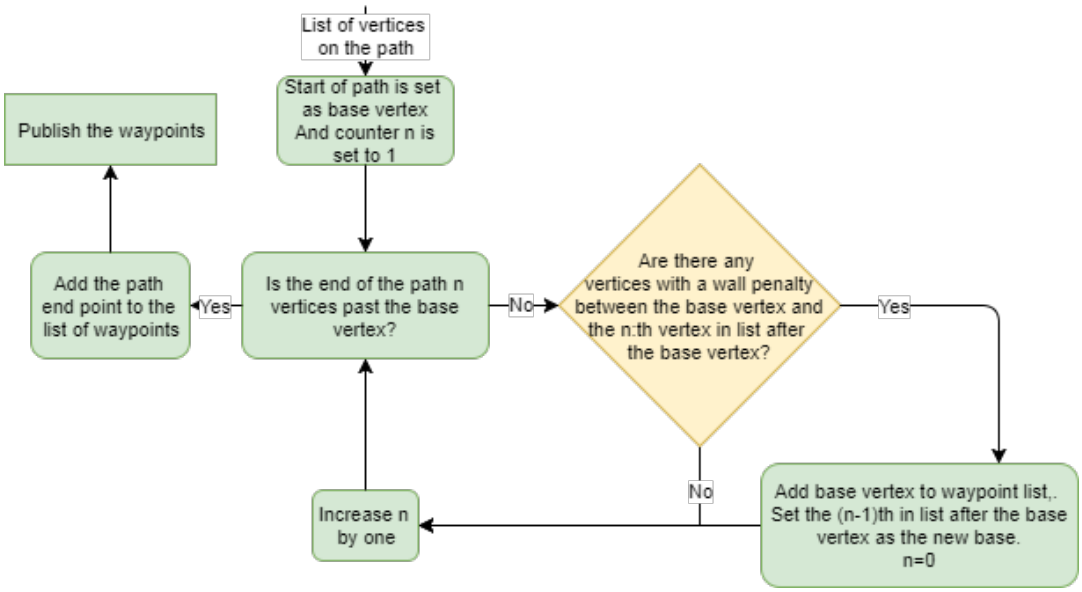


## 4.4 Way-points

When the path has been found the basic A* style path is generated by starting at the goal vertex and picking the next vertex on the path by choosing the one with the lowest m-cost amongst the neighbouring vertices over and over until the start is reached.

All the vertices on the path are added to a list which is then reduced to a list of way points. This is done through a process where any segments of the paths which can be bypassed by traveling in a straight line from the vertex before the segment and the vertex after without passing too close to a wall is removed. This achieved in the following manner:

FIGURE 4.6: The Waypoint generation process.




The test used to determine if there was a vertex with a wall penalty is a simple raycasting algorithm. The algorithm compares the two end vertices, $v_1$ and $v_2$ the space between whom will be investigated. It simply uses the linear equation of the line that passes through both points and checks the penalty status of all the vertices corresponding to the cubes of space that the line passes through.

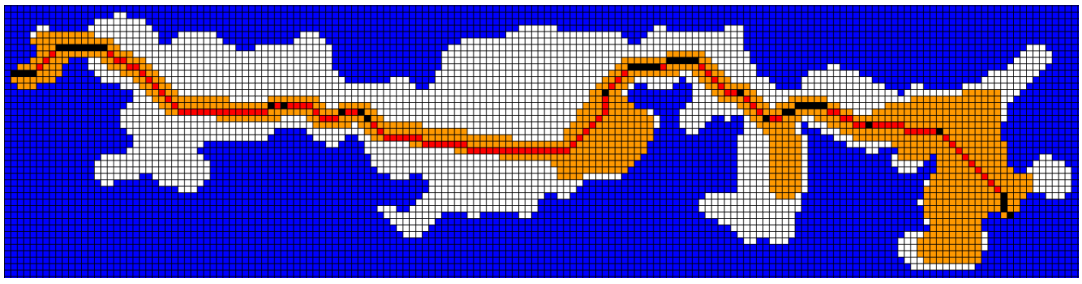


FIGURE 4.7: A path(red) though a 2D map, reduced to way-points(black). Blue vertices are walls, orange free explored and white are free unexplored

This example image showcases the behavior of the waypoint generation, this is in 2D to make it easier to understand. Examples of the waypoint generation working on a 3D map is provided in the results section.

# Chapter 5

# Results

## 5.1 Tests

To try out the behavior of the planner a test map was made with an L-shaped wall obstacle to navigate around. The map was made as a point cloud and converted into an OctoMap and published by the OctoMap server plug-in.

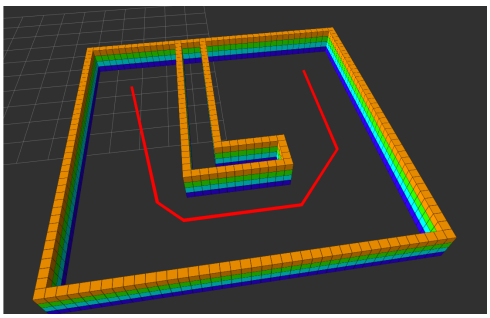
In order to show how the WAPP parameters affect the behavior a series of simulations were made. Each of them were done on the same map with the same start and goal, with the three cost parameters set to different levels. In addition to this one of the configurations were run at two different map resolutions, to illustrate how WAPP would work on a larger map of the same shape.

The following table shows how how the different configurations affect how large a percentage of the map is explored, which illustrates how different parameter settings affect the performance of the program.

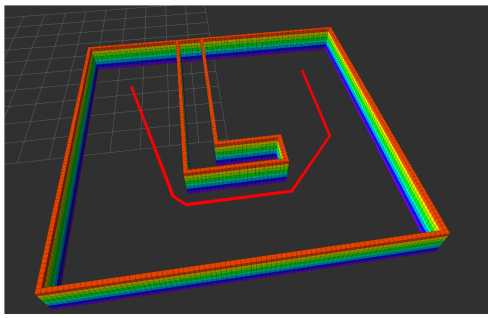
### 5.1.1 Building Interior Map(pseudo-2D)

| WCL | MCL | GCL | Map exp. | Analysis |
|---|---|---|---|---|
| 10 | 1 | 1 | 48.6% | Decent exploration considering the narrow shape of the map |
| 10 | 1 | 1 | 48.0% | Same behavior as on the lower resolution map. Different percentage of explored vertices is due to a lower percentage of wall vertices on this resolution. |
| 10 | 10 | 10 | 53.6% | Slight decrease in efficiency, probably due to more vertices being explored near the corner of the obstacle |
| 10 | 10 | 1 | 74.4% | Same path generated as when all parameters were equal, but with a sharp decrease in efficiency due to lowered WCL. |
| 10 | 1 | 10 | 35.0% | Good path, with good wall clearance and great efficiency, best configuration for this map. |
| 1 | 1 | 10 | 34.1% | Even more efficient, but cutting very close to the corners. |
| 1 | 10 | 1 | 74.1% | Terrible efficiency and a path too near the walls as well. |
| 1 | 10 | 10 | 53.0% | Bad path, average efficiency. |

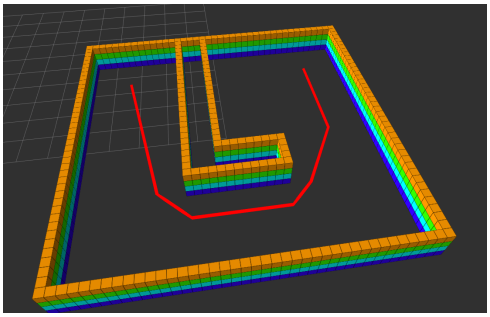
In the following images the paths generated by these simulations are shown.



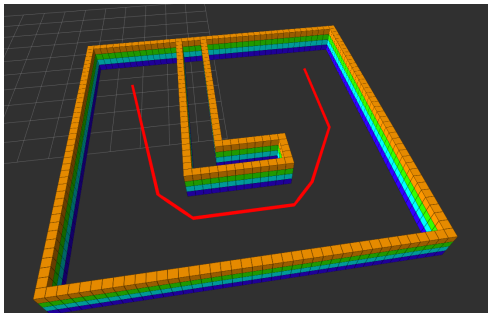
(A) 10,1,1



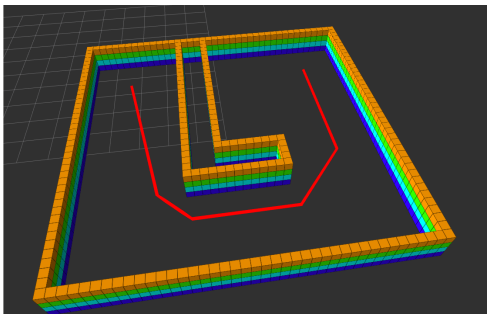
(B) 10,1,1 larger map



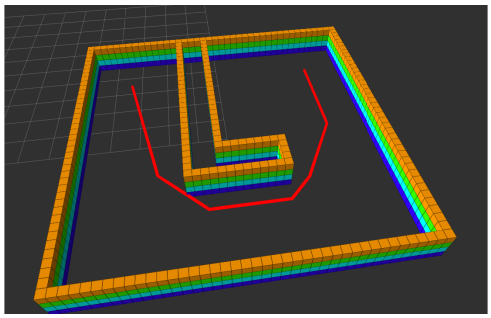
(C) 10,10,10



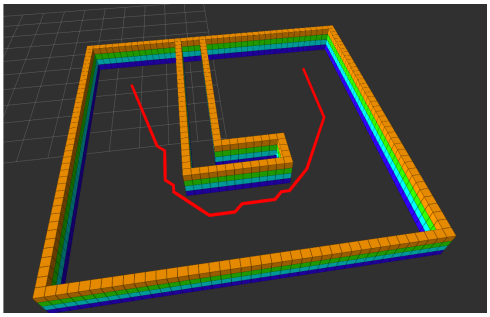
(D) 10,10,1



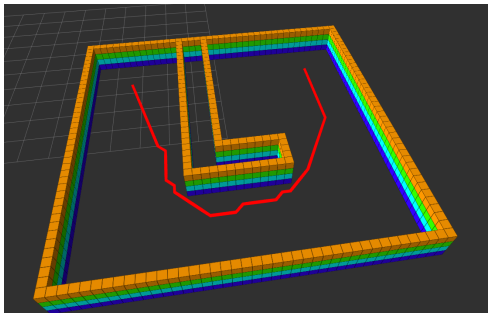
(E) 10,1,10



(F) 1,1,10



(G) 1,10,1

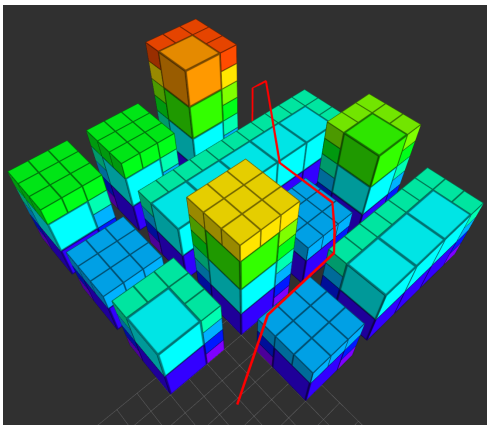


(H) 1,10,10

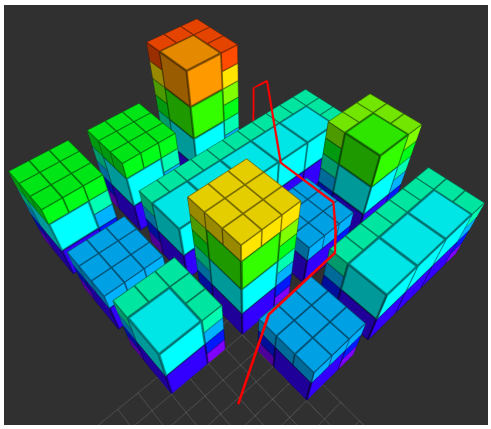
### 5.1.2 Cityscape Map(full 3D)

| WCL | MCL | GCL | Map exp. | Analysis |
|---|---|---|---|---|
| 10 | 1 | 1 | 48.6% | One of the least efficient configurations due to insufficiently low GCL/MCL ratio |
| 10 | 10 | 10 | 38.6% | Slight increase in efficiency, probably due to fewer vertices being explored along the long twisting city paths |
| 10 | 10 | 1 | 55.6% | Terribly inefficient configuration due to a very low GCL/MCL ratio. This is somewhat reduced by the WCL forcing the algorithm to avoid exploring between buildings |
| 10 | 1 | 10 | 4.39% | Good path, clears the city and explores the least area of the map. Great efficiency, best configuration for this map. |
| 1 | 1 | 10 | 4.83% | Nearly as efficient, but lower WCL allows for exploration between buildings which reduces efficiency on this map. |
| 1 | 10 | 1 | 64.5% | Terribly inefficient configuration due to a very low GCL/MCL ratio. |
| 1 | 10 | 10 | 44.3% | Insufficiently low GCL/MCL ratio, low efficiency. |

