Hierarchical D* Lite: A Real-Time 3D Path Planning Algorithm for Unknown Environments

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 paper 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 need to update the map representation 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 parameters such as map size, environment complexity, sensor range, and any restrictions on planning time. Monte Carlo simulations were used to assess performance, and it was found that HD* finds paths within 10% of optimal in under 35 ms for the most complex environments tested.

I. 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 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 rapid replanning is again

needed to ensure the vehicle does not stray too far from the optimal path.

Path planning in known, 2D terrain is a well-studied problem, and various solutions to find the shortest path exist. These include potential field methods [1, 2], visibility graphs [3], and heuristic based planners such as the widely used A* algorithm [4]. There are three significant problems with existing solutions that make them unsuitable for real-time 3D path planning. Heuristic-based algorithms are designed for off-line operation in known environments, so their primary function is to find the true shortest path, with minimal focus on reducing computation time. Additionally, these solutions 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 [3]. Lastly, these approaches are not designed to update paths to accommodate unexpected changes in terrain. When a path needs to be recalculated, previously used information cannot always be reused to speed up the current search.

Real-time path planning in unknown terrain has not been studied as extensively and fewer solutions exist. In this paper a new path planning algorithm is presented, called Hierarchical D* Lite (HD*), that is capable of computing new paths fast enough to operate in real-time. HD* merges D* Lite [5] and Hierarchical Path-Finding A* [6], and introduces changes to improve performance and path quality. Computation times are similar whether used for unknown environments, chasing moving targets, or a combination of the two. The presented algorithm sacrifices optimality for performance, but still finds near-optimal paths within 7 to 35 ms, which is certainly sufficient for real-time performance. In Section II, a brief review of existing literature is presented. Section III provides background information for the algorithms implemented into HD*. Section III A discusses modifications to these algorithms and other specific implementation details of HD*, and Section IV presents performance results of HD*.

II. Background

There has been significant research in path planning in recent years. A few existing methods of path planning were briefly mentioned in Section I, but there are additional relevant approaches worth discussing. Although the path planning method presented in this paper is for a single UAV

in an unknown environment, a wide variety of approaches are discussed below for completeness.

A. Potential Field Methods

Potential field methods haven been used for real-time planning and obstacle avoidance in a wide range of applications, from robotics to spacecraft [1, 2, 7, 8]. By simulating an attractive force around the goal location and repulsive forces around obstacles, the vehicle can follow the field to the goal. However, potential field methods have a tendency to generate local minima that trap the vehicle and prevent it from reaching the goal. The local minima may be removed by the injection of random noise, but this would result in jagged paths that may be infeasible to follow. 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 [1, 9].

B. Sampling-Based Methods

Rapidly Exploring Random Trees (RRTs) [10–14], Probabilistic Roadmaps (PRMs) [14, 15], and Expansive Space Planners (ESPs) [14, 16] all use a probabilistic approach to connect the initial and goal locations. These methods each begin by sampling a random state. RRTs create a tree at the start and goal nodes, and attempt to connect the newly sampled state to the closest state on one of the trees. This causes the trees to expand towards each other, and once the trees are connected, a path between the start and goal nodes has been found. PRMs connect the random sample to all neighboring states, forming a roadmap which can be searched to find the optimal path between the start and goal nodes. ESPs are similar to PRMs, but with a bias towards unexplored regions and a bidirectional search method. These methods can account for vehicle dynamics and be used in real-time scenarios. Their downside lies in the fact that they have no guarantee of optimality, especially when the environment becomes complex and computation times may be prohibitively long [9, 11, 12, 17].

C. Heuristic Methods

Heuristic-based incremental planners, such as D* Lite [5], expand the heuristic-based A* [4] algorithm to reuse previous information, resulting in reduced computation time for the current search. This category of planners attempts to combine the performance of real-time algorithms with the solution quality of heuristic-based algorithms. While this is a promising combination, their performance is not suitable for on-line operation; the performance of such a planner is sufficient for some scenarios [18], but it does not perform well for large 3D maps, where an exponential increase in grid size significantly slows down the search, as does computing the entire path to the goal each time the environment or target changes. This process can be performed significantly faster by initially planning a coarse path and later refining it, as demonstrated by Botea and Müller [6]. This algorithm, known as Hierarchical Path-Finding A* (HPA*), is designed for a 2D implementation, and would need to be expanded to 3D for real-time UAV path planning.

D. Real-Time Methods

Path planning applications typically do not require the entire path to be known immediately. As a result, heuristic approaches can be modified to obtain real-time performance by restricting the lookahead distance of each search. This family of heuristic-based on-line planners, such Real-Time A* (RTA*), Learning RTA* [19], and the Real-Time Edge Follow (RTEF) methods [18] find a prefix of the path and begin to move along it, instead of finding the complete path. This allows the vehicle to begin moving within a constant amount of time regardless of map size and environmental complexity. These are suitable for unknown environments, and some variations are applicable to moving targets. Because the complete path is not found, the prefix being followed often leads to a very suboptimal path [18, 19], and these approaches are unable to account for vehicle dynamics.

E. Optimal Control Theory Methods

The use of pseudospectral (PS) methods [20–23] for optimal control problems has recently gained prominence, and real-time path planning falls under this domain. Thus, PS approaches can be used for real-time planning in obstacle-dense environments. As an optimal control method, this approach makes it easy to incorporate any necessary constraints such as vehicle dynamics or

path requirements. Moreover, these methods are applicable to a variety of different vehicles and problems, making them an attractive approach.

In [20], Gong, Lewis, and Ross successfully simulated this approach for a UAV in an urban environment. The computation time was about 15 seconds, which may be too slow for some applications. It is stated that the best-case time for the PS approach could be about 33 ms, which is comparable to the worst-case time of HD*, although the PS method does have the benefit of handing a variety of constraints.

F. Curve-Based Methods

Trajectory generation and obstacle avoidance through the use of Bézier Curves [24, 25] or circular paths [26] can be quite useful, as they are guaranteed to meet vehicle constraints and ensure evasion. They have the added benefit of supporting situations involving sets of cooperative vehicles and maintaining temporal separation. One downside to such an approach is that the use of collision avoidance maneuvers, as opposed to replanning the entire path, may result in highly suboptimal paths in the presence of numerous obstacles. This occurs because the agent attempts to continue following the existing path, even though it may no longer be near-optimal due to these obstacles. It is also worth noting that without replanning the full path, these algorithms cannot be used to chase moving targets.

The B-spline path template method presented by Jung and Tsiotras in [27] is similar to that used by HD*. Both are based on the D* Lite algorithm, utilize a form of hierarchical planing, and use splines to smooth out paths. Although the B-spline approach has the advantage of accounting for kinematic constraints, HD* provides two improvements upon this method. First, the approach presented in [27] is for generating 2D paths, and does not discuss altitude changes as part of the planning process. On the contrary, HD* was designed for computing paths in 3D. The second is that HD* takes a different approach to hierarchical planning, leading to both improved path quality and reduced computation times. The HD* approach to hierarchical planning is further discussed in Section III A 1.

