Robotics: Science and Systems 2025
Los Angeles, CA, USA, June 21–25, 2025

Efficient Hierarchical Any-Angle Path Planning on Multi-Resolution 3D Grids

Victor Reijgwart, Cesar Cadena, Roland Siegwart and Lionel Ott
Autonomous Systems Lab, ETH Zürich, Switzerland
Email: vreijgwart@rai-inst.com, [cesarc | rolandsi | liott]@ethz.ch

Abstract—**Hierarchical, multi-resolution volumetric mapping approaches are widely used to represent large and complex environments as they can efficiently capture their occupancy and connectivity information. Yet widely used path planning methods such as sampling and trajectory optimization do not exploit this explicit connectivity information, and search-based methods such as A* suffer from scalability issues in large-scale high-resolution maps. In many applications, Euclidean shortest paths form the underpinning of the navigation system. For such applications, any-angle planning methods, which find optimal paths by connecting corners of obstacles with straight-line segments, provide a simple and efficient solution. In this paper, we present a method that has the optimality and completeness properties of any-angle planners while overcoming computational tractability issues common to search-based methods by exploiting multi-resolution representations. Extensive experiments on real and synthetic environments demonstrate the proposed approach's solution quality and speed, outperforming even sampling-based methods. The framework is open-sourced to allow the robotics and planning community to build on our research.**

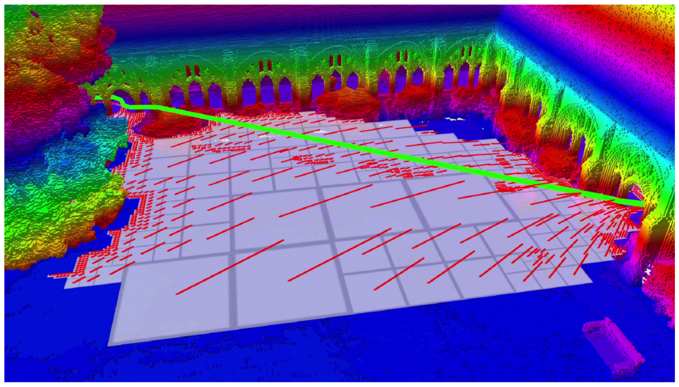
Fig. 1. Illustration of a solution generated by our proposed any-angle path planner, *wavestar*. The shortest path (green) consists of straight-line segments that efficiently traverse free space using a small number of waypoints which tightly fit the obstacles (voxels shaded by height). A 2D slice of the multi-resolution 3D cost field demonstrates how the hierarchical algorithm refines resolution only where necessary, ensuring both efficiency and accuracy.

I. Introduction

A core competency of robots is the ability to autonomously navigate between areas of interest, such as storage spaces, work sites, and inspection points, even if these locations are far apart. Methods for solving this planning problem can be categorized into optimization-, sampling- and search-based approaches. Optimization-based methods produce high-quality, continuous solutions but generally require an initial guess to converge to a good solution. This initial guess is often obtained from a sampling- or search-based planner. Search-based methods generally operate on a graph with a fixed topology, such as a grid map's adjacency graph or a state lattice constructed from motion primitives. Meanwhile, sampling-based methods build the graph by randomly sampling and connecting collision-free robot configurations. Sampling-based approaches are popular in practice due to their ability to find solutions while only sparsely covering large, potentially high-dimensional configuration spaces. However, extracting graphs through random sampling discards much of the information embedded in the volumetric map and neglects its underlying structure. The information contained in discretized maps is finite, yet sampling-based methods are only asymptotically complete and cannot detect infeasibility in finite time. This is particularly problematic in environments with narrow passages, where solving a planning query can take a long time, and feasibility is not guaranteed. This raises a hard-to-answer question in sampling-based methods: How long should one try to find a solution before giving up?

For many applications, a volumetric map's adjacency graph provides a reasonable discretization of the true, continuous search space. Combining this graph with standard search algorithms [5] allows resolution-complete solutions to be found in finite time. While searching for the shortest path, A* [9] and similar methods compute the optimal cost-to-come and predecessor for each explored grid vertex. Their time and space complexity, therefore, scales linearly with the explored volume and cubically with the grid resolution [27]. This is particularly problematic in environments with dead-ends that are deep relative to the grid resolution.

Octree-based maps [11, 26, 22] compactly encode traversability information using multi-resolution. Evaluating A* directly on the adjacency graph of an octree's leaves preserves the completeness of running it on a grid at the highest resolution while offering significant memory and runtime improvements [13]. However, considering only the octree leaves' centers yields suboptimal paths in length and smoothness [2]. As illustrated in Figure 2, post-processing steps such as path-shortening cannot resolve this issue since the paths might not even be close to the true shortest path.

Any-angle planning algorithms, such as Theta* [4], improve path quality by allowing paths to deviate from grid edges, connecting vertices in line of sight with straight lines. This approach can yield paths up to $\approx 13\%$ shorter than those produced by A* [19]. In this paper, we extend Theta*'s cost

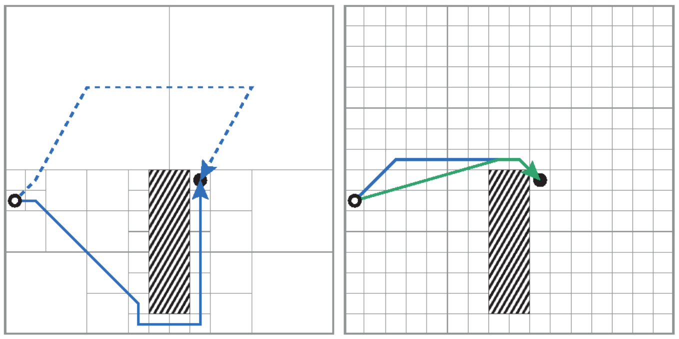

Fig. 2. Comparison of A* on an octree's leaves (left, blue), A* on a fixed-resolution grid (right, blue), and Theta* (right, green). On the octree, A* produces highly suboptimal paths. While its search space includes a path (dashed blue) on the correct side of the obstacle (striped box), this path is ignored due to the detour introduced by passing through the leaves' centers. A* on the grid finds shorter, smoother paths, but still performs worse than Theta*. Our method matches Theta*'s path quality while operating on octrees.

field formulation to efficiently represent large parts of the search space at coarser resolutions. Additionally, we propose a coarse-to-fine search algorithm that starts at the coarsest resolution and refines solutions only in regions requiring higher detail, significantly reducing memory usage and computational cost without sacrificing accuracy.

In summary, the main contribution of this paper is a search-based planner that combines the accuracy of any-angle planning with the efficiency of multi-resolution representations and hierarchical algorithms. Extensive evaluations on synthetic and real-world maps demonstrate that the proposed method retains Theta*'s accuracy while running up to two orders of magnitude faster. Compared to well-established search- and sampling-based planners, it consistently finds near-optimal paths and, in cluttered environments, runs even faster than sampling-based planners. The complete framework is open-sourced[1] to allow the planning and robotics communities to build on these results.

II. Related work

Path planning methods can generally be categorized into optimization-, sampling-, and search-based approaches. Sampling-based methods are most commonly used for global planning, especially in large environments. While very fast, randomized methods such as RRT [16] and RRTConnect [14] provide no guarantees on the quality of their solutions. Variants such as RRT* [12] are guaranteed to converge the optimal solution as the number of samples approaches infinity. However, they do not provide bounds on their intermediate solutions and stopping them after a finite time leads to different paths even when the start and goal positions are the same [6]. A challenge in practice is that the inconsistency of randomized planners worsens in cluttered environments, and finding solutions through narrow passages can take a very long time.

Search-based planners such as A* [9] operate on a discretized search space and are deterministic, complete, and terminate in finite time, explicitly reporting when no solution exists. However, using a fixed-resolution 3D occupancy map's adjacency graph as the space discretization results in runtimes that grow linearly with volume and cubically with resolution, making it impractical for large or high-resolution maps.