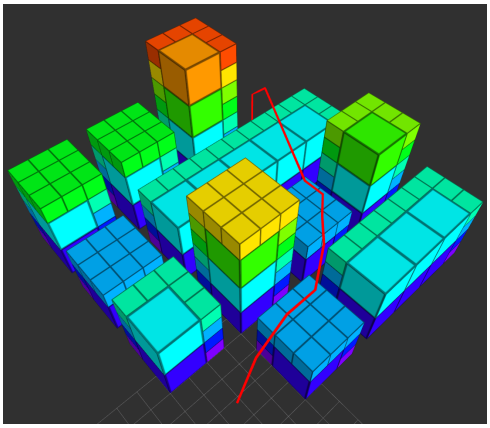
The results from path planning simulations on the cityscape map are similar to those of the building interior map but reveals one big difference. On this map an increased wall penalty actually improves the efficiency of the planner whereas on the interior map there was a decrease in efficiency following a heightened WCL. The reason for this is that, on this map, a higher WCL causes the planner to explore in an arc above the obstacles and avoid the areas in between the buildings. On this map going through the streets in between the buildings is both higher risk as well as longer in distance which means the planner finds a path quicker and more efficiently with a higher WCL. The effects of GCL and MCL on the efficiency is the same as on the previous map.
In the following images the paths generated by these simulations are shown.



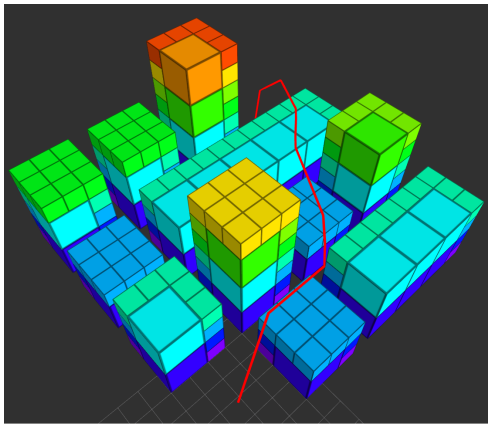
(A) 10,1,1



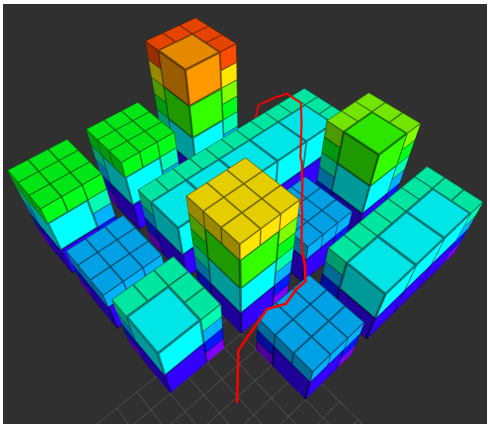
(B) 1,1,1



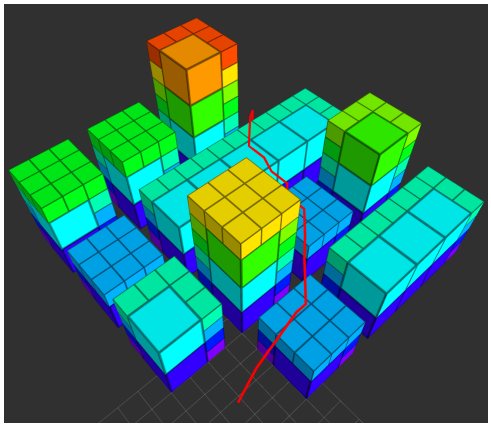
(C) 10,10,1



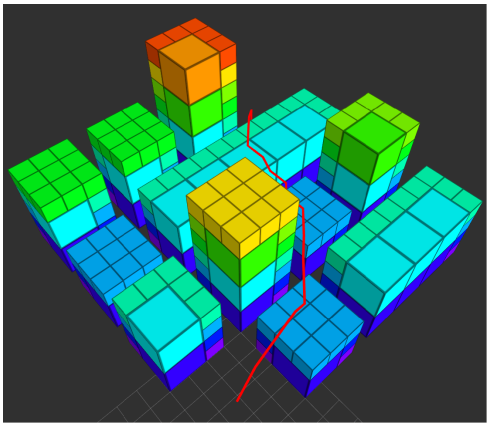
(D) 10,1,10



(E) 1,1,10



(F) 1,10,1



(G) 1,10,10

### 5.1.3 Analysis of the Cost Parameters Impact

**Wall Cost Level** The way the WCL affects the planner behavior is different from the other two parameters in that in most cases there is a certain level were it stops having any impact on the planner what so ever. This threshold varies between maps and depends on how long the longest detour is that would be needed to avoid exploring near a wall vertex.

If the only path there is between the start and the goal goes through a narrow passage where complete wall avoidance is impossible, a high enough WCL will force the program to explore every single vertex in the area before the passage. This will lead to the program planning a path through the passage anyway, if there is a path one will always be returned.

If the WCL is set too low compared to the other parameters it will simply lead to paths being planned near the walls. this might be acceptable in some scenarios, for example if the path is being planned for a slow moving ground vehicle which is in low risk of bumping into walls, and which will not be damaged were it to happen.

One surprising find from the cityscape simulations is an example of how a higher WCL can lead to increased efficiency in some situations. When the shortest path is not the straight line between start and goal due to a clutter of objects blocking the straight line path a higher WCL can prevent the algorithm from exploring winding twisting paths between objects. This reduces the number of computations needed to find the quicker path around the obstacles.

**Move Cost Level** The MCL is what keeps the planner focused on the efficiency of the planned path. This is accomplished by putting the emphasis more on how far from the start a vertex is when it comes to prioritizing which vertex to explore next. The MCL needs to be high enough that enough vertices are explored that the path can actually take the shortest route. If the MCL is too small in comparison to the GCL the program will end up exploring in a straight line, only a single vertex wide, only diverging from this to pass by obstacles. This can lead to suboptimal paths being planned.

Providing that the WCL is set high enough this is largely fixed by the way-point generation, but the possibility of suboptimal paths is still existent.

**Goal Cost Level** As can be seen clearly in the simulation data the GCL is the main parameter when it comes to increasing the computational efficiency of the planner. A higher GCL causes the program to explore fewer vertices where there is an open space to cross on the way toward the goal.

As Previously mentioned in the previous segment a too high GCL compared to the MCL may lead to suboptimal paths, and in addition to this a too strong pull towards the goal may result in a shorter path not being found at all. This can happen if the planner prioritizes vertices nearest a straight line to the goal so much that it fails to find a shorter path.

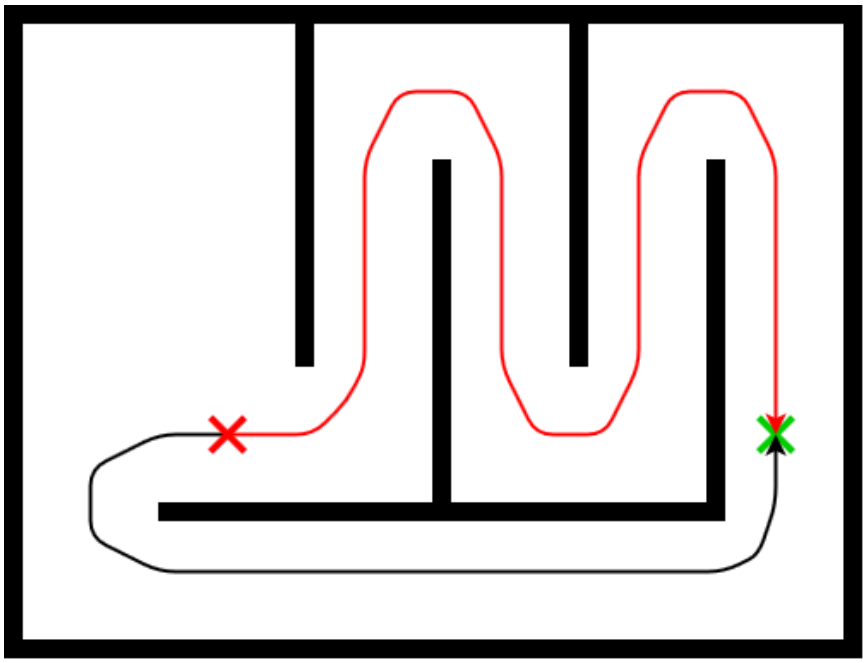


FIGURE 5.3: An example of how a path generated with too high GCL(red) could be suboptimal, as well as a better alternative(black), generated with a lower GCL

Although these risks exist, on most maps the most efficient configuration would be keeping the GCL higher than the MCL.

# Chapter 6

# Future Work

## 6.1 Easily adjustable wall costs

Currently the code allows for easily changing linearly the size of the penalties added to vertices surrounding a wall vertex, but not the number of vertices that gets the penalty. This means that depending on the resolution of the map, the real-world distance of a vertex to the wall that gets an added penalty. The planner could be changed to have a parameter input for a radius for the penalty as well, with the function adding the penalty being modified to accept this input and generate a variable size penalty zone.

## 6.2 Re-planning

Since the code is based on LPA* it is already prepared for re-planning, greatly saving computation time on larger maps. At the moment the code only computes the path once, but it is created to be able to re-use map data that has not changed since the last time the planner was run. This I planned to do, but the time of getting it working wasn't there

## 6.3 Trajectory Planning

The way the planner creates a list of way points without going too near walls, which could either be used as the actual trajectory, or could be the first step to planning a more complex trajectory for a UAV. Such a planner could calculate a trajectory taking into account varying speed, the robots inertia and other factors.

Here the positions of the waypoints far from the walls where possible, and more clustered where walls couldn't be avoided, would make using a polynomial curve fitting algorithm easier to use in order to turn the series of points into a smoother curve for a UAV to follow.

There is also information that would help set upper limits to UAV speed that is generated when the planner generates a path. First of all OctoMap itself makes a difference between open vertices and unknown vertices and even though the planner assumes all non-wall vertices to be open one could tell a UAV to fly slower when entering areas with more unknown vertices. The planner provides info as well when it comes to areas with more walls. This could either be read directly from the wall penalty values in the planners map, or from the waypoint density of the generated path.

# Bibliography

Dijkstra, E. (1959). “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik*.

Gammell, J., S. Srinivasa, and T. Barfoot (2014). “Informed RRT* Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic”. In:

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In:

Hornung, A. et al. (2013). “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees”. In: *Autonomous Robots*.

Koenig, S. and M. Likhachev (2005). “D* Lite”. In:

Koenig, S., M. Likhachev, and D. Furcy (2005). “Lifelong Planning A*”. In:

Lavalle, S. (1998). “Rapidly-exploring random trees: A new tool for path planning”. In:

“ROS”. In: URL: http://www.ros.org.

Stentz, A. (1994). “Optimal and Efficient Path Planning for Partially-Known Environments”. In:

Luleå University of Technology

# *Abstract*

Faculty of science and technology
Department of Computer Science, Electrical and Space Engineering, Signals and Systems.

Master of Aerospace Engineering

**Path Planning with Weighted Wall Regions using OctoMap**

by Jerker Bergström

In the work of the Control Engineering research group of the Department of Computer Science, Electrical and Space Engineering, Signals and systems at Luleå University of Technology a need had arisen for a path planning algorithm. The ongoing research with Unmanned Aerial Vehicles(UAVs) had so far been done with any complicated paths being created manually with waypoints set by the uses. To remove this labourious part of the experimental process a path should be generated automatically by simply providing a program with the position of the UAV, the goal to which the user wants it to move, as well as information about the UAV's surroundings in the form of a 3D map.