III. Hierarchical D* Lite

This paper presents an algorithm called HD*, which combines D* Lite with a modified version of HPA*. D* Lite allows a full path to be replanned quicker than the A* algorithm it is based on. However, D* Lite is still not sufficient for real-time use in a 3D environment, so HD* builds on the approach of HPA* by utilizing hierarchical planning to reduce computation time. HD* uses a novel hierarchical approach which is suitable for any environment, accounts for detected obstacles, and provides the user with flexibility on the tradeoff between path cost and computation time. This results in a heuristic-based algorithm designed for real-time use with unknown environments or moving targets. Section IV shows that compared to an existing 3D version of D* Lite, known as 3D Field D* [28], HD* reduces the number of nodes searched by an order of magnitude. This significantly improves computation speed, but comes with a worst-case increase in path cost of less than 10%.

Before continuing with this section, it is recommended that the reader be familiar with A*, D* Lite, and HPA*, which are the three algorithms that compose the foundation of HD*. References for these algorithms can be found in [4], [5], and [6], respectively. Note that throughout this paper it is assumed that all traversal costs are one for open nodes and infinity for blocked nodes, and that nodes of unknown status are assumed to be open.

A. HD* Implementation

The operation of HD* can now be detailed. The pseudocode can be found in III B, and the complete Python implementation can be found at `https://github.com/mds1/path-planning`.

A simple 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 now explained and resolved. The first subsection that follows details the problems with HPA* and the remedies used by HD*. The following four subsections, III A 2 – III A 5, cover methods implemented to improve the path quality compared to a simple merge of D*

Lite with HD*. The final four subsections, III A 6 – III A 9, cover methods implemented to further improve performance.

1. Hierarchical Modifications

As detailed in [6], HPA* defines transition nodes between adjacent clusters which are then used to run A* and find a coarse path. However, this approach is not ideal for 3D maps or unknown environments. For 3D maps, the extra dimension results in many more node pairs that must be checked to define entrances between clusters, increasing the time needed to complete this step. 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 [31], 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 cover a few nodes this would be an acceptable approach. The level 0 planner would simply route the path around the blocked nodes. When the domain includes large obstacles, such as buildings, this becomes problematic. It is possible that the coarse path may pass through a large obstacle, which can result if the coarse path contains a transition node on either side of the blocked region. Planning a high resolution path between these two nodes may result in a very suboptimal path. Computation time increases with path length, so even if longer, suboptimal paths are acceptable, the high resolution planning may take longer than desired.

Instead of needing to continually redefine entrances, HD* improves the replanning process by

eliminating cluster borders. Instead, it 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 in Fig. 1, which shows two levels of clusters. The most coarse clusters are 6×6 (represented by triangles), and the finer clusters are 3×3 (represented by circles). 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 of the nodes forming a square a distance of 6 nodes away, given by the nodes containing triangles. Upon refinement, the 3×3 level is used, and the successors for this level are the nodes with circles.

Successor nodes can 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. This has the added benefit of requiring less memory, as there is no 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 is solved in the following way. Recall that D* Lite plans from goal to start, and during the coarse planning stage, it is unlikely that the start node becomes a hierarchical successor. To resolve this, each time successor nodes are computed HD* checks 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, which also helps reduce the number of nodes expanded. An example of this is shown in Fig. 2, which uses 3×3 clusters. The triangles represent the nodes available for path

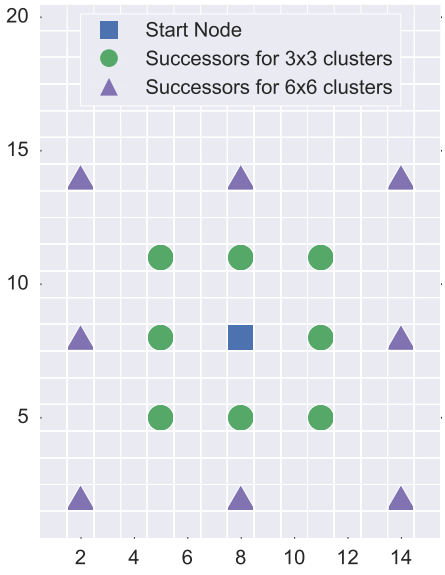


Fig. 1 Improved method for determining hierarchical successors in 2D environments.

planning using the HD* successor definition, and the black squares represent obstacles. The path between $s_{current}$ (the circle) and s_{start} (the diamond) will not pass through node s_1 (the triangle in the bottom row) as there is not line-of-sight between the centers of s_1 and s_{start}. Therefore, the path would normally have to pass through s_2 (the top-middle triangle). Instead, since s_{start} is nearby and has line-of-sight with $s_{current}$, the coarse planner is allowed to jump directly between the two.

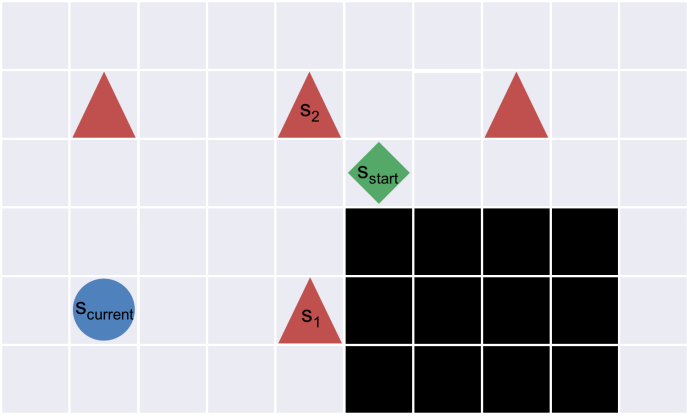


Fig. 2 Example of a situation where HD* can jump directly from $s_{current}$ to s_{start}.

This new approach makes it necessary to consider obstacles when computing coarse paths, which increases computation time but helps produce improved routes. Without this consideration, successor nodes may be within or blocked by obstacles, which can result in low quality or impossible coarse paths. This can increase planning time as these invalid nodes must be skipped, and thus longer low-level paths must be planned. To counter this, HD* first checks line-of-sight using a 3D version of Bresenham's Line Algorithm [32]. 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 time consuming, especially as the distance between the nodes increases at coarser levels. To resolve this, the checks are only performed when at least one of the two nodes being tested for line-of-sight

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.

2. Abstract Levels

HD* abstracts the map into multiple levels, where level 0 is defined as the initial, highest resolution map representation. This provides more flexibility when planning coarse paths, and results in improved path quality during the refinement and smoothing stages, which are discussed in Section III A 3.

Due to the method used to determine successors, abstract levels are 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 n is given by 2^{n+1}, for $n = 1, 2, ..., n_{max}$. 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 $^1/_8$ the maximum map dimension. This cutoff was used to help ensure good quality paths are found, as a grid that is too coarse is more likely to find 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.

3. 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 result in a very conservative path that passes 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, less conservative, and more realistic paths. Once this path is represented as level 0 nodes, it can be smoothed.

The smoothing algorithm is a modified version of the one presented in [29], 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 reached by the agent.

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.

The next step involves generating a centripetal Catmull-Rom spline [33] to smooth any sharp turns from the path. This type of spline is used as it is guaranteed to pass through the specified nodes [34], and will not result in any self-intersections or cusps [33]. 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, 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 resulting spline is represented by three additional points between the second and third input points. Generation of these splines causes an increase in path length but is necessary to create a path that can realistically be followed. Modification of spline generation, paired with a modification of the cost computation discussed in Section III A 4, would allow HD* to account for vehicle dynamics. Here, the parameterization of the spline could be configured to prohibit generation of splines with a radius smaller than the vehicle's minimum

turning radius.

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 III A 8.

