ABSTRACT

Title of thesis: DEVELOPMENT OF A REAL-TIME HIERARCHICAL 3D PATH PLANNING ALGORITHM FOR UNMANNED AERIAL VEHICLES

Matthew David Solomon, Master of Science, 2016

Thesis directed by: Dr. Huan Xu
Department of Aerospace Engineering

Unmanned aerial vehicles (UAVs) frequently operate in partially or entirely unknown environments. As the vehicle traverses the environment and detects new obstacles, rapid path replanning is essential to avoid collisions. This thesis presents a new algorithm called Hierarchical D* Lite (HD*), which combines the incremental algorithm D* Lite with a novel hierarchical path planning approach to replan paths sufficiently fast for real-time operation. Unlike current hierarchical planning algorithms, HD* does not require map corrections before planning a new path. Directional cost scale factors, path smoothing, and Catmull-Rom splines are used to ensure the resulting paths are feasible. HD* sacrifices optimality for real-time performance. Its computation time and path quality are dependent on the map size, obstacle density, sensor range, and any restrictions on planning time. For the most complex scenarios tested, HD* found paths within 10% of optimal in under 35 milliseconds.

Development of a Real-Time Hierarchical 3D Path Planning Algorithm for Unmanned Aerial Vehicles

by

Matthew David Solomon

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2016

Dedication

To my family.

Acknowledgments

I would first like to thank my advisor, Dr. Xu, for providing me the opportunity to work on such an exciting and challenging topic. Her suggestions and input have proven to be invaluable, and I genuinely enjoyed working on this project. I have benefited greatly from having the chance to explore this problem. I would also like to thank all the staff of the Aerospace Engineering Department, including Tom Hurst and Michael Jones. They have been extremely helpful during my time at the University of Maryland and have always been available to answer the many questions I had. Finally, I would like to give a big thanks my parents. Throughout my entire life, they have been there to provide support and assistance in all my endeavors. They have provided me much guidance, and without them I would not be the person I am today.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1: Introduction

Flight paths for unmanned aerial vehicles (UAVs) often consist of traveling from their current location to one or more given goal locations without complete knowledge of the surrounding environment. Because the domain is not entirely known, sensors on the UAV continuously detect new obstacles as the vehicle travels towards the goal, and the original path that was being followed may no longer be valid. In these situations, the UAV must plan a new path sufficiently fast in real-time to avoid collisions with these obstacles.

Real-time replanning is also necessary when chasing a moving target. The path may frequently become invalidated as the target moves, so fast replanning is again needed to ensure the vehicle does not stray too far from the optimal path. In this thesis, a new path planning algorithm is presented that is capable of handling both of these challenges by computing new paths fast enough to operate in real-time. Replanning within 60 milliseconds provides real-time performance for vehicles traveling at moderate speeds [3], and the presented algorithm consistently finds paths even faster. Finding the true shortest path is sacrificed for performance improvements, but the worst case resulting path lengths are within 10% of the optimal path.

1.1 Motivation

Path planning in known, 2D terrain is a well-studied problem [4], and various solutions to find the shortest path exist. Potential field methods guide the vehicle by simulating attractive forces around the goal location and repulsive forces around obstacles. Visibility graphs create edges between the vertices of obstacles, and a 2D implementation guarantees the shortest path will be found [5]. Heuristic based planners, such as the widely used A* algorithm, utilize a user-supplied cost estimate between two states on a map to guide the search from start to goal and are guaranteed to find the shortest path [6].

There are three main problems with existing solutions that make them unsuitable for real-time 3D path planning. Some of these algorithms were designed for off-line operation in known environments, so their primary function is to find the true shortest path, with minimal focus on reducing run-time. Additionally, they are generally designed for 2D environments and do not necessarily carry over well to 3D environments. Visibility graphs, for example, are guaranteed to find the shortest path in 2D, but are frequently unable to find the shortest path in 3D [5] as shown in Section 2.2.2. Lastly, many are not designed to update paths to accommodate unexpected changes in terrain. When a path needs to be recalculated, none of the previously used information gets reused to speed up the current search.

Real-time path planning in unknown terrain has not been studied as extensively, and fewer solutions exist. Because the entire path is frequently not needed immediately, most on-line planners such as Real-Time A* (RTA*) [7] obtain real-time performance by restricting the lookahead of each search. Instead of finding the complete path, they find a prefix of the path and begin to move along it. This approach allows the vehicle to begin moving within a constant amount of time, regardless of the map size used to represent the environment. However, because the complete path is not found, this prefix being followed is often a low-quality, suboptimal path [7–9].

Heuristic based incremental planners, such as D* Lite [4], expand the A* algorithm to reuse previous information to reduce the computation time for the current search. This category of planners attempts to combine the performance of on-line algorithms with the solution quality of heuristic-based algorithms. While this class of planners is promising for smaller environments, it is currently not suitable for real-time operation in its current state; the performance of such a planner is sufficient for some scenarios [9], but it does not perform well for large 3D maps. On a small 2D map represented by a grid, there are no more than eight possible nodes to move to from any given node. However, a 3D cubic map has 26 possible successor nodes. This exponential increase in grid size significantly slows down the search, as does computing the entire path to the goal each time the environment changes. This process can be significantly sped up by initially planning a coarse path and later refining it, as demonstrated by Botea and Müller [2]. This algorithm, known as Hierarchical Path-Finding A* (HPA*) is designed for a 2D implementation, and needs to be expanded to 3D for real-time UAV path planning.

1.2 Thesis Contributions

This thesis presents an algorithm written in Python called Hierarchical D* Lite, or HD*. It combines and modifies D* Lite and HPA*, resulting in a heuristic-based algorithm suitable for real-time use. D* Lite is first used on coarse levels to find a rough path. Next, it is refined into a high-resolution path and then smoothed. This path is followed until either a scheduled replan is called for, the target moves, or new obstacles are discovered that invalidate the path. A new coarse path is then computed, and the process repeats until the goal is reached.

The performance of HD* will be compared to that of 3D Field D* [3], which is an extension of the D* Lite algorithm designed for real-time use. Performance is measured with respect to the number of nodes searched, path computation time, and past cost. We show that HD* reduces the number of node expansions by an order of magnitude to rapidly find paths. This speed comes with a worst-case increase in path cost of less than 10%. The performance of HD* will also be looked at on many randomly generated maps while varying multiple parameters, to see how it performs in different environments. The modifications necessary to obtain real-time performance while providing realistic paths are as follows:

1. An improvement to the approach used by HPA* to create coarse levels. In its current form, the map representation may need to be repaired when new obstacles are detected. Because we are considering completely unknown environments, this can happen often and degrade performance. Instead, an improved

method of defining the high-level search space is presented. The proposed method works for any environment and removes the overhead of repeatedly correcting the map.

2. The coarse paths found in HPA* neglect to consider the known obstacles of the map. This improves performance, but can result in paths that are more suboptimal than necessary, and leaves more work for the refinement stage. HD* utilizes line-of-sight checks to ensure the validity of the computed coarse paths. These checks are performed using Bresenham's Line Algorithm [10].

3. The implementation of user-defined directional cost parameters that can be tailored to the particular situation. Vertical travel is often more costly than planar travel, and sometimes even impossible. This addition allows HD* to handle both of these situations.

4. Path-finding algorithms typically call for path replanning when new obstacles are detected, regardless of their location. This leads to unnecessary replanning. HD* improves upon this by only computing a new path when the detected obstacle blocks the current path.

5. Path smoothing and Catmull-Rom splines are used to reduce path length and provide more realistic looking paths. Path smoothing is only applied in directions of uniform cost to ensure smoothing cannot result in more costly paths.

1.3 Thesis Overview

In Chapter 2, an in-depth overview of the problem is presented along with current existing solutions. Chapter 3 provides a review of the algorithms that compose the foundation of HD*. Chapter 4 discusses the issues that arise when attempting to use these algorithms in unknown 3D environments, and presents the implemented solutions. Chapter 5 explains the various methods used to obtain near-optimal and realistic looking paths, and Chapter 6 presents the methods employed to ensure computation time is sufficiently fast for real-time performance. Chapter 7 compares HD* with 3D Field D* and explores how the performance of HD* varies in a wide range of possible operating configurations. Suggested improvements are presented in Chapter 8.

Chapter 2: Background

2.1 Problem Description

Path planning is the process of determining a series of movements that direct a vehicle, known as the agent, from its current position to the desired goal location, referred to as the target. The path followed should be that with the lowest possible traversal cost, which in this case is the distance traveled, but could also be metrics such as fuel usage, travel time, etc. We start with a completely unknown 3D environment. As the UAV follows a computed path, it uses onboard sensors to detect the surrounding environment and update its knowledge of the terrain. Note that the sensor problem is not addressed in this thesis, and we start with the assumption that the vehicle has adequate sensor data to generate a map of the environment. When obstacles are detected, they may invalidate the current path, and the UAV must plan a new route to the target. Depending on how many obstacles are present, it is possible that replanning may need to occur after nearly every movement made. Thus, it is essential that the replanning step can be executed sufficiently fast. To ensure this, the desired timescale for replanning is on the order of milliseconds.

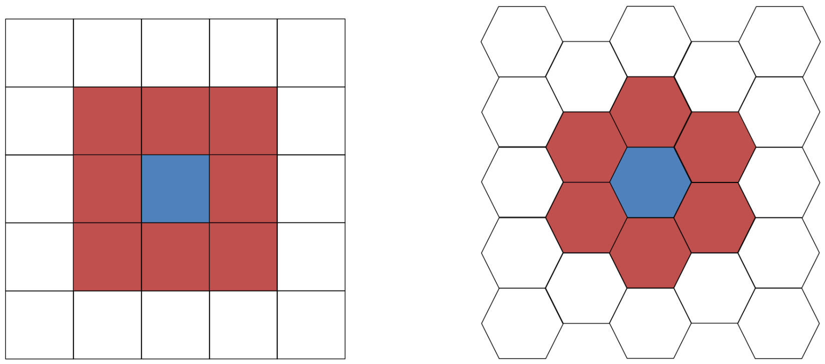
It may be the case that the target is the location of another UAV or some other moving target. In this case, the goal state is continually changing, and the path may need to be constantly recalculated to ensure the agent remains on the optimal path. Furthermore, the particular UAV dynamics can vary, so the algorithm should be able to handle basic constraints such as scaling the cost of altitude changes and preventing vertical movement. Finally, the generated paths should also be realistic, meaning they should avoid frequent heading changes and sharp turns.

2.2 Map Representations

There are many different approaches to representing maps, and the approach used can have a noticeable impact on performance. The representations presented here use a set of nodes, S, where each node $s \in S$ represents a possible state of the agent. A node may either be open or blocked. Open nodes can be occupied by the agent, whereas blocked states contain obstacles and cannot be occupied. The start and goal nodes are given by s_{start} and s_{goal} respectively. Any given node s is connected via an edge to its neighbor $s' \in succ(s)$, where $succ(s)$ defines the set of all successor nodes. The set of successors is dependent on the selected map representation. The cost to travel from s to s' is given by $c(s, s')$, and the travel cost to an occupied node is infinity. A brief overview of commonly used map representations is provided below.

2.2.1 Grids

A common and intuitive representation is a grid, which discretizes the environment into many individual nodes. In a 2D environment, these nodes can be



(a) 2D grid of squares (b) 2D grid of hexagons

Figure 2.1: Examples of grid representations. For a node s shown in blue, its successors s' are shown in red.

squares, hexagons, or triangles, with hexagons and squares being more common for path planning [11]. Sample square and hexagonal grids are shown in Figure 2.1. A node is considered occupied if any area within the node contains an obstacle, which is wasteful when only a tiny portion of the node is blocked [12]. An attractive property of hexagonal grids is that they have six equidistant successors [11]. Unlike hexagons, the successors on square grids are not equidistant, as diagonal successors are separated by a distance of $\sqrt{2}$ instead of one. For 3D representations, cubic nodes are common as they are straightforward to implement, with each node having 26 successor nodes.

Movement on grids is limited to transitions between node centers, thus heading angles become artificially constrained to increments of 45° for square grids and 60° for hexagonal grids. Smoothing the path after it has been found can partially remedy this constraint, but grid based algorithms such as Theta* [13] and Field D* [14] are necessary to allow for true any-angle path planning.

2.2.2 Polygons

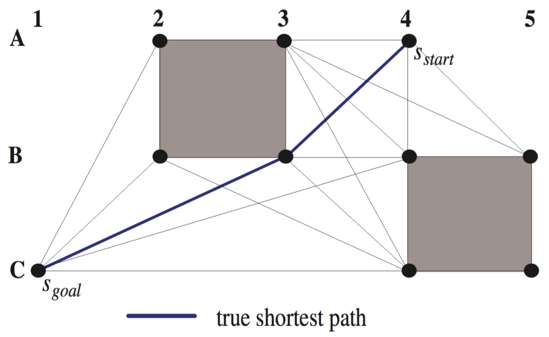
Instead of placing nodes at fixed locations, a polygonal map places nodes at the vertices of obstacles [5,11]. Successor nodes are determined with line-of-sight checks, where the function LineOfSight(s, s') returns true if line-of-sight exists between the two nodes. This is used to create a visibility graph, which defines $succ(s)$ as each node s' for which LineOfSight(s, s') returns true.

In a 2D scenario, the optimal path will always be formed by the edges of the visibility graph, as demonstrated in Figure 2.2(a). This is no longer true when generalized to 3D maps [5], as shown by the map in Figure 2.2(b). The nodes for the visibility graph would be located on the vertices of the polygon, but the shortest path does not include these vertices.

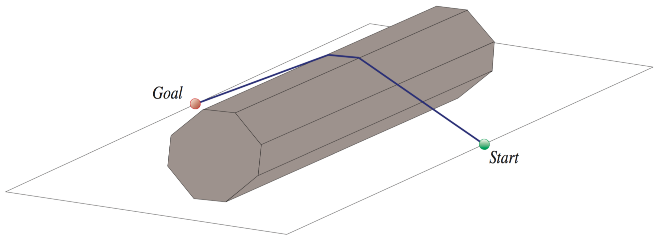
There are a few additional problems to consider. Line-of-sight checks can be time-consuming, especially when the vertices are far apart. This can hinder performance as the map may be frequently updated. Additionally, crowded environments lead to a large number of successor nodes, and more successor nodes leads to slower path-finding. Lastly, determining where to place vertices for round obstacles presents a challenge, and often does not lead to ideal solutions [2,16].

2.2.3 Navigation Meshes

Navigation meshes can be thought of as a variant of polygonal representations. Instead of representing the obstacles with polygons, the traversable areas are represented with polygons. This representation provides the option of placing nodes in



(a) 2D Visibility graphs are guaranteed to find the shortest path [13].



(b) The optimal path in 3D does not always include the vertices of obstacles [15].

Figure 2.2: Examples of Visibility Graphs

the center of the polygons, along the edges, on the vertices, or using a combination of these three. Having many node locations allows for high-quality paths that do not have to depend on the obstacle shapes. Navigation meshes can reduce the size of the path planning space, which simplifies computation, and are useful for video games where the environment may already be represented by sets of polygons. However, they do suffer from many of the some problems as the polygonal representation, including costly visibility checks and frequent updates to the map structure [11,16].

2.3 Review of Existing Solutions

Here, various path planning solutions are presented, and the advantages and disadvantages of each are discussed.

2.3.1 Potential Field Methods

Potential field methods [17, 18] can be used to avoid obstacles in real-time by simulating a potential field. By simulating an attractive force around the goal location and repulsive forces around obstacles, the vehicle can follow the field to the goal. This method is simple, easy to implement, and was originality developed for real-time use in unknown environments, making it an attractive candidate. [18]

However, potential field methods have a tendency to generate local minima that trap the vehicle and prevent it from reaching the goal [17–19]. The local minima may be removed by the injection of random noise, but this results in jagged paths that may be infeasible to follow [19]. An alternative resolution is to include a separate planner to provide information on map connectivity, but this increases computation time and the planner is not guaranteed to be suitable for real-time operation. Even if the global planner is sufficiently fast, the resulting paths from this approach are likely to be suboptimal [17, 20].

2.3.2 Heuristic Planners

Heuristic based planners, such as the A* algorithm, utilize a user-supplied heuristic to guide the vehicle to the goal. The A* algorithm was introduced in 1968 by P. Hart, N.Nilsson, and B. Raphael, [6], and it is a very popular search algorithm in applications such as robotics and video games [13]. It is a complete and optimal algorithm, meaning it will always find the optimal solution if one exists. Heuristic-based planners can be used on a variety of map representations, including grids, polygonal graphs, and navigation meshes. It is important to note that heuristic planners were designed as off-line algorithms, and therefore are not capable of handling map changes. When used in these circumstances, they cannot use any previously obtained information to plan the new path. Instead, the entire path must be replanned from the beginning, which is highly inefficient.

2.3.3 Incremental Planners

Incremental planners, such as D* and D* Lite [4, 21], are extensions of the A* algorithm that can reuse information from previous searches to speed up the current search. The name D* derives from that fact that they are dynamic versions of A* [21]. When a change in the environment is detected, this family of algorithms updates the information obtained from the previous search, essentially transforming the outdated information into accurate information, reducing the time needed to recompute the shortest path.