In addition to simply finding an available path through a 3D environment the path should also be adapted to the risks that the physical environment poses to a flying robot. This was achieved by adapting a previously developed algorithm, which did the simple path planning task well, by adding a penalty weight to areas near obstacles, pushing the generated path away from them. The planner was developed working with the OctoMap map system which represents the physical world by segmenting it into cubes of either open or occupied space. The open segments of these maps could then be used as vertices of a graph that the planning algorithm could traverse. The algorithm itself was written in C++ as a node of the Robot Operating System(ROS) software framework to allow it to smoothly interact with previously developed software used by the Control Engineering Robotics Group.

The program was tested by simulations where the path planner ROS node was sent maps as well as UAV position and intended goal. These simulations provided valid paths, with the performance of the algorithm as well as the quality of the paths being evaluated for varying configurations of the planners parameters. The planner works well in simulation and is deemed ready for use in practical experiments.

# *Acknowledgements*

I would first like to thank my supervisors Emil and George for their help and support throughout the thesis work. Emil especially for his work as the C++ guru of the office, solving any problem that would thwart my progress in no time, and teaching me some of the finer points of the language along the way. I would also like to thank Christoforos Kanellakis who provided lots of help with the map system. Without him wrapping my head around the workings of that less than optimally documented piece of software would have been even more of a pain.

# Contents

**Abstract** iii

**Acknowledgements** v

**1 Goals** 1
- 1.1 Introduction . . . 1
  - 1.1.1 What is a path? . . . 1
  - 1.1.2 What is a path planner? . . . 1
  - 1.1.3 The lay of the land . . . 2
- 1.2 Goal . . . 2
  - 1.2.1 Requirements on the planner . . . 2

**2 Software used** 5
- 2.1 ROS . . . 5
- 2.2 OctoMap . . . 7

**3 Path Planner Algorithms** 9
- 3.1 Path planning in general . . . 9
- 3.2 Algorithm alternatives . . . 9
  - 3.2.1 Simple planners and their flaws. . . . 9
  - 3.2.2 A* basics . . . 10
  - 3.2.3 A* Efficiency and Optimality Issues . . . 10
  - 3.2.4 LPA* . . . 10
  - 3.2.5 D* . . . 11
  - 3.2.6 D* lite . . . 11
  - 3.2.7 Random Sample based solutions . . . 11
- 3.3 Choice of algorithm . . . 12
- 3.4 Differences between WAPP and other A*-derivatives . . . 12
  - 3.4.1 Penalties for moving close to walls . . . 12
  - 3.4.2 Reduction of the Path to Waypoints . . . 13

**4 How the Planner Operates** 15
- 4.1 Overview . . . 15
- 4.2 Initialization . . . 16
  - 4.2.1 Inputs and basic workings . . . 16
  - 4.2.2 Parameters . . . 17
  - 4.2.3 Loading . . . 17
  - 4.2.4 Verification of Start and Goal . . . 18
- 4.3 The Search . . . 20
  - 4.3.1 The Vertex Costs . . . 20
  - 4.3.2 The Priority List . . . 20
  - 4.3.3 The Expand Function . . . 21
  - 4.3.4 The Search . . . 22

4.4 Way-points . . . . . 23

**5 Results** **25**
5.1 Tests . . . . . 25
5.1.1 Building Interior Map(pseudo-2D) . . . . . 25
5.1.2 Cityscape Map(full 3D) . . . . . 27
5.1.3 Analysis of the Cost Parameters Impact . . . . . 29

**6 Future Work** **31**
6.1 Easily adjustable wall costs . . . . . 31
6.2 Re-planning . . . . . 31
6.3 Trajectory Planning . . . . . 31

**Bibliography** **33**

# List of Figures

1.1 An example of a tree being represented as a cubic honeycomb in the OctoMap format . . . . . 2

2.1 Example of what the network of nodes running a robot able to move autonomously to a user specified goal could look like. . . . . 6
2.2 The structure of an octree and the cube shaped space it represents. . . 7
2.3 OctoMap storing one occupied vertex(black) within a larger free(white) space. . . . . 8
2.4 The same object represented as OctoMaps at three different resolutions. 8

3.1 Comparison of a path generated by A* on a 2D grid with 8 directions of movement and the optimal path. . . . . 10
3.2 RRT space filling tree. . . . . 12
3.3 A path(red) though a 2D map, reduced to waypoints(black). Blue vertices are walls, orange free explored and white are free unexplored . 13

4.1 An example of a map corresponding to the real coordinates $x_{min} = y_{max} = 5$, $x_{min} = y_{max} = 5$, with real coordinates shown in red, and the planner coordinates shown in black. . . . . 17
4.2 How penalty is distributed around a wall vertex. . . . . 18
4.3 The Initialization process. . . . . 19
4.4 The vertex updating process for every vertex examined by the expand function . . . . . 21
4.5 The Search process. . . . . 22
4.6 The Waypoint generation process. . . . . 23
4.7 A path(red) though a 2D map, reduced to waypoints(black). Blue vertices are walls, orange free explored and white are free unexplored . 23

5.3 An example of how a path generated with too high GCL(red) could be suboptimal, as well as a better alternative(black), generated with a lower GCL . . . . . 30

# List of Abbreviations

| | |
|---|---|
| **WAPP** | **W**all **A**voiding **P**ath **P**lanner |
| **UAV** | **U**nmanned **A**erial **V**ehicle |
| **WCL** | **W**all **C**ost **L**evel |
| **MCL** | **M**ove **C**ost **L**evel |
| **GCL** | **G**oal **C**ost **L**evel |
| **ROS** | **R**obot **O**perating **S**ystem |

# Chapter 1

# Goals

## 1.1 Introduction

When a robot moves through an environment many different components need to interact. The robot will need to know its position in space, it will need either a map of its surroundings or a way to generate a map through sensory input, and then there is the actual motion. Electrical engines will need to run at appropriate speeds, servos controlling steering or control surfaces will need to move. Then there is keeping track of the direction, speed and acceleration of the robot, all which is needed to get the robot to move in the direction in which its operator wants it to move.

But even if all these things are functioning, where should the robot move to? Suppose it has been given a goal, a point on its map to where it wants to move. Does it just aim for that point and move straight towards it? What if something is in the way? What if the robot is situated in a building with rooms and corridors, or even in a maze?

For the robot to autonomously move towards its goal it needs more detailed instructions, it needs a series of coordinates all in line of sight of each other, starting at the robots position and ending at the goal.

The robot needs a path.

### 1.1.1 What is a path?

What path means in this case is a path though 3D space which a UAV can follow without colliding with obstacles. This could be a smooth, continuous curve generated by taking into the dynamics of flying or, in its most basic form, a simple series of coordinates which a robot could follow sequentially in a series of straight lines. This simple type of path is what is generated by the path planner developed during this thesis work.

The problem that needs addressing is thus how to take information about the physical environment, the position of a robot and it's desired destination and extract a viable path using this data.

### 1.1.2 What is a path planner?

A path planner is a program designed to solve this problem, to figure out a path through its environment without colliding with walls or other obstacles whilst doing so. How this is achieved varies and depends largely on how the map containing the walls and obstacles work, but generally it boils down to representing the map as a weighted graph that can be searched for a path between two of its vertices. This can either be an inherent feature of the map, such as on maps that represents the world as a cubic honeycomb of either free or occupied cubes of space. The free vertices of such maps essentially make up one or more searchable graphs, one if all free vertices

are interconnected, more if there are groups of them that are blocked off from each other.

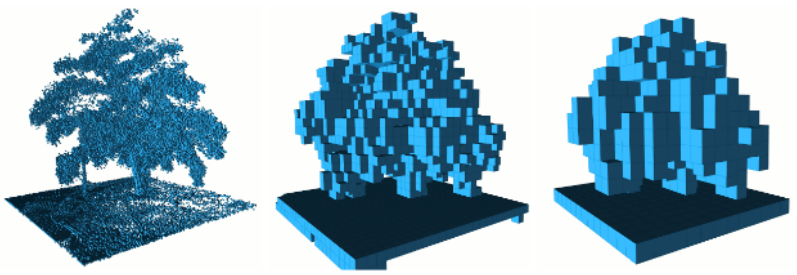


FIGURE 1.1: An example of a tree being represented as a cubic honeycomb in the OctoMap format. [1]

Another method of generating a searchable graph is to use randomly sampled points on the map and see which of them can be connected to neighbouring points without causing a collision. The points that can be connected forms a grid around the various obstacles, and gradually the resolution can be increased by adding new points and connecting them to nearby previously added points. As the point density increases the chances that there is a connection around obstacles between any two points on the map eventually approaches one.

When the graph is established there is a multitude of options for search algorithms, a selection of which will be presented in chapter 3.

### 1.1.3 The lay of the land

In the robotics research group, for which the path planner created during this thesis work was made, there was already a map system in use. This map system is called OctoMapHornung et al., 2013 and is of the previously mentioned type that abstracts the 3D environment into a cubic honeycomb of free or occupied spaces. To be able to integrate the planner with software already in use and to be able to draw on previous knowledge and experience available in the research group, it was deemed that using the OctoMap format as the map input to the planner would be best.

Both the map software and all other software components which the path planner will need to interact with runs under the Robot Operating System (ROS)"ROS". ROS is a software framework for robotics which lets the various programs, called nodes, communicate and interact This means the path planner will need to be a ROS node as well.

## 1.2 Goal

The goal of this thesis work was to implement a functioning path planner that could be of practical use in research and experiments. The path planner should generate effective paths that can be used both as a simple trajectory for UAVs and as a basis for further development.

### 1.2.1 Requirements on the planner

The following requirements for the planner were given:

- The planner should be deterministic(i.e. if there is a path to the goal, the planner should find it and not get stuck in local minima)
- The planner should read 3D grid maps in the OctoMap format, using ROS node(s) to do so to facilitate integration with systems in use.
- The Planner should avoid planning paths near walls, but when no other path is available plan paths through more narrow passages. This is both to reduce the risks that comes with trying to fly too close to a wall, as well as facilitating an easier later implementation of a trajectory planner.

# Chapter 2

# Software used

## 2.1 ROS

The Robot Operating System is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

Even though ROS stands the Robot Operating System, it is not an operating system per se, but rather a software framework, or so called middleware, for use in robotics. The programs running under ROS are called nodes and what ROS mainly does is providing easy data communication between them. The nodes can publish data to topics or subscribe to topics and receive data when it is published to them. The data sent between nodes can be sensor data, coordinates or any info that various nodes might need to share to facilitate the use of a robot and the performance of its various functions.

The following image shows a visualization of how the path planner constructed during this thesis work could fit into a network of ROS nodes.

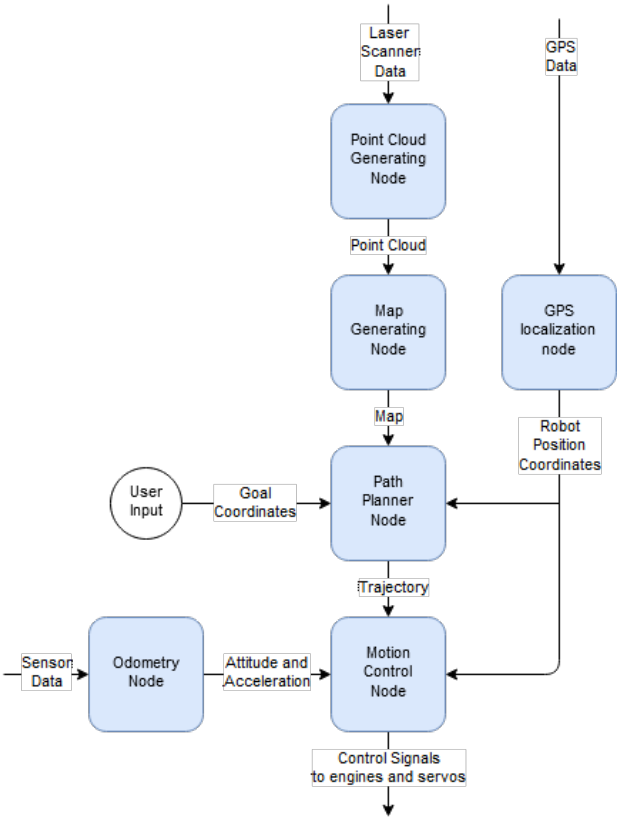



FIGURE 2.1: Example of what the network of nodes running a robot able to move autonomously to a user specified goal could look like.

The path planner node is being fed data though the ROS topic system from three sources. Firstly it receives the goal coordinates from a user input which is the only non-automatic part of this system. In addition to this input the robot receives its own position from a GPS localization node which in turn receives its data directly from the signals from a GPS receiver device on the robot.