Several research efforts have explored hierarchical approaches to improve the scalability of search-based planning. Kambhampati and Davis [13] applied A* to the octree's leaves, compactly representing traversable space and achieving significant efficiency improvements, albeit at the cost of longer, jagged paths. Funk et al. [8] extended this approach to orientation-aware planning in large environments with narrow openings. CFA* [17] proposed a coarse-to-fine strategy, performing a coarse search over large blocks followed by refinement at the grid cell level. HPA* [1] generalized this concept to multi-level hierarchies of pre-processed clusters. Beyond these methods, iterative [10] and information-theoretic [15] approaches have also been proposed. Recently, Du et al. [6] demonstrated how multiple simultaneous weighted-A* searches at different resolution levels can share information to combine their strengths. However, a significant drawback of all these methods is that their path lengths are, at best, equivalent to those of A* on the highest-resolution grid.

Any-angle planners improve upon A* by allowing deviations from the grid's edges, finding up to $\approx 13\%$ shorter paths [19] by better approximating true shortest paths in continuous space, which are *taut* – straight except at inflection points wrapping around obstacles. Theta* [4], a widely adopted any-angle planner, achieves high accuracy in diverse environments [25] by connecting each vertex to its best visible predecessor. While these deviations improve accuracy, Theta* propagates information only along grid edges, enabling simple and efficient implementation. However, in 3D, it incurs significant runtime overhead due to the numerous visibility checks required to ensure vertex-predecessor edges are collision-free. LazyTheta* [19] addresses this limitation with lazy visibility checking, reducing the overhead by an order of magnitude with minimal impact on path quality.

Multi-resolution methods for any-angle planning have also been explored. Chen et al. [2] introduced framed quadtrees, which pad leaf nodes with high-resolution vertices to permit a broader range of angles through each leaf. While effective in 2D, this approach scales poorly for 3D Euclidean shortest paths, multiplying A*'s computational complexity by an additional term that grows quartically with padding resolution. Closest to our work, Faria et al. [7] applied LazyTheta* to octree leaves. However, their method produces arbitrarily suboptimal paths as it considers only leaf centers. In contrast, we explicitly consider the high-resolution vertices within each leaf and dynamically refine the octree to bound the approximation error. Additionally, a custom initialization procedure ensures all potentially optimal inflection points are evaluated. Extensive comparisons and ablations demonstrate that these improvements yield significantly shorter, smoother paths.

[1]https://github.com/ethz-asl/wavestar

III. Problem statement

This paper presents a method for finding collision-free Euclidean shortest paths between a start and goal point in 2D or 3D workspaces, given an occupancy map representing obstacles. To simplify collision checking, we approximate the robot as a bounding sphere and inflate all obstacles by its radius, treating the robot as a point. Like Theta*, the presented algorithm does not account for motion constraints.

IV. Method

In this section, we describe the components of our planner. First, we show how multi-resolution enables compact encoding of intermediate solutions in any-angle planning. Building on this, we present our efficient any-angle path planner, which consists of: i) an approach that ensures all plausible waypoints are efficiently considered, and ii) a coarse-to-fine method to explore the search space spanned by the previously generated waypoints. Formal statements on our method's completeness, optimality and time-complexity are provided in Appendix C.

A. Any-angle planning

Any-angle planners produce shorter and smoother paths than A* by allowing paths to deviate from grid edges (Figure 2 right), better approximating true shortest paths in continuous space. A necessary condition for Euclidean shortest paths is that they are *taut*, i.e. consisting of straight line segments connected at inflection points where the path wraps tightly around obstacles.

We base our planner on Theta* [4], which closely follows A*'s algorithm. Both planners check whether a path through an expanded node can improve the cost-to-come (g cost) of its neighbors. However, Theta* introduces a key improvement: for each neighbor, it performs a visibility check to determine if it can be directly connected to the node's `predecessor`. As illustrated in Figure 3, this eliminates intermediate waypoints, further reducing g costs and avoiding detours.

B. Multi-resolution cost field representation

Search-based planners such as A* and Theta* compute the minimum g cost and best `predecessor` for each (grid) vertex expanded during the search. Since these two properties are often stored together, we refer to their combination as the *cost field*.

By construction, neighboring grid vertices rarely share the same g cost. In Theta*, however, large regions are often dominated by the same `predecessor` (Fig. 3). The g cost of any vertex s is defined as the g cost of its predecessor plus the distance between them. Consequently, the cost field can be compressed losslessly by storing $\texttt{predecessor}(\mathcal{V})$ and $g(\texttt{predecessor}(\mathcal{V}))$ for each region $\mathcal{V}$. The g cost of any vertex s within a region $\mathcal{V}$ can then be retrieved using

$$g(s) = g(s^p) + c(s^p, s) \mid s \in \mathcal{V}, s^p = \texttt{predecessor}(\mathcal{V}) \tag{1}$$

where $c(s^p, s)$ is the Euclidean distance from s^p to s.

To exploit this, we propose to partition the cost field into multi-resolution cubes corresponding to an octree's leaves.

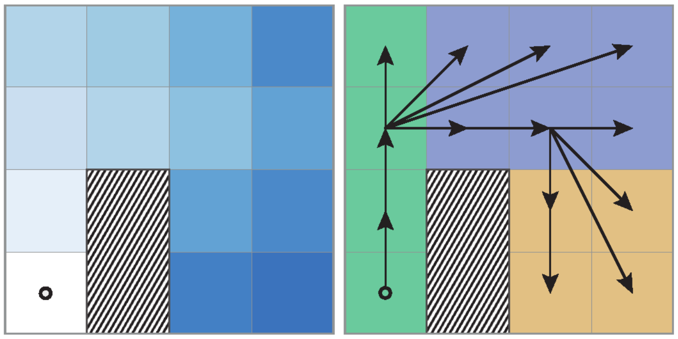
Fig. 3. Illustration of the cost-to-come (g cost) and `predecessor` fields of Theta* in a 2D environment with a single obstacle (striped box). The g cost field (left) changes from cell to cell, while the `predecessor` field (right) is largely constant. All cells to the left of the obstacle (green) are directly visible from the start vertex (black circle) and thus use it as their predecessor. Cells near the top right (purple) connect through the cell at the obstacle's top-left corner, while the rest (gold) connect through the cell at its top-right corner.

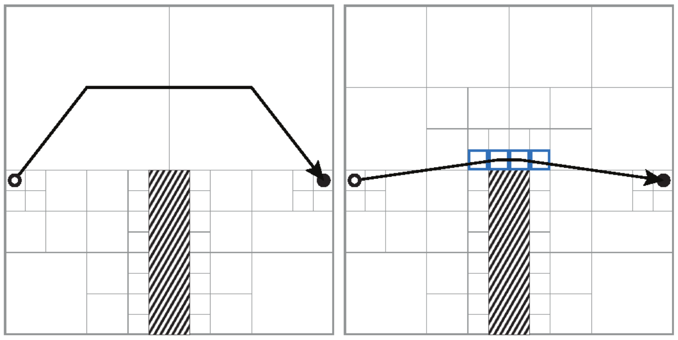
Fig. 4. Illustration of the importance of initializing inflection points. Without initialization (left), the retrieved shortest path may take large detours around obstacles. Initializing the cost field at a higher resolution near obstacles (right) resolves this issue, resulting in shorter, smoother paths. As the number of added subvolumes (blue) is small, the performance overhead remains minimal.

This regular structure enables efficient storage, fast random access, simplified neighborhood operations, and natural alignment between the search space and octree-based traversability maps.

C. Cost field initialization

Just as typical search-based planners where a vertex's predecessor is itself a vertex, we define the predecessors of subvolumes as subvolumes. While a single subvolume could encompass the start, goal, and several inflection points, we initialize the cost field such that each subvolume contains at most one such waypoint to reduce bookkeeping and the runtime complexity of neighborhood operations.

Initializing subvolumes at the minimum resolution required to avoid occupied leaves in the map's octree suffices to guarantee resolution completeness [13]. However, as shown in Figure 4 (left), this approach often results in paths taking significant detours around obstacles. From the definition of taut paths, optimal inflection points can only appear next to obstacles. Thus, considering all vertices of a high-resolution grid that are traversable and adjacent to an obstacle ensures that no inflection points that would be considered by Theta* on the same grid are missed. This initialization can be performed

Algorithm 1: Heuristic-guided search over subvolumes

```
1  open ← $\emptyset$
2  closed ← $\emptyset$
3  $g(s^{\text{start}}) \leftarrow 0$
4  predecessor($\mathcal{V}^{\text{start}}$) ← $s^{\text{start}}$
5  open.*insert*($\mathcal{V}^{\text{start}}$, ComputeFScore($\mathcal{V}^{\text{start}}$))
6  **while** *open* $\neq \emptyset$ **do**
7      $\mathcal{V}$ ← open.*pop*()
8      **if** $s^{goal} \in \mathcal{V}$ **then**
9          **return** *PathFound*
10     **end**
11     closed ← closed $\cup \{\mathcal{V}\}$
12     $g(\mathcal{V}_{\text{center}})$ ← $g$(predecessor($\mathcal{V}$)) +
           $c$(predecessor($\mathcal{V}$), $\mathcal{V}_{\text{center}}$)
13     UpdateSubvolume($\mathcal{V}$, $\mathcal{V}_{\text{root}}$)
14 **end**
15 **return** *NoPathFound*
```

globally or incrementally as the search progresses through the traversability map. In our tests, we adopt the incremental approach, allowing the initialization cost to scale with the explored volume rather than the map's total size.

D. Multi-resolution search

Search-based planning in 3D spaces is computationally challenging as the number of vertices grows rapidly with increasing resolution. To address this, our approach explores the search space at the coarsest possible resolution and dynamically refines it only where needed to maintain accuracy. After initializing the cost field as described in Section IV-C, our any-angle planner computes each subvolume's predecessor($\mathcal{V}$) and g(predecessor($\mathcal{V}$)) by running a modified version of A* over the octree's leaves. Similar to A*, the algorithm uses a min-priority queue (open) to expand elements sorted by their estimated goal-reaching cost (f score). However, unlike A*, the elements in our algorithm are subvolumes that can contain many vertices, which are processed together.

Algorithm 1 shows our planner's main loop. For each expanded subvolume $\mathcal{V}$, the algorithm first checks whether it contains the goal vertex s^{goal} (Li. 8). If so, the search terminates. Otherwise, $\mathcal{V}$ is added to the closed set, and the g cost for its center is computed (Li. 12) and stored for potential use as a predecessor. Finally, the UpdateSubvolume function processes all adjacent subvolumes, which may include 26 or more multi-resolution neighbors.

E. Dynamic refinement

Throughout the majority of the environment, the cost field resolution chosen by the initialization procedure suffices. However, subvolumes are occasionally reachable from multiple predecessors, each better suited for different vertices within the subvolume. For instance, in Figure 5, vertices toward the top right are optimally reached by passing above the obstacle, whereas passing below the obstacle provides shorter paths to the vertices on the bottom right. In such cases, instead of

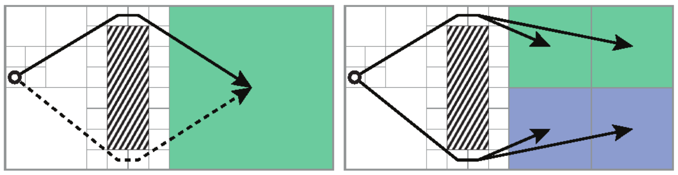

Fig. 5. Illustration of our dynamic refinement procedure. When a new path (dashed arrow) is discovered to a subvolume $\mathcal{V}'$ (left) that has already been reached (solid arrow), we evaluate its effect on the cost to reach each vertex in $\mathcal{V}'$. The algorithm handles three cases: i) the new path reduces the cost for all vertices, replacing the previous path; ii) the new path does not improve any costs and is ignored; iii) some costs improve while others worsen. In this last case (right), $\mathcal{V}'$ is recursively subdivided until each child subvolume is fully resolved under case i or ii. Subvolumes are colored by their predecessor.

Algorithm 2: Recursive octree node updates

```
**Function** UpdateSubvolume($\mathcal{V}$, $\mathcal{V}'$) **is**
    **if** IsLeaf($\mathcal{V}'$) **then**
        Status ← UpdateCost($\mathcal{V}$, $\mathcal{V}'$)
        **if** *Status* = *StrictlyBetter* **then**
            **if** $\mathcal{V}' \in$ *open* **then**
                open.*remove*($\mathcal{V}'$)
            **end**
            open.*insert*($\mathcal{V}'$, ComputeFScore($\mathcal{V}'$))
            **return**
        **else if** *Status* = *NotBetter* **then**
            **return**
        **else** // Status = *Ambiguous*
            open.*remove*($\mathcal{V}'$)
        **end**
    **end**
    **foreach** $\mathcal{V}'^{child} \in \mathcal{V}'$ **do**
        **if** $\mathcal{V}'^{child} \notin$ *closed* **then**
            $s^{p'}$ ← predecessor($\mathcal{V}'$)
            **if** $\mathcal{V}'^{child} \notin$ *open* **then**
                predecessor($\mathcal{V}'^{\text{child}}$) ← $s^{p'}$
                open.*insert*($\mathcal{V}'^{\text{child}}$, ComputeFScore($\mathcal{V}'^{\text{child}}$))
            **end**
            **if** AreAdjacent($\mathcal{V}$, $\mathcal{V}'^{child}$)
             *and* $\mathcal{V}'^{child} \neq \mathcal{V}$ **then**
                UpdateSubvolume($\mathcal{V}$, $\mathcal{V}'^{\text{child}}$)
            **end**
        **end**
    **end**
**end**
**Function** AreAdjacent($\mathcal{V}^a$, $\mathcal{V}^b$) **is**
    sep ← cmax($\mathcal{V}^a_{\text{min}}$, $\mathcal{V}^b_{\text{min}}$) − cmin($\mathcal{V}^a_{\text{max}}$, $\mathcal{V}^b_{\text{max}}$)
    **return** sep.$x \leq 1$ *and* sep.$y \leq 1$ *and* sep.$z \leq 1$
**end**
```

assigning a single suboptimal predecessor to the entire subvolume, we dynamically refine the cost field.

Leveraging the hierarchical structure of octrees, this refinement can efficiently be integrated into our multi-resolution planner through a recursive UpdateSubvolume method (Alg. 2). Starting at the octree's root $\mathcal{V}' = \mathcal{V}_{\text{root}}$, the method handles two cases. If $\mathcal{V}'$ is a leaf, the function calls `UpdateCost` to evaluate whether a path through the expanded subvolume $\mathcal{V}$ or its `predecessor`$(\mathcal{V})$ improves the g cost of $\mathcal{V}'$. Since $\mathcal{V}'$ can contain multiple vertices, the comparison has three possible outcomes: i) using $\mathcal{V}$ or `predecessor`$(\mathcal{V})$ reduces the cost for all vertices in $\mathcal{V}'$, updating its `predecessor` and reprioritizing it in the `open` queue; ii) no costs improve and both predecessors are ignored; or iii) some costs improve while others worsen, flagging $\mathcal{V}'$ for refinement.

In the second case, for non-leaf subvolumes or those flagged for refinement, the function iterates over $\mathcal{V}'$'s children, skipping `closed` nodes. Newly created children inherit their parent's `predecessor` and are added to the `open` queue, ensuring the full region $\mathcal{V}'$ covered is eventually processed. The function then recursively visits each child adjacent to $\mathcal{V}$, until all descendant subvolumes fall under cases 1 or 2. `AreAdjacent` tests whether subvolumes $\mathcal{V}^a$ and $\mathcal{V}^b$ touch or overlap, which holds if the minimum offset between their Axis-Aligned Bounding Boxes (AABBs) is at most 1 along every axis on the highest-resolution grid. This offset is computed from the coefficient-wise max (cmax) and min (cmin) of their AABB corners $\mathcal{V}_{\text{min}}$ and $\mathcal{V}_{\text{max}}$.

The `UpdateCost` function (Alg. 3) implements the comparison between $\mathcal{V}'$'s current `predecessor`, $s^{p'}$, and two candidate predecessors: an inflection point at the center of $\mathcal{V}$, s^c, and `predecessor`$(\mathcal{V})$, s^p. If $s^{p'}$ results in a lower g cost for all vertices $s \in \mathcal{V}'$ compared to s^p and s^c, the function returns *NotBetter*, indicating no changes are required. Conversely, if s^p or s^c provides strictly better costs for all $s \in \mathcal{V}'$, the `predecessor` is updated and *StrictlyBetter* is returned. If neither condition is fully satisfied, the function returns *Ambiguous*, signaling the need for further refinement.

In practice, tolerating small path length suboptimalities may be acceptable, particularly if it leads to efficiency improvements. To quantify this, we define the worst-case suboptimality of $s^{p'}$ relative to an alternative predecessor s^i over $\mathcal{V}'$ as

$$\hat{E}(s^{p'}, s^i, \mathcal{V}') = \max_{s\in\mathcal{V}'} \frac{g(s^{p'}) + c(s^{p'}, s) - g(s^i) - c(s^i, s)}{c(s^{p'}, s)} \tag{2}$$

where $c(s^a, s^b)$ is the straight-line distance from s^a to s^b. Since the error is normalized by edge length, the total accumulated error along the path grows at most proportionally with the path's length.

To bound this error, we introduce ϵ, which represents the worst-case relative path length suboptimality. The function `IsBetterOrSimilar` (Li. 22) applies this threshold to decide whether a new predecessor should be accepted. For $\epsilon = 0$, the function returns `True` only if s^a is a strictly better predecessor than s^b for every vertex in $\mathcal{V}'$. The planner then refines each subvolume until its children are strictly dominated by a single predecessor. In general, `UpdateSubvolume` recurses until the following condition holds for $s^i \in \{s^p, s^c\}$:

$$\hat{E}(\texttt{predecessor}(\mathcal{V}'), s^i, \mathcal{V}') \le \epsilon \tag{3}$$

Algorithm 3: Computing cost updates and heuristics

```
1  Function UpdateCost($\mathcal{V}, \mathcal{V}'$) is
2      $s^c \leftarrow \mathcal{V}_{\text{center}}$
3      $s^p \leftarrow$ predecessor($\mathcal{V}$)
4      $s^{p'} \leftarrow$ predecessor($\mathcal{V}'$)
5      if LineOfSight($s^p, \mathcal{V}'$) then
           // Ray traced connection
6          if IsBetterOrSimilar($s^{p'}, s^p, \mathcal{V}'$) then
7              return NotBetter
8          else if IsBetterOrSimilar($s^p, s^{p'}, \mathcal{V}'$)
            then
9              predecessor($\mathcal{V}'$) $\leftarrow s^p$
10             return StrictlyBetter
11         end
12     else
           // Direct neighbor connection
13         if IsBetterOrSimilar($s^{p'}, s^c, \mathcal{V}'$) then
14             return NotBetter
15         else if IsBetterOrSimilar($s^c, s^{p'}, \mathcal{V}'$)
            then
16             predecessor($\mathcal{V}'$) $\leftarrow s^c$
17             return StrictlyBetter
18         end
19     end
20     return Ambiguous
21 end
22 Function IsBetterOrSimilar($s^a, s^b, \mathcal{V}'$) is
23     if $\forall s \in \mathcal{V}' : g(s^a) + c(s^a, s) <$
          $g(s^b) + c(s^b, s) + \epsilon\ c(s^a, s)$ then
24         return True
25     else
26         return False
27     end
28 end
29 Function ComputeFScore($\mathcal{V}$) is
30     $s^p \leftarrow$ predecessor($\mathcal{V}$)
31     return $\min_{s\in\mathcal{V}} [g(s^p) + c(s^p, s) + h(s)]$
32 end
```

Finally, `ComputeFScore` demonstrates how our multi-resolution planner computes consistent f scores for sorting the `open` queue, by identifying the minimum f score across all vertices in $\mathcal{V}$. Although `IsBetterOrSimilar` and `ComputeFScore` consider subvolumes with many vertices, the computational burden is significantly reduced in practice because $c(s^p, s)$ and $h(s)$ are straight-line distances, requiring only a few critical vertices to be checked.

V. Experiments

In the experiments, we start with an evaluation of our multi-resolution planner's initialization and dynamic refinement procedures. Then, we compare our method to other path-planning approaches based on success rate, path length, and runtime.

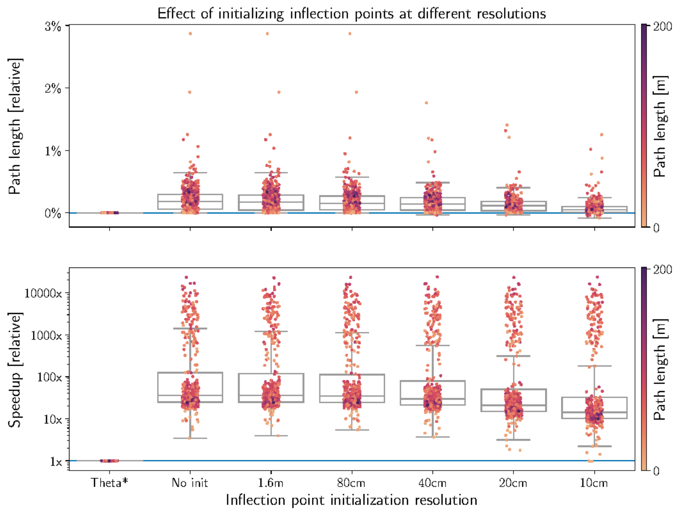

Fig. 6. Initialization procedure's impact on path length and runtime relative to Theta* (blue line). Reading the plot from left to right, we see how increasing the inflection point initialization resolution brings our path lengths (top) closer to Theta* while remaining significantly faster (bottom).

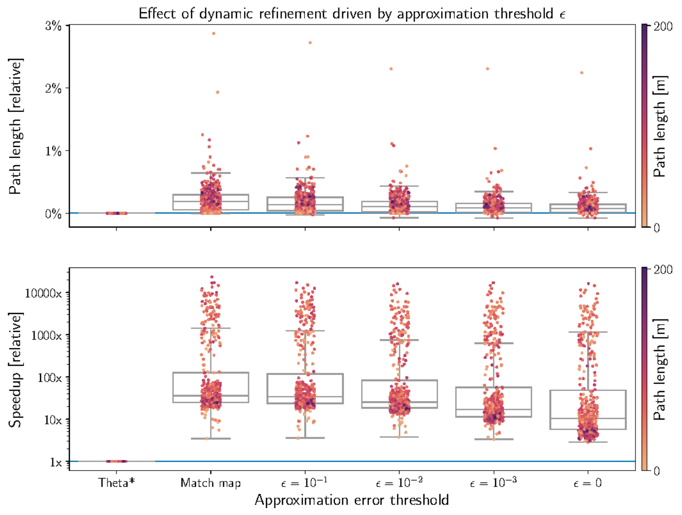

Fig. 7. Ablation showing how the dynamic refinement strategy affects the path length and speedup (log scale) of our method relative to Theta* (blue line). Reading the plot from left to right, we see that as the threshold is tightened, the path lengths decrease while runtime moderately increases.

The result plots share an overall structure, showing the measured quantity on the Y-axis, while the property being varied is along the X-axis. The result distributions are presented as box plots with individual results shown as dots, colored according to the true path length. Where results are relative to a baseline, a blue line indicates baseline performance.

A. Impact of initialization and refinement

The evaluation of our initialization and refinement procedures impact on performance are conducted on five synthetic maps, each measuring $100\,\mathrm{m} \times 100\,\mathrm{m} \times 100\,\mathrm{m}$ mapped at $10\,\mathrm{cm}$ resolution. They are generated by adding 0, 1000, 2000, 3000, and 4000 randomly shaped obstacles to an initially empty volume, representing varying levels of clutter. For each map, 100 random collision-free start-goal pairs (500 total) were sampled.

1) Inflection point initialization: To evaluate the significance of the initialization procedure (Section IV-C), we compare the path lengths and execution times of our planner with and without initialization to `Theta*` running at the highest resolution. As our method allows defining the initialization resolution, we present results for initialization resolutions ranging from $1.6\,\mathrm{m}$ to $10\,\mathrm{cm}$. For this analysis, the dynamic refinement strategy (Section IV-E) is disabled to isolate the effect of initialization.

The results in Figure 6 (top) show that increasing the inflection point initialization resolution moves the path lengths of our planner closer to those of `Theta*`. On average, path lengths converge to values close to `Theta*`, especially for longer paths (deep purple). Outliers primarily correspond to short paths (light orange), where small differences are amplified when normalized by the short path length and runtime values of `Theta*`.

At coarser resolutions ($\geq 80\,\mathrm{cm}$), initializing inflection points yields no notable improvement over `No init` because the cost field's octree conforms to the obstacles in the occupancy map. This results in most occupied cells being surrounded by medium to high-resolution subvolumes, rendering low-resolution initializations redundant. Additionally, the occupancy map and cost field are stored using an optimized octree data structure [18], limiting the coarsest resolution to $6.4\,\mathrm{m}$, implicitly preventing extremely bad solutions. At higher resolutions, our method occasionally discovers slightly shorter paths than `Theta*`, as `Theta*` is not guaranteed to be optimal.

Looking at runtime results (Figure 6 bottom) we see that increasing the initialization resolution generally increases runtime, as higher resolutions require the search to expand more subvolumes and consider more inflection points as predecessors. Note that the clusters of points at the top of the bottom plot correspond to queries in the 0-obstacle environment. Since these involve no obstacles requiring initialization, their speedup is independent of the initialization resolution.

2) Refinement strategy: To evaluate the effect of the dynamic refinement procedure (Section IV-E), we run our planner with different approximation error thresholds ϵ (Eq. (3)) and compare the results to `Theta*`. The initialization procedure (Section IV-C) is disabled to isolate the effect of refinement. We test ϵ values ranging from 10^{-1} to 10^{-3}, along with two special cases: lossless refinement ($\epsilon = 0$) and no refinement, which we refer to as `Match map`. Because the planner can only traverse fully unoccupied cells, the resolution of the cost field must always match or exceed the occupancy map's leaf resolution. When refinement and initialization are disabled, the cost field's subvolumes exactly match the occupancy map's leaves, resulting in the `Match map` configuration.

The results in Figure 7 show that as the threshold ϵ is tightened, the path lengths gradually approach those of `Theta*`. Similar to the initialization ablations, the error relative to `Theta*` is already low for `Match map` because

TABLE I
SUMMARY OF ABLATIONS. PATH LENGTHS RELATIVE TO THETA*.

		Initialization	
Path length $\mu \pm \sigma$		None	10 cm
Refinement	None	0.23% ± 0.43%	0.07% ± 0.13%
	$\epsilon = 10^{-2}$	0.16% ± 0.32%	**0.04%** ± **0.12%**

very coarse free space leaves rarely occur in occupancy maps. This highlights that our multi-resolution, any-angle cost field formulation is inherently accurate, even at the moderate resolutions that dominate most free space. As ϵ is reduced, the relative path lengths generally remain within their ϵ suboptimality thresholds. However, some outliers persist, and the paths do not fully converge to Theta*, even for $\epsilon = 0$, due to the absence of initialization in this ablation, which limits the discovery of critical inflection points. In terms of runtime, reducing ϵ gradually increases computation, as smaller thresholds require more refinement steps.

In conclusion, as shown in Table I, initialization and refinement each reduce the mean and variance of path lengths, with their combination providing the greatest improvement.

B. Comparisons with other planners

The comparisons are performed on maps of four real environments (Mine, Cloister, Math, and Park sequences of the Newer College Dataset [28]) mapped with *wavemap* [22] at 10 cm resolution. These sequences represent constrained indoor (Mine), mixed indoor-outdoor (Cloister), large urban (Math), and large vegetated environments (Park). Obstacles were inflated by 35 cm to account for the robot's radius. For each map, 100 random collision-free start-goal pairs (400 total) were sampled. To assess how each planner handles unsolvable cases, infeasible queries were not filtered out.

We compare the success rates, path lengths, and execution times of our proposed multi-resolution planner to a representative set of search and sampling-based planners. In terms of search-based planners, we implemented fixed-resolution versions of A* [9], Theta* [4], and LazyTheta* [19]. For A*, we used the octile distance heuristic, which is consistent on 26-connected grids, as we found it to run up to 70% faster than using the Euclidean distance heuristic. Additionally, we include the reference implementation of OctreeLazyTheta* [7] as a multi-resolution baseline. For sampling-based planning, we used the RRTConnect [14] and RRT* [12] implementations from the Open Motion Planning Library [24]. While RRTConnect terminates immediately once a path is found, RRT* does not. Therefore, we include three RRT* variants with increasing time budgets, namely RRT* 0.1s, RRT* 1s and RRT* 10s. Note that RRTConnect is also limited to a maximum time budget of 10 s, to keep it from running forever when a planning query is infeasible.

We evaluate three variants of our multi-resolution planner: *Ours*, *OursLazy*, and *OursFast*. *Ours* implements the baseline algorithm as described in the method section. To improve runtime further, *OursLazy* and *OursFast* incorporate lazy visibility checking, which reduces computational overhead by deferring visibility evaluations until necessary. These variants modify

TABLE II
PLANNING SUCCESS RATES PER MAP FOR 100 RANDOMLY SAMPLED QUERIES EACH, INCLUDING INFEASIBLE CASES.

Success rate (%)	Mine	Cloister	Math	Park
A*	**88**	**100**	**98**	**99**
Theta*	**88**	**100**	**98**	**99**
LazyTheta*	**88**	**100**	**98**	**99**
OctreeLazyTheta*	**88**	**100**	**98**	**99**
RRTConnect	**88**	97	**98**	97
RRT* 0.1s	80	37	96	56
RRT* 1s	85	52	97	90
RRT* 10s	**88**	87	**98**	96
Ours	**88**	**100**	**98**	**99**
Ours Lazy	**88**	**100**	**98**	**99**
Ours Fast	**88**	**100**	**98**	**99**

the baseline algorithm as detailed in Appendix B, applying the principles of LazyTheta* [19]. The specific settings we use for our three planner variants are:

- *Ours*: $\epsilon = 10^{-2}, r^{\text{init}} = 10\,\text{cm}$
- *OursLazy*: $\epsilon = 10^{-2}, r^{\text{init}} = 10\,\text{cm}$, lazy visibility checks
- *OursFast*: $\epsilon = 10^{-2}, r^{\text{init}} = 40\,\text{cm}$, lazy visibility checks

To ensure fair comparisons, all planners, including ours, use optimized data structures and subroutines. The fixed-resolution search-based planners store their cost fields using a hashed voxel block data structure [20], while our multi-resolution planner employs a hashed octree data structure [18]. These planners and all RRT variants use *wavemap*'s hierarchical occupancy map and multi-resolution ray tracer for fast traversability and visibility checking. As motivated in Section III, we inflate all obstacles by the robot's radius, allowing it to be treated as a point. We configure OctreeLazyTheta*, which uses Octomap [11] and a custom visibility checker, to match this setup. All experiments are run single-threaded on the same benchmarking server with an Intel i9-9900K CPU and 64 GB of RAM.

1) Success rates: Starting with the success rates shown in Table II, all search-based planners perform equally well. As the start and goal pose pairs can contain infeasible planning queries, even complete planners may fail in some environments. For example, in Mine where none of the planners succeed in more than 88 out of 100 queries due to limited connectivity between areas.

Among the sampling-based planners, RRTConnect achieves the highest success rate, performing almost as well as the search-based planners. Its bidirectional tree growth and lack of rewiring provide an efficiency advantage over RRT* 10s, which comes in a close second (both planners operate within a maximum time budget of 10 s). RRT* 1s trails slightly behind RRT* 10s in simpler environments but struggles in maps that are large (Park) or have narrow passages (Cloister), as these scenarios require extensive sampling to ensure adequate coverage or density. Finally, RRT* 0.1s performs reasonably well only in the Math environment, which consists of wide open spaces with good visibility.

We verified that for every query where at least one planner succeeded, all search-based planners also succeeded. This empirical finding suggests that our multi-resolution planners

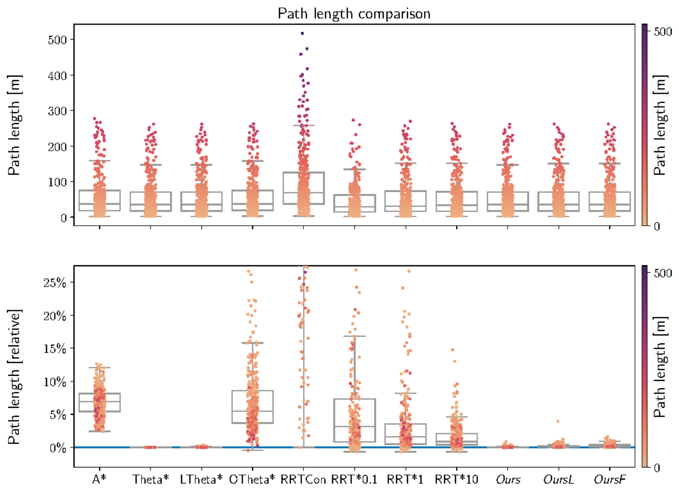

Fig. 8. Path lengths for selected search- and sampling-based planners and three variants of our multi-resolution planner. The upper plot shows absolute lengths, and the lower plot shows lengths relative to `Theta*` (blue line). Outliers for sampling-based planners are partially omitted in the lower plot. In particular, only the bottom few quantiles of `RRTConnect` are visible.

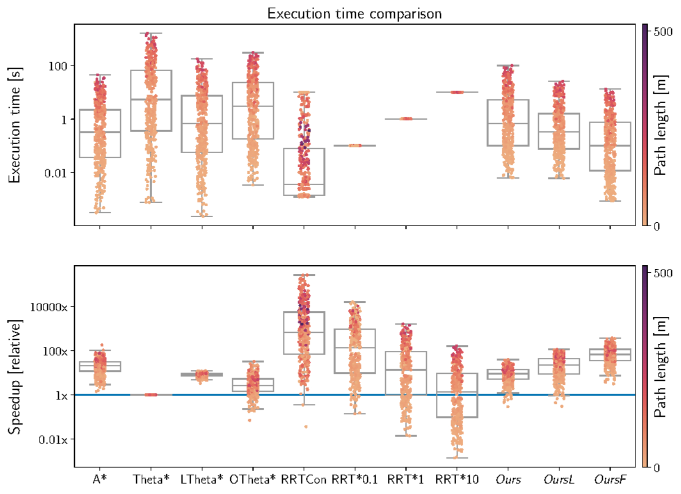

Fig. 9. Execution times for selected search- and sampling-based planners and three variants of our multi-resolution planner. The upper plot shows absolute times (log scale), and the lower plot shows speedups relative to `Theta*` (blue line).

TABLE III
AVERAGE PATH LENGTHS PER MAP FOR QUERIES WHERE ALL PLANNERS SUCCEEDED.

Mean path length (m)	Mine	Cloister	Math	Park
A*	15.96	20.49	45.10	106.05
Theta*	**14.87**	**19.06**	**42.20**	**99.12**
LazyTheta*	<u>14.88</u>	**19.06**	42.22	<u>99.14</u>
OctreeLazyTheta*	16.34	20.42	44.67	103.63
RRTConnect	30.49	39.49	73.83	155.89
RRT* 0.1s	17.50	19.59	44.05	106.52
RRT* 1s	15.81	19.35	43.04	101.41
RRT* 10s	15.16	19.19	42.61	100.04
Ours	14.89	**19.06**	<u>42.21</u>	99.15
Ours Lazy	14.91	<u>19.07</u>	42.26	99.21
Ours Fast	14.95	19.08	42.28	99.25

maintain the completeness guarantee of their fixed-resolution counterparts. Additionally, there were no cases where a sampling-based planner found a solution that the search-based planners could not. This supports the idea that the adjacency graph of an occupancy map provides a reliable approximation of the solution space.

2) Path length: Moving on to the path quality evaluations, we compare the average path lengths for all planners. To ensure fairness, only queries where all planners succeeded are included, avoiding bias toward planners that fail more often on longer paths. As shown in Table III, `Theta*` consistently finds the shortest paths on all maps. `LazyTheta*` follows closely, with paths only 0.03% longer on average, and *Ours* achieves similar results. *OursLazy* and *OursFast* also perform well, with *OursFast* producing slightly longer paths but never exceeding `Theta*` by more than 0.5%. In contrast, the RRT* variants gradually increase in path length as their time budgets decrease. `A*` and `OctreeLazyTheta*` also yield noticeably longer paths, as `A*` is constrained to a 26-connected grid and `OctreeLazyTheta*` restricts paths to octree leaf centers. Finally, `RRTConnect` produces the longest paths, which, on average, are almost twice as long as those of `Theta*`.

The absolute and relative path length distributions in Figure 8 provide additional insights. The evaluated path lengths ranged from 0 to 500m, with shorter paths being more frequent due to the smaller connected areas in maps like `Mine` and `Cloister`. Most planners find reasonable paths, but `RRTConnect` stands out with significantly longer paths and high variance. `RRT* 0.1s` appears to find slightly shorter paths on average, a bias explained by its failure to solve queries with distant start and goal pairs.

For relative path lengths, `Theta*` consistently finds the shortest paths and is closely followed by `LazyTheta*`. While RRT* occasionally surpasses `Theta*`, it is less consistent overall. For instance, `RRT* 10s` features outliers with paths up to 1.8 times longer than `Theta*`. `RRTConnect` exhibits extreme variance, with paths up to 16.8 times longer than `Theta*`. This highlights the importance of RRT*'s tree rewiring for improving path quality. As predicted by Nash et al. [19], `A*` paths are far from optimal, with most paths being at least 2% longer and some reaching the theoretical worst-case of 13%. Similarly, `OctreeLazyTheta*` can introduce significant detours, sometimes exceeding 25%. Finally, *Ours* closely matches `Theta*` on average, demonstrating high consistency and very few outliers. *OursLazy* introduces slight variability due to lazy visibility checking, producing paths that are marginally less direct. Also reducing the inflection point initialization resolution in *OursFast* results in slightly larger detours around obstacles. Nonetheless, paths produced by *OursFast* remain suitable for many practical applications.

3) Runtime: The last metric we evaluate is execution time, starting with the average runtime of each planner in each environment, as shown in Table IV. All runs are included in the averages to capture both successful and unsuccessful queries, while the planning times of the RRT* variants are not listed as they are constant. `Theta*` is the slowest planner by a large margin. Enabling lazy visibility checking (`LazyTheta*`) improves its runtime by 6 to 9 times, but it remains signif-

TABLE IV
AVERAGE EXECUTION TIMES PER MAP FOR 100 RANDOMLY SAMPLED QUERIES (RRT* VARIANTS OMITTED DUE TO FIXED RUNTIMES).

Mean execution time (s)	Mine	Cloister	Math	Park
A*	0.20	1.15	1.89	8.19
Theta*	2.70	30.29	61.18	281.38
LazyTheta*	0.42	3.66	6.89	31.51
OctreeLazyTheta*	2.88	10.79	13.12	64.94
RRTConnect	1.24	1.63	**0.21**	**0.39**
Ours	0.51	2.69	3.86	20.22
Ours Lazy	0.20	0.82	1.12	5.72
Ours Fast	**0.10**	**0.36**	0.51	2.76

icantly slower than `A*`. Interestingly, `OctreeLazyTheta*` is not faster than `LazyTheta*`, indicating that octree-based representations alone do not outperform optimized fixed-resolution methods. However, substantial speedups can be achieved through careful multi-resolution design. On average, *Ours* is 8 times faster than `Theta*` in confined environments and 15 times faster in large open spaces. *OursLazy* achieves similar gains, being 2 to 6 times faster than `LazyTheta*`. *OursFast* and `RRTConnect` are the fastest overall. *OursFast* is up to 12 times faster than `RRTConnect` in confined environments like `Mine`, while `RRTConnect` is up to 7 times faster in large open spaces like `Park`.

Figure 9 shows runtime distributions, absolute with logarithmic scale (top) and relative to `Theta*` (bottom). For search-based planners, execution time is strongly correlated with path length. In contrast, `RRTConnect` shows no meaningful correlation, and the RRT* variants maintain constant runtimes. Notably, the speedup of all planners over `Theta*` increases with path length, reflecting `Theta*`'s poor scalability.

Looking at our proposed planners, *Ours* and *OursLazy* are rarely slower than `Theta*` and achieve speedups of up to 60 and 100 times, respectively. *OursFast* consistently outperforms all other search-based planners, with speedups ranging from 5 to 800 times. We can also see that our method's runtimes are more predictable than those of sampling-based methods, in part due to being able to detect infeasible queries. This result underscores the effectiveness of leveraging multi-resolution to balance efficiency and path quality.

VI. LIMITATIONS

The main limitation of our method is its restriction to Euclidean cost formulations. This is because, like Theta*, our multi-resolution extension relies on the triangle inequality to find any-angle paths efficiently [19]. We also focused on Euclidean workspaces, to reliably and efficiently provide global paths as inputs to trajectory optimizers or local planners in navigation tasks [21, 29, 3].

VII. CONCLUSION

In this paper, we presented *wavestar*, a search-based global planning method for Euclidean workspaces that utilizes an octree-like structure to improve planning speed in occupancy maps. We extend the ideas from any-angle planners to hierarchical representations to exploit spatial sparsity by generalizing the concept of inflection points from fixed-resolution grids to a hierarchical representation.

Extensive evaluations and comparisons to search-based methods show that we achieve paths of competitive quality but at a substantially reduced computational cost. This demonstrates that exploiting the inherent sparsity of real environments does not significantly impact accuracy, while providing significant computational benefits. Additionally, in contrast to sampling-based methods, our approach can detect when a query is infeasible, while also producing high-quality paths. Overall, these results show that our approach combines the benefits and guidance of search-based methods with the speed of sampling-based methods. This makes it suitable as a first step for many navigation systems, where a coarse initial path through 3D space is needed.

ACKNOWLEDGMENTS

We would like to thank Helen Oleynikova, Alexander Liniger and the reviewers for their valuable suggestions and feedback on the manuscript.

REFERENCES

[1] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *J. Game Dev.*, 1(1):1–30, 2004.

[2] D. Z. Chen, R. J. Szczerba, and J. J. Uhran. A framed-quadtree approach for determining Euclidean shortest paths in a 2-D environment. *IEEE Transactions on Robotics and Automation*, 13(5):668–681, 1997.

[3] P. Chen, Y. Jiang, Y. Dang, T. Yu, and R. Liang. Real-time efficient trajectory planning for quadrotor based on hard constraints. *JINT*, 2022.

[4] K. Daniel, A. Nash, S. Koenig, and A. Felner. Theta*: Any-Angle Path Planning on Grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[6] W. Du, F. Islam, and M. Likhachev. Multi-resolution A*. In *Proceedings of the International Symposium on Combinatorial Search*, volume 11, pages 29–37, 2020.

[7] M. Faria, R. Marín, M. Popović, I. Maza, and A. Viguria. Efficient Lazy Theta* Path Planning over a Sparse Grid to Explore Large 3D Volumes with a Multirotor UAV. *Sensors*, 19(1):174, 2019.

[8] N. Funk, J. Tarrio, S. Papatheodorou, P. F. Alcantarilla, and S. Leutenegger. Orientation-Aware Hierarchical, Adaptive-Resolution A* Algorithm for UAV Trajectory Planning. *IEEE Robotics and Automation Letters*, 8(10): 6723–6730, 2023.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[10] F. Hauer, A. Kundu, J. M. Rehg, and P. Tsiotras. Multi-scale perception and path planning on probabilistic obstacle maps. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4210–4215, 2015.

[11] A. Hornung, K. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees. *Autonomous Robots*, 2013.

[12] Sertac K. and Emilio F. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[13] S. Kambhampati and L. Davis. Multiresolution path planning for mobile robots. *IEEE Journal on Robotics and Automation*, 2(3):135–145, 1986.

[14] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation*, volume 2, pages 995–1001 vol.2, 2000.

[15] D. T. Larsson, D. Maity, and P. Tsiotras. Information-Theoretic Abstractions for Planning in Agents With Computational Constraints. *IEEE Robotics and Automation Letters*, 6(4):7651–7658, 2021.

[16] S. M. LaValle. *Planning algorithms*. Cambridge University Press, New York, 2006.

[17] J.-Y. Lee and W. Yu. A coarse-to-fine approach for fast path finding for mobile robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5414–5419, 2009.

[18] K. Museth. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics*, 32 (3), 2013.

[19] A. Nash, S. Koenig, and C. Tovey. Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):147–154, 2010.

[20] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics*, 32(6):1–11, 2013.

[21] M. Nieuwenhuisen and S. Behnke. Local multiresolution trajectory optimization for micro aerial vehicles employing continuous curvature transitions. In *IROS*, 2016.

[22] V. Reijgwart, C. Cadena, R. Siegwart, and L. Ott. Efficient volumetric mapping of multi-scale environments using wavelet-based compression. In *Robotics: Science and Systems*, 2023.

[23] S. J. Russell, P. Norvig, and E. Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[24] I. A. Şucan, M. Mark, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012.

[25] T. Uras and S. Koenig. An Empirical Comparison of Any-Angle Path-Planning Algorithms, September 2021.

[26] E. Vespa, N. Nikolov, M. Grimm, L. Nardi, P. Kelly, and S. Leutenegger. Efficient Octree-Based Volumetric SLAM Supporting Signed-Distance and Occupancy Mapping. *IEEE Robotics and Automation Letters*, 2018.

[27] Liron Y., Alberto B., and Guillermo S. O(N) implementation of the fast marching algorithm. *Journal of Computational Physics*, 212(2):393–399, 2006.

[28] L. Zhang, M. Camurri, and M. Fallon. Multi-Camera LiDAR Inertial Extension to the Newer College Dataset. *CoRR*, 2021.

[29] B. Zhou, J. Pan, F. Gao, and S. Shen. RAPTOR: Robust and Perception-Aware Trajectory Replanning for Quadrotor Fast Flight. *TRO*, 2021.

APPENDIX

A. A brief introduction to Theta*

Given that our method's cost field formulation and search algorithm can be seen as a multi-resolution extension of Theta*, we briefly summarize the algorithm and relevant terminology in this section. Theta* [4] is an any-angle path-finding extension to the A* [9] search algorithm. Just like A*, it only propagates information along grid edges. The key distinction between the two search algorithms lies in how they select each vertex's predecessor. Since A* only considers each vertex's direct neighbors, the paths it returns are strictly composed of grid edges. Theta* additionally considers connections to each direct neighbor's best predecessor if it is within the vertex's line of sight. This allows Theta* to deviate from the grid and find paths up to $\approx 13\%$ shorter than those found by A*, at the cost of increased runtime due to the additional visibility checks.

The main loop of A* and Theta*, shown in Algorithm 4, is identical. Both search algorithms store two values per vertex, namely the vertex's cost-to-come (g cost) and an index or pointer to its best `predecessor`. Furthermore, both algorithms use a priority queue (`open`) to expand vertices in order of their minimum f score, where $f(s) = g(s)+h(s)$ with $h(s)$ a consistent heuristic function. Using a consistent heuristic guarantees that a node is only expanded from the queue once its optimal g cost and `predecessor` have been found [23]. A closed set (`closed`) can therefore be used to track and explicitly skip updates of already expanded nodes (Line 13). It also means that both algorithms can terminate immediately once the goal vertex s^{goal} is expanded (Line 8). Since the paths found by A* can only contain edges of the 26-connected grid, using the octile distance to the goal, $h(s) = \|s^{\text{goal}} - s\|_{\text{oct}}$, is consistent. In contrast, Theta* must use the Euclidean distance, $h(s) = \|s^{\text{goal}} - s\|_2$, because its paths are not constrained to the grid's edges.

As highlighted earlier, the key difference between A* and Theta* is how they find each vertex's best predecessor. When expanding vertex s, the function `UpdateCost` is called for each neighboring vertex s' to check if using s could lead to a shorter path. In that check A* only considers connecting s' directly to s (Algorithm 5, note that $c(s^a, s^b)$ refers to the edge cost between two vertices s^a and s^b). As shown in Algorithm 6, Theta* considers connections from s' to both s and `predecessor`(s). By virtue of the triangle inequality, a connection to `predecessor`(s) – when available – is guaranteed to yield a candidate g cost that is equal to or lower

Algorithm 4: Heuristic-guided search over vertices

```
open ← $\emptyset$
closed ← $\emptyset$
$g(s^{\text{start}}) \leftarrow 0$
predecessor($s^{\text{start}}$) ← $s^{\text{start}}$
open.$insert(s^{\text{start}}, g(s^{\text{start}}) + h(s^{\text{start}}))$
while $open \neq \emptyset$ do
    $s$ ← open.$pop()$
    if $s = s^{goal}$ then
        return PathFound
    end
    closed ← closed ∪ $\{s\}$
    foreach $s' \in neighbors(s)$ do
        if $s' \notin closed$ then
            if $s' \notin open$ then
                $g(s') \leftarrow \infty$
                predecessor($s'$) ← NULL
            end
            UpdateVertex($s, s'$)
        end
    end
end
return NoPathFound
Function UpdateVertex($s, s'$) is
    Status ← UpdateCost($s, s'$)
    if Status = Changed then
        if $s' \in open$ then
            open.$remove(s')$
        end
        open.$insert(s', g(s') + h(s'))$
    end
end
```

Algorithm 5: Definitions for A*

```
Function UpdateCost($s, s'$) is
    if $g(s) + c(s, s') < g(s')$ then
        predecessor($s'$) ← $s$
        $g(s') \leftarrow g(s) + c(s, s')$
        return Changed
    end
    return Unchanged
end
```

Algorithm 6: Definitions for Theta*

```
Function UpdateCost($s, s'$) is
    $s^p$ ← predecessor($s$)
    if LineOfSight($s^p, s'$) then
        // Ray traced connection
        if $g(s^p) + c(s^p, s') < g(s')$ then
            predecessor($s'$) ← $s^p$
            $g(s') \leftarrow g(s^p) + c(s, s')$
            return Changed
        end
    else
        // Direct neighbor connection
        if $g(s) + c(s, s') < g(s')$ then
            predecessor($s'$) ← $s$
            $g(s') \leftarrow g(s) + c(s, s')$
            return Changed
        end
    end
    return Unchanged
end
```

than a direct connection to s. Therefore, Theta* only considers connecting to direct neighbor s when its `predecessor`(s) is not visible from s'.

B. Lazy visibility checking extension

As demonstrated by LazyTheta*[19], lazy visibility checking can improve the runtime of Theta* [4] by over an order of magnitude without significantly increasing path length. This technique can also be applied to our multi-resolution any-angle planner to achieve similar benefits. To implement this extension, we modify the `UpdateCost` function (Alg. 3) by assuming that `LineOfSight`($s^p, \mathcal{V}'$) is always true during the initial cost update (Line 5). When the search later expands $\mathcal{V}'$ (Alg. 1 Li. 7), this assumption is verified. If the visibility check fails, the `predecessor` of $\mathcal{V}'$ is updated to the best direct neighbor, introducing only a minor detour as the best neighbor's center is typically very close to the optimal path. The remainder of the algorithm remains unchanged.

C. Formal statements

To the authors' best knowledge, no proofs or formal guarantees on completeness, optimality, or time complexity have appeared in prior literature for either Theta* [4] or Lazy-Theta* [19]. In the statements that follow, we therefore discuss observations applying to both Theta* and our multi-resolution extension.

1) Completeness: Theta* operates on a supergraph of the grid's 26-connected adjacency graph considered by A*, as its additional line-of-sight check can only *add* edges. Since each vertex receives at most one additional edge, the supergraph's branching factor remains finite. Furthermore, all edge weights are non-zero. Therefore, Theta* preserves A*'s resolution completeness guarantee.

Our method losslessly converts the traversability grid into octree leaves and defines any two leaves as adjacent if they are both traversable and touch in any form, i.e., have an overlapping face, edge, or corner. For any two connected vertices on the fixed-resolution grid, the respective octree leaves that cover them will also be connected. Our representation, therefore, maintains the connectivity of the original space, preserving the resolution completeness of running A* on the original high-resolution grid.

2) Optimality: We build on the previous section's insight that Theta* operates on a supergraph of A*'s graph. New edges are only introduced if they reduce the evaluated vertex's cost-to-come. Moreover, the chosen Euclidean distance heuristic remains admissible and consistent. Theta* is therefore guaranteed to find paths that are at least as good as those of A* and, as proven in [19], it can find solutions that are up to $\approx 13\%$ shorter.

Naively applying A* or Theta* to the center points of an octree's leaves results in paths that can be arbitrarily suboptimal with respect to their fixed-resolution counterparts, as illustrated in Figure 2. Our method addresses this problem by leveraging an octree while still considering the high-resolution vertices covered by each leaf. Our algorithm's initialization procedure (Section IV-C) guarantees that all potentially optimal predecessors, at the chosen initialization resolution, are considered. During the multi-resolution search, our dynamic refinement procedure (Section IV-E) increases the resolution of each subvolume until the cost to reach each high-resolution vertex it covers is suboptimal by a factor of at most 2ϵ. This factor of two follows from the fact that Equation (3) is applied pairwise between all successive candidates. Since the goal is itself a vertex, the total path length will be within 2ϵ of Theta* and, by extension, A*.

Note that setting $\epsilon = 0$ and matching the initialization resolution to the fixed-resolution grid yields paths whose length is almost identical to those of Theta*, but that sometimes differ in terms of chosen predecessors. We believe this follows from the fact that Theta* is not guaranteed to find the optimal *any-angle* Euclidean shortest path. The predecessors chosen by Theta* and our algorithm are sensitive to the order in which the vertices or subvolumes are expanded, and this order will differ slightly due to small numerical and discretization differences in our multi-resolution formulation.

3) Time Complexity: Our method leverages a data structure that combines hash maps with fixed-maximum-depth octrees [18], achieving the same $O(1)$ access and insertion complexity as regular grids. Consequently, the time complexity of expanding a subvolume in our method matches Theta*'s vertex expansion complexity. Theta* performs up to 26 predecessor-visibility checks per expanded vertex, each potentially having a complexity that grows linearly in the checked distance. However, very long-distance visibility checks can be avoided without significantly affecting path length, since the detour introduced by splitting a long edge into two and aligning their middle vertex to the grid is negligible. Our implementations of Theta* and our multi-resolution extension therefore reject checks beyond a fixed maximum distance, ensuring their worst-case runtime complexity matches that of A* on a regular grid. Experimentally, we demonstrate that our method's runtime is substantially lower in practice (Table III).