Rapidly Exploring Random Trees (RRTs) are another type of incremental algorithm. They randomly pick samples from the map to generate short paths organized into a tree [22]. Trees are expanded from the start and goal states, and for each sampled state the planner tries to connect the state to the closest point on each tree. If the state can connect both trees, a path has been found [20]. RRTs tend to prioritize areas of the map that have not been explored, and will gradually improve the resolution of the tree as necessary. They avoid the rapid growth in the number of states that arises when using large grids. RRTs can also reuse information from previous searches and are comparable to D* Lite in regards to planning time. However, they may also suffer from the local minima problem, generally do not find optimal solutions, and are not very reactive to changing environments [20, 23].

While incremental planners are not real-time planners by their nature, a sufficiently fast implementation can act as one [9, 24]. For smaller 2D implementations, they may be fast enough to serve as a real-time planner, but performance begins to suffer as the environment grows larger and more complex. Even though they can reuse information from previous searches, planning the entire path to the goal still can be quite time-consuming.

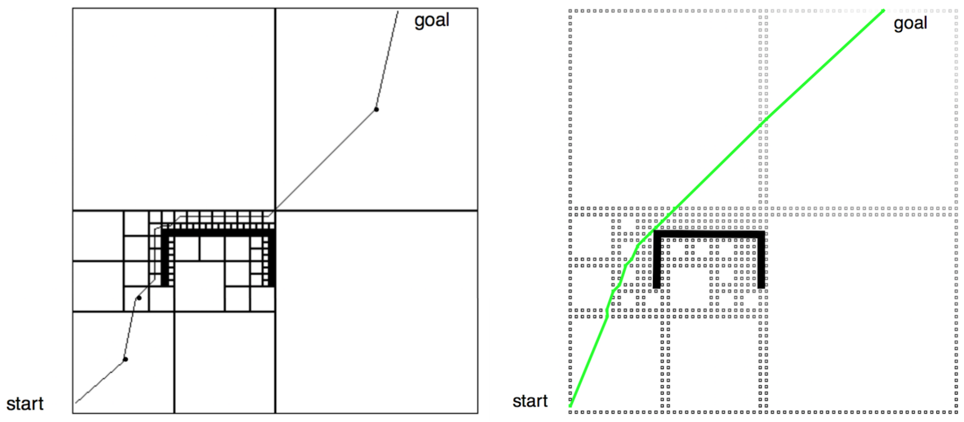
2.3.4 Real-Time Planners

Instead of planning the entire path to the goal, real-time algorithms such as Real-Time A* (RTA*) and Learning Real-Time A* (LRTA*) [7, 8] only plan a prefix of the path and begin to follow it. The agent searches a local region, finds the optimal path within that region, and begins to follow it. A new partial path is computed when the current path becomes blocked, or the agent leaves the previously defined search space. The size of the local search space can be varied to guarantee planning is completed within a set time requirement. The full path is generally not needed immediately, so this approach allows the agent to begin moving within a fixed amount of time, regardless of environmental complexity or distance to the target [24].

For each encountered node, a value is stored which estimates the distance from that node to the target. These values are continuously updated as the vehicle travels and learns more about its surroundings [7,8]. However, memory requirements scale with the number of nodes and can become demanding on large maps [24], especially considering solutions are often suboptimal [7]. Although real-time planners are not optimal algorithms, they are complete algorithms so are guaranteed to find a solution if one exists.

2.3.5 Hierarchical Planners

Hierarchical planners can be used to reduce map complexity and simplify the path-finding process. The idea is that the number of nodes used to represent the environment directly correlates to the time spent finding a path, as more nodes means more states need to be searched. Furthermore, regular grids are not very efficient for representing vast amounts of open space, as many cells are used to represent a large open area [1]. The goal of hierarchical planning is to fix these two problems to reduce the number of nodes, thereby reducing both the search space and computation time [2]. This is achieved by representing large regions of space using a single node. The procedure is analogous to planning a cross-country road trip by car. Searching all types of roads creates an enormous search space and makes it difficult to find the best path. Instead, the search space is greatly simplified by using hierarchical planning. First, a route to the highway is found using local roads. Next, the highways are used to cross the country, and lastly another local path from the highway to the goal is found. Hierarchical planners cannot act as a standalone



(a) Quadtree Path (b) Framed Quadtree Path

Figure 2.3: Paths Generated by Quadtrees [1]

planner, and still rely on another algorithm such as A* to compute paths.

2.3.5.1 Quadtrees and Octrees

Quadtrees and octrees [25] are based on the recursive decomposition of a grid into smaller grids. Quadtrees and octrees refer to the 2D and 3D implementation of this concept, respectively. This approach initially splits the map into four evenly sized squares or eight evenly sized cubes. If any of the blocks are occupied, it is further divided in the same manner, and the process is repeated until the highest allowed resolution is reached, or every node is either fully blocked or fully unblocked. The benefit of this approach is that large regions of open space are represented by one cell, which significantly simplifies path-finding. As is typical with a grid representation, the node centers are used for path planning. This can produce suboptimal paths, particularly when crossing the larger cells [1, 2].

Framed quadtrees [1] have been proposed as a method to provide shorter and more realistic paths by adding a perimeter composed of the highest resolution cells around each block. This allows for smoother and shorter paths to be planned. A comparison of the paths generated by quadtrees and framed quadtrees is shown in Figure 2.3. Note that the small gray squares around the border of Figure 2.3(b) represent the added border of framed quadtrees. Due to having many additional higher resolution cells, framed quadtrees require more memory, especially in crowded environments. The benefits gained via framed quadtrees diminish as the environment becomes more cluttered [1].

The biggest problem with quadtrees and framed quadtrees is that when the environment is unknown, the additional step of updating the map representation is required before the path is replanned. It has been shown in [1] that this results in a longer replanning time than that of a standard grid representation.

2.3.5.2 Hierarchical Path-Finding A*

Another method to simplify the search space is Hierarchical Path-Finding A* (HPA*) [2]. Instead of recursively dividing the map like quadtrees, HPA* takes the opposite approach and starts with the original full-sized map. The nodes that compose the original map are the highest resolution that is used to represent the environment. The algorithm abstracts this existing map representation into multiple coarse levels, with each level composed of many of the original high-resolution nodes. Level 0 represents the full, high-resolution map, and each subsequent level is more coarse than the one beneath it. HPA* was designed for 2D maps, and uses sets of level 0 nodes to define transitions between adjacent clusters. These transitions represent the nodes and their successors for the abstract levels. Transitions are defined by regions of adjacent nodes that are unblocked in each of the neighboring clusters. If there is a stretch of open nodes, only one or two of them are selected to be used as transitions to reduce the search space.

The coarse planning does not consider the details of the map, allowing a rough path to be rapidly computed. The agent then plans the level 0 paths between the cluster transitions as needed, and these paths can be found quickly over short distances. If obstacles are detected along the current path, a new abstract path is computed. This approach considerably improves performance because the coarse planner has a greatly simplified search space, and the highest resolution planner only needs to plan over small distances. While this method results in suboptimal paths, it can decrease planning time by an order of magnitude. Additionally, path smoothing on both the coarse and fine levels can bring path lengths to within just 1% of the optimal path length when the environment is known [2].

2.4 Chosen Approach

The preceding review of map representations and planning algorithms shows that no approach is ideal for every situation, and a trade-off between path optimality and performance is necessary. Potential field methods are first eliminated from consideration due to their downsides. Using artificial potential fields to guide the search means the local minima problem must be solved, and this brings either increased computation time or low-quality paths. We can also eliminate RRTs as they have been shown to be slower than hierarchical planners and do not always find near-optimal paths [20]. A hierarchical approach can be combined with an incremental planner, but not with real-time planners since by nature these only plan on a local level. That leaves a combination approach or a real-time planner as the remaining two choices.

D* Lite was compared with the real-time algorithm LRTA* by Sven Koenig in [26], and found some interesting results. Koenig compares the algorithms in two types of environments. The first map is a maze, which detracts from the accuracy of the heuristic values, and the second map contains random obstacles, resulting in a fairly accurate heuristic. It is found that the performance of LRTA* is highly dependent on the environment, whereas the performance of D* Lite seems to be more consistent. D* Lite often outperforms LRTA* in the maze, because the misleading heuristic confuses LRTA* due to the fact it only plans partial paths. On the other hand, LRTA* consistently outperforms D* Lite in the random environment.

In the real-world, the terrain will likely fall between these two extremes. The real-time planner has the benefit of ensuring any limitations on planning time are met, but it is shown in [26] that the performance of LRTA* can vary significantly depending on the specific implementation details. When the environment is unknown, LRTA* poses the risk of taking long and undesirable paths, and therefore cannot guarantee consistent and predictable performance. Incremental planners, however, will be more consistent in finding near-optimal path lengths for a variety of environments. Although the incremental planner requires more planning time as the map size increases, this can be compensated for with the inclusion of a hierarchical approach. For these reasons, a hybrid algorithm combining a heuristic based incremental planner with a hierarchical planner will be used. This choice naturally leads a cubic grid representation, as this is the simplest way to define the hierarchy of levels. With this approach, it is essential that paths are smoothed to obtain realistic and high-quality solutions.

D* Lite will be the foundation for the planner, as it is an improvement over both A* and D* in terms of performance [4]. The approach of HPA* will be the foundation of the hierarchical planner, due to the increase in on-line planning time experienced by quadtrees. This new algorithm will be known as Hierarchical D* Lite, or HD*.

Chapter 3: Overview of Implemented Algorithms

Before discussing how HD* works, it is essential to understand the foundation it is built on. The heuristic based A* algorithm will first be described, followed by an explanation of the incremental planner D* Lite. Lastly, the principles behind HPA* are covered. Modifications have been made to these algorithms to make them suitable for unknown 3D environments, and those changes will be presented in Chapter 4.

3.1 A*

Until the development of A*, there was no central theory used to guide the search of a minimum cost path [6]. Methods used to find shortest paths did not consider computational practicality and required every node of the map to be searched. The methods that did consider performance used domain specific knowledge to reduce the number of nodes searched, but were unable to find the shortest path. A* combines these two approaches, resulting in a complete and optimal algorithm that does not need to search the entire map.

Algorithm 1 A* Algorithm

The functions used to manage the priority queue are: U.Insert(s, k) inserts node s with key k into priority queue U. U.Top() deletes and returns the node with the smallest priority in U. U.remove(s) removes s from U.

```
1: function UpdateVertex($s, s'$)
2:     $g_{old} = g(s')$
3:     ComputeCost($s, s'$)
4:     if $g(s') < g_{old}$ then
5:         if $s' \in U$ then
6:             U.remove($s'$)
7:         U.Insert($s', f(s')$)
8: function ComputeCost($s, s'$)
9:     if $g(s) + c(s, s') < g(s')$ then        ▷ Found new shortest path from $s_{start}$ to $s'$
10:        $bptr(s') = s$
11:        $g(s') = g(s) + c(s, s')$
12: function BuildPath( )
13:     $path = s_{goal}$
14:     while $s \neq s_{start}$ do
15:         $s = bptr(s)$
16:         $path = path \cup s$
17:     return $path$
18: function Main( )
19:     $U = closed = \emptyset$
20:     $g(s_{start}) = 0$
21:     $bptr(s_{start}) = NULL$
22:     U.Insert($s_{start}, f(s_{start})$)
23:     while $U \neq \emptyset$ do
24:         $s$ = U.Top()
25:         if $s = s_{goal}$ then
26:             $path$ = BuildPath()
27:             return $path$
28:         $closed = closed \cup \{s\}$
29:         for all $s' \in succ(s)$ do
30:             if $s' \notin closed$ then
31:                 if $s' \notin U$ then
32:                     $g(s') = \infty$             ▷ Initialize nodes when first encountered
33:                     $bptr(s') = NULL$
34:                 UpdateVertex($s, s'$)
35:     return  no path exists
```

A* works by keeping track of four different values for every node $s \in S$. These values are:

1. The g-value, $g(s)$, which is an estimate of the cost from the start node to node s. This cost is denoted $c(s_{start}, s)$. The g-values are initialized to infinity.

2. The heuristic, $h(s, s_{goal})$, is an estimate of $c(s, s_{goal})$ provided by the user. The heuristic must be admissible, meaning it never overestimates the true cost. It must also be consistent, which is true if and only if it obeys the triangle inequality $h(s, s'') \leq h(s, s') + h(s', s'')$. An admissible heuristic is required to guarantee the shortest path is found. The Euclidean distance is a commonly used heuristic when the agent can move in any direction.

3. The f-value, which is an estimate of the smallest cost of moving from s to the goal state, and is defined as $f(s) = g(s) + h(s, s_{goal})$

4. The backpointer of a node, given by $bptr(s) \in succ(s)$, points to the parent node of s. Recall that $succ(s)$ represents all the successors of a given node. The backpointer is used to extract the shortest path once the search is complete. It is initialized to $NULL$ when s is first encountered, and is computed using (3.1).

$$bptr(s) = \begin{cases} NULL & \text{if } s = s_{goal} \\ \text{argmin}_{s' \in succ(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases} \tag{3.1}$$

The open set U is a priority queue sorted by f-values, and the node with the minimum f-value is expanded next. Initially, it only contains s_{start}. Expansion of a node s occurs when each successor of s is operated on in order to determine their f-values. This begins in line 29 of Algorithm 1, which provides a complete implementation of the A* algorithm. A* also maintains a closed list, which is a list of all nodes that have already been expanded. The algorithm operates by removing the state with the smallest f-value from U, moving it to the closed list, and expanding that state. It terminates when the goal state is expanded, indicating the shortest path was found and can be extracted. It may also terminate when U becomes empty, in which case no path exists.

3.2 D* Lite

D* Lite is an extension of A* that is fundamentally the same with the benefit of faster replanning. Whereas A* searches from start to goal, D* Lite [4] searches from goal to start. This is because D* Lite requires the root node to remain the same for each subsequent search in order to speed up replanning [26]. The start node continually changes as the agent moves towards the goal. On the other hand, the goal node is often a fixed location, making it an ideal candidate for the root node. The restriction arises from the fact that obstacle detection occurs near the vehicle within the range of its sensors. Planning from start to goal would require the detected changes to be propagated throughout the entire path, but planning from goal to start reduces the impact of these changes [27]. Because of this requirement D* Lite cannot find only a prefix of the path, as the prefix would begin at the goal node and therefore be useless to the vehicle. This restriction on the root node prevents incremental search algorithms from being combined with real-time search algorithms.

D* Lite makes no assumptions about how the map costs change, and allows for costs to either increase or decrease. All traversal costs are one for open nodes and infinity for blocked nodes. Nodes of unknown status are assumed to be open. Because the search direction is reversed from A*, the g-values now represent estimates of distances to the goal.

The g-values, f-values, and backpointers are computed for each node the same way as they are in A*, with the addition of a new variable, known as the right-hand side value. Often referred to as the rhs-value, this value is based on the g-values of $succ(s)$ and must always satisfy the relationship in 3.2. Its value is dependent on the neighbors of a node, making it more informed than the g-value because it looks one step ahead. This introduces the concept of consistency, where a node is considered consistent if $g(s) = rhs(s)$. When $g(s) \neq rhs(s)$ the node is called inconsistent, and can be categorized as overconsistent when $g(s) > rhs(s)$, and underconsistent when $g(s) < rhs(s)$. Note that if $rhs(s) = \infty$, no node will point to s as its minimum cost successor, and therefore $bptr(s) = NULL$.

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in succ(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases} \tag{3.2}$$

The concept of consistency results in a modification to the operation of the priority queue. It now contains only the inconsistent nodes whose g-values need to be made consistent. The queue is lexicographically sorted by the key values $k(s)$ of each node, which are defined in 3.3.

$$k(s) = \begin{bmatrix} k_1(s) \\ k_2(s) \end{bmatrix} = \begin{bmatrix} \min(g(s),\ rhs(s)) + h(s, s_{start}) \\ \min(g(s),\ rhs(s)) \end{bmatrix} \tag{3.3}$$

D* Lite finds the shortest path by executing the Main() function of Algorithm 2. This first initializes the problem using the steps in lines 3-8. All g-values and rhs-values are set to infinity, except for $rhs(s_{goal})$, which is initialized to 0. This makes s_{goal} the only inconsistent vertex and therefore it gets inserted into U. For large maps, it is not practical to initialize these values to infinity for all nodes, as this may require significant amounts of memory. Instead, nodes can be initialized when they are encountered for the first time. Next, the algorithm finds the shortest path between the start and goal locations. This step is equivalent to running a reversed version of A* and therefore is a complete and optimal algorithm. Thus, if the execution of ComputeShortestPath() results in $g(s_{start}) = \infty$, then the cost estimate from the start position to the goal is infinity, indicating that no path exists.