The map data needed by the planner is provided by a map generating node which transforms a point cloud message into a map format that the planner can use. This point cloud is supplied by a node which receives measurement data from a rotating laser scanner which measures the distance between the scanner and objects and walls in proximity tot he robot. This measurement data is published as a ROS topic and made available to the map generating node.

When provided with its three needed inputs the planner node generates a path which is published as the desired trajectory for the robot. This trajectory is then received by a motion control node which uses the trajectory, the robot position and odometry data and computes how the robot shall move and when, and proceeds to send the required control signals to the various servos and motors that need to be activated to move the robot along the path.

This is just an example of how it could work, and all the nodes work in the same fashion no matter what nodes were publishing the input data or subscribing to the output data. This makes a system running ROS highly flexible allowing different configurations of nodes to smoothly interact.

The reason for using ROS is mainly that is provides easy integration with existing software as well as with future developments within the robotics group. There are several other similar types of robotics middleware that could replace ROS but since this work was done to provide a working piece of software to a research group where ROS was being used these alternatives were neither investigated nor seriously considered, due to practicality and time constraints.

The program in its entirety was written in C++ in the ROS environment.

## 2.2 OctoMap

The map that the path planner will navigate will be come in the form of an OctoMap, published in ROS by the OctoMap server plug-in. The OctoMap library uses an octree based cubic grid system to represent three dimensional space, with data structures and mapping algorithms provided in C++. An octree is a data structure that divides the space into cubes, subdivided into smaller octant cubes, which in turn are subdivided into octants and so on until a set minimum cube size is reached. Cubes whose constituent octants are all either occupied or open can be saved as a single element. This results in a great reduction in file size on maps with large areas where all vertices are either occupied or open.

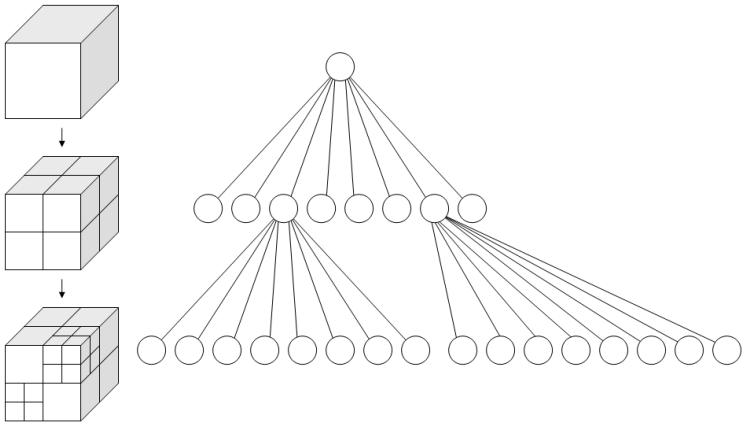


FIGURE 2.2: The structure of an octree and the cube shaped space it represents.[1]

This allows OctoMap to efficiently store a model of a 3D environment that can rapidly be sent between robots running ROS, or between nodes on a single ROS using device. This compression affects how the OctoMap messages are stored and sent, but have no real impact on the workings of the planner algorithm, to which the OctoMap seems to be made up of equal sized cubes of a size corresponding to the maps resolution.

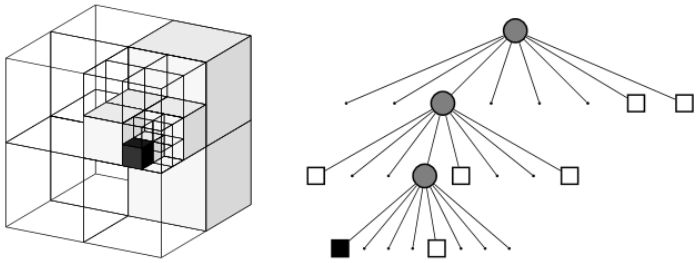


FIGURE 2.3: OctoMap storing one occupied vertex(black) within a larger free(white) space. [2]

OctoMap allows for three different states for each division of space: free, occupied and unknown. The unknown status was not used in this thesis work, but it could be useful at a later stage when planning the velocity of a UAV in different areas of the map.

The OctoMap plugin for ROS can either open an OctoMap from file or create one out of a point cloud. The point cloud is the list of coordinates where a detector such as a laser scanner has detected a surface. When generating an OctoMap from a point cloud one can set any resolution deemed suitable to the task at hand, depending on such factors as available computational power and the size of the robot that is to navigate the map. When loading a map from a file the minimum cube size can be set to that of the file or larger, allowing for quicker processing of maps of unnecessary high resolution.

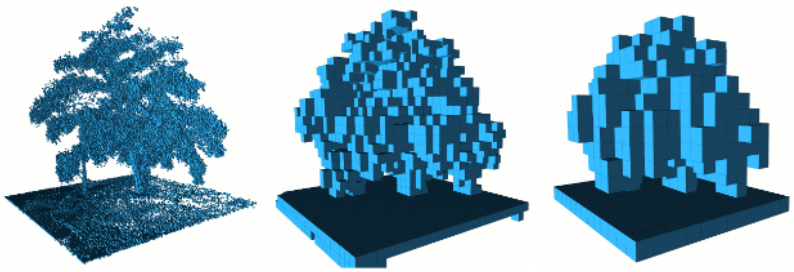


FIGURE 2.4: The same object represented as OctoMaps at three different resolutions. [3]

# Chapter 3

# Path Planner Algorithms

## 3.1 Path planning in general

The area of path planning came about as a subject within the area of Graph Theory, itself an area within discrete mathematics. More specifically the planning of paths between two points on a map is the solving of Graph Theories shortest path problem, how to find the cheapest path between two points on a weighted graph. There are several ways of doing this and countless variations of path planning algorithms has been developed over the years.

## 3.2 Algorithm alternatives

There are several methods of finding a path. The problem comes in two parts, firstly the generation of a graph that contains a path around any obstacles to the goal, and secondly the search algorithm which tries to find the best path through the grid once it has been established. The algorithms discussed in this chapter are mainly solutions to the second problem, that of efficiently finding a good path through a large graphs with many vertices. However some other solutions which focus more on the graph generation will receive a short mention as well.

### 3.2.1 Simple planners and their flaws.

One of the first solutions to the shortest path problems was Dijkstra's Algorithm (Dijkstra, 1959), conceived by Edsger Dijkstra in the late 1950s. Dijkstra's Algorithm finds the shortest path by simply searching all paths in order of their cost, starting at one end of the path. This means that whilst the search will find the shortest path in the graph, it will search aimlessly for it and it will end up searching vast areas of the map unnecessarily. This means that for any applications other than searching for paths through narrow, non branching corridors with the start at a dead end it is highly inefficient.

One example of a simple method of path planning that goes a completely different way is finding a path by simulating a potential field. Here the robot is made to act as a charged particle in a potential field with the goal attracting it and obstacles repulsing it. This solution can create a very quick planner that is clearly aimed at the goal and does not spend unnecessary computational power on searching irrelevant areas of the map. However, this approach is not certain to find a path to the goal, since it can lead to the robot getting stuck in local minima.

Both the very basic algorithms that go straight for the goal and the ones that explore the map methodically has serious flaws, and a combination of the two approaches is needed.

### 3.2.2 A* basics

One of the first solutions that markedly improved Dijkstra's algorithm was A*Hart, Nilsson, and Raphael, 1968, first described in 1968. This algorithm introduced a heuristic to Dijkstra's algorithm that made the planner give priority in the order of vertex exploration based on the distance to the goal in addition to the length of the path. Basically A* works like a variant of Dijkstra's that tries to explore only vertices relevant to the search for a path between two points. This algorithm was highly successful, and variants of it are in common use to this day.

### 3.2.3 A* Efficiency and Optimality Issues

A* and it's descendants will always generate a path if there is one on the grid, however there are a few things that determines how close this path will be to the optimal path. Firstly there is the resolution of the map. A higher resolution will lead to paths closer to the optimum, at the cost of longer computation times. Since the number of vertices in the map increase cubically to the increase in resolution this is a major factor. A lower resolution map will generate a path much quicker, but this comes at the cost of producing a path further from the optimum path, and risks closing off narrow passages that cannot be seen at too low a resolution.

Another obstacle that makes the path suboptimal is the fact that A* operating in a square or cube grid is limited to moving in a set number of angles. This is solved in a variety of ways, usually by either checking for optimal parts in a small area at a time while the planner is running, or by post processing of the generated path.

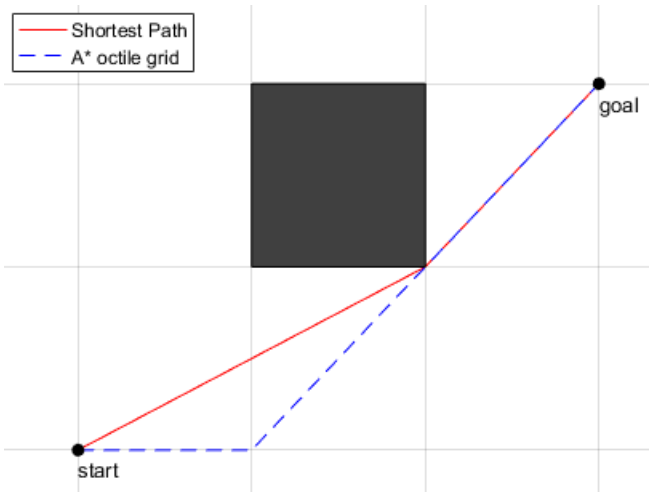



FIGURE 3.1: Comparison of a path generated by A* on a 2D grid with 8 directions of movement and the optimal path.[1]

When it comes to performance in terms of the computational weight of the algorithm the main issue with A* however is the fact that the basic algorithm is designed to solve the problem once only, and to get an updating path one has to run the algorithm over and over. This is one of the main reasons plain A* has to a large extent been replaced by updated variants of the algorithm.

### 3.2.4 LPA*

The lack of support for map changes in A* is dealt with in a very efficient manner by the LPA*Koenig, Likhachev, and Furcy, 2005, or lifelong planning A* algorithm.

LPA* deals with this in the following way: When the occupation status of a vertex changes, that is, a wall vertex is added or removed, the surrounding vertices are checked to see if the move distance from the start is consistent with those of the vertices that surround them. This means that vertices look at their neighbours to check if their data is up to date, or if they have been blocked off. For each vertex that was found to have changed the neighbours of that vertex is then checked, forming a wave that identifies all vertices whose data is now outdated, add them as potential targets for exploration again. This leads to only the information that is outdated is discarded, and the segments of the map which were explored stays that way for the next search, making the re-planning more efficient.

When this method is applied one has to take into consideration the direction of the search as well. If the map is generated or updated by a scanner positioned on the robot the planning should generally be done from the goal in the direction of the robot position. The reason for this is that when a vertex occupation status changes near the robot the direction of the wave of vertex recalculations will move toward the robot and only a mall part of the map will need to be recalculated at a single time. Thus the rule is generally that the direction of the search should go from the end of the path where vertices are more likely to change towards the end where this is less likely.

### 3.2.5 D*

D*Stentz, 1994, or Dynamic A* is a variant of A* made to be used with a moving end of the path, with the algorithm handling the re-planning this requires. D* works better than multiple runs of A*, but uses a fairly complex and inefficient algorithm using pointers at every vertex pointing to a parent vertex.

### 3.2.6 D* lite

D* liteKoenig and Likhachev, 2005 is effectively a combination of D* and LPA*, implementing the behavior and support for a moving end of the path from the former, and the efficient re-planning system of the latter. The efficiency of D* lite has led it to become somewhat of an industry standard over the last decade.

### 3.2.7 Random Sample based solutions

One way to solve path planning problems used by algorithms such as RRTLavalle, 1998, or Rapidly-exploring Random Tree, is to make use of random samples to build a graph on which the path later is found. These methods are not restricted to a grid in the same manner as A* and can produce highly efficient paths if left to generate a complex enough map. However, these methods in their basic state are direction-less and wastes time on probing every area as frequently.

There are improvements to these methods, which fall into two categories. The first involves speeding up the convergence to optimality by restricting samples to an area nearby the path when it has been found. One example of this is Informed RRTGammell, Srinivasa, and Barfoot, 2014, which greatly improves the speed of which an optimal path is found by forcing the algorithm to sample in an area restricted by the outer surface of an ellipsoid based on the shape of the best known path.