It is important to note that although smoothing results in shorter paths, it does not always result in the true shortest path. The path found with a grid representation may be optimal for traveling between node centers, but node centers do not always compose the true shortest path [35].

4. Cost Computation

Although HD* is not designed to account for the UAV's dynamics, it should be capable of 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 straight up, 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 are 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.

Depending on the environment, increasing c_z can result in the planner preferring strictly vertical

movement. Many UAVs, particularly fixed wing aircraft, are unable to change their altitude in this manner. 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. Instead, the only way to change altitude is through vertical diagonal movement. To implement these changes, the way D* Lite computes cost is modified for HD*. The cost between two nodes is now computed with the function ComputeCost(s, s'), which accounts for these scale factors. This cost computation function is the same when computing paths at any level, with the addition of an initial line-of-sight check for the coarse paths.

As discussed in Section III A 3, modification of this cost computation step is the second change necessary to expand HD* to account for vehicle dynamics. The updated cost computation would involve developing a more robust and comprehensive function to map the vehicle's state and constraints into this cost calculation, and should assign a cost of infinity to impossible maneuvers. This is effectively an extension of the concept already implemented into the algorithm, where, for example, the cost of purely vertical traversal can be set to infinity.

5. 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. This is implemented by extending the footprint of obstacles to include the surrounding nodes within the safety margin.

6. Data Structures

When finding a path, a large percentage of time is spent managing the priority queue, so the data structure chosen to manage the queue is important. Analysis of the D* Lite algorithm reveals that popping the best node from U occurs once for each node visited, while node insertion and membership checks occur up to 26 times for each popped node. Updating the priority of a node

is the least common, which is fortunate since this is an expensive operation. A binary heap is a common choice, as it provides $O(\log n)$ performance for the first three operations [36]. HD* uses a binary heap, but supplements it with a hash table to give $O(1)$ performance for membership checks.

7. Heuristic Choice

An admissible heuristic is required to find the shortest path between the start and goal nodes. HD* uses the Euclidean distance between two points, which represents the smallest possible cost between two nodes and is both an admissible and consistent heuristic. A heuristic which underestimates the true distance or knows it exactly is guaranteed to find the shortest path, and the more it underestimates the distance the more nodes it will search [4]. This is evident in the limiting case when the heuristic is zero, and A* reduces into Dijkstra's algorithm [37], which expands every node until it finds the shortest path.

Alternatively, 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 [4]. Since the ability to find a true shortest path was already sacrificed by implementing hierarchical planning, a slightly non-admissible heuristic may be safely used to help reduce computation time. Inflating the heuristic also helps speed up the search 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 [38]. It has been proven in [39] 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 1, where L_{opt} represents the optimal path length and L_{mod} represents the path found when the heuristic is modified

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

HD* uses a default value of $\epsilon = 0.01$, and the tradeoff between ϵ, path cost, and performance is explored in Section IV.

8. Path-Finding Process

HD* computes the entire coarse path from goal to start, but only a portion needs to be refined to get the UAV moving, and the remainder can be refined as needed. Furthermore, when traveling in an unknown environment the planned coarse path may soon become invalid. Thus, refining the entire path up front may be wasteful. Instead, refinement is only performed on 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, as a lot of time may be spent refining the path. Alternatively, this choice of d_r may be too conservative for situations when the agent has a small search radius. To counter these situations and provide flexibility, HD* allows the user to specify the value of d_r, providing control over the trade-off between path quality and computation time. This trade-off is explored in Section IV.

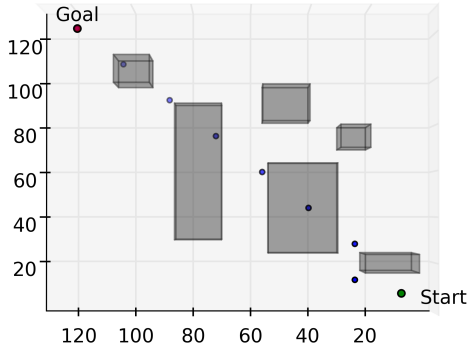
Once the lowest resolution path is computed, the portion within d_r is refined. The segment is continually refined by the process presented in Section III A 3 until represented in the highest resolution nodes. Next, the entire path is smoothed. Once complete, the path is represented by many closely spaced 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 Fig. 3 from the top down view of a $128 \times 128 \times 128$ 3D map, where level 3 is the most coarse level with nodes 16 units apart. Level 2 nodes are 8 units apart, and level 1 nodes are 4 units apart. Circles represent the nodes composing the path. The agent follows the resulting trajectory until the goal is reached, or one of three situations occur:

1. new obstacles invalidate the current path;

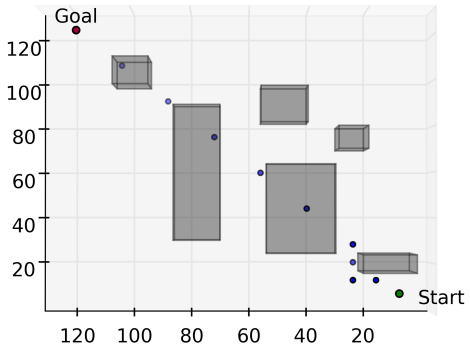
2. the target location has moved;

3. the UAV is halfway through refinement region.

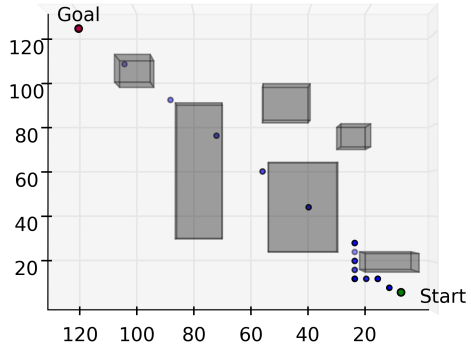
When scenarios 1 or 2 occur, a new coarse path is planned and the process restarts. Scenario 3 occurs when neither of the two preceding scenarios forces a replan before the midway point of the refinement region is reached. Scheduling replans halfway through the refinement region helps ensure



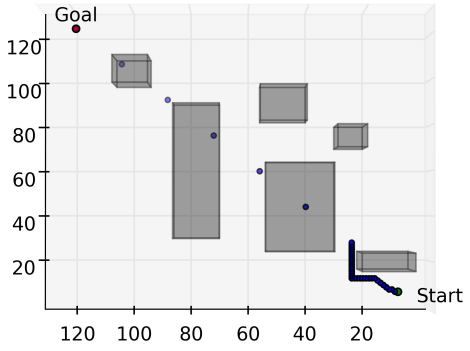
(a) The lowest resolution path is found with level 3. Some nodes pass through obstacles, as the UAV is unaware of those obstacles.



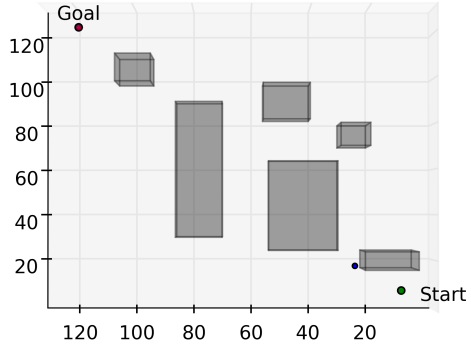
(b) Next, level 2 nodes are used to refine the segment of the path within the refinement distance.