If a path does exist, line 27 is executed and the vehicle moves from its current location to the successor node which lies on the optimal path. The environment is then scanned for changes, and if new obstacles are detected UpdateVertex() is called for each changed node. This ensures 3.2 remains satisfied for each node and adjusts their status in U as needed. The key values of each node in U are then updated, transforming the outdated queue into a current one. ComputeShortestPath() is executed again, and the process is repeated until the target is reached or a path no longer exists. There are a few optimizations that can be made to D* Lite, which are discussed in detail in [4].

Algorithm 2 D* Lite Algorithm

In addition to the functions used in A*, D* Lite also uses additional functions to manage U. U.Pop() removes and returns the node number of the element with smallest priority in U, and U.TopKey() refers to the key values of the node returned from U.Pop().

```
 1: function CalcKey(s)
 2:     return $[\min(g(s), rhs(s)) + h(s_{start}, s);\ \min(g(s), rhs(s))]$
 3: function Initialize( )
 4:     $U = \emptyset$
 5:     for all $s \in S$ do $rhs(s) = g(s) = \infty$
 6:     $rhs(s_{goal}) = 0$
 7:     U.Insert($s_{goal}$, CalcKey($s_{goal}$))
 8: function UpdateVertex(u)
 9:     if $u \neq s_{goal}$ then $rhs(u) = \min_{s' \in succ(u)}(c(u, s') + g(s'))$
10:     if $u \in U$ then U.Remove($u$)
11:     if $g(u) \neq rhs(u)$ then U.Insert($u$, CalcKey($u$))
12: function ComputeShortestPath( )
13:     while U.TopKey() < CalcKey($s_{start}$) or $rhs(s_{start}) \neq g(s_{start})$ do
14:         $u$ = U.Pop()
15:         if $g(u) > rhs(u)$ then
16:             $g(u) = rhs(u)$
17:             for all $s \in succ(u)$ do UpdateVertex($s$)
18:         else
19:             $g(u) = \infty$
20:             for all $s \in succ(u) \cup \{u\}$ do UpdateVertex($s$)
21: function Main( )
22:     Initialize()
23:     ComputeShortestPath()
24:     while $s_{start} \neq s_{goal}$ do
25:         if $g(s_{start}) = \infty$ then
26:             return no path exists
27:         $s_{start} = \text{argmin}_{s' \in succ(s_{start})}(c(s_{start}, s') + g(s'))$
28:         Move to $s_{start}$
29:         Scan environment for new obstacles
30:         if new obstacles exist then
31:             for all edges $(u, v)$ with changed costs do
32:                 Update traversal cost $c(u, v)$
33:                 UpdateVertex($u$)
34:             for all $s \in U$ do
35:                 U.Update($s$, CalcKey($s$))
36:             ComputeShortestPath()
```

A 2D example of D* Lite is now stepped through in Figure 3.1 to show how paths are computed. The top image of Figure 3.1 shows the initial state of the map. The heuristic used is the maximum absolute value of the difference between the x- and y-coordinates of s_{start} and s, with s_{start} = B1 and s_{goal} = E3. A bold border around a node indicates it will be the next node expanded. The vehicle is aware of the two obstacles at B2 and C2. During the initialization step, the g-values and rhs-values are set to infinity for each node. The rhs-value of the goal is then set to 0 and added to U with a key of $k(s_{goal}) = [3;\ 0]$. In Step 1, ComputeShortestPath() is called. This call to ComputeShortestPath() is identical to a reversed A* search. This step pops s_{start} from U, and the key values of its successors are computed and inserted into U. Notice that cells D2 and D3 are tied for the minimum priority, so D2 is arbitrarily selected to be expanded next. Step 2 shows the results after D2 is expanded, and the key values of its successors are computed and inserted into U. D3 now has the minimum priority, but all of its successors are also shared by D2. Thus, in Step 3 we see no changes to the node priorities. Node C1 is expanded in Step 4, which results in s_{start} being the next node expanded. Expansion of s_{start} results in the start node becoming consistent, as shown in Step 5, which terminates the ComputeShortestPath() function. The path can now be extracted by following the nodes that results in the minimum cost of $c(s_{start}, s') + g(s')$. The cost to transition to any successor node, including diagonals, is 1 in this example, as indicated by the heuristic definition. This results in a path of B1 $\rightarrow$ C1 $\rightarrow$ D2 $\rightarrow$ E3, which the robot will begin to follow.

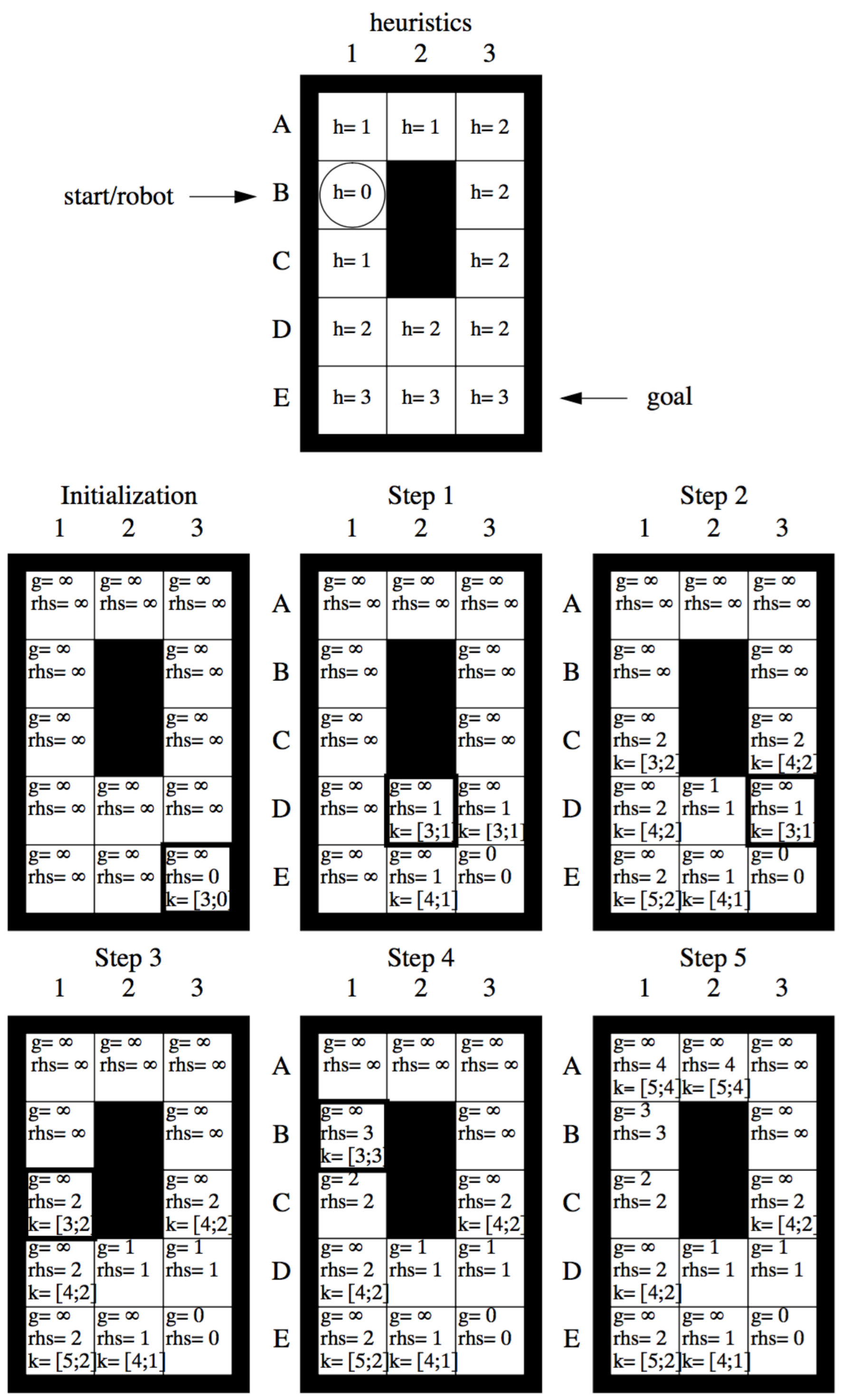


Figure 3.1: D* Lite operation: finding the initial path [4]

3.3 Hierarchical Path-Finding A*

As discussed in Section 2.3.5, HPA* [2] creates abstract levels to simplify the search space. These abstract levels are composed of clusters of the high-resolution nodes that represent the original map. Level 0 represents the highest map resolution, with each successive level being coarser than the one below it. The number of levels can be configured to increase with map size to help minimize search times. A coarse abstract path is planned first, which is then refined using the finer abstract levels. Level 0 nodes are then used to compute the short paths between the abstract clusters.

To plan the coarse path, sets of entrances are defined between neighboring clusters using the adjacent level 0 nodes. Consider the 2D case of two neighboring clusters c_1 and c_2. The clusters share adjacent lines of nodes l_1 and l_2, where each line is in one cluster. For a node $s \in l_1 \cup l_2$, $symm(s)$ defines the symmetric node of s with respect to the border of the two clusters. The nodes s and $symm(s)$ therefore represent adjacent nodes that are not in the same cluster. With this definition an entrance e can be defined as a set of nodes meeting the following conditions:

1. The border limitation condition states that an entrance e is defined only along l_1 and l_2 and cannot extend beyond these bounds.

2. The symmetry condition states that $\forall\ s \in l_1 \cup l_2 : s \in e \Leftrightarrow symm(s) \in e$.

3. The obstacle free condition states that an entrance cannot contain any blocked nodes.

4. The maximality condition states that an entrance is extended as far as possible in both directions until one of the previous conditions is broken.

HPA* defines transitions within each entrance and A* is run on these transition nodes to find the coarse path. For entrances with a length greater than a defined threshold, two transitions are defined at the ends of the entrance. Below this length, one transition is placed at the center. Figure 3.2 shows the transitions for a map split into 10×10 clusters with a length threshold of 6 nodes. Black squares represent obstacles, gray squares represent the transitions, and gray lines represent traversable edges. The curved intra-cluster edges shown in the top-right of this figure do not represent an exact path, but are intended to show that those nodes are successors of one another.

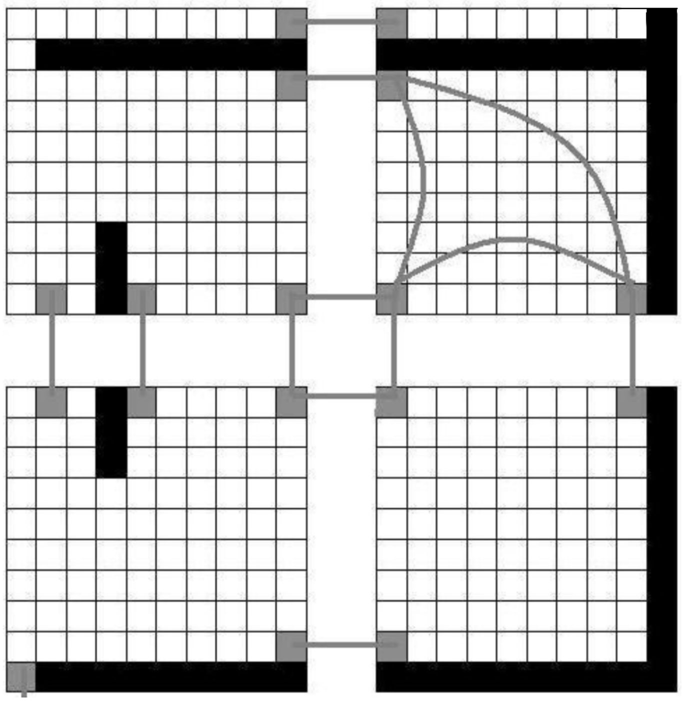


Figure 3.2: Defining transitions on an abstract level. For simplicity, the intra-cluster edges are only shown in the top-right cluster [2].

The shortest coarse path is first computed using the transition nodes, neglecting the details of the individual nodes composing the clusters. This path can be refined to lower levels if desired, and is then computed at the highest resolution as needed. This approach reduces the computational effort required. It is much easier to find a path between small segments of the coarse path than through the entire high-resolution map. When a path becomes invalid, HPA* discards it and computes a new coarse path. In unknown environments, new entrances and transitions will likely need to be found if obstacles occupy the current transitions [2].

Chapter 4: Expansion to 3D

The most straightforward way to implement hierarchical levels in 3D is through a grid composed of cubic nodes, which is the map representation used by HD*. This allows for D* Lite to easily be expanded to 3D with each node represented as a tuple of its coordinates, given by (x, y, z). The successors of a node simply become the 26 nodes that surround it. On the other hand, extending HPA* to 3D introduces some challenges, which are explained and resolved below.

4.1 HPA* Problems

As detailed in Section 3.3, HPA* defines transition nodes between adjacent clusters which are then used to run A* and find a coarse path [2]. However, this approach is not ideal for 3D maps or unknown environments. For 3D maps, there are many more node pairs that must be checked to define entrances between clusters, which increases the time needed to complete this step. For example, a 2D implementation with 8×8 clusters only has eight pairs of nodes shared by neighboring clusters. Neighboring clusters in a 3D version with $8 \times 8 \times 8$ clusters share 64 pairs. Compounded over every pair of adjacent clusters, this significantly impacts the time required to identify transitions. This is not a problem when the environment is completely known, as the entrances would only need to be identified once, but for unknown environments this method will not suffice.

When the environment is unknown, the UAV has no initial knowledge of which nodes are open. Once a set of transition points are defined, the UAV will likely discover that some of these transitions are blocked. If the current path uses any of these blocked nodes, it becomes invalid and needs to be recomputed. Without a new set of entrances, the new path may be much longer than the previous one. Another possibility is that every transition of a cluster is blocked and now a path cannot be found. To resolve these problems the transition nodes must be redefined, but this is a costly process. As seen with quadtrees in [1], devoting time to repairing the map representation can significantly degrade performance. Instead, a new method of determining coarse paths is needed for real-time hierarchical planning.

Another issue with HPA* is the lack of consideration for known obstacles. If the obstacles are small and only span one or two nodes this would be an acceptable approach. The level 0 planner would simply route the path around these obstacles. When the domain includes large obstacles, such as buildings, this becomes problematic. The coarse path may pass through a large obstacle, with one transition node on either side of it. Planning a high resolution path between these points may result in a very suboptimal path. Computation time increases with path length, so even if longer paths are acceptable, the high resolution planning may take longer than desired.

4.2 Hierarchical Modifications

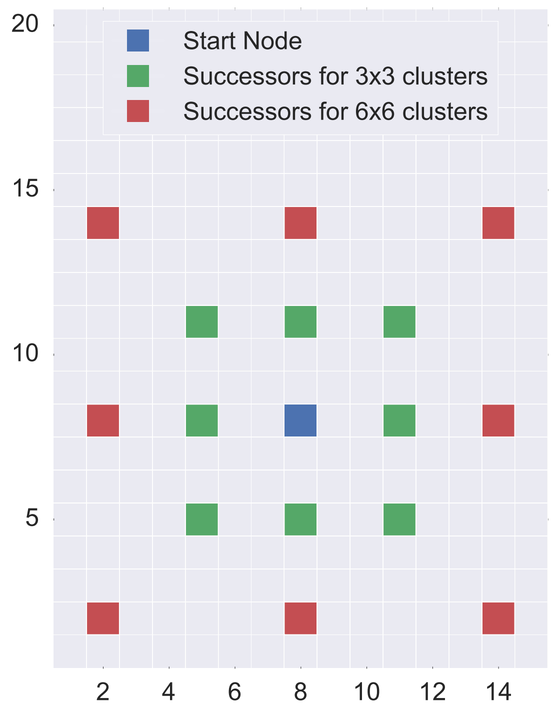


Figure 4.1: Improved method for determining hierarchical successors in 2D environments.

Instead of continually redefining entrances between clusters, HD* does not need the cluster borders and instead only uses the dimensions of the clusters to identify successors. The abstract successor nodes of s are the 26 nodes that form a cube around s at a distance equal to the cluster dimension. To clarify, a simple 2D example will be used. Consider the map of Figure 4.1, which has two levels of clusters. The most coarse clusters are 6×6, and the finer clusters are 3×3. The start location is $(8, 8)$. When searching for a coarse path, the highest level is used first. Therefore, the successors of the start location are all the nodes forming a square a distance of 6 nodes away, given by the red nodes. Upon refinement, the

3×3 level is used, and the successors for this level are shown by the green nodes.

Successor nodes can now be rapidly computed from any node and planning is no longer limited to sets of entrances. With this successor definition, a new coarse path can be planned immediately when the current path becomes invalid, without necessitating a map correction. It has the added benefit of requiring less storage, as we do not need to maintain the list of valid transition nodes and their successors. Instead, HD* only stores the dimensions of the clusters at each level.