FIGURE 3.2: RRT space filling tree.[2]

The other method for improving the use of Algorithms such as RRT is by first using a low-resolution run of A* or a similar planner, and then using RRT to improve upon the path by using a similar restriction as Informed RRT.

## 3.3 Choice of algorithm

Since the planner should work on the OctoMap square grid maps generated in various experiments A* and it's derivatives are especially suited for this due to them being made for square grid as well as them being deterministic, and always finding a path if there is one. In addition the heuristic-system for A*-derived planners are highly modifiable which makes the adaption of the wall avoidance objective fairly straight forward to implement. A variant of A*, similar to D*lite in its mechanism, was chosen due to this, and since it is a very well established and widely used it seems like an excellent options for an OctoMap reading path planner. This path planner algorithm will be referred to as WAPP, which stands for Wall Avoiding Path Planner

## 3.4 Differences between WAPP and other A*-derivatives

### 3.4.1 Penalties for moving close to walls

To add a penalty for going close to walls an additional cost is added to moving to the vertices closest to the wall. The penalties are added to all vertices that lies within the boundaries of a sphere, in this case with a diameter of seven vertices. The distance this represent in reality is dependent on map resolution, and with a high resolution map the radius of the sphere of added wall penalties would probably need to be increased. On the other hand, the use of high resolution maps is unmotivated unless ones robot was very small and nimble and able to pass though passages, which might need less free space, so perhaps this is less of an issue than it might initially appear.

### 3.4.2 Reduction of the Path to Waypoints

WAPP will be used to generate paths for UAVs in the form of a trajectory made up of waypoints, published in the MultiDOFJointTrajectory Message format. This reduction of an A* style path to a list of waypoints serves different purposes.

First and foremost it solves the problem of the planner being restricted when it comes to which directions it can plan in. The 45° minimum angle for turns allowed in a cube grid is removed by letting the path go between vertices that are not neighbours. This solution needs to examine fewer number of vertices to interpolate between them compared to some any-angle A* variants which continuously do interpolation during the expansion part of the algorithm.

Secondly the waypoint generation will only reduce series of path vertices to straight lines if said vertices are not in the proximity of a wall. This leads to a higher waypoint density in areas with a larger collision risk for a UAV, which can make it easier to implement a trajectory planner that converts the path into a continuous curve, leaving a curve fitting algorithm more freedom to go off the path where there are no walls nearby. Or if a UAV simply is set to pass by every single waypoint on the list, it will be forced to more more slowly and carefully in areas where there are walls to avoid. Thirdly the waypoint generation process will compensate for suboptimal paths which can be generated when setting the goal distance heuristic to be the dominant factor in the exploration prioritization.

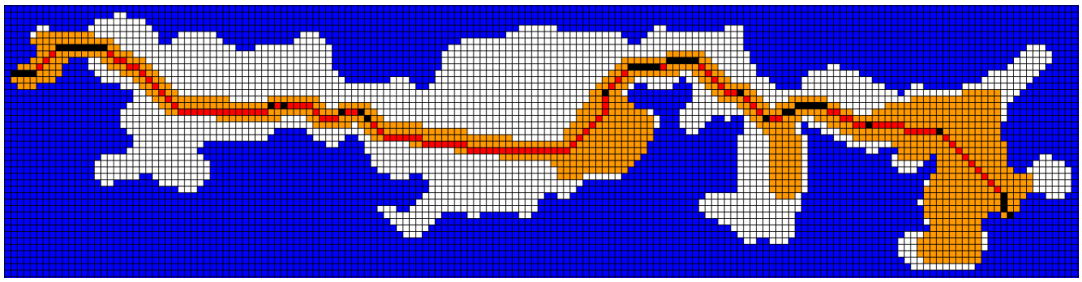


FIGURE 3.3: A path(red) though a 2D map, reduced to waypoints(black). Blue vertices are walls, orange free explored and white are free unexplored

Note that the example image is in 2D rather than 3D. This is merely to make the behaviour of the waypoint generation easier to understand for the viewer. The algorithm works in the same manner on a 3D map as when generating waypoints on a flat map only one OctoMap block deep.

# Chapter 4

# How the Planner Operates

## 4.1 Overview

The path planner can be divided into three parts, The Initialization, the Search and the waypoint generation.

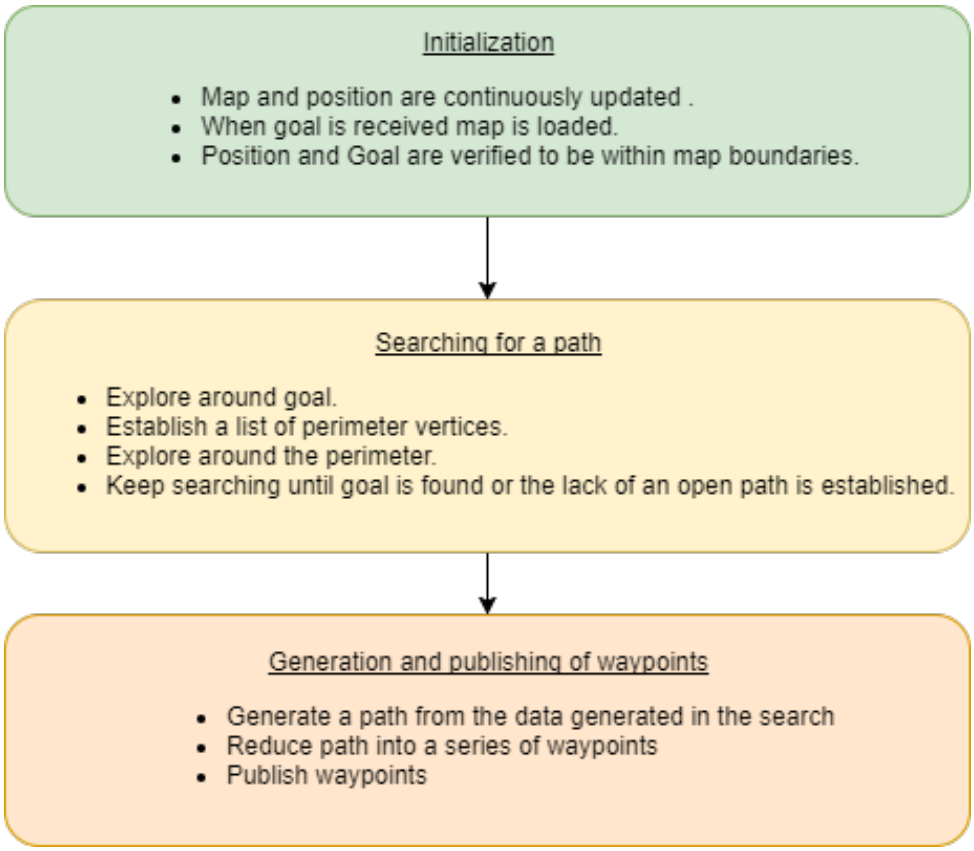
- The initialization contains all the things that need to take place before the actual path planning can take place. This includes the node receiving the coordinates for the current robot position and goal, as well as the loading of the map. This part makes sure the requirements for the path planner has been fulfilled, and if this is not the case the program will wait until that is the case.

- The Search is the core of the program, which searches the map, exploring until a path to the goal has been found. It does this by searching the map, first expanding around the start, assigning values to the vertices based on their distance to the goal(referred to as the g-cost) and the length of path needed to move to them from the start(referred to as the m-cost). The program keeps track of the vertices who received these values and picks the next centre of expansion from them based on the two costs. The program repeats this until the goal has been reached or it has been established that there is no path that leads to the goal.

- The waypoint generation is done if a path to the goal was found during the search process in order to make the path smoother and better for both direct UAV use as well as for use as a basis for more advanced UAV motion planning.

  It finds the path by picking from amongst its neighbours the one with the lowest m-cost, with represents the path length, then does the same with that neighbour, and continues as such until the robot position has been reached.

  It then uses the list of these vertices and removes any vertices that can be bypassed by moving in a straight line between the previous vertex and the next on the list. After this is done the waypoints are published to a ROS topic to be used by other nodes.




## 4.2 Initialization

### 4.2.1 Inputs and basic workings

WAPP has three inputs which it receives by subscribing to three ROS topics to which other nodes can publish them. These are the map, the position of the robot and the goal to which the robot's operator wants the robot to move.

The map is an OctoMap which is sent in a compact serialized format to ensure quick transfer rates. The two coordinates come in the format geometry_msgs:: TransformStamped which is a ROS format used by nodes to transmit both the position and the orientation of objects. In this case though only the position coordinates contained within these messages are read.

The map and the position are continually updated and saved while the program waits for the operator to send it a request for a path to be generated. This request is the act of sending the planner a goal coordinate by publishing it to the topic which WAPP node is subscribed to. Every time a new goal is sent to the WAPP node a path will be generated.

When the node receives the goal coordinates it first checks that an OctoMap as well as the position coordinates have already been received and saved. If this is confirmed the program first goes ahead and loads the planner parameters.

### 4.2.2 Parameters

The parameters are the settings that dictate the behavior of the planner, and during the initialization they are loaded. These parameters are loaded from a .launch file in the WAPP nodes ROS directory and are set by the operator beforehand. The search process is guided by three motivations: Moving towards the goal, keeping the path of minimal length and keeping the path away from walls and obstacles. The parameters are the weight values which guide this behavior, the goal-cost level(GCL), the move-cost level(MCL) and the wall-cost level(WCL).

These parameters are simply integer values that set the behaviour of the planner algorithm, the ratio between them is what affects how WAPP prioritizes between the three aforementioned motivations.

If the launch file containing the parameters is missing the parameters are set to hard-coded default values of GCL=1, MCL=1, WCL=10.

### 4.2.3 Loading

When the parameters have been loaded, the map can be deserialized and properly loaded. First of all the planner begins with identifying the dimensions of the map and it's resolution using built in functions of the OctoMap package for C++. The planner map is created as a three dimensional data structure with the same amount of vertices as the OctoMap, but with the three coordinates all starting at 0 and the resolution being 1. This is in contrast to the OctoMap which allows for negative coordinates and resolutions smaller or larger than 1. This coordinate transform is done in the following manner:

$$x_{planner} = (x_{OctoMap} + x_{min,OctoMap}) * s + 0.5 \tag{4.1}$$

$$y_{planner} = (y_{OctoMap} + y_{min,OctoMap}) * s + 0.5 \tag{4.2}$$

$$z_{planner} = (z_{OctoMap} + z_{min,OctoMap}) * s + 0.5 \tag{4.3}$$

Where s is the size of the OctoMap cubes

The 0.5 comes from the fact that WAPP works with the centers of the blocks in OctoMap, and without it the grid would be off one half vertex.

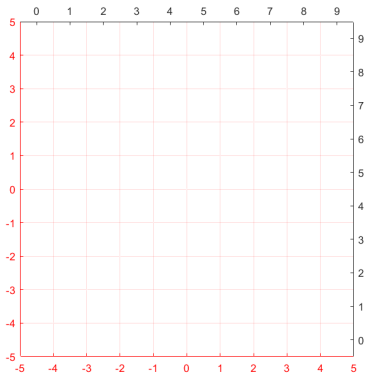



FIGURE 4.1: An example of a map corresponding to the real coordinates $x_{min} = y_{max} = 5$, $x_{min} = y_{max} = 5$, with real coordinates shown in red, and the planner coordinates shown in black.

The reason for creating an internal map in this manner is that the vertices of the map needs to be able to store several values in addition to their occupation status, these are previously mentioned g-cost and m-cost, a tag to keep track of previously expanded vertices as well as the penalty a vertex receives for being in near proximity of a wall vertex.

This wall penalty is added to the vertices during loading in the following manner: As the map is loaded into this data structure, every time a wall vertex is added a wall proximity penalty is added to vertices in a sphere surrounding it. The diameter of the sphere is seven vertices, so that vertices whose distance to the wall vertex is less than 3.5 times the OctoMap cube size will receive a penalty. The sphere in which the costs are added contains two smaller spheres that adds higher penalties, to the effect that the penalty at the edge of the largest sphere is simply the WCL, whilst vertices nearer to the wall vertex receive as penalty the WCL multiplied by higher numbers, as shown below.