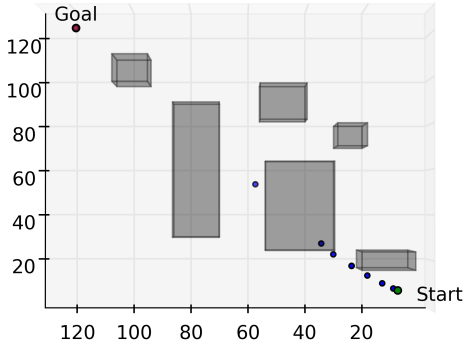
(c) Level 1 nodes are used to further refine the path.



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



(e) The path is then smoothed, which leaves only three nodes to define the path.



(f) Finally, additional points are generated with a centripetal Catmull-Rom spline. These points are used to create the trajectory.

Fig. 3 Path Finding and Refinement Process for HD*

high-quality paths.

9. 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 implementation is not a hard restriction in the sense that the total path planning time will nearly always exceed t_{max}, but it does allow for some flexibility in limiting the time spent planning.

However, with a few extra steps, this approach can be easily modified to effectively act as a hard limit tailored to a specific vehicle. First, hardware-in-the-loop simulations using a map representative of the expected environment will allow for more accurate path-finding time statistics to be determined. Next, a function can be developed that maps the vehicle's state to the max allowable path computation time. With these two pieces, and the knowledge that the next level of refinement will always be computed if the elapsed time is less than t_{max}, adjust t_{max} accordingly. For example, consider a case where the simulated path finding times are found be normally distributed with a standard deviation of x ms, and the maximum allowable path-finding time is y ms (although this is more likely to be based on velocity). Now, setting $t_{max} = y - 2x$ ensures the path-finding time will be less than the hard limit of y ms 95.4% of the time, and setting $t_{max} = y - 3x$ increases this to 99.7% of the time. Additionally, the decision of whether or not to compute another iteration can be modified from a simple $t_{elapsed} \leq t_{max}$ check, to something that accounts for these standard deviations as well, such as $t_{elapsed} + 3x \leq t_{max}$.

B. HD* Pseudocode

This section provides the relevant pseudocode for HD*.

Algorithm 1 HD* Algorithm

class CreateLevel
    Class containing the following functions: Initialize(), CalcKey(), AddNode(), RemoveNode(), PopNode(), UpdateVertex(), ComputeCost(), Succ(), ComputeShortestPath()
    U, $entryfinder$, k_m, g, rhs, and $bptr$ are class variables

function SetupLevels( )
    Calculate number of levels and the successor distance at each level
    Define a variable L for each level, where L is an instance of the CreateLevel class
    return $L_1, L_2, ..., L_{nLevels}$

function FindPath(L)
    d = EuclideanDistance(s_{start}, s_{goal})
    $path = [s_{start}, s_{goal}]$
    if $d < 28$ **then** ▷ Distance too short to benefit from hierarchical planning
        $path = L_0$.ComputeShortestPath($path$)
        return $path$
    for $level_a = nLevels$ **to** 0 **do**
        if $d >= 7 \times L_{level_a}$.length **then** ▷ Length is the successor distance
            $path = L_{level_a}$.ComputeShortestPath($path$)
            break
    while t_{max} has not been exceeded **do**
        $path_{refined}$ = segment of path that lies within the refinement distance d_r
        for $level_b = level_a - 1$ **to** 0 **do** ▷ Start refinement from next lowest level
            $path_{refined} = L_{level_b}$.ComputeShortestPath($path_{refined}$)
    Splice $path_{refined}$ into the beginning of $path$
    return $path$

```
function GenerateTrajectory(path)
    newpath = ∅
    for i = 0 to length(path) − 1 do
        s = path_i
        s′ = path_{i+1}
        d_x = s_x − s′_x;  d_y = s_y − s′_y;  d_z = s_z − s′_z
        d_max = max(|d_x|, |d_y|, |d_z|)
        if d_max ≤ 1 then                          ▷ Occurs when spline points are close together
            newpath = newpath ∪ {s′}
        else
            for j = 1 to d_max do
                f_x = d_x/d_max;  f_y = d_y/d_max;  f_z = d_z/d_max
                u_x = s_x + j × f_x
                u_y = s_y + j × f_y
                u_z = s_z + j × f_z
                newpath = newpath ∪ {(u_x, u_y, u′_z)}
    return newpath

function Main(L)
    L = SetupLevels()
    Scan environment for obstacles
    while s_start ≠ s_goal do
        path = FindPath(L)
        path = SmoothPath(path)
        path = CatmullRomSpline(path)
        path = GenerateTrajectory(path)
        s_last = s_start
        dfs = 0                                    ▷ Tracks distance traveled since path was found
        while s_start ≠ s_goal  and path is valid do
            Move to next point in path
            s_start = current location
            Scan for new obstacles and determine if they block the current path
            if current path is blocked then
                for all nodes u with changed costs do
                    Update traversal cost of u
                Path is no longer valid
            dfs = EuclideanDistance(s_start, s_last)
            if dfs ≥ d_r/2  or goal has moved then
                Path is no longer valid
```

Algorithm 2 Path Smoothing Algorithm

This version assumes $c_x = c_y = 1$

function SmoothPath($[s_0, s_1, ..., s_n]$) ▷ Nodes are (x, y, z) coordinate tuples
  $k = 0$
  $p_k = s_0$
  for $i = 1$ **to** $n - 1$ **do**
    $x_1 = p_{k,x}$; $y_1 = p_{k,y}$; $z_1 = p_{k,z}$
    $x_2 = s_{i+1,x}$; $y_2 = s_{i+1,y}$; $z_2 = s_{i+1,z}$
    if ($z_1 \neq z_2$ **and** $c_z \neq 1$) **or not** LineOfSight(p_k, s_{i+1}) **then**
      $k += 1$
      $p_k = s_i$
  $k += 1$
  $p_k = s_n$
  $path = [p_0, p_1, ...p_k]$
  return $path$

Algorithm 3 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.

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

Algorithm 4 Cost Computation

This version assumes $c_x = c_y = 1$

function EUCLIDEANDISTANCE(s, s')
    return $\sqrt{(s_x - s'_x)^2 + (s_y - s'_y)^2 + (s_z - s'_z)^2}$
function COMPUTECOST(s, s')
    if s' contains an obstacle **then**
        return ∞
    else if computing cost for the coarse path **then**
        if within search radius **and not** LineOfSight(s, s') **then**
            return ∞
    if $s_z \neq s'_z$ **then**
        $costFactor = c_z$
    else
        $costFactor = 1$ ▷ Default planar cost
    $cost = costFactor \times$ EuclideanDistance(s, s')
    return *cost*

Algorithm 5 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

$entryfinder = \emptyset$ ▷ Hash table mapping tasks in U to entries
function REMOVENODE(u)
    Delete entry in $entryfinder$ with key u
function ADDNODE(k_1, k_2, u) ▷ Add node or update key of existing node
    if $u \in entryfinder$ **then**
        RemoveNode(u)
    Add key u with value $[k_1, k_2, u]$ to $entryfinder$
    Add entry $[k_1, k_2, u]$ to U
function POPNODE( )
    while True **do**
        $k_1,\ k_2,\ u = U$.Pop()
        if $u \in entryfinder$ **then**
            Delete entry in $entryfinder$ with key u
            return $[k_1, k_2, u]$

IV. Simulation Results

The performance of HD* was analyzed in a variety of scenarios. 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* is quantified; then it is compared to 3D Field D* [28], which

is another real-time planning algorithm. Next, various parameters are modified to determine how performance can vary in different scenarios. These parameters and their default values are listed in Table 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.

Table 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
Safety Margin	None

A. Quantifying Degree of Suboptimality

To measure the variability in cost, the path cost found by HD* with an empty initial map is compared to that found by A* with full knowledge of the environment. A* uses an admissible heuristic to ensure the optimal path is 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 2.

Table 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

With fewer obstacles, 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

Table 3 HD* Path Cost vs. Optimal Path Cost

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

of obstacles. Section IV C discusses some changes can be made to the default HD* implementation if shorter path lengths are desired.

B. Comparison to 3D Field D*

In [28], 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 demonstrates whether HD* improves upon similar existing algorithms. To compare HD* with 3DF, the experimental setup used in [28] was replicated. (N.B. 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 3. Note that HD* did not use splines for the tests with 50% obstacle density, as the splines sometimes pass through obstacles in denser environments. Path length is not presented as it is not reported in [28]. It is found that HD* computes paths faster than 3DF, and the time savings increase as obstacle density ρ increases. HD* is 12% faster with $\rho = 20\%$, and 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.

C. HD* Performance Analysis

Next, the performance of HD* is analyzed for a wide range of possible configurations. 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 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.

It is worth mentioning the performance of recent state of the art planners as a reference point for this section. The computation times for two different types of planners presented in [40] are given. The best-case time for the Rapidly Exploring Random Tree planner was approximately 28 ms, and for the Probabilistic Roadmap planner the best-case time was approximately 56 ms. Note that these planners have the advantage of also accounting for kinodynamic constraints, whereas HD* does not.

Fig. 4 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 is expected, as it can be sees 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 significant 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 implies the goal can be 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,

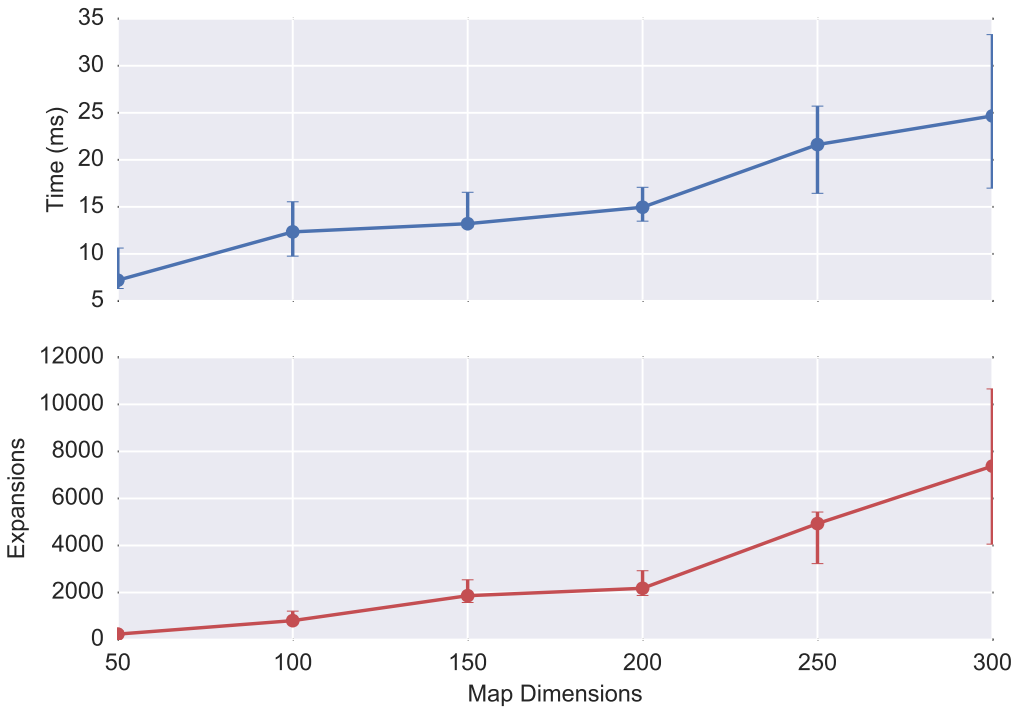


Fig. 4 Effect of Map Dimensions on Performance

leading to routes that are more likely to be suboptimal and therefore require frequent replanning and more node expansions.

The effect of obstacle density can be seen in Fig. 5. 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 the heuristic 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 [4]. This relationship is explored in Fig. 6, 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 its value increases. The planning

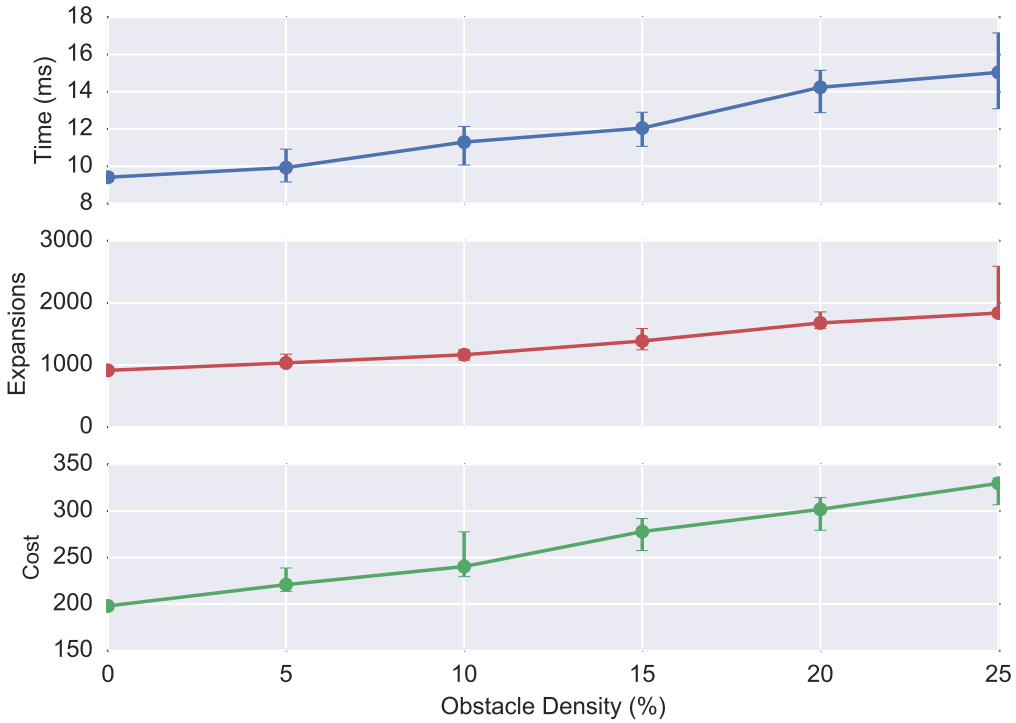


Fig. 5 Effect of Obstacle Density on Performance

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 might be due to the random maps, as it was proved in [39] that an increase in ϵ does result in longer paths. Therefore, it may be concluded 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 Fig. 7. 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.

The impact of r_s and d_r can be seen in Fig. 8. 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

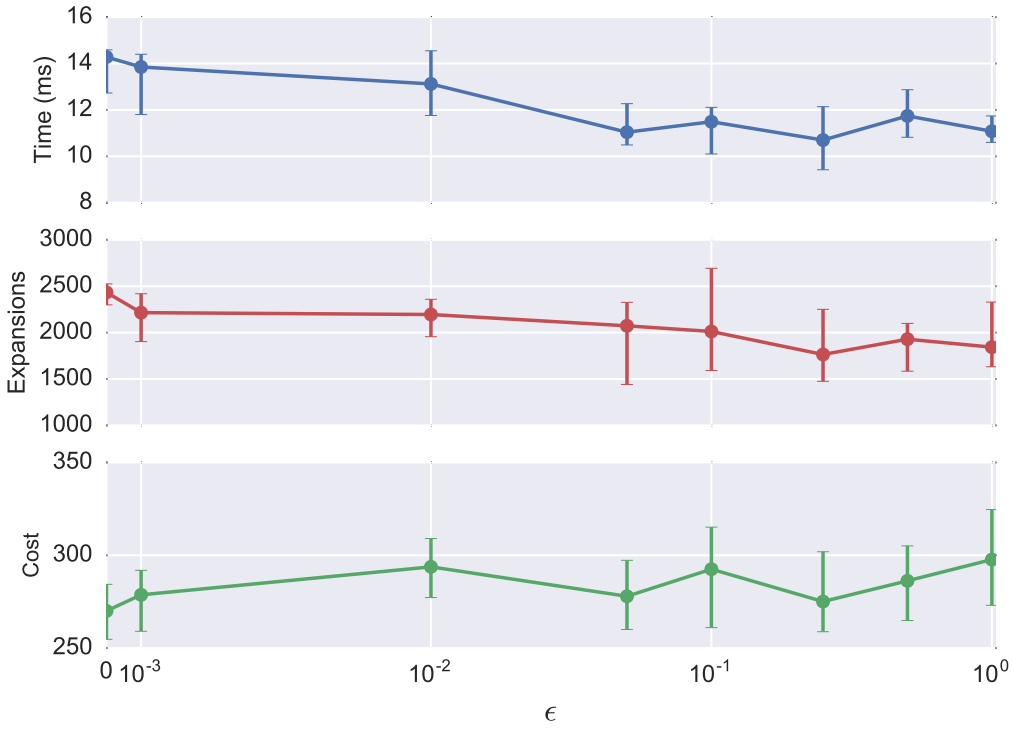


Fig. 6 Effect of heuristic scale on performance with randomly generated maps.

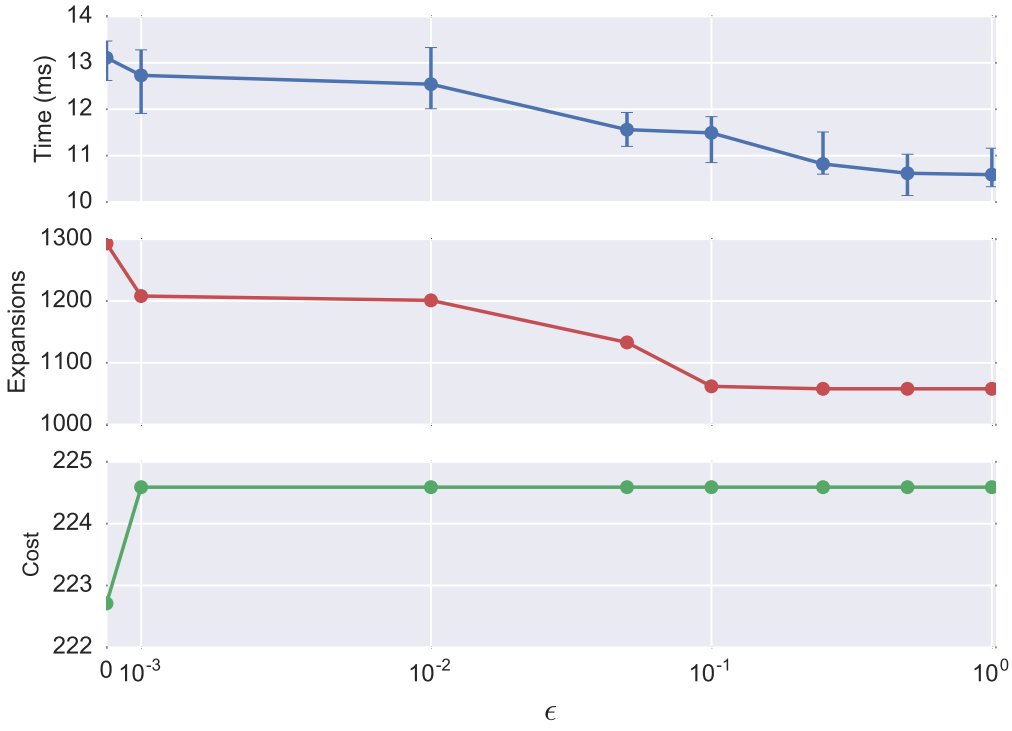


Fig. 7 Effect of heuristic scale on performance with each test performed on the same map.

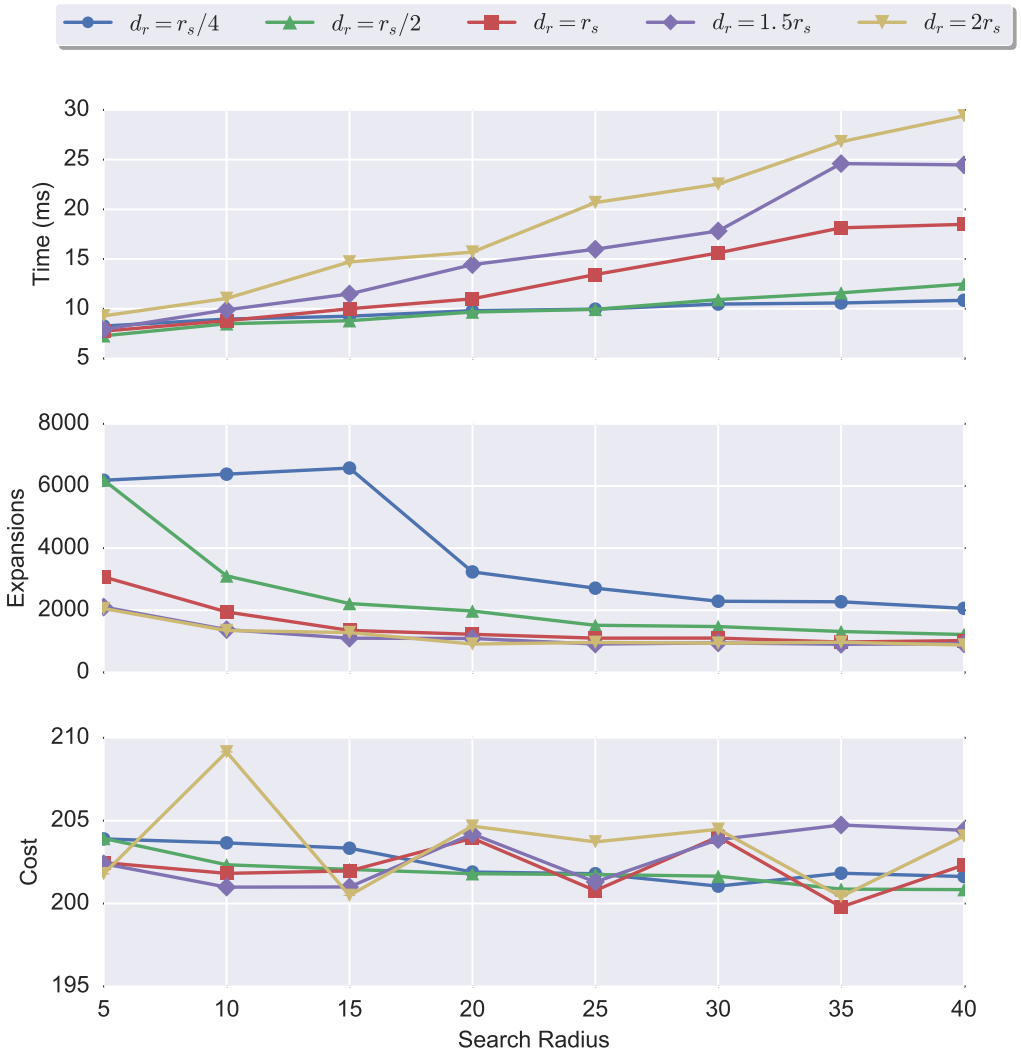


Fig. 8 Effect of Search Radius and Refinement Distance on Performance

maintain consistency between trials, splines were not used for any of the tests performed to generate Fig. 8.

There are a few insights gained from this figure. In the previous tests, the number of node expansions was directly correlated with computation time, but the opposite is true here. It is known 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, Fig. 8 leads to the conclusion that the refinement process dominates node expansions in regards to impact on planning time.

Intuitively, it would be expected that larger values of r_s and d_r would result in shorter paths,

Table 4 Performance of Paths Shown in Figures 9 and 10

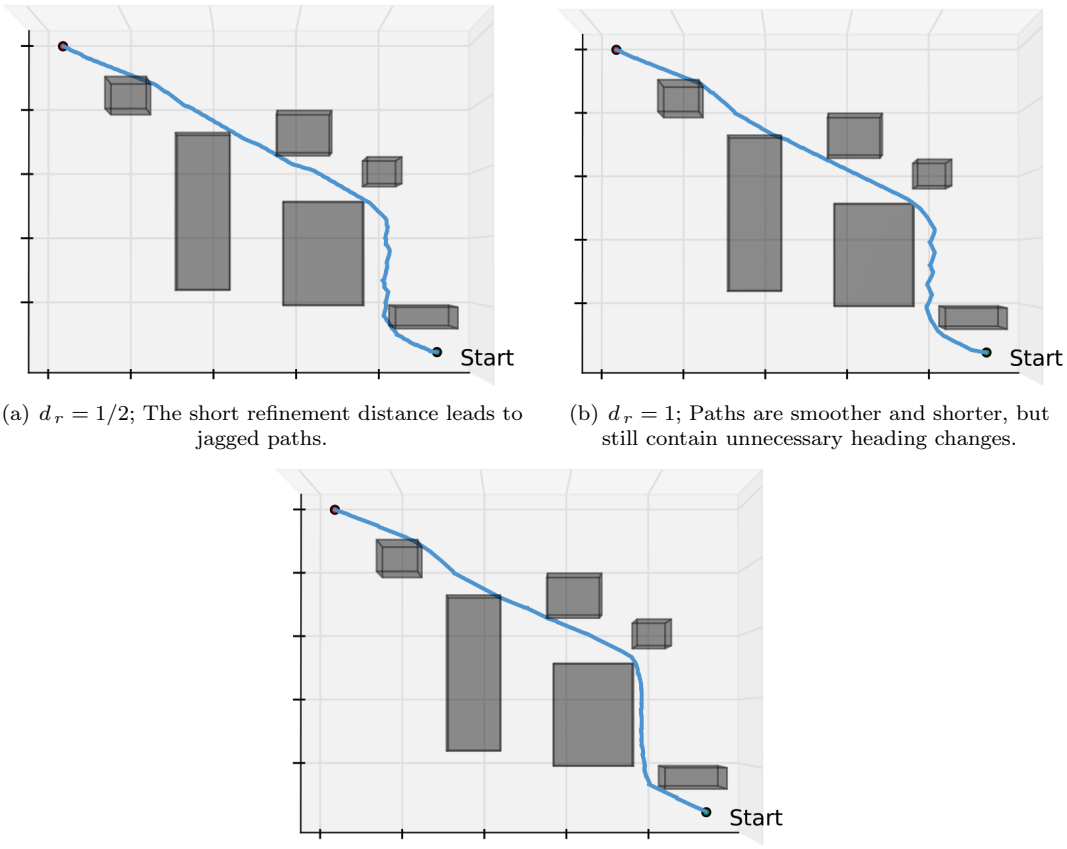
Figure Number	r_s	d_r	Expansions	Time (ms)	Cost
9(a)	20	$^1/_2$	6049	14.20	443
9(b), 10(b)	20	1	3684	16.40	429
9(c)	20	2	4092	22.47	399
10(a)	10	1	6434	13.32	423
10(c)	30	1	3698	19.71	408

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 below by comparing results on a map designed to represent a portion of a city. A $256 \times 256 \times 256$ environment designed to be representative of a city is used. The examples presented show a top-down view of the map and use $c_z = 2$ and therefore the paths do not change elevation. Elevation changes in the paths would be observed for $c_z = 1$, but for clarity of exposition paths of constant altitude are shown.

Fig. 9 compares the resulting paths when varying r_s and Fig. 10 shows the impact of varying d_r. Table 4 provides the performance data for these maps. From the figures the correlation between both r_s and d_r with path length becomes clear. Figures 9(c) and 10(c) represent the shortest and most realistic paths, suggesting that a larger search radius and refinement distance are key to producing quality paths.

V. Conclusion

This paper 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. In the best-case scenario, HD* can find paths within 7 milliseconds that are within 1% of optimal, and in more complicated environments paths are found in under 35 milliseconds and are within 10% of optimal. Larger sensor ranges and time limits allow



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

(b) $d_r = 1$; Paths are 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.

Fig. 9 Paths produced when $r_s = 20$ for varying values of d_r

for a greater refinement distance to be used, which can significantly improve path quality, but this comes at the cost of increased computation time.

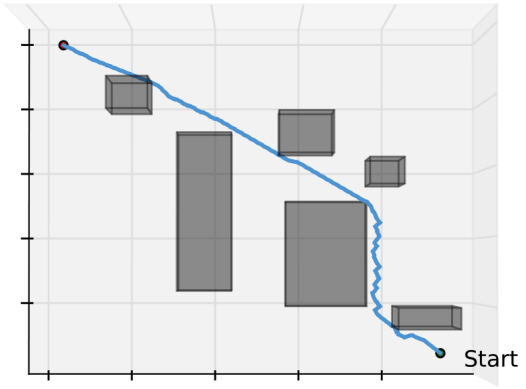
References

[1] Khatib, O., "Real-time obstacle avoidance for manipulators and mobile robots," *International Journal of Robotics Research*, Vol. 5, 1986.

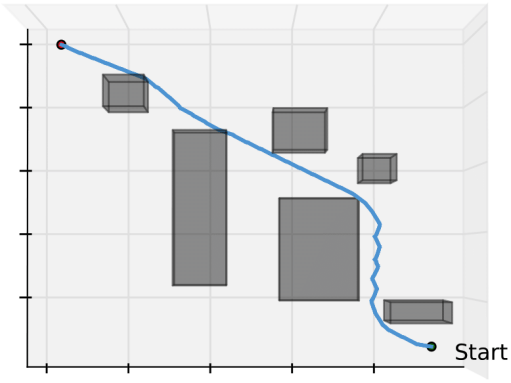
[2] Pimenta, L. C. A., Fonseca, A. R., Pereira, G. A. S., Mesquita, R. C., Silva, E. J., Caminhas, W. M., and Campos, M. F. M., "Robot Navigation Based on Electrostatic Field Computation," *IEEE Transactions on Magnetics*, Vol. 42, 2006.

[3] Lozano-Pérez, T. and Wesley, M. A., "An algorithm for planning collision-free paths among polyhedral obstacles," *Communications of the ACM*, Vol. 22, 1978.

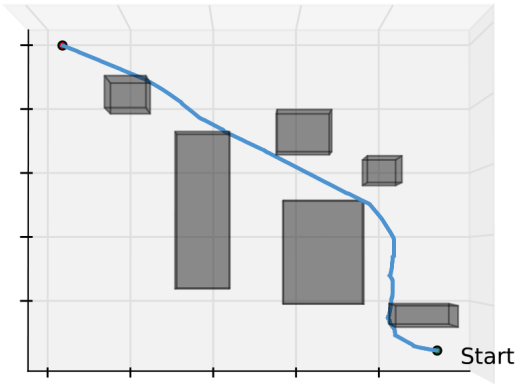
[4] Hart, P. E., Nilsson, N. J., and Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, Vol. 2, 1968.



(a) $r_s = 10$; A small sensor range results in a jagged traverse, as paths are suboptimal and frequently replanned.



(b) $r_s = 20$; The increased search radius produces more informed paths with less replanning, but still contains frequent heading changes.



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

Fig. 10 Paths produced when $d_r = 1$ for varying values of r_s

[5] Koenig, S. and Likhachev, M., "Fast Replanning for Navigation in Unknown Terrain," *IEEE Transactions On Robotics*, Vol. 21, No. 3, 2002, pp. 354–363.

[6] Botea, A., Müller, M., and Schaeffer, J., "Near Optimal Hierarchical Path-Finding," *Journal of Game Development*, Vol. 1, 2004.

[7] Lopez, I. and Mclnnes1, C. R., "Autonomous Rendezvous Using Artificial Potential Function Guidance," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 18, 1995.

[8] Spencer, D. A., "Automated Trajectory Control Using Artificial Potential Functions to Target Relative Orbits," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 39, 1995.

[9] Knepper, R. A., *On the Fundamental Relationships Among Path Planning Alternatives*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2011.

[10] Boeuf, A., Cortés, J., Alami, R., and Siméon, T., "Enhancing Sampling-Based Kinodynamic Motion Planning for Quadrotors," *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems,*

IROS 2015.

[11] Luders, B., Ellertson, A., and How, J. P., "Wind Uncertainty Modeling and Robust Trajectory Planning for Autonomous Parafoils," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 39, 2016.

[12] Farinella, J., Lay, C., and Bhandari, S., "UAV Collision Avoidance using a Predictive Rapidly-Exploring Random Tree (RRT)," *AIAA Infotech @ Aerospace*, San Diego, California, 2016.

[13] Levine, D., Luders, B., and How, J. P., "Information-Rich Path Planning with General Constraints using Rapidly-exploring Random Trees," *AIAA Infotech@Aerospace 2010*, Atlanta, Georgia, 2010.

[14] LaValle, S., *Planning Algorithms*, Cambridge University Press, 2006.

[15] Kavralu, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H., "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," *IEEE Transactions on Robotics and Automation*, Vol. 12, 1996.

[16] Hsu, D., Kindel, R., Latombe, J.-C., and Rock, S., "Randomized Kinodynamic Motion Planning with Moving Obstacles," *The International Journal of Robotics Research*, Vol. 21, 2002.

[17] Bruce, J. and Veloso, M., "Real-Time Randomized Path Planning for Robot Navigation," *Proceedings of IROS-2002*, Switzerland, 2002.

[18] Undeger, C. and Polat, F., "Real-Time Edge Follow: A Real-Time Path Search Approach," *IEEE Transactions on Systems, Man, and Cybernetics: Part C*, Vol. 37, 2007.

[19] Korf, R. E., "Real-Time Heuristic Search," *Artificial Intelligence*, Vol. 42, 1990.

[20] Gong, Q., Lewis, L. R., and Ross, I. M., "Pseudospectral Motion Planning for Autonomous Vehicles," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 32, 2009.

[21] Fahroo, F. and Ross, I. M., "Costate Estimation by a Legendre Pseudospectral Method," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 24, 2001.

[22] Fahroo, F. and Ross, I. M., "Direct Trajectory Optimization by a Chebyshev Pseudospectral Method," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 25, 2002.

[23] Gong, Q., Ross, I. M., and Fahroo, F., "Costate Estimation by a Chebyshev Pseudospectral Method," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 33, 2010.

[24] Choe, R., Puig-Navarro, J., Cichella, V., Xargay, E., and Hovakimyan, N., "Cooperative Trajectory Generation Using Pythagorean Hodograph Bezier Curves," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 39, 2016.

[25] Mehdi, S. B., Choe, R., Cichella, V., and Hovakimyan, N., "Collision Avoidance through Path Replanning using Bezier Curves," *AIAA Guidance, Navigation, and Control Conference*, Kissimmee, Florida, 2015.

[26] Saunders, J., Beard, R., and McLain, T., "Obstacle Avoidance Using Circular Paths," *AIAA Guidance, Navigation and Control Conference and Exhibit*, Hilton Head, South Carolina, 2007.

[27] Jung, D. and Tsiotras, P., "On-Line Path Generation for Small Unmanned Aerial Vehicles Using B-Spline Path Templates," *Journal of Guidance, Control, and Dynamics*, Vol. 36, 2013.

[28] Carsten, J., Ferguson, D., and Stentz, A., "3D Field D*: Improved Path Planning and Replanning in Three Dimensions," *International Conference on Intelligent Robots and Systems*, 2006.

[29] Daniel, K., Nash, A., Koenig, S., and Felner, A., "Theta*: Any-Angle Path Planning on Grids," *Journal of Artificial Intelligence Research*, Vol. 39, 2010.

[30] Koenig, S., "A Comparison of Fast Search Methods for Real-Time Situated Agents," *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2004, pp. 864–871.

[31] Yahja, A., Stentz, A., Singh, S., , and Brumitt, B. L., "Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments," *Proceedings, IEEE Conference on Robotics and Automation*, Leuven, Belgium, 1998.

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

[33] Yuksel, C., Schaefer, S., and Keyser, J., "On the Parameterization of Catmull-Rom Curves," *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, ACM, New York, NY, USA, 2009, pp. 47–53.

[34] Rabin, S., "A* Aesthetic Optimizations," *Game Programming Gems*, Charles River Media, 2000.

[35] Ferguson, D. and Stentz, A., "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.

[36] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms*, MIT Press, 2009.

[37] Murphy, E., *Planning and Exploring Under Uncertainty*, Ph.D. thesis, University of Oxford, Somerville College, Oxford, England, 2010.

[38] Rina Dechter, J. P., "Generalized best-first search strategies and the optimality of A*," *Journal of the ACM*, Vol. 32, 1985.

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

[40] Boeuf, A., Cortés, J., Alami, R., and Siméon, T., "Enhancing Sampling-Based Kinodynamic Motion Planning for Quadrotors," *IEEE/RSJ International Conference on Intelligent Robots and Systems*,

2015.