There is one problem with this approach that fortunately has a simple solution. When planning the path from goal to start, it is unlikely that the start node ends up as a hierarchical successor. To resolve this, each time successor nodes are computed we check the distance between the start node and the node being expanded. If it is within a distance of twice the cluster size, it is considered a successor. This distance was chosen in order to prevent the need for a final short transition to reach the start node. This also helps reduce the number of nodes expanded. An example of this is shown in Figure 4.2, which uses 3×3 clusters. The black squares represent obstacles. The path between $s_{current}$ and s_{start} will not pass through node s_1, as there is not line-of-sight between s_1 and s_{start}. Therefore, the path would normally have to pass through s_2. Instead, since s_{start} is nearby and has line-of-sight with $s_{current}$, we allow the coarse planner to jump directly between the two.

The downside to this approach is that D* Lite may not always be able to reuse information to speed up the replanning of coarse paths. A new path may be needed at anytime, and because the successors are dependent on the UAV's location it is likely that a different set of successors will be used. If information from previous

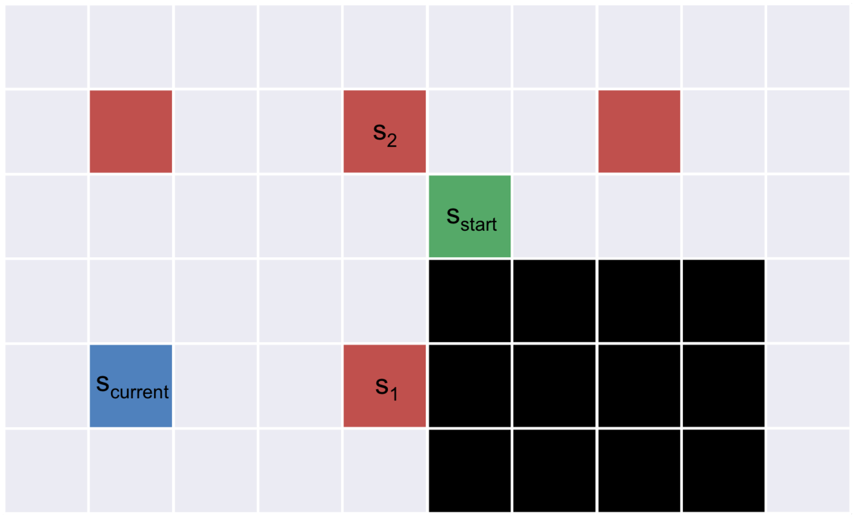


Figure 4.2: During hierarchical planning, HD* can jump directly from $s_{current}$ to s_{start} when the two nodes are nearby and have line-of-sight.

searches cannot be reused, it seems as if A* would be sufficient. However, when the goal location is nearby, hierarchical planning provides little to no benefit, and these situations see performance improvements when using D* Lite over A*. Furthermore, the first call to ComputeShortestPath() is identical to a reversed A* search, so it is acceptable to use D* Lite when A* is sufficient.

This new approach makes it necessary to consider obstacles when computing coarse paths, which increases computation time but helps produce improved routes. Without this, successor nodes may be within or blocked by obstacles, which can result in low quality or impossible coarse paths. As a result, HD* first checks line-of-sight using Bresenham's Line Algorithm [10]. This 2D algorithm was expanded to 3D, and the implemented version is presented in Algorithm 3.

Algorithm 3 3D Version of Bresenham's Line Algorithm

```
1: function LineOfSight($(x_1, y_1, z_1), (x_2, y_2, z_2)$)
2:     $d_x = x_2 - x_1;\ d_y = y_2 - y_1;\ d_z = z_2 - z_1$
3:     $a_x = 2|dx|;\ a_y = 2|dy|;\ a_z = 2|dz|$
4:     $s_x = \text{sgn}(x);\ s_y = \text{sgn}(y);\ s_z = \text{sgn}(z)$
5:     if $a_x \geq \max(a_y, a_z)$ then                      ▷ change in $x$ is the greatest
6:         $y_d = a_y - a_x/2$
7:         $z_d = a_z - a_x/2$
8:         while $x_1 \neq x_2$ do
9:             if $y_d \geq 0$ then $y_1 += s_y;\ y_d -= a_x$          ▷ move in y-direction
10:            if $z_d \geq 0$ then $z_1 += s_z;\ z_d -= a_x$          ▷ move in z-direction
11:            $x_1 += s_x$                                          ▷ move in x-direction
12:            $y_d += a_y;\ z_d += a_z;$
13:            Check status of node at $s = (x_1, y_1, z_1)$
14:            if $s$ contains an obstacle then
15:                return false
16:    else if $a_y \geq \max(a_x, a_z)$ then
17:        $x_d = a_x - a_y/2$
18:        $z_d = a_z - a_y/2$
19:        while $y_1 \neq y_2$ do
20:            if $x_d \geq 0$ then $x_1 += s_x;\ x_d -= a_y$
21:            if $z_d \geq 0$ then $z_1 += s_z;\ z_d -= a_y$
22:            $y_1 += s_y$
23:            $x_d += a_x;\ z_d += a_z;$
24:            Check status of node at $s = (x_1, y_1, z_1)$
25:            if $s$ contains an obstacle then
26:                return false
27:    else if $a_z \geq \max(a_x, a_y)$ then
28:        $x_d = a_x - a_z/2$
29:        $y_d = a_y - a_z/2$
30:        while $z_1 \neq z_2$ do
31:            if $x_d \geq 0$ then $x_1 += s_x;\ x_d -= a_z$
32:            if $y_d \geq 0$ then $y_1 += s_y;\ y_d -= a_z$
33:            $z_1 += s_z$
34:            $x_d += a_x;\ y_d += a_y;$
35:            Check status of node at $s = (x_1, y_1, z_1)$
36:            if $s$ contains an obstacle then
37:                return false
38:    return true
```

If the two nodes do not have line-of-sight, the traversal cost is said to be infinity and the nodes are not considered successors of one another. While it is possible that a small obstacle between them is easy to navigate around, it is simpler to assume such a path is blocked. In the case where this assumption is wrong, it will later be mostly or completely corrected by path smoothing, and therefore is a safe simplifying assumption to make.

This check is only performed for the highest level of path-finding, as it is already known whether or not line-of-sight exists when paths are refined to lower levels. These line-of-sight checks can be costly, especially as the distance between the nodes increases at coarser levels. To counter this, the checks are only performed when at least one of the nodes is within the search radius of the UAV. The environment may be partially known outside of this radius, but as shown later, the resulting paths from this setup provide a suitable balance between computation time and path length.

Chapter 5: Methods Used to Improve Path Quality

The framework is now set up to adapt and merge D* Lite and HPA* into HD*, but there are many improvements that can be made. This chapter will discuss the optimizations implemented to produce shorter and more realistic looking paths, and Chapter 6 will detail the techniques used to obtain performance suitable for real-time use.

5.1 Abstract Levels

With hierarchical planning, there is no reason to restrict the map to just one abstraction. In fact, it is beneficial to have more levels since this provides more flexibility when planning the coarse paths. When the start and goal nodes are on opposite sides of a large map, a very coarse abstract level is desired to quickly find a path. It is also possible that on this same map the next target is close by, in which case the coarse level is not useful. In this situation it is ideal to have multiple abstract levels, and the one used to find the initial coarse path is dependent on how far away the goal is. Multiple abstract levels also results in improved quality during the path refinement and smoothing stages, which are discussed in Section 5.2

Therefore, HD* abstracts the map into multiple levels, where level 0 is the initial map resolution. Due to the method used to determine successors, the abstract levels are easily defined by the distance between a node and its successors. This successor distance is measured in the number of nodes, and the distance used for each level is given by 2^{n+1}, where n is the level number. Therefore, level 1 successors are a distance of $2^{1+1} = 4$ nodes away, level 2 successors are 8 nodes away, level 3 successors are 16 nodes away, and so on. The maximum level used by HD* is the level in which the distance between successor nodes is approximately ⅛ the maximum map dimension. This cutoff was used to ensure the paths are good quality, as a grid that is too coarse provides provides poor paths. This limit also helps reduce the time spent on line-of-sight checks, which take more time to execute as the distance between nodes increases.

For example, a map with an initial size of 128 in each dimension would have four levels. Level 0 consists of the individual nodes composing the map. Levels 1, 2, and 3 use successor nodes as stated above. At level 3, the successor nodes are 16 nodes apart, which is ⅛ of 128, so no additional levels are created.

5.2 Path Refinement and Smoothing

The use of multiple abstract levels allows the inclusion of a refinement stage to increase the benefits of path smoothing. The initial coarse path is likely to pass widely around obstacles, and smoothing would not do much to resolve this if the nodes are far apart. To improve the smoothing quality, the initial coarse path is repeatedly refined until it is represented by the highest resolution nodes. This refinement is performed by using the next level down to compute the shortest path between successive nodes of the current level. If the initial coarse path is a level 3 path, the first stage of refinement is performed by finding the shortest path between each pair of these nodes using level 2 nodes. With the coarse path represented by level 2 nodes, the process is repeated using level 1 nodes. Finally, it is repeated one more time until represented by the level 0 nodes. This increases the number of nodes used to define the path, which increases the quality of path smoothing and therefore provides shorter and more realistic paths. Once represented as level 0 nodes, the path is smoothed.

The smoothing algorithm is presented in Algorithm 4. It is a modified version of the one presented in [13], and works as follows. The path is input as a series of nodes. Starting from the first node in the path, s_0, the smoothing algorithm checks for line-of-sight to the third node in the path, s_2. If line-of-sight exists, the intermediary node s_1 is removed from the path. This process is repeated until the goal node is reached or there is not line-of-sight from the first node. If there is not line-of-sight to a node, the process now restarts from that node and continues in the same fashion. When the status of a node is unknown, it is assumed to be open. Therefore, when smoothing the path, line-of-sight is assumed to exist between nodes that have yet to be explored.

Smoothing is only used in directions of uniform cost in order to ensure this process does not increase path cost. This can occur when altitude changes are expensive. Smoothing a quick and steep traverse into a gradual slope results in the UAV traveling upwards for a longer time. HD* considers the cost of altitude changes to be the same regardless of the path angle relative to the ground, thus using more distance to change altitude would result in an increased cost.

Algorithm 4 Path Smoothing Algorithm

This version assumes $c_x = c_y = 1$

```
 1: function SmoothPath($[s_0, s_1, ..., s_n]$)    ▷ Nodes are $(x, y, z)$ coordinate tuples
 2:     $k = 0$
 3:     $p_k = s_0$
 4:     for $i = 1$ to $n - 1$ do
 5:         $x_1 = p_{k,x}$ ; $y_1 = p_{k,y}$ ; $z_1 = p_{k,z}$
 6:         $x_2 = s_{i+1,x}$ ; $y_2 = s_{i+1,y}$ ; $z_2 = s_{i+1,z}$
 7:         if $(z_1 \neq z_2$ and $c_z \neq 1)$ or not LineOfSight$(p_k, s_{i+1})$ then
 8:             $k += 1$
 9:             $p_k = s_i$
10:     $k += 1$
11:     $p_k = s_n$
12:     $path = [p_0, p_1, ...p_k]$
13:     return $path$
```

The next step involves generating a centripetal Catmull-Rom spline [28] to smooth any sharp turns from the path. This type of spline is used as it is guaranteed to pass through the specified nodes [16], and will not result in any self-intersections or cusps [28]. To create the splines, four consecutive points are used as the input, and the result is a spline between the second and third points. To generate a spline between the first two nodes, we simply input the first node as both the first and second points. Likewise, the spline between the last two nodes is created by using the final node for the third and fourth input points. The spline is represented by three additional points between the second and third input points, with pseudocode presented in Algorithm 5. Generation of these splines brings an increase in path length but is necessary to create a path that can realistically be followed.

In the final step, the points that define that path are used to generate a trajectory composed of (x, y, z) coordinates, with each coordinate no more than one unit away from the previous one. The number of coordinates generated between two successive nodes in the path is equal to the maximum absolute value of the differences in the x-, y-, and z-coordinates. If two consecutive nodes in the path are located at $(9, 2, 6)$ and $(6, 10, 7)$, then there will be 8 points used to define the trajectory between them. This trajectory is followed until an obstacle is detected or the halfway point of the refinement region is reached. The refinement region is explained, along with a walkthrough of the path-finding process, in Section 6.3.

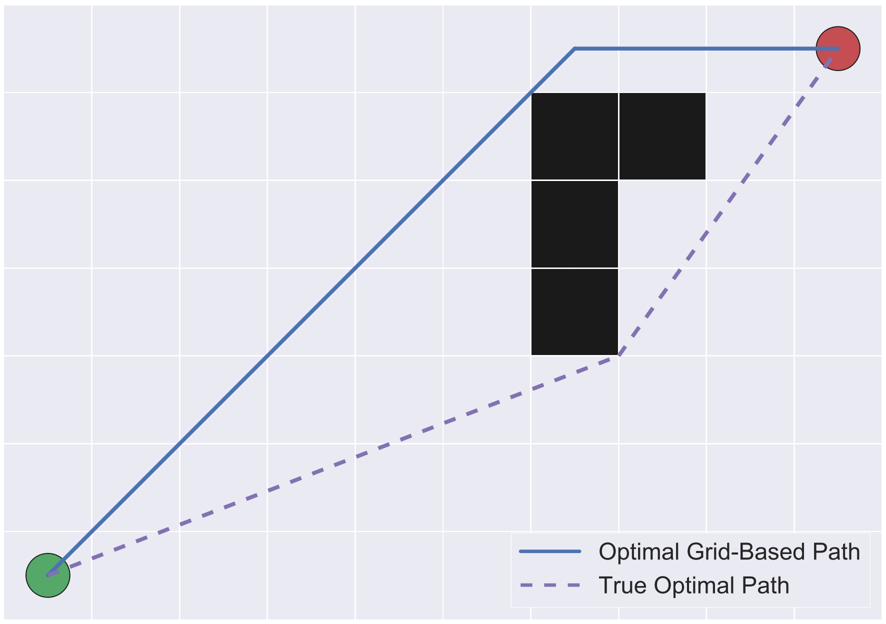


Figure 5.1: Optimal path produced by grid-based planners compared to the true optimal path.

It is important to note that although smoothing will result in shorter paths, this does not mean it always results in the true shortest path. The path found by using a grid representation may be the optimal path found by traveling between node centers, but node centers do not always compose the true shortest path. This is shown in Figure 5.1, where the optimal path between two locations cannot be found, and smoothing does not reduce path length. Any angle planners such Theta* [13] and Field D* [14] are better at finding a near-optimal path in these situations. These algorithms are more computationally expensive as they require frequent line-of-sight checks or interpolation to determine what angle to travel at. Path lengths may be shorter with these approaches, but Chapter 7 shows that HD* outperforms 3D Field D* [3] and still produces near-optimal path lengths.

Algorithm 5 Catmull-Rom Spline Generation

t_i determines the parameterization, $\alpha = 0.5$ results in the centripetal parameterization used by HD*, p_1 and p_2 are a pair of (x, y, z) input control points. HD* uses $nPts = 5$, resulting in three points between p_1 and p_2.

```
1: function ParameterValues($t_i, p_1, p_2, \alpha$)
2:     $dx = p_{1,x} - p_{2,x}$ ; $dy = p_{1,y} - p_{2,y}$ ; $dx = p_{1,z} - p_{2,z}$
3:     return $\left(\sqrt{dx^2 + dy^2 + sz^2}\right)^{\alpha} + t_i$
4: function CatmullRomPoints($p_0, p_1, p_2, p_3, nPts$)
5:     $t_0 = 0$
6:     for $i = 1$ to 3 do $t_i$ = ParameterValues($t_{i-1}$, $p_{i-1}$, $p_i$, $\alpha$)
7:     if $t_0 = t_1$ then $t_1 = 10^{-8}$                ▷ to avoid divide by zero error
8:     if $t_2 = t_3$ then $t_3 = t_2 + 10^{-8}$
9:     $t$ is a linearly spaced column vector between $[t_1, t_2]$ with $nPts$ elements
10:    $L_{01} = \frac{t_1 - t}{t_1 - t_0} \times p_0 + \frac{t - t_0}{t_1 - t_0} \times p_1$
11:    $L_{12} = \frac{t_2 - t}{t_2 - t_1} \times p_1 + \frac{t - t_1}{t_2 - t_1} \times p_2$
12:    $L_{23} = \frac{t_3 - t}{t_3 - t_2} \times p_2 + \frac{t - t_2}{t_3 - t_2} \times p_3$
13:    $L_{012} = \frac{t_2 - t}{t_2 - t_0} \times L_{01} + \frac{t - t_0}{t_2 - t_0} \times L_{12}$
14:    $L_{123} = \frac{t_3 - t}{t_3 - t_1} \times L_{12} + \frac{t - t_1}{t_3 - t_1} \times L_{23}$
15:    $C = \frac{t_2 - t}{t_2 - t_1} \times L_{012} + \frac{t - t_1}{t_2 - t_1} \times L_{123}$
16: function CatmullRomSpline($path$)
17:     if $path$ has less than 3 nodes then
18:         Not enough nodes to generate a spline
19:         return $path$
20:     else
21:         Copy first and last nodes such that $path_0 = path_1$ and $path_{n-1} = path_n$
22:         $C = \emptyset$
23:         $k = \text{length}(path) - 3$
24:         for $i = 1$ to $k$ do
25:             $c$ = CatmullRomPoints($path_i$, $path_{i+1}$, $path_{i+2}$, $path_{i+3}, nPts$)
26:             $C = \{C\} \cup \{c\}$
27:         return $C$
```

5.3 Cost Computation

Although HD* is not designed to account for the UAV's dynamics, it should be capable of at least handing some basic constraints. The cost to change elevation can vary depending on the particular vehicle, and a planner that neglects this will not always produce the optimal path. Likewise, some UAVs cannot travel vertically, so a path that includes vertical movement is useless for these vehicles. Therefore, it is essential that the planner factor in these cost constraints.

First, the default cost of traversal between two nodes must be defined. This is simply the Euclidean distance between the two nodes. If line-of-sight does not exist or the target node has an obstacle, the cost is infinity. Assuming the target node is open, the final traversal cost is determined by multiplying the distance by a directional cost scale factor, given by c_x, c_y, and c_z for the x-, y-, and z-directions respectively. These scale factors have a default value of one and can be configured by the user to modify the cost of travel in each direction. The predominant scale factor is used when computing cost. If the cost of the z-direction is set to 2 and the UAV is traveling upwards at some angle relative to the ground, the cost is still 2 and is not reduced by the fact that travel is not vertical.

Algorithm 6 Cost Computation

```
This version assumes $c_x = c_y = 1$
 1: function EuclideanDistance($s, s'$)
 2:     return $\sqrt{(s_x - s'_x)^2 + (s_y - s'_y)^2 + (s_z - s'_z)^2}$
 3: function ComputeCost($s, s'$)
 4:     if $s'$ contains an obstacle then
 5:         return $\infty$
 6:     else if computing cost for the coarse path then
 7:         if within search radius and not LineOfSight($s, s'$) then
 8:             return $\infty$
 9:     if $s_z \neq s'_z$ then
10:         $costFactor = c_z$
11:     else
12:         $costFactor = 1$                                ▷ Default planar cost
13:     $cost = costFactor \times$ EuclideanDistance($s, s'$)
14:     return $cost$
```

Depending on the environment, increasing c_z can result in the planner preferring strictly vertical movement. Many UAVs are unable to change their altitude in this manner, so this must be restricted as necessary. A boolean variable called $restrictVerticalMovement$ is used to control this, and when used, vertical successor nodes are no longer included in the set of successors. To implement these changes, the way D* Lite computes cost is modified. The cost between two nodes $c(s, s')$ was previously the distance between the nodes. We replace this with the function ComputeCost(s, s'), which is defined in Algorithm 6. The function shown assumes the cost in the x- and y-directions is one, and the z-direction cost is varied. This is a reasonable assumption the majority of the time but can be easily altered if desired by modifying the "if" statements beginning on line 9. This cost computation function is the same when computing paths at any level, with the addition of a line-of-sight check for the coarse paths.

5.4 Safety Margin

Because HD* searches for the shortest path, it is likely that portions of the path will be directly adjacent to obstacle edges and corners. For a realistic implementation, this is dangerous and increases the risk of collision. It is also possible that the generated path passes through a region too narrow to safely pass through. Thus, a safety margin is needed to prevent the UAV from getting too close to obstacles. The safety margin is defined as the distance, in nodes, to maintain between the agent and an obstacle. To implement it, we extend the footprint of obstacles to include the surrounding nodes within the safety margin.

Chapter 6: Methods Used to Improve Performance

6.1 Data Structures

When finding a path, a large percentage of time is spent managing the priority queue, so the data structure chosen to manage this queue is an important decision. The four primary operations performed on the queue are:

1. Pop best node, which is the action of removing the best node from the priority queue and returning the node number and key values.

2. Find node, which checks whether a node is a member of the priority queue.

3. Add node, in which is a new node is added to the priority queue.

4. Update node, which updates a node's priority when it is already in the queue and the new key value is different from the existing value.

To determine the ideal data structure both the frequency of these operations and their time complexities must be considered. The different data structures and their worst case performance for each operation [27, 29] are compared in Table 6.1. For heaps, the time complexities of "Pop Node" and "Add Node" include the time to restore the heap properties. The hybrid structure uses a binary heap to add nodes and pop the best node, and a hash table to check for membership.

We refer to Algorithm 2 to determine the frequency of these operations. Line 14 of the pseudocode shows that the pop operation occurs once for each node visited. Lines 17 and 20 show that UpdateVertex(s) is called once for each successor, which is up to 26 times per node. Consequently, the speed of the membership check and node insertion operations are important, as they occur frequently. The update node operation on line 35 is the least common, which is fortunate since Table 6.1 shows that it is an expensive two-step operation.

Unsorted arrays can be removed from consideration because of their slow membership test, and sorted arrays can be eliminated due to their slow insertion. The Fibonacci heap seems efficient in theory but is complicated to implement and not as fast in practice [29, 30]. This leaves the binary heap and hybrid structure as the remaining options. The binary heap may be simpler to implement, but a hybrid structure was chosen for its superior performance when finding and updating nodes. Algorithm 7 shows the pseudocode used to implement this structure. HD* is written in Python so the binary heap is implemented via the “heapq” module, and the hash table is implemented using a dictionary.

Table 6.1: Comparison of asymptotic upper bound running times between different data structures for priority queue operations.

Data Structure	**Pop Best Node**	**Find Node**	**Add Node**	**Update Node**
Unsorted Array	$O(n)$	$O(n)$	$O(1)$	Find: $O(n)$ Update: $O(1)$
Sorted Array	$O(1)$	$O(\log n)$	$O(n)$	Find: $O(\log n)$ Update: $O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	Find: $O(\log n)$ Update: $O(\log n)$
Fibonacci Heap	$O(\log n)$	$O(\log n)$	$O(1)$	Find: $O(\log n)$ Update: $O(1)$
Hybrid Structure	$O(\log n)$	$O(1)$	$O(\log n)$	Find: $O(1)$ Update: $O(\log n)$

Algorithm 7 Priority Queue Operations for Hybrid Data Structure

The entries in *entryfinder* and U are formatted as $[k_1, k_2, u]$, where k_1 and k_2 represent the priorities and u is the node

```
1: $entryfinder = \emptyset$                ▷ Hash table mapping tasks in $U$ to entries
2: function RemoveNode($u$)
3:     Delete entry in $entryfinder$ with key $u$
4: function AddNode($k_1, k_2, u$)          ▷ Add node or update key of existing node
5:     if $u \in entryfinder$ then
6:         RemoveNode($u$)
7:     Add key $u$ with value $[k_1, k_2, u]$ to $entryfinder$
8:     Add entry $[k_1, k_2, u]$ to $U$
9: function PopNode( )
10:    while True do
11:        $k_1,\ k_2,\ u = U$.Pop()
12:        if $u \in entryfinder$ then
13:            Delete entry in $entryfinder$ with key $u$
14:            return $[k_1, k_2, u]$
```

The data structure used to keep track of the cost, g-value, rhs-value, and backpointer for each node must also be determined. $O(1)$ performance is desired for lookup of these values to minimize the time spent on these operations. From

Table 6.1 we saw that the use of hash tables for the hybrid structure allowed for $O(1)$ lookup of a value. For smaller maps, hash tables are suitable to store these four values. However, for larger maps this is not always feasible, as the storage requirements would get very large. In Python, the hash table for a $256 \times 256 \times 256$ map would contain nearly 17 million keys and takes 805 MB of storage for the dictionary overhead alone. Since each node has four values associated with it, the actual storage space would be even greater, particularly when map size increases. Instead, Python's "defaultdict" container is used to store these values. This provides $O(1)$ lookup like a standard dictionary, but does not require an entry for each node. If the lookup key, which is the node, does not exist in the dictionary, a default value is returned. The default values are 1 for the cost, infinity for the g-values and rhs-values, and "None" for the backpointers.

6.2 Heuristic Choice

As described in Section 3.1, an admissible heuristic is required to find the shortest path between the start and goal. In this implementation, the Euclidean distance between two points represents the smallest possible cost between two nodes and is both an admissible and consistent heuristic. In reality, distances will generally be longer than this estimate due to the presence of obstacles. A heuristic which underestimates the true distance is guaranteed to find the shortest path, and the more it underestimates the distance the more nodes it will search [6]. This is evident in the limiting case when the heuristic is zero, and A* reduces into Dijkstra's algorithm [27], which expands every node until it finds the shortest path.

On the other hand, a non-admissible heuristic overestimates the distance between two nodes, which results in fewer nodes being expanded. The downside to this is that finding the shortest path is no longer guaranteed [6]. Since the ability to find a true shortest path was already sacrificed by implementing hierarchical planning, it is worth making the heuristic slightly non-admissible to help reduce computation time. Inflating the heuristic also helps speed up the search is when there are multiple paths with the same cost, which is common on uniform cost grids. An admissible heuristic will spend more time deciding between paths of equal or near equal cost than a non-admissible heuristic [31]. For these reasons, it is often beneficial to increase the heuristic value to improve performance. It has been proven in [32] that for a consistent heuristic which has been multiplied by a factor of $(1+\epsilon)$, the resulting path is guaranteed to be within $(1+\epsilon)$ times the shortest path. This is expressed in 6.1, where L_{opt} represents the optimal path length and L_{mod} represents the path found when the heuristic is modified.

$$L_{opt} \leq L_{mod} \leq L_{opt}(1+\epsilon) \tag{6.1}$$

HD* uses a default value of $\epsilon = 0.01$, which would guarantee path lengths within 1% of optimal if hierarchical techniques were not used. The tradeoff between ϵ, path cost, and performance is explored in Chapter 7.

6.3 Path-Finding Process

HD* computes the entire coarse path from goal to start, but this does not mean the entire path must be refined as well. Only a portion needs to be refined to get the UAV moving in the right direction, and the remainder can be refined as necessary. Furthermore, when the environment is unknown outside of the search radius of the UAV, the planned coarse path may soon become invalid. Therefore, refining the entire path up front is wasteful. Instead, we only refine the section of the path that lies within the refinement distance d_r. This distance is expressed as a function of the sensor range, r_s, of the UAV. An intuitive choice would be to set $d_r = r_s$, but this may not be ideal when r_s is large. A large sensor range causes the time spent in the refinement to become greater, thereby increasing the planning time. Alternatively, this choice of d_r may be too conservative for situations when the agent has a small search radius. To counter these situations, HD* allows the user to specify the value of d_r, providing control over the trade-off between path quality and computation time. The value selected for d_r can significantly affect both path length and computation time, and we explore this relationship in Chapter 7.

Once the lowest resolution path is computed, the portion within d_r is refined. The segment is continually refined by the process presented in Section 5.2 until represented in the highest resolution nodes. Next, the entire path is smoothed. Once smoothing is complete, the path is represented by many closely spaced level 0 nodes within the refinement region, and fewer nodes elsewhere. Splines are then generated and the exact coordinates defining the trajectory are computed. This process is shown in Figure 6.1 from the top down view of a $128 \times 128 \times 128$ 3D map. The UAV follows the resulting trajectory until one of four possible situations occur:

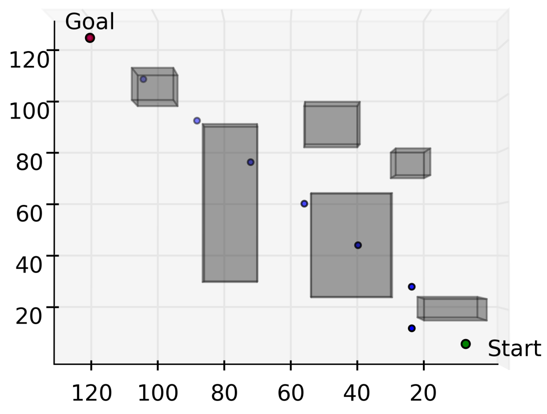
1. New obstacles are detected that invalidate the current path.
2. The target location has moved.
3. The UAV reaches the midway point of the refinement region.
4. The goal is reached.

When scenarios 1 or 2 occur, a new coarse path is planned, and the process restarts. Scenario 3 occurs when neither of the two preceding scenarios forces a replan before the midway point of the refinement region is reached. Scheduling replans halfway through the refinement region helps ensure high-quality paths. If scenario 4 occurs, no further planning is needed and the algorithm terminates.

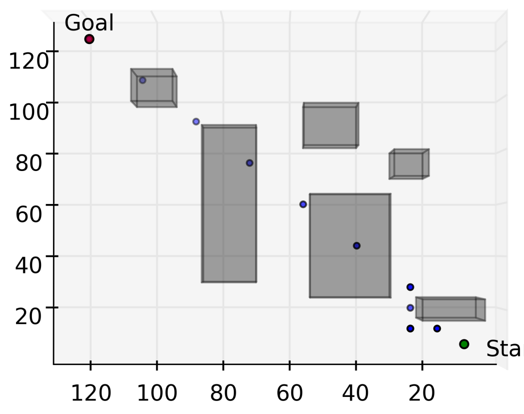
6.4 Time Restrictions

When there is a restriction on planning time, it is possible that the time required to complete all of these steps may be too long. Therefore HD* allows the user to set a time limit, t_{max}, on the path-finding process. This limit does not affect the smoothing, spline generation, and trajectory generation steps, as these are necessary to produce realistic paths. Instead, the limit affects the path refinement process.

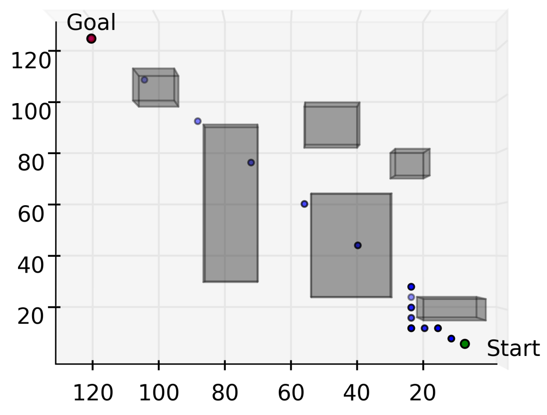
Once HD* finds a coarse path, the elapsed time is compared to the value of t_{max}. If the elapsed time is greater than t_{max}, the function exits and passes the coarse path to the smoothing function. If it is less than t_{max}, one additional level of refinement is completed. HD* again checks the elapsed time and executes another level of refinement if t_{max} has not been exceeded. This process repeats until the path has been refined down to level 0, or the time limit was reached. This approach is not a hard restriction in the sense that the total path planning time will exceed t_{max}, but it still allows for some flexibility in limiting the time spent planning.



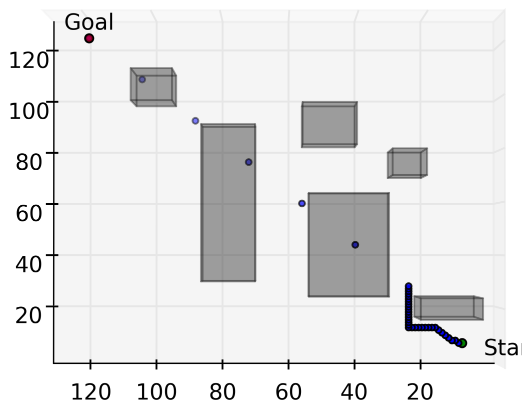
(a) First, the lowest resolution path is found using level 3 with nodes 16 units apart. Blue circles represent the nodes composing the path. Some nodes pass through obstacles, as the UAV is currently unaware of those obstacles.



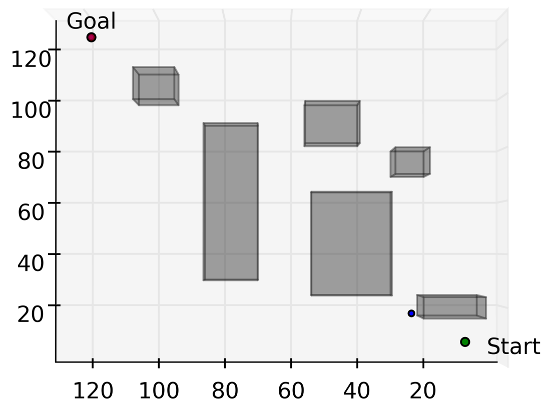
(b) Next, level 2 nodes a distance of 8 units apart are used to refine the segment of the path within the refinement distance.



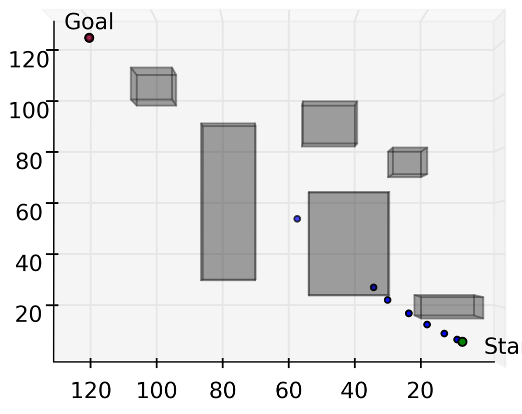
(c) Level 1 nodes a distance of 4 units apart are used to further refine the path.



(d) The next refinement stage uses the highest resolution nodes.



(e) The path is then smoothed. Only one node is needed to define the path because the UAV currently assumes line-of-sight exists from the blue node to the goal.



(f) Finally, additional points are generated with a centripetal Catmull-Rom spline. These points are used to create the trajectory composed of (x, y, z) coordinates that the UAV will follow.

Figure 6.1: Path Finding and Refinement Process

Chapter 7: Experimental Results

The performance of HD* was analyzed in a variety of situations. Performance is measured with respect to path cost, the number of nodes expanded, and mean path computation time. The values reported for computation time and are given in milliseconds and refer to the CPU time. It include all stages of the path-finding process, from the initial coarse path to the final trajectory. Therefore, the computation time represents the delay between requesting a new path and obtaining a path to follow. First, the suboptimality of HD* will be quantified; then it will be compared to 3D Field D*, which is another real-time planning algorithm. Next, various parameters will be modified to determine how performance can vary in different

Table 7.1: Default Testing Parameters

Parameter	**Value**
Map Size	$150 \times 150 \times 150$
Obstacle Density (ρ)	15%
Heuristic Scale (h_s)	1.01 ($\epsilon = 10^{-2}$)
Cost of Altitude Change (c_z)	2
Search Radius (r_s)	20
Refinement Distance (d_r)	r_s
Maximum Path-Finding Time (t_{max})	No limit
Restrict Vertical Movement	True

Table 7.2: HD* Path Cost vs. Optimal Path Cost

Obstacle Density (%)	5	10	15	20	25
Optimal Cost	198.39	198.92	199.78	201.39	201.80
HD* Cost	199.73	203.81	212.33	216.13	220.73
Percent Increase	0.68	2.46	6.28	7.32	9.38

scenarios. These parameters and their default values are listed in Table 7.1, and are the values used for each test case unless otherwise specified. Finally, a few example paths will be presented. Results were obtained on a 2.6 GHz Intel Core i5 MacBook Pro with 8 GB of RAM.

7.1 Quantifying Degree of Suboptimality

To measure the variability in cost, we compare the path cost found by HD* with an empty initial map to that found by A* with full knowledge of the environment. A* used an admissible heuristic to ensure the optimal path was found. The percentage differences were compared using random maps with obstacle densities ranging from 5% – 25%. At each density, 25 trials were run, and the median values are reported. Path lengths are reported without Catmull-Rom splines as the extra length generated by the splines may exacerbate the difference and is not fundamental to the planning procedure. The results are shown in Table 7.2.

We see that with fewer obstacle HD* finds paths that are very close to optimal, and the deviations become greater as the obstacle density increases. This trend is expected, as HD* must find a path fast enough to operate in real-time, and the effort required to find the optimal path increases with the number of obstacles. Section 7.3 discusses some changes can be made to the default HD* implementation if shorter path lengths are desired.

7.2 Comparison to 3D Field D*

In [3], the 3D Field D* (3DF) algorithm was tested on a $150 \times 150 \times 150$ map. This algorithm uses interpolation based planning to remove the limitation that the agent must transition between node centers when planning a path. 3DF and HD* are both based on D* Lite, so comparing the two allows us to determine if HD* improves upon similar existing algorithms. To compare HD* with 3DF, the experimental setup used in [3] was replicated. A sensor range of seven units was used, and the map was assumed to be initially empty. The path was updated as changes in the environment were detected. The agent began in the center of the environment at $(75, 75, 75)$ with the goal location at $(150, 75, 75)$. Vertical movement is allowed and $c_x = c_y = c_z = 1$. The obstacle densities tested were $\rho = 20\%$ and $\rho = 50\%$. Because 3DF is conceptually similar to D* Lite but with the addition of interpolation, a comparison with D* Lite is not presented as its performance will be similar to 3DF.

The results are presented in Table 7.3. Note that HD* did not use splines for the tests with 50% obstacle density, as the splines pass through obstacles in dense

Table 7.3: Comparison of HD* with 3D Field D*

Algorithm	**Obstacle Density (%)**	**Nodes Expanded**	**Run Time (ms)**
3D Field D*	20	25,200	11.07
HD*	20	2,293	9.67
3D Field D*	50	62,400	20.80
HD*	50	5,090	15.34

environments. This problem is discussed more in Section 8.1. Path length is not presented as it is not reported in [3], aside from the fact that the vehicle moved at least 75 units each time. We see that HD* finds path faster than 3DF, and the time savings increase as ρ increases. HD* is over 12% faster with $\rho = 20\%$, and about 26% faster when $\rho = 50\%$. Additionally, the number of nodes expanded by HD* is an order of magnitude less than 3DF. The difference in node expansions would become more drastic as map size increases. Although not evident from this data, the runtime of 3DF will increase as map size increases, as it must replan each path at the highest resolution. HD* is capable of providing more consistent performance in a variety of environments due to the hierarchical planning, which is the biggest benefit it provides over 3DF.

7.3 HD* Performance Analysis

Next, the performance of HD* is analyzed for a wide range of possible configurations to determine how performance varies. Twenty-five trials were run for each configuration, and the median values are reported. Error bars represent the 25th and 75th percentile values. The trials use randomly generated obstacles of size $5 \times 5 \times 5$, and parameters will be analyzed in the order they appear in Table 7.1. The start and goal locations are always located in opposite corners of the map, with $s_{start} = (5, 5, d_z/2)$ and $s_{goal} = (d_x - 5, d_y - 5, d_z/2)$ where the map dimensions are given by (d_x, d_y, d_z). The impact of directional cost factors and vertical movement restriction are not presented as they do not have a significant impact on performance. Directional costs alter the path but do not have a large impact on computation time, and the restriction of vertical movement simply removes two successor nodes.

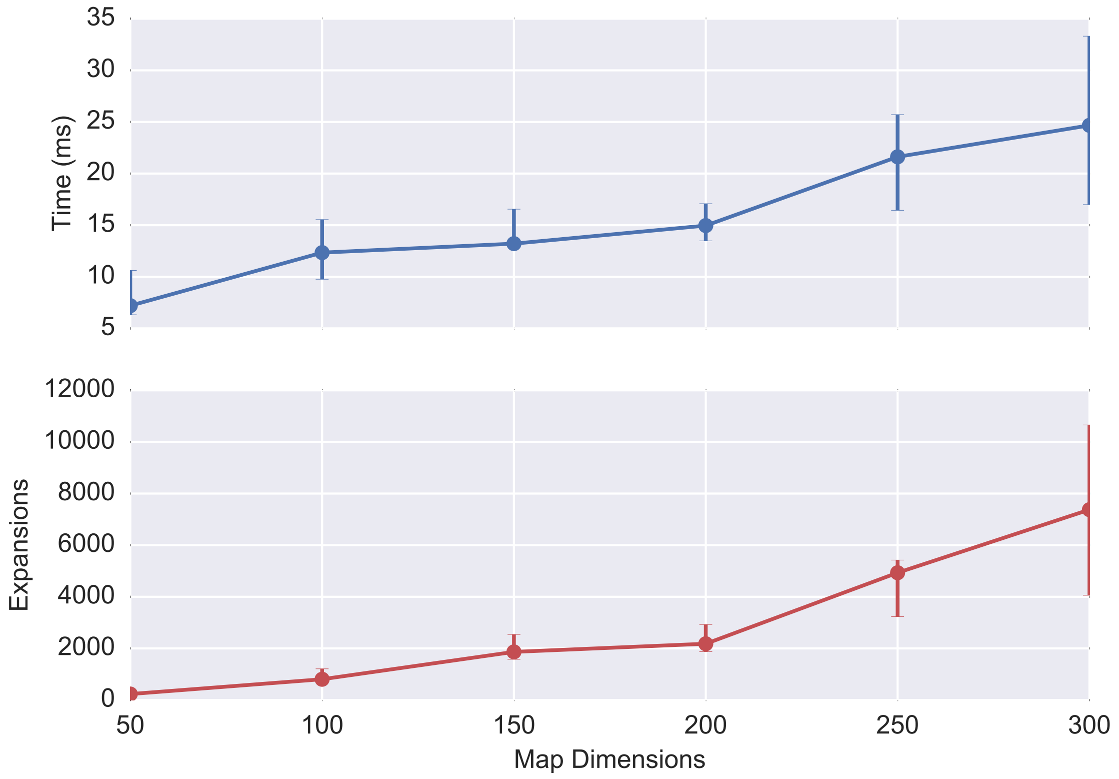


Figure 7.1: Effect of Map Dimensions on Performance

Figure 7.1 shows the impact of performance when map size is varied. The x-, y, and z- dimensions are equal for each trial, and tests were performed at values of 50, 100, 150, 200, 250, and 300. The results match what should be expected for an algorithm that must plan the entire path. We see that planning time is correlated with the number of nodes expanded, and both increase with map size. However, planning time does not increase as fast as map size. The planning time for the largest map is less than 3.5 times that of the smallest map, which is impressive considering the larger map has 216 times more nodes.

The increase in node expansions can be attributed to two main factors. The first is simply that a larger map means the goal is further away and there are more nodes to explore. The second factor is that the sensor range remains constant as map size increases. When the map dimensions are $50 \times 50 \times 50$, the search radius covers 40% of the length of the map and can see 6.4% of all the nodes. Comparatively, these values decrease to 6.67% of the length and 0.03% of the entire map for the largest map tested. Such little foresight on the larger maps results in less informed paths, leading to routes that are more likely to be suboptimal and therefore require frequent replanning and more node expansions.

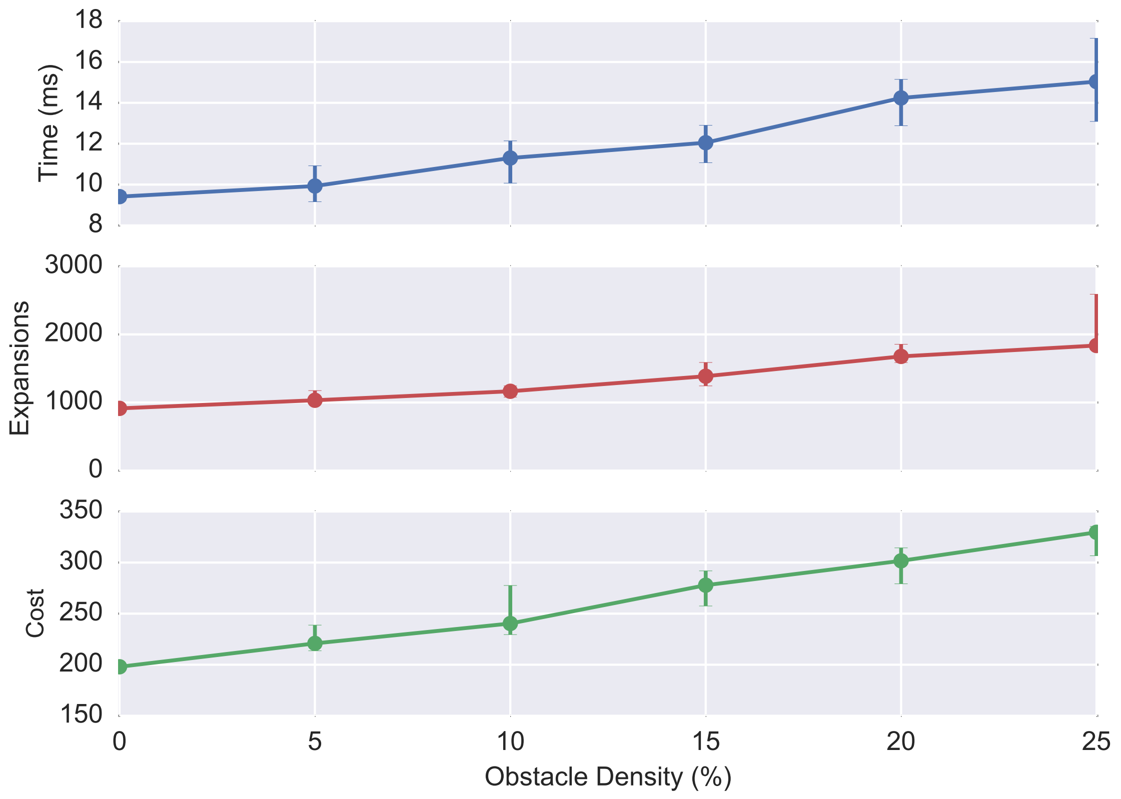


Figure 7.2: Effect of Obstacle Density on Performance

The effect of obstacle density can be seen in Figure 7.2. Obstacle densities from 0% – 25% were tested in increments of 5%. As expected, the more complex the environment is, the longer planning takes and the longer the resulting path lengths. If the planning needs to be sped up, the values of h_s, r_s, d_r, and t_{max} may be modified to decrease planning times, but this frequently comes with an increase in path cost.

Recall that when increasing h_s, the algorithm overestimates the distance to the goal and is more aggressive in pushing the search towards the goal. This results in following a path that may not be optimal, but in return it reduces the number of nodes expanded [6]. This relationship is explored in Figure 7.3, where the heuristic is multiplied by a scale factor of $(1 + \epsilon)$. The admissible heuristic is given by the case when $\epsilon = 0$. The tested ϵ values were 0, 10^{-3}, 10^{-2}, 0.05, 0.1, 0.25, 0.5, and 1.

It is evident from the figure that the impact of ϵ diminishes as it increases. The planning time and node expansions decrease until around $\epsilon = 0.05$, at which point results remain fairly constant with a few fluctuations attributed to the random maps. Interestingly, there does not seem to be a correlation with the cost. This must be due to the random maps, as it was proved in [32] that an increase in ϵ does result in longer paths. Therefore, we can conclude that because the number of node expansions reaches a limit, the cost increase should also reach a limit. To confirm this, one of the randomly generated maps was used to repeat the test, with the results shown in Figure 7.4. In this case, the maximum path cost is rapidly reached at $\epsilon = 0.001$, at which point computation time and the number of node expansions can be reduced without a corresponding cost increase. Above $\epsilon = 0.01$ the number of expansions remains constant, confirming that there is a limit to the trade-off experienced by inflating ϵ.

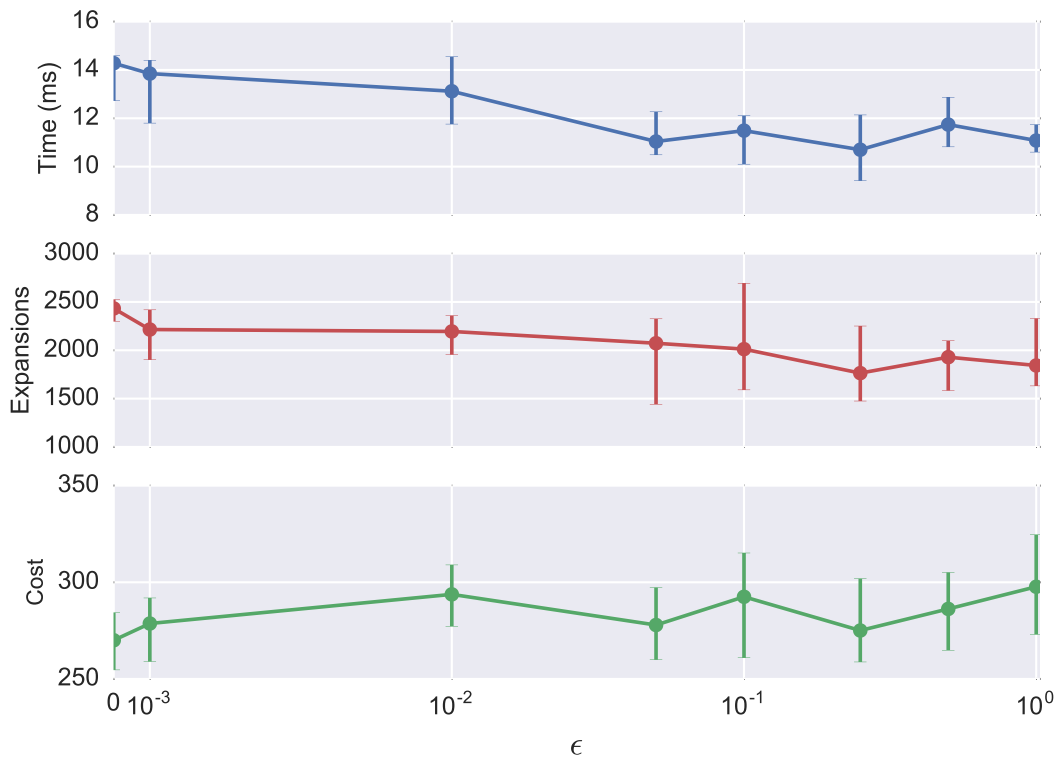


Figure 7.3: Effect of heuristic scale on performance with randomly generated maps.

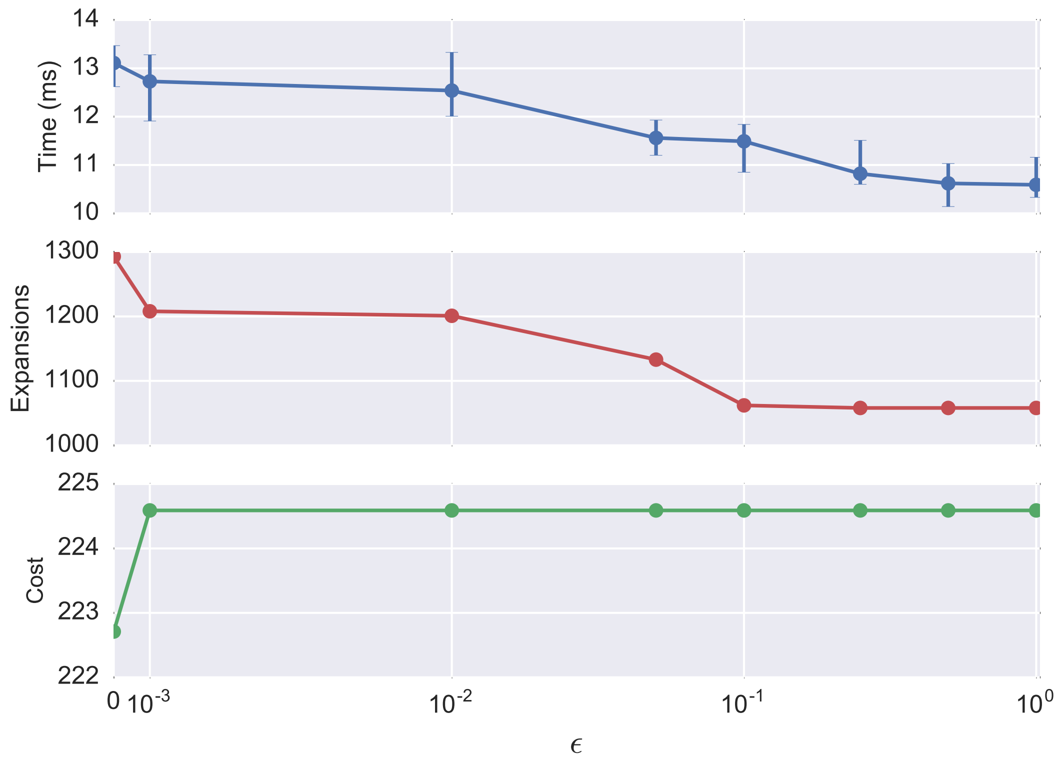


Figure 7.4: Effect of heuristic scale on performance with each test performed on the same map.

The impact of r_s and d_r can be seen in Figure 7.5. The value tested for r_s were 5 – 40 in increments of five, and the values of d_r were $r_s/4$, $r_s/2$, r_s, $1.5r_s$, and $2r_s$. When the search radius and refinement distances are small, the splines frequently pass through obstacles. This problem was seen earlier when running the comparison tests to 3DF, and is elaborated upon in the following chapter. To maintain consistency between trials, splines were not used for any of the tests performed to generate Figure 7.5.

There are a few insights gained from this figure. In the previous tests, the number of node expansions was correlated with computation time, but the opposite is true here. As the number of node expansions decreases, planning time increases. We know that as r_s and d_r increase, less replanning is necessary because more of the map can be seen at any given time, resulting in fewer node expansions. However, these larger values of r_s and d_r also result in more time dedicated to the refinement stage. Thus, Figure 7.5 leads to the conclusion that the refinement process dominates node expansions in regards to impact on planning time.

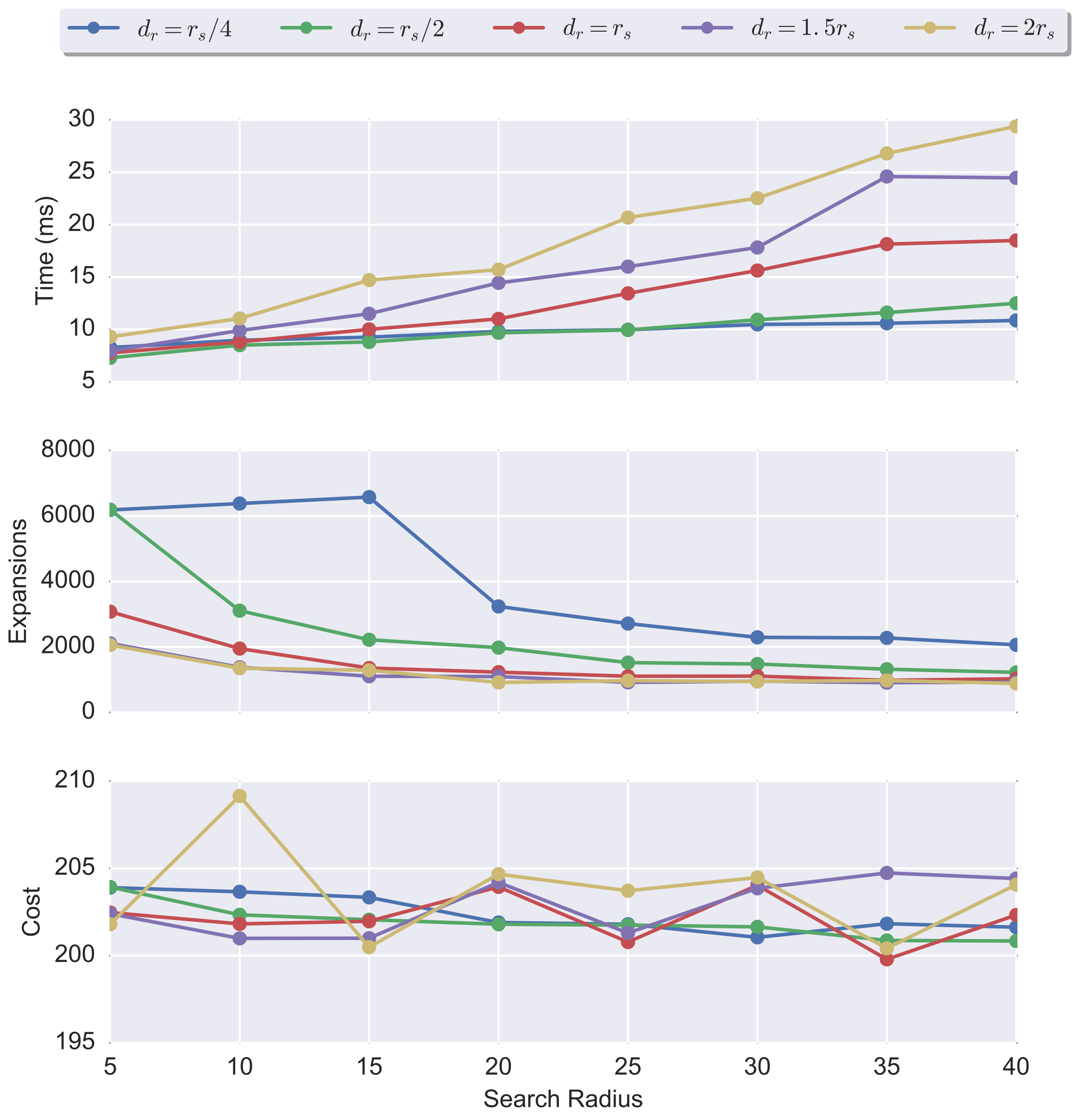


Figure 7.5: Effect of Search Radius and Refinement Distance on Performance

Intuitively, it would be expected that larger values of r_s and d_r would result in shorter paths, but it appears there is no correlation between the two. This was also observed during the heuristic tests and is again suspected to be the result of the randomly generated maps that were used. It is possible that the layout of the maps results in similar paths being favored regardless of these parameters. This is tested in Section 7.4 by comparing results on a map designed to represent a portion of a city.

The final parameter to examine is the maximum time limit t_{max} applied to planning, which is shown in Figure 7.6. A limit of zero milliseconds will only find the most coarse path, which is not refined before being passed on to the remainder of the path-finding process. As the limit increases, the planner has more time to refine the path, which is expected to result in a shorter path. From the figure, it appears the path length only begins to decrease after about 4 ms. By comparing the x-axis to the time plot in Figure 7.6, the relationship between the configured limit and actual computation time can be seen. The figure indicates that HD* needs a minimum of about 6 ms to find a path to begin following.

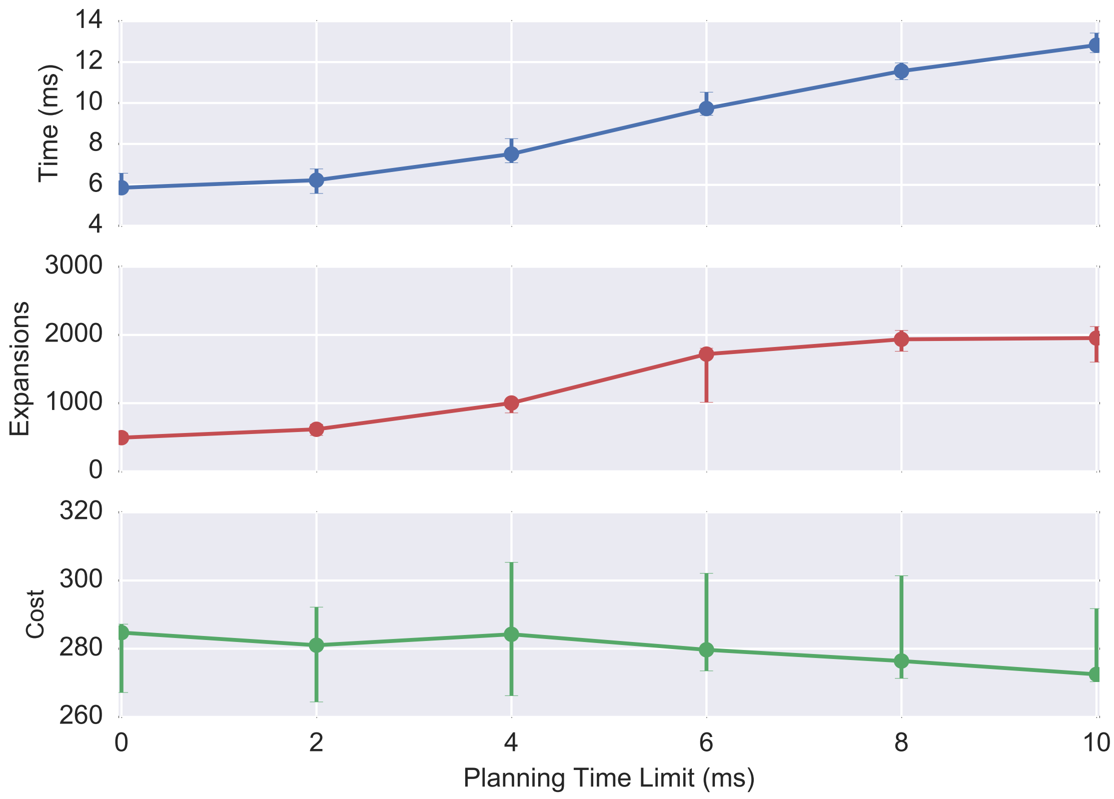


Figure 7.6: Performance Impact of Restrictions on Planning Time

7.4 Example Paths Produced by HD*

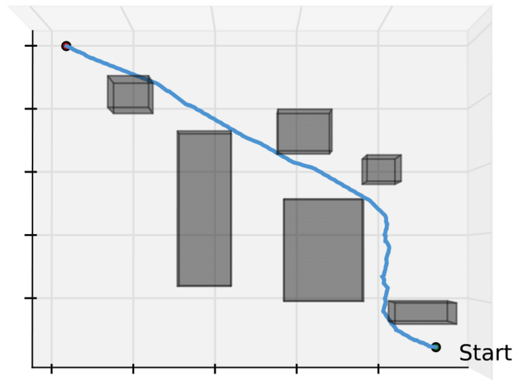
This section presents various paths produced by HD* in a $256 \times 256 \times 256$ environment designed to be representative of a city. The examples presented use $c_z = 2$ and therefore the paths do not change elevation. If we had set $c_z = 1$ there would be elevation changes, but for clarity paths of constant altitude are shown. Figure 7.7 compares the resulting paths when varying r_s and Figure 7.8 shows the impact of varying d_r. This was done in response to the results of Figure 7.5. Table 7.4 provides the performance data for these maps. The figures show a top-down view of the map, and the correlation between both r_s and d_r with path length

Table 7.4: Performance of Paths Shown in Figures 7.7 and 7.8

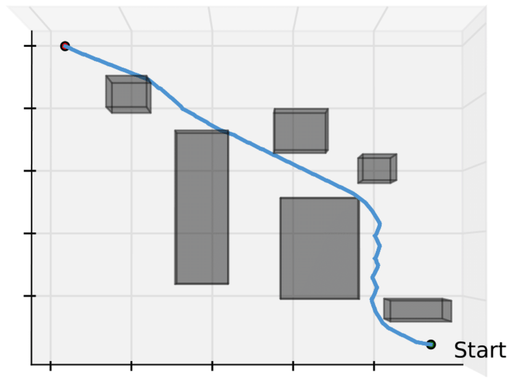
Figure Number	$\boldsymbol{r_s}$	$\boldsymbol{d_r}$	**Expansions**	**Time (ms)**	**Cost**
7.7(a)	20	1/2	6049	14.20	443
7.7(b), 7.8(b)	20	1	3684	16.40	429
7.7(c)	20	2	4092	22.47	399
7.8(a)	10	1	6434	13.32	423
7.8(c)	30	1	3698	19.71	408

becomes clear. Figures 7.7(c) and 7.8(c) represent the shortest and most realistic paths, suggesting that a larger search radius and refinement distance are key to producing quality paths.

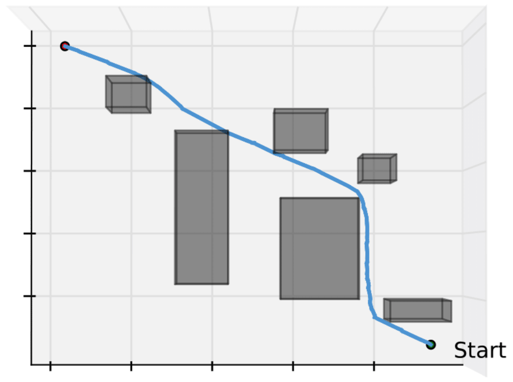
Figure 7.9 shows how the paths vary for different safety margins. The use of a safety margin does not impact performance, and simply expands the footprint of obstacles by treating the surrounding nodes as if they had infinite cost. The same city map is used but at half the size to more clearly see the gap between the path and the obstacles. These paths used $r_s = 20$ and $d_r = 2r_s$, as this combination has been shown to produce high-quality paths for the examples above. The figure shows that the path can vary widely depending on the chosen safety margin. We can see that even with the addition of splines, the generated path may still result in sharp turns that may be difficult or impossible to execute. A revision to HD* that allows the current heading of the agent to be considered should resolve this issue and is discussed in the following chapter.



(a) $d_r = 1/2$; The short refinement distance leads to jagged paths.

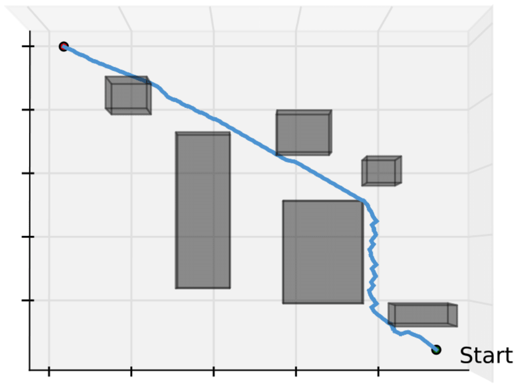


(b) $d_r = 1$; Paths are a little smoother and shorter, but still contain unnecessary heading changes.

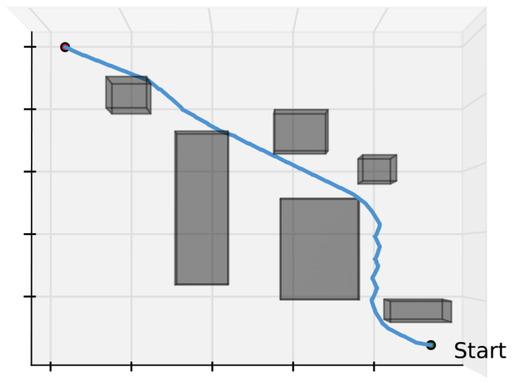


(c) $d_r = 2$; A larger refinement distance leads to ideal paths with no unnecessary heading changes.

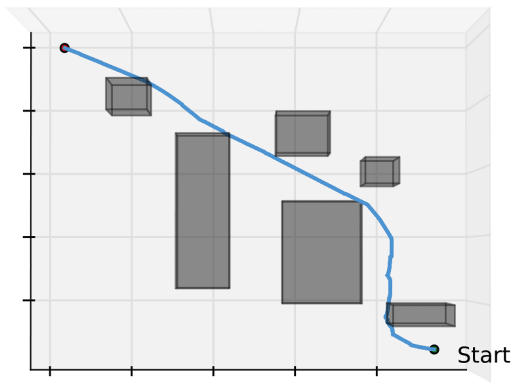
Figure 7.7: Paths produced when $r_s = 20$ for varying values of d_r



(a) $r_s = 10$; A small sensor range results in suboptimal paths that frequently need to be replanned, leading to a jagged traverse.

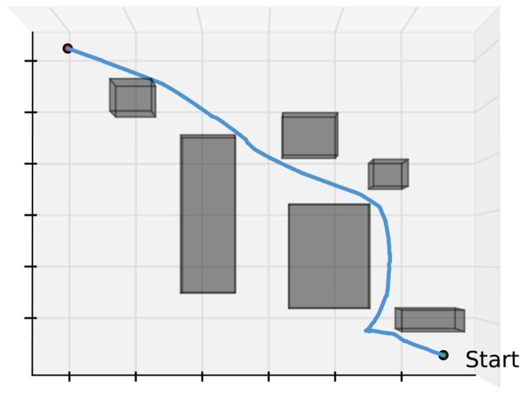


(b) $r_s = 20$; Note this is the same as Figure 7.7(b). The increased search radius produces better, more informed paths, but it still contains frequent heading changes.

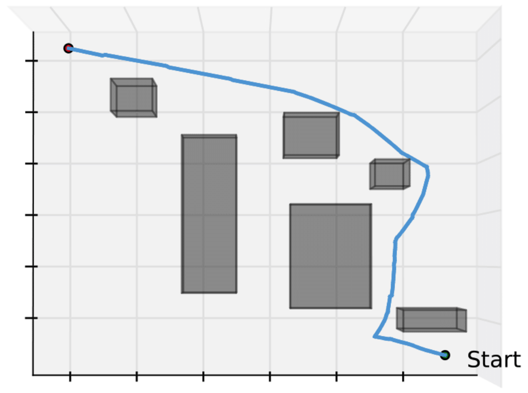


(c) $r_s = 30$; Path quality is further improved, and can be made smoother if a larger sensor range or refinement distance is used.

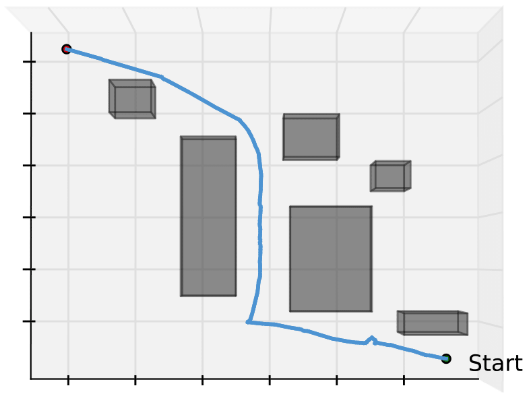
Figure 7.8: Paths produced when $d_r = 1$ for varying values of r_s



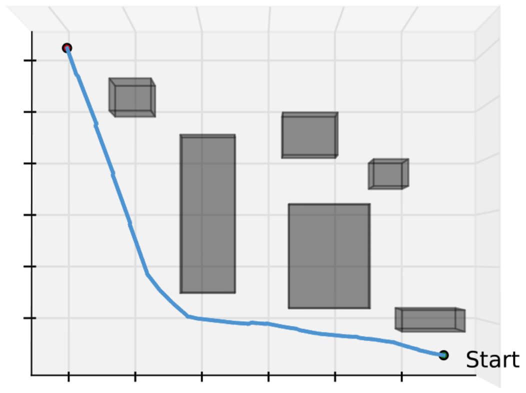
(a) Safety Margin = 2; Here, the UAV goes past the first gap and makes a sharp turn to gain enough clearance between the two obstacles.



(b) Safety Margin = 3; In the top right, we see the UAV can no longer fit through the second gap and must take an alternative route.



(c) Safety Margin = 4; The first gap has a width of eight units, so the UAV abruptly turns around to follow a new path when it learns the gap is too narrow.



(d) Safety Margin = 5; Traveling around all of the obstacles is now the only route that provides sufficient clearance.

Figure 7.9: Paths produced by various safety margins.

Chapter 8: Future Work and Conclusion

8.1 Future Work

Although HD* is capable of producing realistic, near-optimal paths in real-time, there are still a few aspects that can be improved. As briefly discussed earlier, when using a small search radius or in an obstacle dense environment, the spline generation sometimes results in turns that pass through nearby obstacles or violate the safety margin. This is because they do not follow the path exactly, so the spline path cannot always be guaranteed to be free of obstacles. A possible solution is to inflate the selected safety margin, to ensure extra surrounding nodes are open to contain the spline path. Alternatively, another method for the splines can be used which allows a turning radius to be input. This could help keep tighter curves while also accounting for the actual turning radius of the vehicle. It has also been shown that splines are not sufficient to prevent all sharp turns, so factoring in the current direction of travel would further improve path quality.

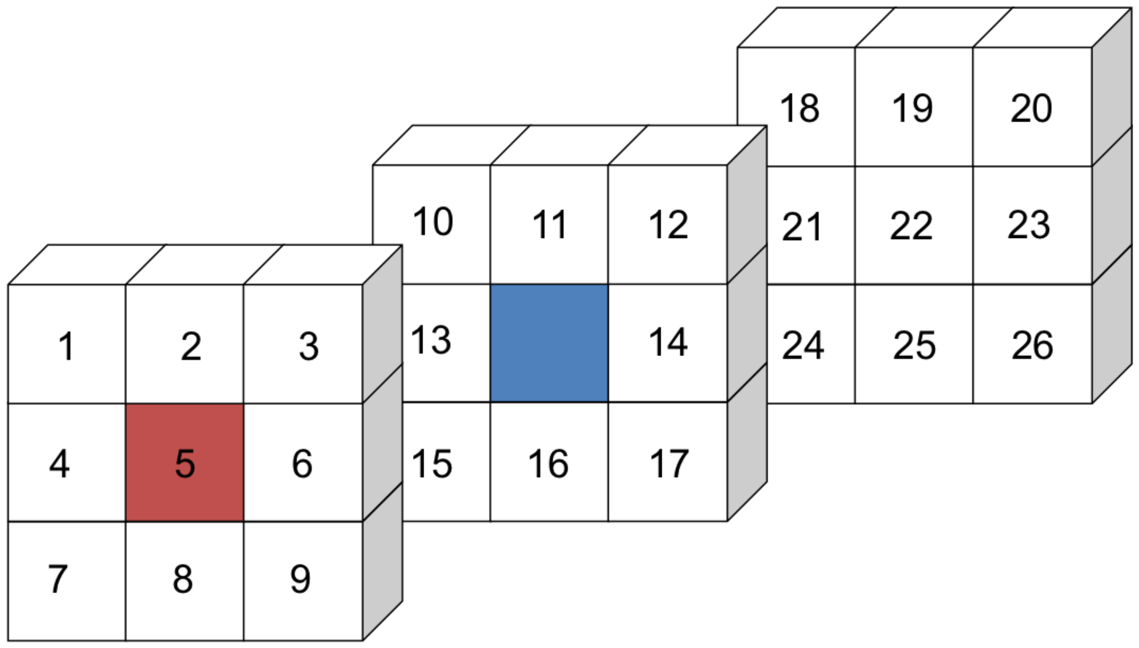


Figure 8.1: A possible approach to reduce the number of successor nodes. After the red node is expanded, the blue node has the lowest priority and is expanded next. Nodes 1-17 are all accessible from the red node, so we can conclude that, if expanded, their priority will not be lower than the blue node's priority. Therefore, only nodes 18-26 need to be considered when expanding the blue node.

The time required to find the shortest path, whether coarse or fine, can be reduced if we can limit the number of successor nodes. Every time a node is expanded up to 26 successor nodes need to be examined, but only a few of them will bring the agent closer to the goal. A method of filtering the successor nodes to only include those that are expected to have a lower cost would reduce computation time. This concept is used by Jump Point Search (JPS) [33] to speed up the path-finding process. A 3D example of how this approach would work is as follows. Consider the scenario shown in Figure 8.1. The red node has just been expanded and resulted in the blue node possessing the lowest priority. The blue node is then popped from the queue and expanded. All of the cubes shown in this figure represent the successors of the blue node. Normally, each cube shown in the figure is expanded. However, we can deduce that nodes 1-9 will not have the lowest priority after expansion. They are successors of the red node, so if one of them did have a lower priority, it would have been expanded instead of the blue node. The same logic can be applied to nodes 10-17, as they are also successors of the red node, and we can also omit them from expansion. Therefore, only nodes 18-26 need to be expanded, and we have reduced the number of successors from 26 to 9. JPS takes this a step further and says that, of the remaining nodes, only node 22 is the least expensive to get to via the blue node from the red node. The surrounding nodes are excluded from consideration because there are alternative paths to get there. Thus, we can jump from the red node directly to node 22. This technique has only been used on 2D grids with uniform cost, so it remains to be seen how effective a 3D implementation would be, and if it can be modified to support directional cost scale factors. Application of this technique may also be used to help prevent sharp turns by restricting the backwards and sideways nodes from being considered successors.

An alternate approach to hierarchical planning that is worth considering is Anytime Truncated D* (ATD*) [34]. An anytime algorithm quickly provides an initial suboptimal path, and iteratively improves the solutions as time permits. This is similar to the iterative refinement process used by HD*. Truncated D* limits the propagation of cost changes using a suboptimality bound, which speeds up replanning and guarantees a solution within the bound. These approaches are merged to develop ATD* in [34], which is shown to be superior to current real-time algorithms in unknown environments. It is tested on a robot using (x, y, heading) as the coordinate system, and HD* can be modified to operate in these coordinates as well to compare the two algorithms.

8.2 Conclusion

This thesis presents a path-finding algorithm for use in unknown environments that improves upon current real-time algorithms. A new hierarchical planning approach is used to allow for rapid replanning without first requiring map corrections. It implements directional cost factors, path smoothing, and spline generation to create realistic paths that are not limited to transitions between node centers. The optimality of the produced paths depends on the sensor range, refinement distance, and planning time restrictions, but paths are no more than 10% longer than the optimal path. Larger sensor ranges and time limits allow for a greater refinement distance to be used, which can significantly improve path quality. On a cubic map 300 units in each direction paths are produced in under 35 milliseconds, and path computation is faster on smaller maps.

Appendix A: HD* Pseudocode

The full pseudocode for HD* is presented below[1].

Algorithm 8 HD* Algorithm

```
1: class CreateLevel
2:     Class containing the following functions: Initialize(), CalcKey(),
           AddNode(), RemoveNode(), PopNode(), UpdateVertex(),
           ComputeCost(), Succ(), ComputeShortestPath()
3:     $U$, $entryfinder$, $k_m$, $g$, $rhs$, and $bptr$ are class variables
4: function SetupLevels( )
5:     Calculate number of levels and the successor distance at each level
6:     Define a variable $L$ for each level, where $L$ is an instance of the CreateLevel class
7:     return $L_1, L_2, ..., L_{nLevels}$
8: function FindPath(L)
9:     $d$ = EuclideanDistance($s_{start}, s_{goal}$)
10:    $path = [s_{start}, s_{goal}]$
11:    if $d < 28$ then            ▷ Distance too short to benefit from hierarchical planning
12:        $path = L_0$.ComputeShortestPath($path$)
13:        return $path$
14:    for $level_a = nLevels$ to 0 do
15:        if $d >= 7 \times L_{level_a}$.length then        ▷ Length is the successor distance
16:            $path = L_{level_a}$.ComputeShortestPath($path$)
17:            break
18:    while $t_{max}$ has not been exceeded do
19:        $path_{refined}$ = segment of path that lies within the refinement distance $d_r$
20:        for $level_b = level_a - 1$ to 0 do        ▷ Start refinement from next lowest level
21:            $path_{refined} = L_{level_b}$.ComputeShortestPath($path_{refined}$)
22:    Splice $path_{refined}$ into the beginning of $path$
23:    return $path$
```

[1]The complete Python implementation can be seen at `https://github.com/mds1/path-planning`.

```
24: function GenerateTrajectory($path$)
25:     $newpath = \emptyset$
26:     for $i = 0$ to length($path$) $- 1$ do
27:         $s = path_i$
28:         $s' = path_{i+1}$
29:         $d_x = s_x - s'_x;\ d_y = s_y - s'_y;\ d_z = s_z - s'_z$
30:         $d_{max} = \max(|d_x|,\ |d_y|,\ |d_z|)$
31:         if $d_{max} \leq 1$ then          ▷ Occurs when spline points are close together
32:             $newpath = newpath \cup \{s'\}$
33:         else
34:             for $j = 1$ to $d_{max}$ do
35:                 $f_x = d_x/d_{max};\ f_y = d_y/d_{max};\ f_z = d_z/d_{max}$
36:                 $u_x = s_x + j \times f_x$
37:                 $u_y = s_y + j \times f_y$
38:                 $u_z = s_z + j \times f_z$
39:                 $newpath = newpath \cup \{(u_x,\ u_y,\ u'_z)\}$
40:     return $newpath$
41: function Main(L)
42:     $L$ = SetupLevels()
43:     Scan environment for obstacles
44:     while $s_{start} \neq s_{goal}$ do
45:         $path$ = FindPath($L$)
46:         $path$ = SmoothPath($path$)
47:         $path$ = CatmullRomSpline($path$)
48:         $path$ = GenerateTrajectory($path$)
49:         $s_{last} = s_{start}$
50:         $dfs = 0$                    ▷ Tracks distance traveled since path was found
51:         while $s_{start} \neq s_{goal}$ and path is valid do
52:             Move to next point in $path$
53:             $s_{start}$ = current location
54:             Scan for new obstacles and determine if they block the current path
55:             if current path is blocked then
56:                 for all nodes $u$ with changed costs do
57:                     Update traversal cost of $u$
58:                 Path is no longer valid
59:             $dfs$ = EuclideanDistance($s_{start}, s_{last}$)
60:             if $dfs \geq d_r/2$ or goal has moved then
61:                 Path is no longer valid
```

Bibliography

[1] Alex Yahja, Anthony Stentz, Sanjiv Singh, , and Barry L. Brumitt. Framed-quadtree path planning for mobile robots operating in sparse environments. In *Proceedings, IEEE Conference on Robotics and Automation*, Leuven, Belgium, 1998.

[2] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1, 2004.

[3] Joseph Carsten, Dave Ferguson, and Anthony Stentz. 3d field d*: Improved path planning and replanning in three dimensions. In *International Conference on Intelligent Robots and Systems*, 2006.

[4] Sven Koenig and Maxim Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions On Robotics*, 21(3):354–363, 2002.

[5] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22, 1978.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 2, 1968.

[7] Richard E. Korf. Real-time heuristic search: First results. In *AAAI-87 Proceedings*. AAAI, 1987.

[8] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42, 1990.

[9] Cagatay Undeger and Faruk Polat. Real-time edge follow: A real-time path search approach. *IEEE Transactions on Systems, Man, and Cybernetics: Part C*, 37, 2007.

[10] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4, 1965.

[11] Amit Patel. Amit's game programming information. available online at `http://theory.stanford.edu/~amitp/GameProgramming/`, 2000.

[12] Robin R. Murphy. *Introduction to AI Robotics.* MIT Press, Cambridge, MA, USA, 1st edition, 2000.

[13] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39, 2010.

[14] Dave Ferguson and Anthony Stentz. The field d* algorithm for improved path planning and replanning in uniform and non-uniform cost environments. Cmu-tr-ri-05-19, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.

[15] Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *AAAI Conference on Artificial Intelligence*, 2010.

[16] Steve Rabin. A* aesthetic optimizations. In *Game Programming Gems.* Charles River Media, 2000.

[17] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5, 1986.

[18] Ding Fu-guang, Jiao Peng, Bian Xin-qian, and Wang Hong-jian. Auv local path planning based on virtual potential field. In *Proceedings of the IEEE International Conference on Mechatronics & Automation*, 2005.

[19] Thomas Hellström. Robot navigation with potential fields. Issn-0348-0542, December 2011.

[20] Ross A. Knepper. *On the Fundamental Relationships Among Path Planning Alternatives.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2011.

[21] Anthony Stentz. The focussed d* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995.

[22] Steven LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[23] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of IROS-2002*, Switzerland, 2002.

[24] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving target d* lite. *Proceedings of 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, 2010.

[25] Hanan Samet. An overview of quadtrees, octrees, and related hierarchical data structures. *NATO ASI Series*, 1988.

[26] Sven Koenig. A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 864–871, 2004.

[27] Elizabeth Murphy. *Planning and Exploring Under Uncertainty.* PhD thesis, University of Oxford, Somerville College, Oxford, England, 2010.

[28] Cem Yuksel, Scott Schaefer, and John Keyser. On the parameterization of catmull-rom curves. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 47–53, New York, NY, USA, 2009. ACM.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2009.

[30] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1, 1986.

[31] Judea Pearl Rina Dechter. Generalized best-first search strategies and the optimality of a*. *Journal of the ACM*, 32, 1985.

[32] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[33] Daniel Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[34] Sandip Aine and Maxim Likhachev. Truncated incremental search. *Artificial Intelligence*, 234, 2016.