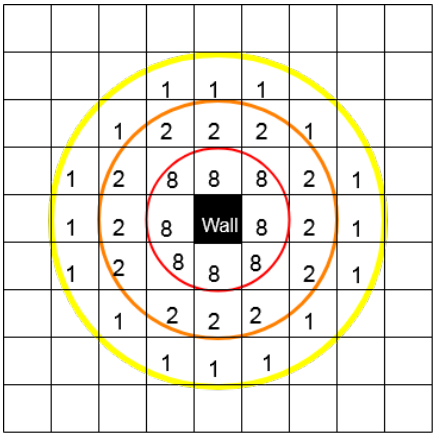



FIGURE 4.2: How penalty is distributed around a wall vertex.

These penalties are cumulative, so that vertices in the proximity of several wall vertices will receive large penalties. The effect this penalty has on the behavior of the path planner will be discussed in the subsection dedicated to the Search part of the program.

### 4.2.4 Verification of Start and Goal

When the map has been loaded the start and the goal are verified to be within the map boundaries by making sure the following conditions are true:

$$x_{min,OctoMap} \leq x_{position} \leq x_{max,OctoMap} \tag{4.4}$$

$$y_{min,OctoMap} \leq y_{position} \leq y_{max,OctoMap} \tag{4.5}$$

$$z_{min,OctoMap} \leq z_{position} \leq z_{max,OctoMap} \tag{4.6}$$

$$x_{min,OctoMap} \leq x_{goal} \leq x_{max,OctoMap} \tag{4.7}$$

$$y_{min,OctoMap} \leq y_{goal} \leq y_{max,OctoMap} \tag{4.8}$$

$$z_{min,OctoMap} \leq z_{goal} \leq z_{max,OctoMap} \tag{4.9}$$

This simply means that if the user feeds the program a goal or position coordinate that lies outside the physical area represented in the OctoMap the program will wait until a proper one has been sent.

When this is done the start and goal coordinates are converted in the same fashion as the map coordinates:

$$x_{start,planner} = (x_{position} + x_{min,OctoMap}) * s + 0.5 \tag{4.10}$$

$$y_{start,planner} = (y_{position} + y_{min,OctoMap}) * s + 0.5 \tag{4.11}$$

$$z_{start,planner} = (z_{position} + z_{min,OctoMap}) * s + 0.5 \tag{4.12}$$

$$x_{goal,planner} = (x_{goal} + x_{min,OctoMap}) * s + 0.5 \tag{4.13}$$

$$y_{goal,planner} = (y_{goal} + y_{min,OctoMap}) * s + 0.5 \tag{4.14}$$

$$z_{goal,planner} = (z_{goal} + z_{min,OctoMap}) * s + 0.5 \tag{4.15}$$

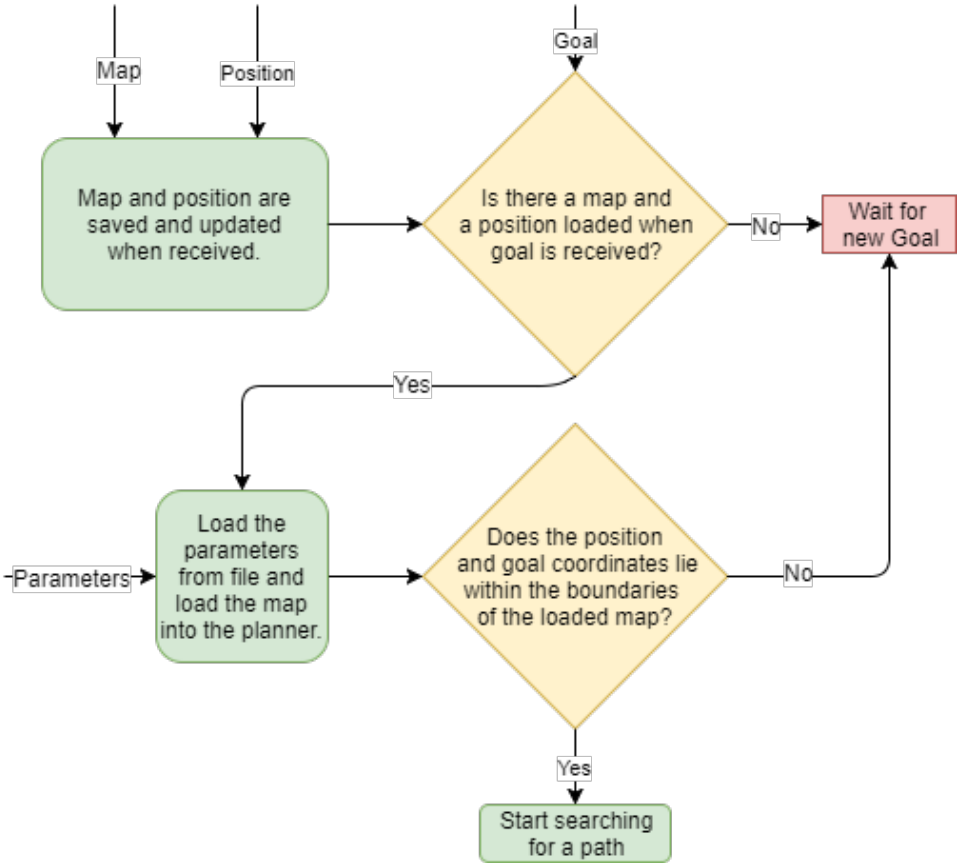



FIGURE 4.3: The Initialization process.

## 4.3 The Search

The search is done by exploring around a starting point, and repeatedly expanding the explored area by searching around its perimeter. To explain how this expansion works a few fundamental parts of the mechanism has to be explained first:the Vertex cost values, the priority list and the expand function.

### 4.3.1 The Vertex Costs

To differentiate between vertices in order to chose where in the map to explore next two costs are associated with the vertices, the g-cost and the m-cost. These values are assigned to a vertex by the expand function, and thus is only set for the explored map vertices The g-cost is for a vertex, v, is computed using the following formula:

$$gcost_v = 10\sqrt{(x_{goal} - x_v)^2 + (y_{goal} - y_v)^2 + (z_{goal} - z_v)^2} \tag{4.16}$$

The m-cost is for a vertex v computed using the following formula:

$$mcost_v = 10\sqrt{(x_{ec} - x_v)^2 + (y_{ec} - y_v)^2 + (z_{ec} - z_v)^2} + mcost_{ec} + pen_v \tag{4.17}$$

Where the index ec denotes values related to the expansion centre vertex from which v was explored, and pen is the wall proximity penalty added to the vertex during loading.

The multiplication by ten in both cases is to allow the number to be converted to an integer without losing too much accuracy. This is an efficiency measure, allowing the program to run almost exclusively integer operations which are faster to compute.

### 4.3.2 The Priority List

The priority list is basically a list of all the vertices that lies on the outer surface of the explored part of the map. This area, and the list with it, grows when the expand function picks one vertex at the surface as an expansion centre and adds its non-wall, unexplored neighbours to the list.

The list is used to sort out which vertex is to be the next centre of expansion, henceforth to be referred to as the expansion centre. In the list entries the coordinates of the vertices are saved, along with the sum of their g-cost and their m-cost, as well as the g-cost alone. An example of one of an entry of the vertex v to the priority list:

$$((mcost_v + gcost_v), gcost_v, x_v, y_v, z_v) \tag{4.18}$$

Everytime a new vertex is added to the list the list is sorted, first by their first values, the sum of g-cost and m-cost, and in case of a tie it is broken by g-cost.

### 4.3.3 The Expand Function

The expand function takes as its input the coordinates of a vertex on the outer surface of the explored area, the expansion centre. Around this vertex the surrounding box of vertices $v_{ijk}$ are investigated:

$v = (i, j, k)$
$(x_{ec}) - 1 \leq i \leq (x_{ec} + 1)$
$(y_{ec}) - 1 \leq j \leq (y_{ec} + 1)$
$(z_{ec}) - 1 \leq k \leq (z_{ec} + 1)$

If a vertex is occupied, or if it is the expansion centre itself it is ignored. If the vertex passes these tests and has not been expanded previously both the g-cost and m-cost of the vertex are set, and the vertex is added to the priority list.

If a vertex has been expanded previously it is only re-expanded only in the following case:

$$mcost_v > 10\sqrt{(x_{ec} - x_v)^2 + (y_{ec} - y_v)^2 + (z_{ec} - z_v)^2} + mcost_{ec} + pen_v \tag{4.19}$$

In other words, if it would result in the vertex getting a lower m-cost than it currently has. In this case it's m-cost is updated to the lower value, and it is re-added to the priority list. This is done to ensure that the final path is indeed the shortest one.

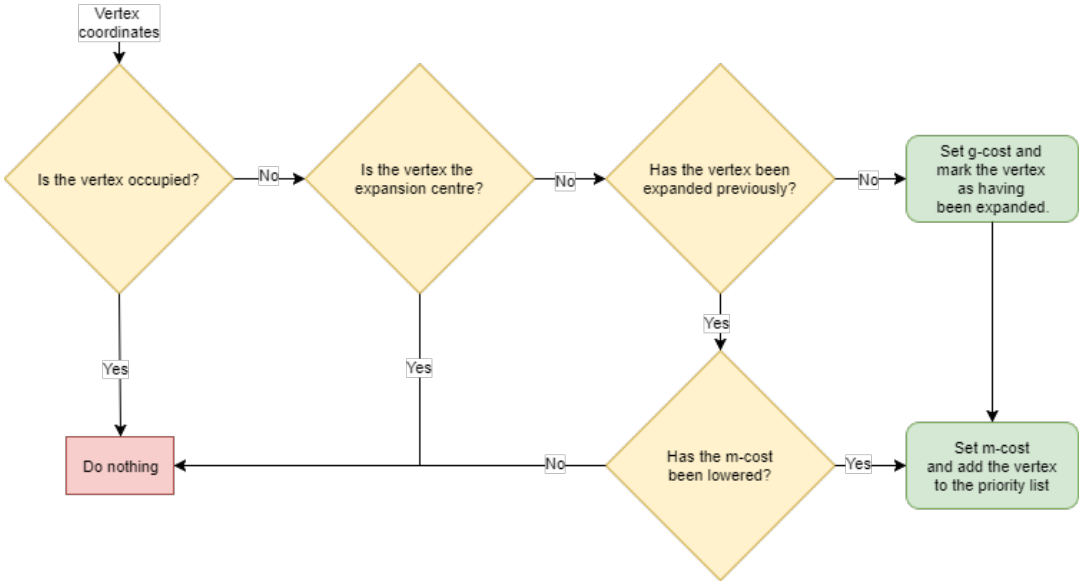


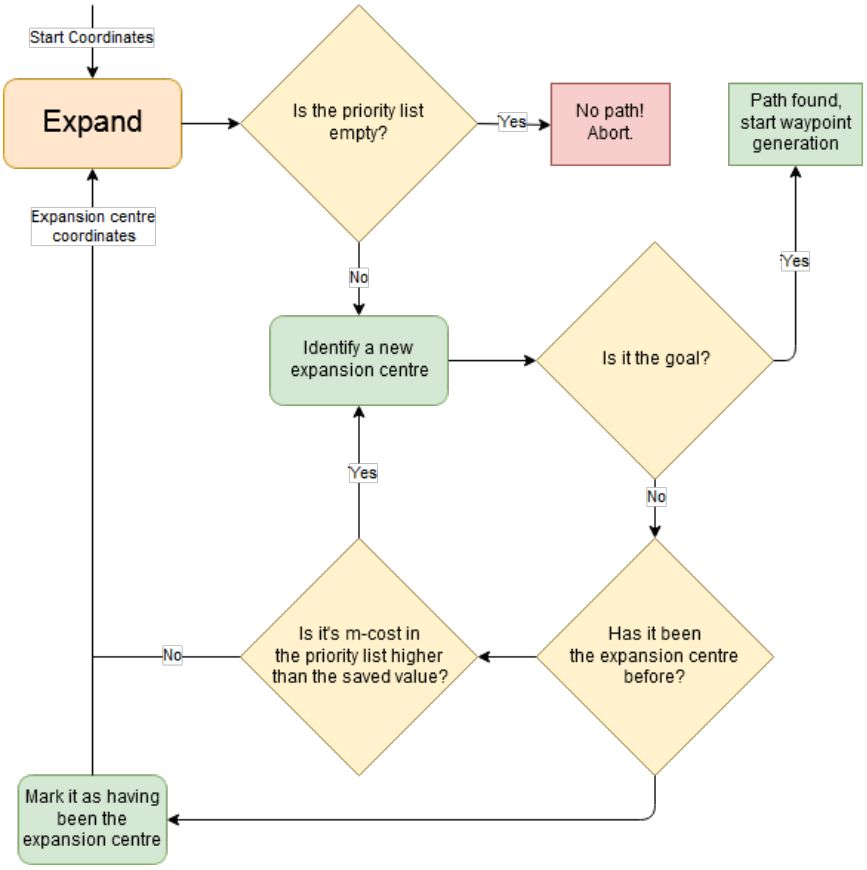

FIGURE 4.4: The vertex updating process for every vertex examined by the expand function

### 4.3.4 The Search

WAPP starts searching in one end of the path, here set to the position. The start coordinate is sent to the expand function as the first expansion centre, which adds the surrounding non-wall vertices to the priority list. The program then sorts the priority list, picks the vertex with the lowest sum of g-cost and m-cost, uses that as the next priority centre.

