MeshA*: Efficient Path Planning With Motion Primitives

Marat Agranovskiy[1], **Konstantin Yakovlev**[1, 2]

[1]St. Petersburg State University
[2]CogAILab
agrinscience@gmail.com, k.yakovlev@spbu.ru

Abstract

We study a path planning problem where the possible move actions are represented as a finite set of motion primitives aligned with the grid representation of the environment. That is, each primitive corresponds to a short kinodynamically-feasible motion of an agent and is represented as a sequence of the swept cells of a grid. Typically, heuristic search, i.e. A*, is conducted over the lattice induced by these primitives (lattice-based planning) to find a path. However, due to the large branching factor, such search may be inefficient in practice. To this end, we suggest a novel technique rooted in the idea of searching over the grid cells (as in vanilla A*) simultaneously fitting the possible sequences of the motion primitives into these cells. The resultant algorithm, MeshA*, provably preserves the guarantees on completeness and optimality, on the one hand, and is shown to notably outperform conventional lattice-based planning (x1.5-x2 decrease in the runtime), on the other hand.

Code — https://github.com/PathPlanning/MeshAStar
Extended version — https://arxiv.org/abs/2412.10320

Introduction

Kinodynamic path planning is a fundamental problem in AI, automated planning, and robotics. Among the various approaches to tackle this problem, the following two are the most widespread and common: sampling-based planning (Karaman and Frazzoli 2011; Sakcak et al. 2019) and lattice-based planning (Pivtoraiko and Kelly 2005; Pivtoraiko, Knepper, and Kelly 2009). The former methods operate in continuous space, rely on the randomized decomposition of the problem into smaller sub-problems, and are especially advantageous in high-dimensional planning (e.g., planning for robotic manipulators). Still, they provide only probabilistic guaranties of completeness and optimality. Lattice-based planners rely on the discretization of the workspace/configuration space and provide strong theoretical guaranties with respect to this discretization. Consequently, they may be preferable when the number of degrees of freedom of the agent is not high, such as in mobile robotics, where one primarily considers the coordinates and the heading of the robot. In this work, we focus on the lattice-based methods for path planning in (x, y, θ).

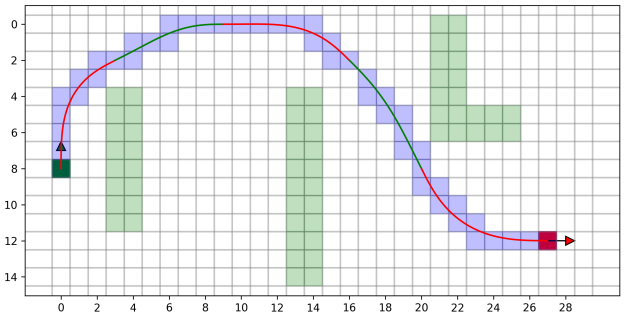

Figure 1: Example of the path planning problem. The workspace is discretized to a grid, where the green cells correspond to the obstacles and the white ones represent the free space. The green cell with an arrow denotes the start state (position and heading), while the red cell – the goal one. The path is composed of the primitives (green and red segments). The swept cells are shown in blue.

Lattice-based planning methods reason over the so-called *motion primitives* – the precomputed kinodynamically-feasible motions from which the sought path is constructed – see Fig. 1. Stacked motion primitives form a *state lattice*, i.e., a graph, where the vertices correspond to the states of the agent and the edges correspond to the motion primitives. A shortest path on this graph may be obtained by algorithms, such as A* (Hart, Nilsson, and Raphael 1968), that guarantee completeness and optimality. Unfortunately, when the number of motion primitives is high (which is not uncommon in practice), searching over the lattice graph becomes computationally burdensome. To this end, in this work, we introduce a novel perspective on lattice-based planning.

We leverage the assumption that the workspace is represented as an occupancy grid (a standard practice for path planning) and search over this grid in a cell-by-cell fashion to form a sequence of cells such that a sought path, i.e. a sequence of the motion primitives, may be fitted into this sequence of cells. We introduce a dedicated technique to reason simultaneously about the grid cells and the motion primitives that pass through these cells within the search process.

Such reasoning allows us to decrease the branching factor, on the one hand, and to maintain the theoretical guarantees, on the other hand. Empirically, we show that the introduced path planning method, called MeshA*, is notably faster than the conventional A* and its lazy variant (this holds when the weighted heuristic is utilized as well).

Related Work

Various approaches to pathfinding consider a mobile agent's kinodynamic constraints (González et al. 2016). One prominent group is sampling-based planning. Classical representatives include the RRT algorithm (LaValle and Kuffner 2001), which rapidly explores the configuration space, its anytime modification RRT* (Karaman and Frazzoli 2010) aimed at converging to optimal solutions, and variants like Informed RRT* (Gammell, Srinivasa, and Barfoot 2014), which incorporates heuristics, RRT-Connect (Kuffner and LaValle 2000) for fast bidirectional planning, and $\mathrm{RRT^X}$ (Otte and Frazzoli 2014) for fast re-planning.

Another direction is lattice-based planning, where a set of motion primitives that respect the constraints of the agent is constructed, and then a suitable sequence of these primitives is sought. Primitives can be generated using B-splines (Flores and Milam 2006), the shooting method (Jeon, Karaman, and Frazzoli 2011), the covering method (Yakovlev et al. 2022), learning-based techniques (De Iaco, Smith, and Czarnecki 2019), and others. In this work, we follow the lattice-based approach, but assume the primitives are given in advance (for experiments, we use a simple Newton optimization method (Nagy and Kelly 2001)).

Our focus is on reducing path planning search effort. Similar goals are addressed by approaches like Jump Point Search (Harabor and Grastien 2011), which significantly accelerates search on an 8-connected grid, and the well-known WA* (Ebendt and Drechsler 2009), which provides suboptimal solutions more quickly using weighted heuristics.

Problem Statement

Consider a point-sized mobile agent moving in a 2D workspace $W \subset \mathbb{R}^2$ composed of a free space W_{free} and obstacles W_{obs}. The workspace is tessellated into a grid, where each cell (i, j) is either free or blocked.

State Representation. The state of the agent is defined by a 3D vector (x, y, ϕ), with coordinates $(x, y) \in W$ and heading angle $\phi \in [0, 360°)$. We assume that the latter can be discretized into a set $\Theta = \{\phi_1, ..., \phi_k\}$, allowing us to focus on discrete states $s = (i, j, \theta)$ that correspond to the centers of the grid cells $(i, j) \in \mathbb{Z}^2$, with $\theta \in \Theta$.

Motion Primitives. The kinematic constraints and physical capabilities of the mobile agent are encapsulated in *motion primitives*, each representing a short kinodynamically-feasible motion (i.e., a continuous state change). We assume these primitives align with the discretization, meaning that transitions occur between two discrete states, such as (i, j, θ) and (i', j', θ'). In other words, each motion starts and ends at the center of a grid cell, with its endpoint headings belonging to the finite set Θ. Each primitive is additionally associated with the *collision trace*, which is a sequence of cells swept by the agent when executing the motion, and *cost* which is a positive number (e.g. the length of the primitive).

For a given state $s = (i, j, \theta)$ there is a finite number of motion primitives that the agent can use to move to other states. Moreover, we assume that there can be no more than one primitive leading to each other state, which corresponds to the intuitive understanding of a primitive as an elementary motion. We also consider that the space of discrete states, along with the primitives, is *regular*; that is, for any $\delta_i, \delta_j \in \mathbb{Z}$ if there exists a motion primitive connecting (i, j, θ) and (i', j', θ') then there also exists one from $(i + \delta_i, j + \delta_j, \theta)$ to $(i' + \delta_i, j' + \delta_j, \theta')$. While the endpoints of such primitives are distinct, the motion itself is not. This means that the collision traces of these primitives differ only by a parallel shift of the cells, and their costs coincide. Thus, we can consider a canonical set of primitives, *control set*, from which all others can be obtained through parallel translation. We assume that such a set is finite and is computed in advance according to the specific motion model of the mobile agent. To avoid ambiguity, we clarify the usage of the term "primitive". Unless specified otherwise, it refers to a specific motion between two discrete states. When we intend to refer to a motion template, we will explicitly state that the primitive is *from the control set*. Such a template, instantiated at a discrete state, yields an exact primitive.

Path. A *path* is a sequence of motion primitives, where the adjacent ones share the same discrete state. Its *collision trace* is the union of the collision traces of the constituent primitives (shown as blue cells in Fig. 1). A path is *collision-free* if its collision trace consists of free cells only.

Problem. The task is to find a collision-free path from a given start state s_0 to a goal state s_f. We wish to solve this problem optimally, i.e. to obtain the least cost path, where the cost of the path is the sum of the costs of its primitives.

Method

A well-established approach to solve the given problem is to search for a path on a *state lattice* graph, where the vertices represent the discrete states and edges – the primitives connecting them. In particular, heuristic search algorithms of the A* family can be used for such pathfinding.

These algorithms iteratively construct a search tree composed of the partial paths (sequences of the motion primitives). At each iteration, the most prominent partial path is chosen for extension. Extension is done by expanding the path's endpoint – a discrete state (i, j, θ). This expansion involves considering all the primitives that can be applied to the state, checking which ones are valid (i.e. do not collide with the obstacles), computing the transition costs, filtering out the duplicates (i.e. the motions that lead to the states for which there already exist paths in the search tree at a lower or equal cost), and adding the new states to the tree. Indeed, as the number of available motion primitives increases, the expansion procedure (which is the main building block of a search algorithm) becomes computationally burdensome, and the performance of the algorithm degrades.

Partially, this problem can be addressed by the *lazy* approach, where certain computations associated with the expansion are postponed, most often – collision checking.

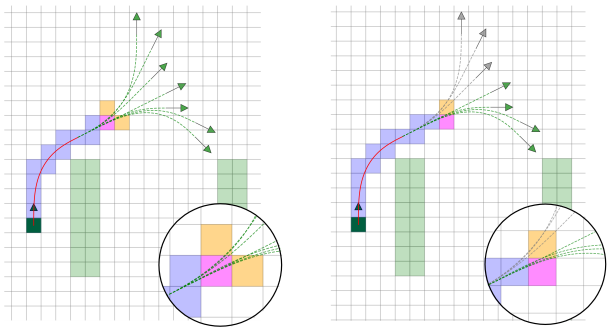

Figure 2: Definition of cell successors during the construction of a collision trace.

However, in environments with complex obstacle arrangements, searches that exploit lazy collision checking often extract invalid states (for which the collision check fails). Consequently, a significant amount of time is wasted on extra operations with the search tree, which is also time-consuming.

We propose an alternative approach. Instead of searching at the level of primitives, we search at the level of individual cells. During the search, we simultaneously reason about the motion primitives that can pass through the cells. This is achieved by defining a new search element that combines a cell with a set of primitives and a proper successor relationship. This approach allows for obtaining optimal solutions faster by leveraging its cell-by-cell nature.

Fig. 2 illustrates the intuition behind our search. On the left, a partial collision trace ends at a magenta cell under expansion. Assume this cell is reached by a red primitive, followed by any of the green ones, as their collision traces do not differ before this point. We store information about all primitives passing through a cell, forming what we call an *extended cell*. This augmentation allows us to infer a small set of successors: the orange cells along the paths of the green primitives. For these successors, we generate corresponding extended cells, propagating the information about the primitives that pass through them. Later, when the search expands one such successor (e.g., the cell to the right, see the right panel of Fig. 2), its propagated information (gray primitives are excluded, as they led to a different cell) determines the cell above as the only valid continuation.

The Search Space

To define the elements of our search space, first, consider a set of n arbitrary motion primitives from the control set, $prim_1$, ..., $prim_n$, and an index $k \in \mathbb{N}$ such that $\forall i \in \{1, \dots, n\} : \; k < U_{prim_i}$, where U_{prim_i} is the number of cells in the collision trace of $prim_i$. Now, the following set of pairs is called the **configuration of primitives**:

$$\Psi = \{(prim_1, k), (prim_2, k), ..., (prim_n, k)\},$$

and used to define elements of the search space.

Definition 1 (Extended cell)**.** *An element of our search space is an **extended cell**, formally defined as a tuple of a specific grid cell (i, j) and a configuration of primitives Ψ:*

$$u = (i, j, \Psi)$$

Algorithm 1: Building the Initial Configuration

Input: Discrete angle $\theta \in \Theta$
Function name: InitConf(θ)

```
Ψ ← ∅
for all prim ∈ ControlSet do
    if prim emerges in θ then
        Ψ.add{(prim, 1)}
return Ψ
```

Recall that primitives from the control set act as motion templates, which, when instantiated at specific discrete states, yield specific primitives. Thus, the conceptual meaning of an extended cell is as follows: for each pair $(prim, k) \in \Psi$, we consider a copy of $prim$ traversing cell (i, j) such that this cell is the k-th in its collision trace. Thus, an extended cell captures information both about the grid cell itself and about the motion primitives passing through it. The magenta cells in Fig. 2 with green primitives passing through them illustrate this concept. The *projection* of the extended cell $u = (i, j, \Psi)$ is the grid cell (i, j).

In practice, instead of storing the configuration of primitives directly, a single number, assigned by indexing all possible configurations, can be used (see Extended version).

Definition 2 (Initial Configuration)**.** *For a given heading θ, the initial configuration Ψ_θ is defined as:*

$$\Psi_\theta = \{(prim_1, 1), \dots, (prim_r, 1)\},$$

where $prim_1, \dots, prim_r$ are all primitives from the control set with initial heading θ (see details in Algorithm 1).

An extended cell containing such a configuration is an *initial extended cell*. The latter can be viewed as a direct analog to a discrete state.

Successors

To define the successor relationship between the extended cells, let us first denote the *displacement* by

$$\Delta_k^{prim} = (i' - i, j' - j)$$

This is the change in grid coordinates when transitioning from the k-th cell to the $(k + 1)$-th cell in the collision trace of $prim$. Throughout the paper, we call the process of making such a transition a *step along the primitive*. Thus, each step along the primitive is characterized by its displacement.

Intuitively, the successors of an extended cell are obtained simply as the results of steps along each primitive from the current configuration (see Fig. 2). The formal definition is divided into two parts, depending on the type of successor.

Definition 3 (Initial Successor)**.** *Let $u = (i, j, \Psi)$ be an extended cell. Initial extended cell $v = (i', j', \Psi_\theta)$ is called an **initial successor** of u if the following condition holds:*

$$\exists (prim, k) \in \Psi \;\; \textit{such that:} \;\; k = U_{prim} - 1,$$

$$\Delta_k^{prim} = (i' - i, j' - j) \;\; \textit{and } prim \textit{ ends at angle } \theta$$

This condition ensures the existence of $prim$ in Ψ that completes at (i', j') with heading θ. At this point, it can be extended by a whole bundle of primitives from Ψ_θ, all starting with the same angle θ.

Definition 4 (Regular Successor)**.** *Let $v = (i', j', \Psi')$ and $u = (i, j, \Psi)$ be extended cells, with v not being initial. Then v is a* ***successor*** *of u if the following hold simultaneously:*

1. ***Predecessor Completeness:*** *For all primitives in Ψ', their preceding cells must exist in the predecessor.*

   $\forall (prim, k) \in \Psi'$ *the following holds:*

   $(prim, k-1) \in \Psi$ *and* $(i' - i, j' - j) = \Delta^{prim}_{k-1}$

2. ***Successor Coverage:*** *All valid continuations from Ψ to the successor's projection must be included in Ψ'.*

   $\forall (prim, k) \in \Psi$ *such that* $k < U_{prim} - 1$ *and*

   $\Delta^{prim}_{k} = (i' - i, j' - j)$ *the following holds:*

   $(prim, k+1) \in \Psi'$

Condition (2) requires that all primitives of the predecessor leading to the projection cell of the successor are present in its configuration, while Condition (1) ensures that there are no other primitives in the successor.

Transition Costs

To ensure compatibility with standard lattice-based methods, transition cost between extended cells is derived from the costs of the underlying motion primitives. Let c_{prim} denote the cost of a primitive $prim$ from the control set.

Definition 5 (Transition Cost)**.** *Let v be a successor of an extended cell u. The transition cost from u to v is defined as:*

$$cost(u, v) = \begin{cases} c_{prim}, & \text{if } v \text{ is an initial successor} \\ 0, & \text{otherwise} \end{cases}$$

In the first case, $prim$ is a primitive terminating at v that satisfies the conditions of Definition 3. If multiple such primitives exist, any of their costs can be chosen. We will show later that this ambiguity cannot arise in any relevant case.

Algorithm 2 details the successor generation for a given extended cell $u = (i, j, \Psi)$. It uses a dictionary `Confs` (line 2) to temporarily store the forming configurations of non-initial successors, mapped by their displacement from u. A set `Successors` is initialized (line 3) to hold the resulting pairs of all successors and their transition costs.

The main loop (lines 4-15) iterates over each pair $(prim, k)$ in the configuration Ψ. For each pair, it computes the displacement (a, b) to the next cell in the primitive's collision trace (line 5). This displacement determines the grid coordinates $(i+a, j+b)$ of a potential successor cell relative to u. Based on whether the primitive terminates (checked in line 6), one of two cases is handled.

Case 1: Initial Successor (lines 6-10). If a primitive is at its final step ($k = U_{prim} - 1$), an initial successor is generated and immediately stored in `Successors` (line 10) according to Definition 3. The angle θ at which the primitive ends determines the initial configuration $\Psi_1 = \text{INITCONF}(\theta)$ for this successor (line 8). The transition cost is set to c_{prim}, the cost of the primitive itself, as this motion is now complete (see Definition 5).

Case 2: Regular Successor (lines 11-15). If a primitive does not terminate ($k < U_{prim} - 1$), it contributes to a non-initial, or regular, successor. The displacement (a, b) is used as a key in the `Confs` dictionary. If this displacement is encountered for the first time (line 11), a new entry is created, and its configuration is initialized with the current primitive at its next step, $(prim, k+1)$ (lines 12-13). If the key (a, b) already exists (line 14), it signifies that multiple primitives from Ψ lead to the same grid cell $(i + a, j + b)$. In this case, $(prim, k+1)$ is added to the existing configuration for this displacement (line 15). Thus, for each displacement (a, b), the `Confs` dictionary groups all non-terminating primitives from Ψ whose next step corresponds to this displacement. This process forms the complete configuration for each potential non-initial successor and precisely matches the conditions of Definition 4.

Finally (lines 16-19), the algorithm iterates through the `Confs` dictionary. For each displacement (a, b) and its aggregated configuration Ψ_2, it forms a single non-initial successor $v_2 = (i + a, j + b, \Psi_2)$. The transition cost to such a successor is 0, per Definition 5. Thus, the pair $(v_2, 0)$ is added to the `Successors` set (line 19).

Algorithm 2: Generating Successors of an Extended Cell

Input: Extended cell u
Output: Set of pairs of successors and transition costs
Function name: GETSUCCESSORS(u)

```
1:  i, j, Ψ ← u
2:  Confs ← {}                                  ▷ Dictionary
3:  Successors ← ∅                              ▷ Set of successors and costs
4:  for all (prim, k) ∈ Ψ do
5:      (a, b) ← Δ^prim_k
6:      if k = U_prim − 1 then                  ▷ Case 1
7:          θ ← angle at which prim ends
8:          Ψ_1 ← INITCONF(θ)
9:          v_1 ← (i + a, j + b, Ψ_1)
10:         Successors.add{(v_1, c_prim)}
11:     else if (a, b) ∉ Confs then             ▷ Case 2
12:         conf_new ← {(prim, k + 1)}
13:         Confs[(a, b)] ← conf_new
14:     else
15:         Confs[(a, b)].add{(prim, k + 1)}
16: for all (a, b) ∈ Confs do
17:     Ψ_2 ← Confs[(a, b)]
18:     v_2 ← (i + a, j + b, Ψ_2)
19:     Successors.add{(v_2, 0)}
20: return Successors
```

MeshA*

Having defined the elements of the search space as well as the successor relationship, we obtain a directed weighted graph. We term this the *mesh graph* to distinguish it from both the environment's grid representation and the standard lattice graph of motion primitives. This graph structure proves essential for our subsequent theoretical analysis.

We can utilize a standard heuristic search algorithm, i.e. A*, to search for a path on this graph. We will refer to this approach as *MeshA**. Next we will show that running MeshA* leads to finding the optimal solution, which is equivalent to the one found by A* on the lattice graph.

Algorithm 3: Trajectory Reconstruction

Input: Path $u_1, u_2, \ldots, u_N$ in the mesh graph between initial extended cells
Output: Trajectory (chain of primitives)

```
Traj ← ∅
p ← u_1
for all l = 2, 3, ..., N do
    if u_l is initial then                    ▷ u_l is next initial after p
        i_1, j_1, Ψ_{θ_1} ← p
        i_2, j_2, Ψ_{θ_2} ← u_l
        s_1 ← (i_1, j_1, θ_1)                 ▷ Obtain discrete states
        s_2 ← (i_2, j_2, θ_2)
        prim ← primitive from s_1 to s_2
        Traj.add{prim}
        p ← u_l
return Traj
```

Theoretical Results

We now establish the equivalence between searching on the mesh graph and searching on the state lattice, which infers that finding an optimal path on the state lattice is equivalent to finding an optimal path on the mesh graph followed by trajectory reconstruction.

This section provides only proof sketches for brevity, while the detailed proofs are in the Extended version.

Lemma 1 (Uniqueness of Path Cost)**.** *For any path on the mesh graph starting from an initial extended cell, the cost of each transition is uniquely defined.*

Proof Sketch. The only non-zero costs occur on transitions into initial extended cells. We prove by contradiction that the primitive inducing such a transition is unique. The key insight is that any primitive can be traced backward along the mesh path to its origin. If two distinct primitives induced the same transition, this backward tracing procedure would show they connect the same discrete start and end states, which contradicts our problem statement. □

Theorem 2 (From State Lattice to Mesh Graph)**.** *Let there be a trajectory on the state lattice from a discrete state $s_a = (i_a, j_a, \theta_a)$ to $s_b = (i_b, j_b, \theta_b)$. Then there exists a path on the mesh graph from the initial extended cell $u_a = (i_a, j_a, \Psi_{\theta_a})$ to another initial one $u_b = (i_b, j_b, \Psi_{\theta_b})$ that satisfies the following conditions:*

1. *The cost of this path is equal to the cost of the trajectory.*
2. *The projections of the vertices along this path precisely form the collision trace of this trajectory.*

Proof Sketch. Proof is constructive, by induction on the number of primitives. For the base case of a single primitive, the path is constructed by starting at u_a and iteratively applying the successor definition to step along the primitive's collision trace cell by cell. Each step to a non-final cell of the trace generates a regular successor with zero cost. The final step generates an initial successor u_b with cost c_{prim}, ensuring the total path cost matches the primitive. The inductive step shows that a path for a longer trajectory is simply the concatenation of the mesh paths for its constituent primitives, ensuring the total cost and trace are preserved. □

Theorem 3 (From Mesh Graph to State Lattice)**.** *For any path on the mesh graph from an initial extended cell $u_a = (i_a, j_a, \Psi_{\theta_a})$ to another initial $u_b = (i_b, j_b, \Psi_{\theta_b})$, Algorithm 3 reconstructs a trajectory composed of primitives (corresponding to the path in the state lattice) that transition from the discrete state $s_a = (i_a, j_a, \theta_a)$ to $s_b = (i_b, j_b, \theta_b)$ and satisfy the following conditions:*

1. *The cost of this trajectory is equal to the cost of the path in the mesh graph.*
2. *The collision trace of this trajectory coincides with the projections of the vertices along this mesh graph path.*

Proof Sketch. The proof is constructive, formalizing the reconstruction algorithm 3. The core idea is that any mesh path between initial cells is uniquely decomposed into segments, each connecting two consecutive initial cells in the path. We establish that each segment corresponds to a single, unique motion primitive. Concatenating these primitives reconstructs the full trajectory, and the cost is preserved as it is only incurred at the end of each segment. □

Theorem 4 (Equivalence of Optimal Pathfinding)**.** *The search for a least-cost, collision-free trajectory between discrete states $s_a = (i_a, j_a, \theta_a)$ and $s_b = (i_b, j_b, \theta_b)$ is equivalent to performing two steps:*

1. *Find a least-cost path on the mesh graph between the corresponding initial extended cells $u_a = (i_a, j_a, \Psi_{\theta_a})$ and $u_b = (i_b, j_b, \Psi_{\theta_b})$. This path must be collision-free, meaning the projection of every vertex on the path is a free grid cell.*
2. *Recover the trajectory from this path using Algorithm 3.*

The resulting trajectory will be optimal and collision-free.

Proof Sketch. The proof rests on the bidirectional, cost-preserving correspondence from Theorems 2 and 3. Theorem 2 guarantees that any optimal collision-free trajectory can be mapped to a mesh path of identical cost. Since the path's projections match the trajectory's trace, this path is also collision-free. Conversely, Theorem 3 ensures that any optimal collision-free mesh path can be reconstructed into a collision-free trajectory of the same cost. Since a path exists in one space if and only if a path of the same cost exists in the other, their optimal costs must be identical, making the proposed two-step procedure both correct and complete. □

On The Efficiency Of MeshA*

MeshA* is not a new search algorithm, but the standard A* applied to a novel search space, the mesh graph (hence, we omit its pseudocode). Theorem 4 established that A* on the mesh graph (MeshA*) is equivalent to A* on the state lattice (LBA*). We now show how the inherent structure of the mesh graph enables cell-level search-space pruning techniques unavailable to LBA*. We believe this structural advantage is the main reason MeshA* notably outperforms LBA*, as shown later in the Empirical Evaluation section.

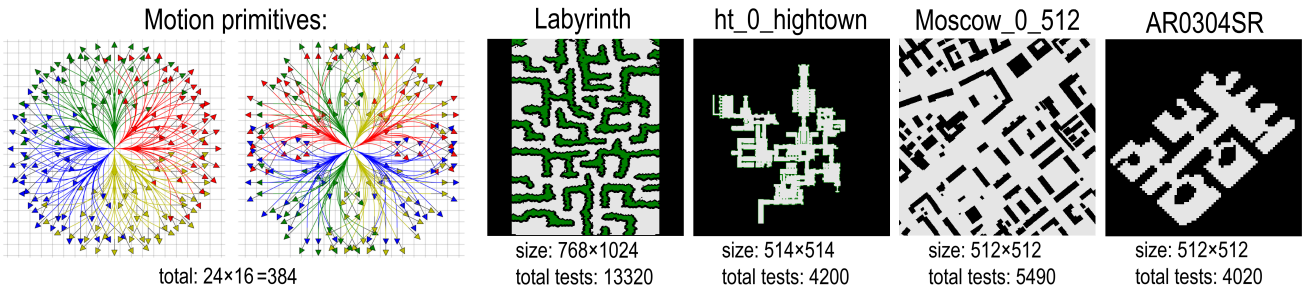

Figure 3: Experimental setup: control set and MovingAI maps.

We first define the heuristic function for MeshA*. To this end, we introduce a new concept related to an extended cell.

Definition 6 (Primitive Endpoints)**.** *For a non-initial extended cell $u = (i, j, \Psi)$, the set $\texttt{Finals}(u)$ is defined as:*

$$\texttt{Finals}(u) = \{(u_{prim}, c_{prim}) \mid \exists (prim, k) \in \Psi\}$$

where u_{prim} is the initial extended cell where the instance of $prim$ passing through u terminates.

Intuitively, $\texttt{Finals}(u)$ captures the information about completing each of the primitives passing through u. This is used to define the heuristic function for MeshA*.

Definition 7 (Heuristic for MeshA*)**.** *Let h_{LBA*} be an (admissible and consistent) heuristic function for lattice-based A* (LBA*). Then, for any extended cell $u = (i, j, \Psi)$, the heuristic function for the MeshA* is:*

$$h(u) = \begin{cases} h_{LBA*}(i, j, \theta), & \text{if } u \text{ is initial with } \Psi = \Psi_\theta \\ \min_{(u_0, c_0) \in \textit{Finals}(u)} \{h(u_0) + c_0\}, & \text{otherwise} \end{cases}$$

This definition preserves the admissibility and consistency of h_{LBA*}. For initial cells, it matches h_{LBA*} directly. For non-initial cells, since any path from u must continue through one of its primitive endpoints u_0 (at a cost of c_0 to complete), taking the minimum $\min(c_0 + h(u_0))$ over all such options enforces consistency by definition.

On the other hand, such a heuristic leads to a more extensive pruning of the search space for MeshA* for the following reason. For any non-initial cell u $h(u)$ estimates the cost of the best possible trajectory that can be completed from u. Crucially, this allows us to evaluate the promise of each potential primitive endpoint in $\texttt{Finals}(u)$ independently. If one primitive leads towards a region with a high heuristic cost, the search will naturally deprioritize such paths, effectively abandoning that branch of the search long before the primitive is fully traversed. In other words, unlike LBA* the introduced method, MeshA*, may detect the unperspective search branches much earlier, thanks to the cell-by-cell nature of the search.

Next, the structure of the mesh graph also enables a powerful terminal pruning rule. A non-initial extended cell u can be safely pruned (i.e., its expansion skipped) if all of its endpoint cells in $\texttt{Finals}(u)$ have already been expanded. Indeed, since any path from u must pass through one of these endpoints, and our search strategy with a consistent heuristic guaranties optimality (or bounded suboptimality in the weighted case) without re-openings (Chen and Sturtevant 2021), further exploration from u is redundant.

Empirical Evaluation

Setup. In the experiments we utilize 16 different headings and generate 24 motion primitives for each heading using the car-like motion model. Experiments are conducted on four MovingAI benchmark (Sturtevant 2012) maps of varying topology. Start-goal pairs are taken from benchmark scenarios, with three randomly generated headings per pair, yielding over 25,000 instances in total. See Figure 3.

The following algorithms were evaluated:

1. **LBA*** (short for Lattice-based A*): The standard A* algorithm on the state lattice, serving as the baseline.
2. **LazyLBA***: The same algorithm that conducts collision-checking lazily.
3. **MeshA*** (ours): Running A* on the mesh graph.

The cost of each primitive is its length, and the heuristic is the Euclidean distance. We test heuristic weights $w \in \{1, 1.1, 2, 5, 10\}$, where larger weights speed up search but increase solution cost (Ebendt and Drechsler 2009).

To ensure a fair comparison, we implement all algorithms in C++ with identical data structures (e.g., priority queue) and the same underlying A* logic for both the mesh graph (MeshA*) and the state lattice (LBA*). Both implementations include *g-value pruning* (i.e., avoiding exploration a state that already has a known better g-value). To avoid `DecreaseKey` operations in the `PriorityQueue` we perform this "lazily": all successors are added to `OPEN`, but a search node is discarded upon extraction if its state is already in `CLOSED`. This strategy correctly guarantees optimality (for $w = 1$) and bounded suboptimality (for $w > 1$) without re-opening (Chen and Sturtevant 2021).

All experiments were conducted on AMD EPYC 7742 under identical conditions. We primarily measure runtime, as other metrics like nodes generated and expansions are not directly comparable due to fundamentally different search approaches: LBA* expands motion primitives while MeshA* expands extended cells. Additionally, we report *checked cells* (total cells in collision traces examined) as a processor-independent metric that highlights MeshA*'s cell-by-cell search advantage.

Results. Fig. 4 illustrates the median runtime of each algorithm as a ratio to LBA*. That is, the runtime of LBA* is considered 100% in each run, and the runtime of the other solvers is divided by this value. Thus, the lower the line is

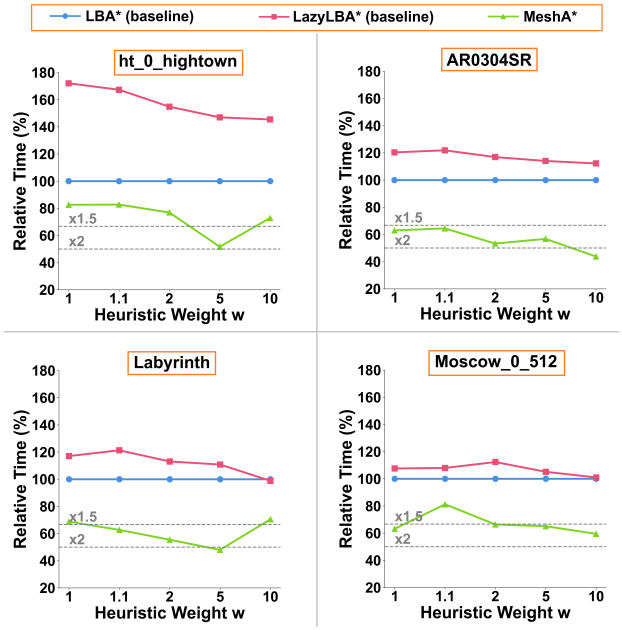

Figure 4: Median runtime of the evaluated algorithms (relative to the runtime of LBA*). The lower – the better. 50% corresponds to x2 speed-up.

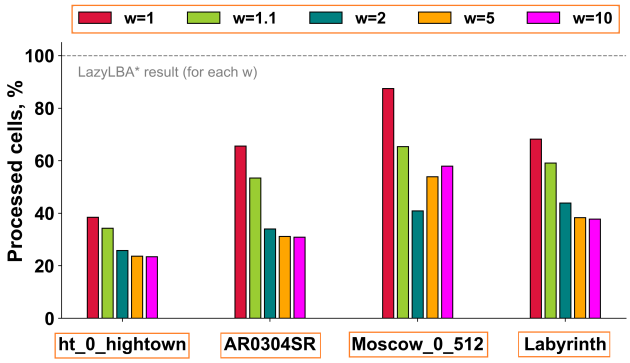

Figure 5: Median number of processed grid cells (sum over all collision traces) for MeshA* relative to LazyLBA*

Algorithms	ht_0_hightown			
	$w = 1$	$w = 2$	$w = 5$	$w = 10$
LBA*	100.0	105.7	110.0	113.2
LazyLBA*	100.0	105.7	110.0	113.2
MeshA*	100.0	109.2	117.6	122.3

Table 1: Median relative costs of the trajectories found as a percentage of the optimal cost.

on the plot – the better. Clearly, MeshA* consistently outperforms the baselines. For $w = 1$, it consumes on average 60%-80% of the LBA* runtime (depending on the map). That is, the speed-up is up to 1.5x. When the heuristic is weighted, this gap gets even more pronounced, and we can observe x2 speed-up.

Interestingly, LazyLBA* consistently performs worse than LBA*. This can be attributed to the simplicity of collision checking in our experiments, which involves verifying that the cells in the collision trace of each primitive are not blocked. This is a quick check in our setup, and postponing it does not make sense. Moreover, lazy collision checking begins to incur additional time costs from numerous pushing and popping of unnecessary vertices to the search queue, which standard LBA* prunes through collision checking. This effect is especially visible on the `ht_0_hightown` map, which contains numerous narrow corridors and passages where many primitives lead to collisions.

Fig. 5 demonstrates another crucial advantage of MeshA*. Remarkably, MeshA* processes only $\approx$ 50% of the cells that LazyLBA* examines. We compare against LazyLBA* as it processes fewer cells than standard LBA* by deferring collision checks until primitive expansion. This significant reduction highlights MeshA*'s cell-by-cell search advantage: it can terminate unpromising primitives early, while LBA* must process entire primitives even when only their initial segments are relevant (see prev. section).

The costs of the obtained solutions are summarized in Table 1 for the `ht_0_hightown` map (results for other maps are similar and provided in the Extended version). Each cell shows the median cost relative to the optimal one, thus lower values are better (100% indicates optimal solutions). As expected, LBA*, LazyLBA*, and MeshA* are optimal.

Another observation is that MeshA*'s solution costs increase slightly more with rising w than LBA*'s. This is because LBA* computes the heuristic only twice per primitive (start/end), while MeshA* computes it at every cell. Thus, the weight w has a more pronounced effect on MeshA*. This same property, however, explains MeshA*'s improved performance with weighted heuristics: the more frequent evaluations allow for more aggressive pruning of unpromising directions, achieving the demonstrated 2x speedup.

Overall, our experiments confirm that the suggested search approach, MeshA*, consistently outperforms the baselines and is much faster in practice (1.5x faster when searching for optimal solutions and 2x faster when searching for the suboptimal ones).

Conclusion

In this paper, we have considered a problem of finding a path composed of the motion primitives that are aligned with the grid representation of the workspace. We have suggested a novel way to systematically search for a solution by reasoning over the sequences of augmented grid cells rather than sequences of motion primitives. The resultant solver, MeshA*, is provably complete and optimal, and is notably faster than the regular A* searching on motion primitives.

Despite having considered path planning in 2D when the agent's state is defined as (x, y, θ), the idea that stands behind MeshA* is applicable to pathfinding in 3D as well as to the cases where the agent's state contains additional variables as long as they can be discretized. Adapting MeshA* to such setups is a prominent direction for future work.

Acknowledgements

This work was supported by the Ministry of Economic Development of the Russian Federation (Agreement No. 139-15-2025-007, dated April 16, 2025; ID: 000000C313925P3O0002)

References

Chen, J.; and Sturtevant, N. 2021. Necessary and Sufficient Conditions for Avoiding Reopenings in Best First Suboptimal Search with General Bounding Functions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35: 3688–3696.

De Iaco, R.; Smith, S. L.; and Czarnecki, K. 2019. Learning a Lattice Planner Control Set for Autonomous Vehicles. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, 549–556.

Ebendt, R.; and Drechsler, R. 2009. Weighted A* search – unifying view and application. *Artificial Intelligence*, 173(14): 1310–1342.

Flores, M.; and Milam, M. 2006. Trajectory generation for differentially flat systems via NURBS basis functions with obstacle avoidance. In *2006 American Control Conference*, 7 pp.–.

Gammell, J. D.; Srinivasa, S. S.; and Barfoot, T. D. 2014. Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2997–3004.

González, D.; Pérez, J.; Milanés, V.; and Nashashibi, F. 2016. A Review of Motion Planning Techniques for Automated Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4): 1135–1145.

Harabor, D. D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding On Grid Maps. *Proceedings of the AAAI Conference on Artificial Intelligence*.

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

Jeon, J. h.; Karaman, S.; and Frazzoli, E. 2011. Anytime computation of time-optimal off-road vehicle maneuvers using the RRT*. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, 3276–3282.

Karaman, S.; and Frazzoli, E. 2010. Incremental Sampling-based Algorithms for Optimal Motion Planning. *ArXiv*, abs/1005.0416.

Karaman, S.; and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7): 846–894.

Kuffner, J. J.; and LaValle, S. M. 2000. RRT-connect: An efficient approach to single-query path planning. *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, 2: 995–1001 vol.2.

LaValle, S.; and Kuffner, J. 2001. Randomized Kinodynamic Planning. *I. J. Robotic Res.*, 20: 378–400.

Nagy, B.; and Kelly, A. 2001. Trajectory generation for car-like robots using cubic curvature polynomials. *Field and Service Robots*.

Otte, M. W.; and Frazzoli, E. 2014. RRTX: Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles. In *Workshop on the Algorithmic Foundations of Robotics*.

Pivtoraiko, M.; and Kelly, A. 2005. Efficient constrained path planning via search in state lattices. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 1–7. Munich Germany.

Pivtoraiko, M.; Knepper, R. A.; and Kelly, A. 2009. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(3): 308–333.

Sakcak, B.; Bascetta, L.; Ferretti, G.; and Prandini, M. 2019. Sampling-based optimal kinodynamic planning with motion primitives. *Autonomous Robots*, 43(7): 1715–1732.

Sturtevant, N. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2): 144 – 148.

Yakovlev, K. S.; Andreychuk, A. A.; Belinskaya, J. S.; and Makarov, D. A. 2022. Safe Interval Path Planning and Flatness-Based Control for Navigation of a Mobile Robot among Static and Dynamic Obstacles. *Automation and Remote Control*, 83(6): 903–918.