This is repeated until one of the following things happen: The list is empty when the program tries to extract a new expansion centre from it. This means there is no open, unblocked path between the start and the goal. The other thing is that the new expansion centre is the goal, which means a path has been found.

FIGURE 4.5: The Search process.

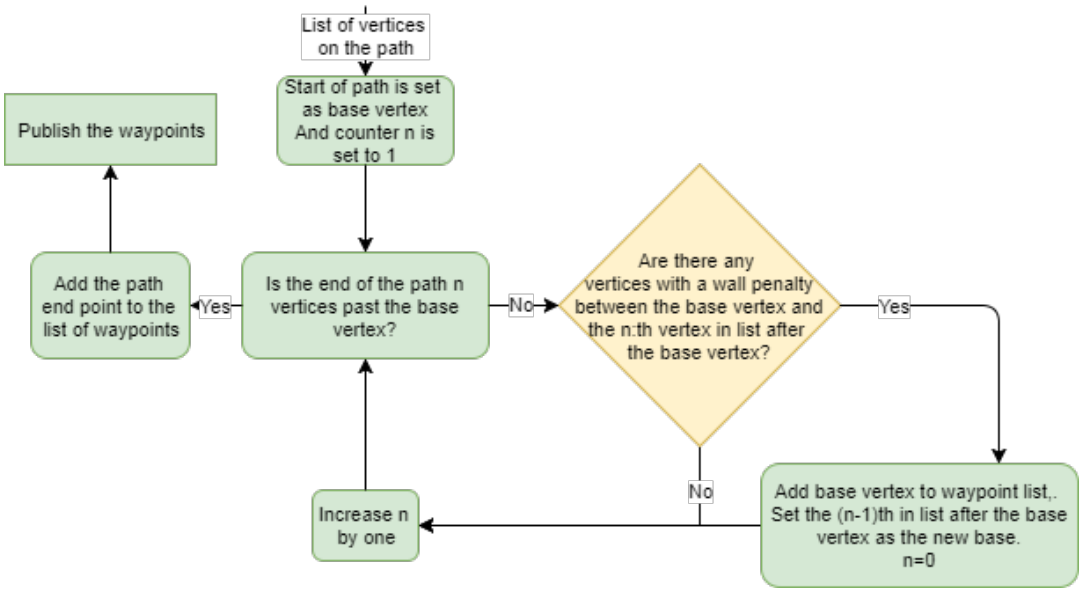


## 4.4 Way-points

When the path has been found the basic A* style path is generated by starting at the goal vertex and picking the next vertex on the path by choosing the one with the lowest m-cost amongst the neighbouring vertices over and over until the start is reached.

All the vertices on the path are added to a list which is then reduced to a list of way points. This is done through a process where any segments of the paths which can be bypassed by traveling in a straight line from the vertex before the segment and the vertex after without passing too close to a wall is removed. This achieved in the following manner:

FIGURE 4.6: The Waypoint generation process.




The test used to determine if there was a vertex with a wall penalty is a simple raycasting algorithm. The algorithm compares the two end vertices, $v_1$ and $v_2$ the space between whom will be investigated. It simply uses the linear equation of the line that passes through both points and checks the penalty status of all the vertices corresponding to the cubes of space that the line passes through.

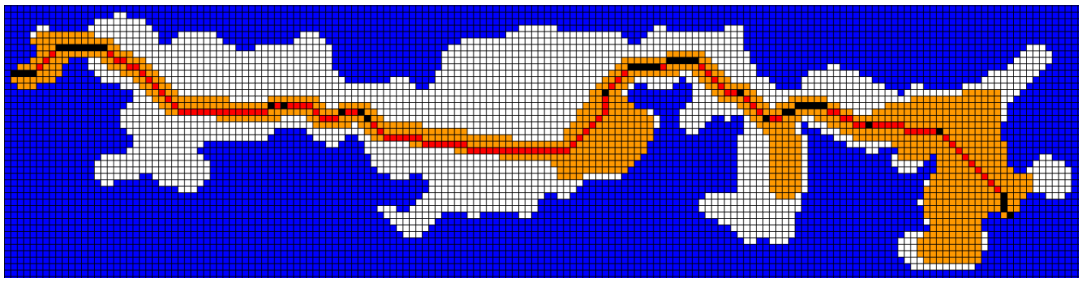


FIGURE 4.7: A path(red) though a 2D map, reduced to way-points(black). Blue vertices are walls, orange free explored and white are free unexplored

This example image showcases the behavior of the waypoint generation, this is in 2D to make it easier to understand. Examples of the waypoint generation working on a 3D map is provided in the results section.

# Chapter 5

# Results

## 5.1 Tests

To try out the behavior of the planner a test map was made with an L-shaped wall obstacle to navigate around. The map was made as a point cloud and converted into an OctoMap and published by the OctoMap server plug-in.

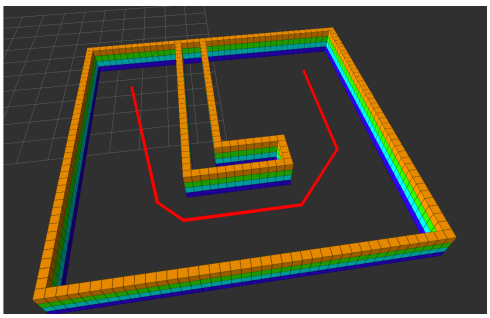
In order to show how the WAPP parameters affect the behavior a series of simulations were made. Each of them were done on the same map with the same start and goal, with the three cost parameters set to different levels. In addition to this one of the configurations were run at two different map resolutions, to illustrate how WAPP would work on a larger map of the same shape.

The following table shows how how the different configurations affect how large a percentage of the map is explored, which illustrates how different parameter settings affect the performance of the program.

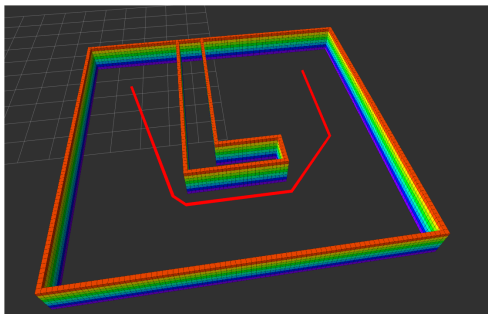
### 5.1.1 Building Interior Map(pseudo-2D)

| WCL | MCL | GCL | Map exp. | Analysis |
|---|---|---|---|---|
| 10 | 1 | 1 | 48.6% | Decent exploration considering the narrow shape of the map |
| 10 | 1 | 1 | 48.0% | Same behavior as on the lower resolution map. Different percentage of explored vertices is due to a lower percentage of wall vertices on this resolution. |
| 10 | 10 | 10 | 53.6% | Slight decrease in efficiency, probably due to more vertices being explored near the corner of the obstacle |
| 10 | 10 | 1 | 74.4% | Same path generated as when all parameters were equal, but with a sharp decrease in efficiency due to lowered WCL. |
| 10 | 1 | 10 | 35.0% | Good path, with good wall clearance and great efficiency, best configuration for this map. |
| 1 | 1 | 10 | 34.1% | Even more efficient, but cutting very close to the corners. |
| 1 | 10 | 1 | 74.1% | Terrible efficiency and a path too near the walls as well. |
| 1 | 10 | 10 | 53.0% | Bad path, average efficiency. |

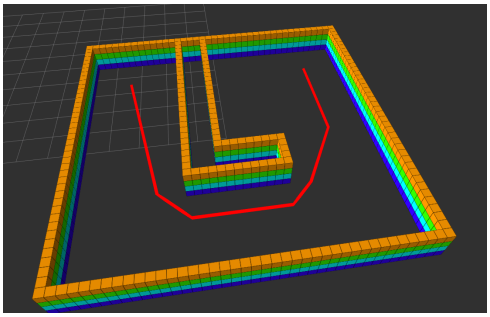
In the following images the paths generated by these simulations are shown.



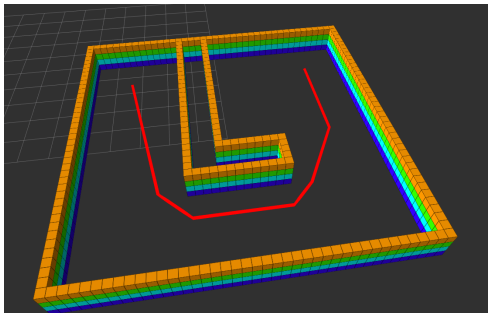
(A) 10,1,1



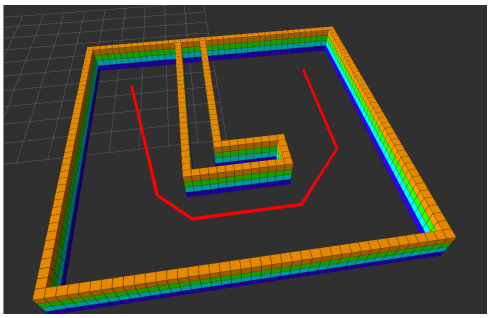
(B) 10,1,1 larger map



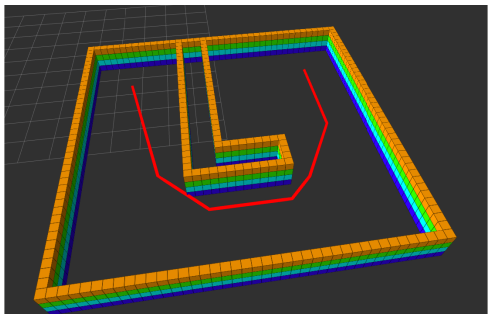
(C) 10,10,10



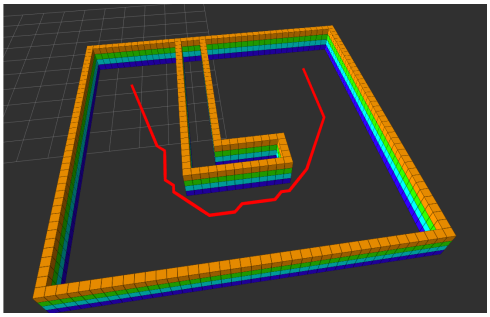
(D) 10,10,1



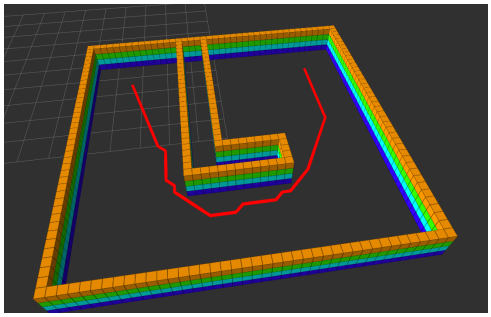
(E) 10,1,10



(F) 1,1,10



(G) 1,10,1

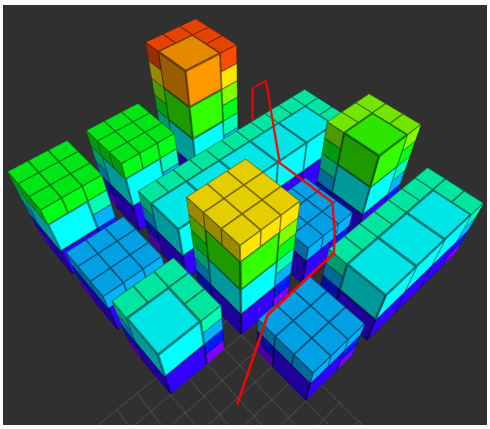


(H) 1,10,10

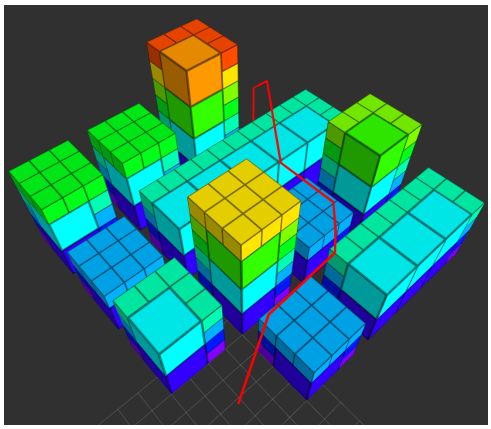
### 5.1.2 Cityscape Map(full 3D)

| WCL | MCL | GCL | Map exp. | Analysis |
|---|---|---|---|---|
| 10 | 1 | 1 | 48.6% | One of the least efficient configurations due to insufficiently low GCL/MCL ratio |
| 10 | 10 | 10 | 38.6% | Slight increase in efficiency, probably due to fewer vertices being explored along the long twisting city paths |
| 10 | 10 | 1 | 55.6% | Terribly inefficient configuration due to a very low GCL/MCL ratio. This is somewhat reduced by the WCL forcing the algorithm to avoid exploring between buildings |
| 10 | 1 | 10 | 4.39% | Good path, clears the city and explores the least area of the map. Great efficiency, best configuration for this map. |
| 1 | 1 | 10 | 4.83% | Nearly as efficient, but lower WCL allows for exploration between buildings which reduces efficiency on this map. |
| 1 | 10 | 1 | 64.5% | Terribly inefficient configuration due to a very low GCL/MCL ratio. |
| 1 | 10 | 10 | 44.3% | Insufficiently low GCL/MCL ratio, low efficiency. |

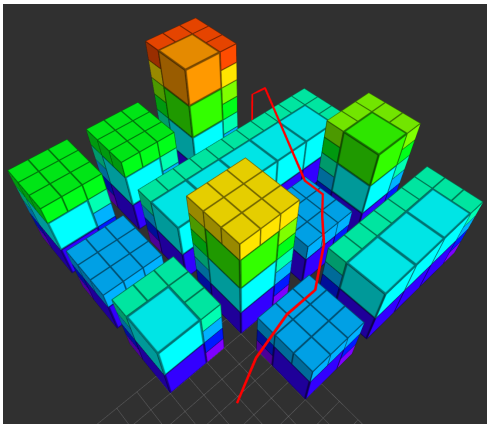
The results from path planning simulations on the cityscape map are similar to those of the building interior map but reveals one big difference. On this map an increased wall penalty actually improves the efficiency of the planner whereas on the interior map there was a decrease in efficiency following a heightened WCL. The reason for this is that, on this map, a higher WCL causes the planner to explore in an arc above the obstacles and avoid the areas in between the buildings. On this map going through the streets in between the buildings is both higher risk as well as longer in distance which means the planner finds a path quicker and more efficiently with a higher WCL. The effects of GCL and MCL on the efficiency is the same as on the previous map.
In the following images the paths generated by these simulations are shown.



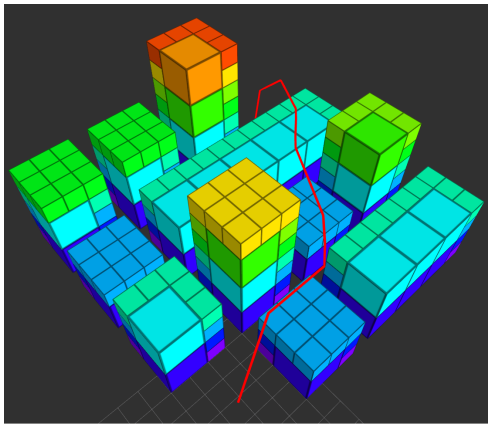
(A) 10,1,1



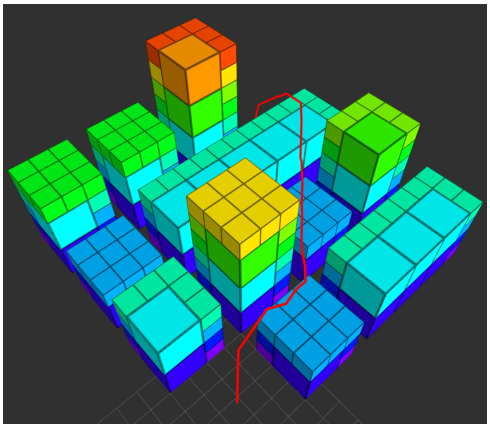
(B) 1,1,1



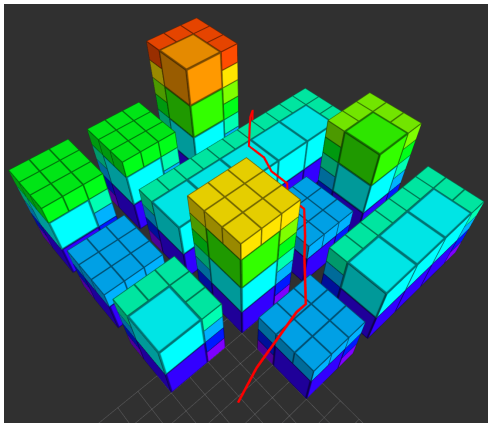
(C) 10,10,1



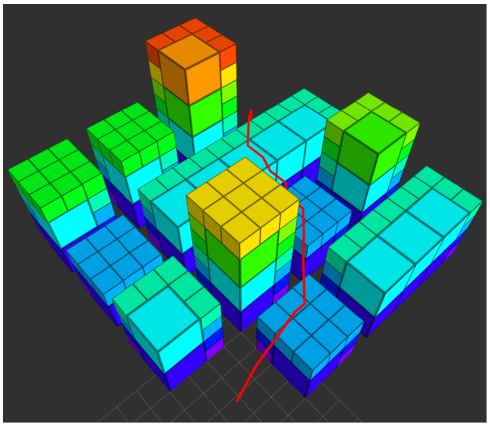
(D) 10,1,10



(E) 1,1,10



(F) 1,10,1



(G) 1,10,10

### 5.1.3 Analysis of the Cost Parameters Impact

**Wall Cost Level** The way the WCL affects the planner behavior is different from the other two parameters in that in most cases there is a certain level were it stops having any impact on the planner what so ever. This threshold varies between maps and depends on how long the longest detour is that would be needed to avoid exploring near a wall vertex.

If the only path there is between the start and the goal goes through a narrow passage where complete wall avoidance is impossible, a high enough WCL will force the program to explore every single vertex in the area before the passage. This will lead to the program planning a path through the passage anyway, if there is a path one will always be returned.

If the WCL is set too low compared to the other parameters it will simply lead to paths being planned near the walls. this might be acceptable in some scenarios, for example if the path is being planned for a slow moving ground vehicle which is in low risk of bumping into walls, and which will not be damaged were it to happen.

One surprising find from the cityscape simulations is an example of how a higher WCL can lead to increased efficiency in some situations. When the shortest path is not the straight line between start and goal due to a clutter of objects blocking the straight line path a higher WCL can prevent the algorithm from exploring winding twisting paths between objects. This reduces the number of computations needed to find the quicker path around the obstacles.

**Move Cost Level** The MCL is what keeps the planner focused on the efficiency of the planned path. This is accomplished by putting the emphasis more on how far from the start a vertex is when it comes to prioritizing which vertex to explore next. The MCL needs to be high enough that enough vertices are explored that the path can actually take the shortest route. If the MCL is too small in comparison to the GCL the program will end up exploring in a straight line, only a single vertex wide, only diverging from this to pass by obstacles. This can lead to suboptimal paths being planned.

Providing that the WCL is set high enough this is largely fixed by the way-point generation, but the possibility of suboptimal paths is still existent.

**Goal Cost Level** As can be seen clearly in the simulation data the GCL is the main parameter when it comes to increasing the computational efficiency of the planner. A higher GCL causes the program to explore fewer vertices where there is an open space to cross on the way toward the goal.

As Previously mentioned in the previous segment a too high GCL compared to the MCL may lead to suboptimal paths, and in addition to this a too strong pull towards the goal may result in a shorter path not being found at all. This can happen if the planner prioritizes vertices nearest a straight line to the goal so much that it fails to find a shorter path.

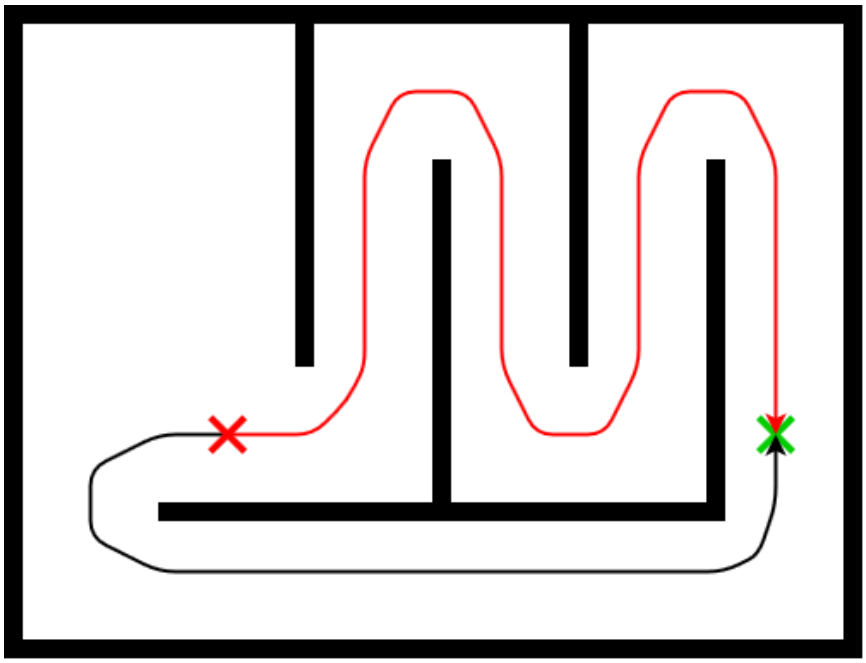


FIGURE 5.3: An example of how a path generated with too high GCL(red) could be suboptimal, as well as a better alternative(black), generated with a lower GCL

Although these risks exist, on most maps the most efficient configuration would be keeping the GCL higher than the MCL.

# Chapter 6

# Future Work

## 6.1 Easily adjustable wall costs

Currently the code allows for easily changing linearly the size of the penalties added to vertices surrounding a wall vertex, but not the number of vertices that gets the penalty. This means that depending on the resolution of the map, the real-world distance of a vertex to the wall that gets an added penalty. The planner could be changed to have a parameter input for a radius for the penalty as well, with the function adding the penalty being modified to accept this input and generate a variable size penalty zone.

## 6.2 Re-planning

Since the code is based on LPA* it is already prepared for re-planning, greatly saving computation time on larger maps. At the moment the code only computes the path once, but it is created to be able to re-use map data that has not changed since the last time the planner was run. This I planned to do, but the time of getting it working wasn't there

## 6.3 Trajectory Planning

The way the planner creates a list of way points without going too near walls, which could either be used as the actual trajectory, or could be the first step to planning a more complex trajectory for a UAV. Such a planner could calculate a trajectory taking into account varying speed, the robots inertia and other factors.

Here the positions of the waypoints far from the walls where possible, and more clustered where walls couldn't be avoided, would make using a polynomial curve fitting algorithm easier to use in order to turn the series of points into a smoother curve for a UAV to follow.

There is also information that would help set upper limits to UAV speed that is generated when the planner generates a path. First of all OctoMap itself makes a difference between open vertices and unknown vertices and even though the planner assumes all non-wall vertices to be open one could tell a UAV to fly slower when entering areas with more unknown vertices. The planner provides info as well when it comes to areas with more walls. This could either be read directly from the wall penalty values in the planners map, or from the waypoint density of the generated path.

# Bibliography

Dijkstra, E. (1959). “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik*.

Gammell, J., S. Srinivasa, and T. Barfoot (2014). “Informed RRT* Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic”. In:

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In:

Hornung, A. et al. (2013). “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees”. In: *Autonomous Robots*.

Koenig, S. and M. Likhachev (2005). “D* Lite”. In:

Koenig, S., M. Likhachev, and D. Furcy (2005). “Lifelong Planning A*”. In:

Lavalle, S. (1998). “Rapidly-exploring random trees: A new tool for path planning”. In:

“ROS”. In: URL: http://www.ros.org.

Stentz, A. (1994). “Optimal and Efficient Path Planning for Partially-Known Environments”. In: