Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische Friedrich-Wilhelms-Universität Bonn

Bachelor Thesis

Dynamic UAV Trajectory Planning using Multi-Resolution A*

Author:
Björn Seine

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Dr. Marcell Missura

Date: June 12, 2022

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Place, Date

Signature

Abstract

Autonomous operation of unmanned aerial vehicles (UAVs) in obstacle-rich environments requires algorithms that can quickly generate dynamically feasible, efficient trajectories. Generally, search-based algorithms become significantly more expensive with increasing dimensionality of the search space, making them unfeasible for frequent replanning of higher order trajectories.

We investigate using Multi-Resolution A*, an improvement of the A* algorithm that plans on several levels of resolution simultaneously, to quickly calculate trajectories in a 6D search space. Trajectories are modeled as a series of motion primitives generated by applying constant acceleration for a fixed time interval. Resolution levels are differentiated by the duration of their motion primitives, and we decide for each expansion at what resolution we expand. We develop several policies for that choice as well as functions to detect and limit search stagnation. Finally, we compare the performance of our design choices to each other, and compare our method to A* searches at different resolutions and to a local multiresolution approach.

Contents

1 Introduction

Unmanned aerial vehicles (UAVs) are being used for an increasing number of tasks, including exploration, inspection, or delivery in a wide variety of settings. Many of these tasks require autonomous navigation in unknown or partially unknown environments. To enable this, algorithms capable of calculating fast, dynamically viable trajectories within a short time frame are required. This is a difficult task, since the search times required by traditional search algorithms like A* tend to increase exponentially with the dimensionality of the search space. Many existing methods circumvent this issue by planning in a 3D position space and refining the solution into a dynamically feasible trajectory; however, the optimal trajectory may differ significantly from the optimal 3D path.

Liu et al. [5] propose a search-based method to generate resolution-complete (i.e. optimal in the discretized search space) higher order trajectories. This approach is based on short-duration motion primitives generated by applying constant control inputs for a short time period. They generate a state lattice graph by unrolling these motion primitives from a start state, and extract optimal trajectories from the graph using A*. However, due to the high dimensionality of the search space, this approach is computationally expensive, making it unfeasible for frequent re-planning of long paths at a fine resolution.

Du et al. [2] propose Multi-Resolution A* (MRA) to increase the speed of A* searches by planning at multiple levels of resolution simultaneously, allowing for fast search progress using coarse resolutions without sacrificing the ability to precisely plan using a fine resolution where required. We apply MRA to the motion primitive-based approach proposed by Liu et al. [5] to allow for fast calculation of efficient second-order trajectories. We develop and test several functions that determine the resolution at which each expansion is executed, along with several minor design adjustments. We also implement several methods to deal with search stagnation.

In this thesis, we will first define the problem we aim to solve (Chapter 2). We will then discuss a variety of existing approaches to motivate our work (Chapter 3). Afterwards, we will introduce the basic structure of our method, followed by a discussion of specific design choices we made (Chapter 4). This is followed by an evaluation of our design choices, leading into a comparison to an A* approach using

different resolutions and a local multiresolution approach proposed by Schleich and Behnke [8] (Chapter 5). Finally, we discuss the results of our work, as well as potential future improvements to be investigated (Chapter 6).

2 Problem Definition

We begin by defining the problem we aim to solve. For comparability, we base our notation on the works of Schleich and Behnke [8] and Liu et al. [5].

Our goal is to find an efficient second-order trajectory between two given states for an unmanned aerial vehicle (UAV) within a reasonable time frame. We are specifically planning for multicopters, which can change their acceleration vector without significant dynamic restrictions. Our approach is not directly applicable to winged UAVs as their flight dynamics do not match our model.

We model the state s of the UAV as a 6-tupel $s = (p, v) \in \mathbb{R}^6$ consisting of a 3D position p and 3D velocity v. While higher order representations could be used to model the dynamics of a real system even more closely, we restrict ourself to second-order systems to limit the dimensionality of the search space. We do not model the orientation (yaw) of the vehicle as it is not relevant for the planning task; it can be calculated in a post-processing step based on the desired trajectory. Additionally, we assume that the velocity component of the goal state is zero. This restriction permits us to use simpler heuristic functions which can be computed more quickly. However, the core concept of our method also allows for goal states with non-zero velocities.

Trajectories are modeled as a chain of motion primitives, generated by applying a constant acceleration u for a short time interval τ from an initial state s. The corresponding motion primitive $F_{u,s}$ linking s to the successor state $s' = F_{u,s}(\tau)$ can be expressed as a time-parameterized polynomial

$$F_{u,s}(t) = \begin{pmatrix} p + tv + \frac{t^2}{2}u \\ v + tu \end{pmatrix}, \text{ for } t \in [0, \tau]. \tag{2.1}$$

We define a finite control set $\mathcal{U} \subset \mathbb{R}^4$ of controls composed of discrete acceleration vectors and corresponding time intervals. By unrolling the motion primitives generated by these controls from an initial state $s_0 = (p_0, v_0)$, we generate a state lattice graph $\mathcal{G}(\mathcal{S}, \mathcal{E})$, where $\mathcal{S}$ denotes the resulting set of discretized states, and $\mathcal{E}$ denotes the set of motion primitives connecting the states. Fig. 2.1, taken from [5], shows a 2D example of motion primitives extending from an initial state based on different acceleration controls.

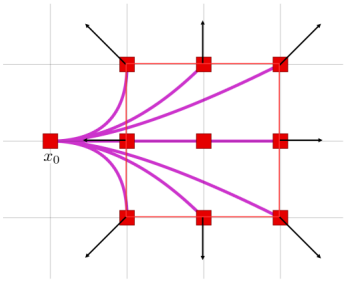

Figure 2.1: Motion primitives (magenta) resulting from application of nine different acceleration controls (black arrows) to an initial state x_0 with non-zero velocity towards the right. Red squares mark the position components of the discretized states. Image taken from [5].

We restrict the permissible velocity components to a free space $\mathcal{V}_{\text{free}} = [-v_{\max}, v_{\max}]^3$ based on the system's dynamics, not allowing velocities along individual axes to exceed a maximum velocity $v_{\max}$. The free position space $\mathcal{P}_{\text{free}} \subset \mathbb{R}^3$ is similarly restrained based on the map borders, known obstacles as well as safety margins around obstacles. Let $\mathcal{X}_{\text{free}} \subset \mathbb{R}^6$ be the free region of the state space, that is the space of states $s = (p, v)$ with position components $p \in \mathcal{P}_{\text{free}}$ and velocity components $v \in \mathcal{V}_{\text{free}}$. Thus, $\mathcal{X}_{\text{free}} := \mathcal{P}_{\text{free}} \times \mathcal{V}_{\text{free}} = \mathcal{P}_{\text{free}} \times [-v_{\max}, v_{\max}]^3$.

A motion primitive $F_{u,s}(t)$ is valid if $F_{u,s}(t) \in \mathcal{X}^{\text{free}}$ for all $t \in [0, \tau]$. A path from an initial state s_0 to a goal state s_{goal} in the state lattice graph $\mathcal{G}$ is valid if every motion primitive on the path is valid.

We define the cost of a motion primitive as the weighted sum of the control effort and the primitive duration:

$$C(F_{u,s}) = \|u\|_2^2 \tau + \rho\tau, \tag{2.2}$$

with the weight $\rho = 2$. The cost of a path is the sum of the primitive costs.

The goal is to find a valid path between an initial state s_0 and a goal state s_{goal} with low costs within a short planning time frame. In the interest of short planning times we allow for slightly suboptimal paths.

3 Related Works

Many tasks for which modern UAVs are being used require autonomous navigation in large, initially unknown or partially unknown environments including dynamically changing obstacles. For this purpose, algorithms that can quickly generate fast, collision-free, dynamically feasible trajectories are necessary. This is a challenging task due to the vast number of possible trajectories.

Sampling-based approaches such as Rapidly-exploring Random Tree Star (RRT*) can be used to quickly produce feasible solutions to a trajectory generation problem and can improve on the solution quality with additional computation time. Karaman et al. [4] propose one such algorithm, as do Zhang et al. [9]. However, due to the random nature of sampling-based approaches, the initial solution is likely far from optimal, and the time required to reach an acceptable solution may be too long for frequent replanning.

Traditional search-based algorithms typically require exponentially more time with increasing dimensionality of the search space, which makes them ill-suited to the generation of higher order trajectories. For this reason, many existing approaches first plan a path in 3D space and then refine this path into a viable trajectory. Jamieson and Biggs [3] generate smooth trajectories following a series of waypoints, though they do not consider obstacle avoidance. Cimurs and Suh [1] generate locally time-optimal trajectories from a path planner output using Bezier curves, their method includes obstacle avoidance. Because these approaches do not take the system dynamics into account, the resulting trajectory may be significantly slower than the optimal trajectory, and also require significantly more effort (resulting in a reduced operational time for the UAV). Fig. 3.1 taken from [5] illustrates the impact of ignoring system dynamics.

Liu et al. [5] propose a search-based approach that plans trajectories as series of short duration motion primitives, formed by applying a constant control input for a fixed time. The motion primitives induce a finite state lattice discretization on the state space, which then allows an optimal trajectory to be calculated by a graph search algorithm. The authors use A* search to calculate their solution. Because they include the system dynamics in the search instead of a post-processing step, they can generate resolution-complete (meaning optimal in the discretized search space) trajectories. However, this approach results in a very high dimensional

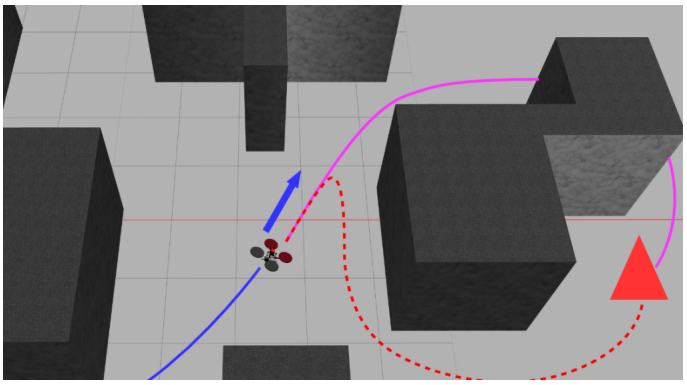

Figure 3.1: Generating trajectories based on the shortest path (red trajectory) can result in slow, high-effort trajectories, while a planner that takes system dynamics such as initial velocity (blue arrow) into account can produce significantly smoother and faster trajectories (magenta). Image taken from [5].

search space. Because of this, the A* search is too slow for frequent replanning of longer trajectories.

Schleich and Behnke [8] propose a local multiresolution approach to improve on the planning times required by [5]. Building on the motion primitives formulation proposed by Liu et al. , they propose a local multiresolution state lattice that uses motion primitives with a longer duration at greater distances from the UAV. By allowing certain sub-optimal expansions (Level-Based Expansion Scheme), this approach can significantly improve the required search times, allowing for frequent replanning of longer trajectories. This approach sacrifices some path quality through the local multiresolution as well as the level-based expansion scheme. Additionally, local multiresolution allows planning at a fine resolution only in the proximity of the UAV. While this is usually not a problem in unknown terrain because sensor-based knowledge of distant obstacles is not precise enough to enable precise planning, in partially known terrain the ability to plan trajectories through small openings and to precise destinations would be beneficial.

Du et al. [2] propose Multi-Resolution A* to accelerate the search speed of A* while maintaining bounded suboptimality for the solution. They use multiple resolution levels simultaneously, with nodes being shared among all resolution levels that include the node. Each resolution level maintains a separate priority queue, and a node that is expanded at a given resolution is only closed at that resolution. The finest resolution level ("anchor level") includes all nodes and acts as an optimal A* search, which is used to guarantee bounded suboptimality of the search result. Coarser resolutions function as suboptimal A* searches since they sort their priority queues by a weighted sum $g + wh$. A queue choice function determines which queue should be expanded from at any given time, while a

condition tied to the anchor level search ensures bounded suboptimality. We will discuss the Multi-Resolution A* algorithm in more detail in Sec. 4.2. By applying Multi-Resolution A* to the approach proposed by Liu et al. [5], we hope to match or exceed the performance achieved by Schleich and Behnke [8] without sacrificing planning resolution at a distance from the UAV. Measures of performance are trajectory costs and planning times.

4 Method

Our method aims to apply the work by Du et al. [2] to 6D trajectory planning using short-duration motion primitives as described by Liu et al. [5], with the goal of shortening planning times sufficiently for frequent replanning. We define our resolution levels as proposed in [8], with the exception that we do not reduce the velocity resolution. In the following sections, we will detail the changes and improvements implemented to facilitate this intent.

We will first discuss how we define our resolution levels and state lattice graph (Sec. 4.1). Next, we will explain the basic structure of the Multi-Resolution A* algorithm (Sec. 4.2). This is followed by a discussion of our Goal Queue, a modification designed to ensure a path can be found even when primarily planning at a coarse resolution that does not include any states in the goal area (Sec. 4.3). Afterwards, we discuss our policies determining the resolution for each expansion (Sec. 4.4). Finally, we discuss our options for detecting stagnating searches due to local minima, along with countermeasures designed to prevent unnecessary expansions in such a case (Sec. 4.5).

4.1 Resolution Levels and State Lattice

As detailed in Chapter 2, the search space is discretized and the state lattice constructed based on short-duration motion primitives, generated by applying constant acceleration for a fixed time interval.

Our approach is based on the idea of planning on multiple levels of resolution in tandem, allowing fast search progress using a coarse resolution while preserving the ability to plan at a finer resolution where required. We define multiple resolution levels differentiated by their control sets $\mathcal{U}_n \subseteq \mathcal{U}$ and the states included in the level $\mathcal{S}_n \subseteq \mathcal{S}$. We call the finest resolution level "anchor level" and assign it the index 0.

For a given resolution level n, the control set $\mathcal{U}_n$ is formed by allowing accelerations of $\pm u_n$ or 0 along individual axis for a fixed time τ_n. For non-anchor resolution levels, the acceleration is halved and the time interval doubled when

compared to the previous level:

$$u_n := 0.5^n u_0,\ \tau_n := 2^n \tau_0, \text{ for } n > 0. \tag{4.1}$$

The 6D search space is discretized based on the position- and velocity resolutions Δ_{p_min} and Δ_{v_min} at anchor resolution level. These result directly from the acceleration u_0 and time step τ_0 at anchor resolution:

$$\Delta_{p\_min} = \frac{1}{2}\tau_0^2 u_0\,, \quad \Delta_{v\_min} = \tau_0 u_0 \tag{4.2}$$

Choosing an acceleration $u_0 = 2.0m/s^2$ and time step $\tau_0 = 0.5s$ at anchor resolution gives us $\Delta_{p\_min} = 0.25m$ and $\Delta_{v\_min} = 1.0m/s$.

The state space $\mathcal{S}_n$ of a given resolution level n consists of those states whose position components lie on a grid with resolution $2^n\Delta_{p\_min}$ and whose velocity components lie on a grid with resolution Δ_{v_min}. It can easily be seen that with this definition, the state space of a coarser resolution $\mathcal{S}_{n+1}$ is a subset of that of the next finer resolution, $\mathcal{S}_n$. We do not limit the velocity resolution for coarser resolution levels as to not restrict the state space too much.

Start and goal poses are rounded so their positions and velocities lie on the anchor level grids. As described in Chapter 2, we generate the state lattice graph $\mathcal{G}(\mathcal{S}, \mathcal{E})$ by unrolling motion primitives from the start state $s_{start} \in \mathcal{S}_0$. $\mathcal{S} = \mathcal{S}_0$ denotes the set of discretized states, and $\mathcal{E}$ denotes the set of motion primitives connecting the states. For a given state $s \in \mathcal{S}_0$, the motion primitives originating from it are defined by the control sets $\mathcal{U}_n$ for all resolution levels n with $s \in \mathcal{S}_n$. It should be noted that a resolution n motion primitive $F_{u,s}$ originating from $s \in \mathcal{S}_n$ and using controls $u \in \mathcal{U}_n$ does not necessarily end in a state $s' = F_{u,s}(\tau_n)$ in $\mathcal{S}_n$; however, s' is guaranteed to be in $\mathcal{S}_0$.

In the interest of computational efficiency, we reformulate these ideas to use integer based calculations for all operations besides collision checking. States are saved based on the anchor space discretization, meaning one positional unit is equivalent to Δ_{p_min}, and a velocity unit is Δ_{v_min}. Controls are saved as a 5-tupel (x, y, z, t, d) , with $u = (x, y, z)^3 \in \{-1, 0, 1\}^3$. The real acceleration u_{real} can be calculated as $u_{\text{real}} = uu_0/d$, the real time step as $\tau_{\text{real}} = t\tau_0$. The successor $s' = (p', v')$ can be calculated as

$$\begin{pmatrix} p' \\ v' \end{pmatrix} = \begin{pmatrix} p + 2tv + t^2u/d \\ v + ut/d \end{pmatrix} \tag{4.3}$$

. In our work t and d are both equal to 2^n at resolution n, but different values

could be used to allow for different controls at given resolutions. The control set must conform to the maximum acceleration allowed by system dynamics, in our set that is guaranteed if 0 does not exceed the limit.

Within a given resolution level, not all states are connected by a chain of motion primitives. Notably, at anchor resolution, a motion primitive changes the (discretized) position along one axis by an even amount if and only if the velocity component along that axis also changes by an even amount. To address this, we change our goal position to a goal region formed by all states with the same velocity as the goal state and whose discretized position component only differs by one along each axis; that is to say, the goal position's 26-neighborhood.

4.2 Multi-Resolution A*

Algorithm 1 Multi-Resolution A* (adjusted from [2])

```
1:  procedure MAIN
2:      g(s_start) = 0; g(s_goal) = ∞
3:      bp(s_start) = bp(s_goal) = null
4:      goal = s_goal
5:      for i = 0, ..., n-1 do
6:          OPEN_i ← ∅
7:          CLOSED_i ← ∅
8:          if i ∈ GetSpaceIndices(s_start) then
9:              Insert s_start in OPEN_i with Key(s, i).
10:     while OPEN_i ≠ ∅ for at least one i ∈ {0, ..., n − 1} do
11:         i ← ChooseQueue()
12:         if i > 0 then
13:             if g(goal) ≤ OPEN_i.MinKey() then
14:                 Return path pointed by bp(g(goal))
15:             else
16:                 s = OPEN_i.Pop()
17:                 ExpandState(s, i)
18:                 Insert s into CLOSED_i
19:         else
20:             if g(goal) ≤ w_2 * OPEN_0.MinKey() then
21:                 Return path pointed by bp(g(goal))
22:             else
23:                 s = OPEN_0.Pop()
24:                 ExpandState(s, 0)
25:                 Insert s into CLOSED_0
```

Algorithm 2 Key & Expand State (adjusted from [2])

```
1: procedure KEY(s, i)
2:     if i = 0 then
3:         return g(s) + h(s)
4:     else
5:         return g(s) + w_1 h(s)
6: procedure EXPANDSTATE(s, i)
7:     for all s' ∈ Succs(s, i) do
8:         if s' was never generated then
9:             g(s') = ∞; bp(s') = null
10:        if g(s') > g(s) + c(s, s') then
11:            g(s') = g(s) + c(s, s'); bp(s') = s
12:            for all i ∈ GetSpaceIndices(s') do
13:                if s' ∉ CLOSED_i then
14:                    Insert/Update s' in OPEN_i with Key(s', i)
15:            if s' = s_goal or s' ∈ 26Neighborhood(s_goal) then
16:                if g(s') < g(goal) then
17:                    goal = s'
```

As mentioned in Chapter 3, the Multi-Resolution A* algorithm [2] forming the basis of our method is designed to perform a search on several resolution levels in tandem to accelerate the search. Alg. 1 demonstrates the main algorithm, using n resolution levels.

We save nodes in a vector in order of their creation, and a hashmap maps their state to their index. For each node, we also remember at which resolutions it has been expanded. In the pseudocode this is represented as lists CLOSED_i for readability, though our implementation saves the information directly in each node.

We use a separate priority queue OPEN_i for each resolution level. Nodes in the priority queues are sorted by `Key`(). As shown in Alg. 2, at anchor resolution the key of a node s is the sum of the path costs $g(s)$ to s and the the estimated goal distance $h(s)$, whereas at coarser resolutions the key is the weighted sum $g(s) + w_1 h(s)$. Our code also allows a weight at anchor resolution, which is used to run a bounded suboptimal A* search for evaluation. If two nodes have identical keys, the tie is broken by $h(s)$. We use the Linear Quadratic Minimum Time heuristic suggested by Liu et al. [5].

In Lines 2-9 in Alg. 1 we prepare the search by initializing g values and back pointers for the start and goal node as well as OPEN and CLOSED for each resolution, and inserting s_{start} into the queues at all resolutions which include

s_{start}. The variable *goal* represents the node in the goal region which has the smallest g value; it initially points at s_{goal} with a g value of infinity and is updated whenever we find a cheaper path to a state in the goal region.

The algorithm terminates if all priority queues are empty or one of the completion criteria in Line 13 or 20 are met.

In Line 11, the algorithm chooses the priority queue which should be expanded next. Several options for the queue choice policy will be discussed in Sec. 4.4. By limiting expansions based on the anchor queue, the algorithm guarantees that the solution returned is within the suboptimality bound w_2 of the optimal solution at anchor resolution. The search at anchor resolution is equivalent to a pure A* search, and another queue may not be expanded if the top element's key is more than a factor w_2 larger than the key of the anchor queue's top element. Lu et al. [2] check this suboptimality condition in Line 12, however, we instead include it in `ChooseQueue`(). This enables us to make a more informed choice and also simplifies the implementation of several policies.

Depending on wether anchor resolution (Lines 20-25) or another resolution (Lines 13-18) was chosen for expansion, the algorithm now checks a slightly different termination criterium, comparing the top key of the chosen queue with the cost of the current best solution. If the algorithm does not terminate, the top element of the chosen queue is removed and expanded at the corresponding resolution, then marked as closed for this resolution.

The expansion process is detailed in Alg. 2. The successors $\texttt{Succs}(s, i)$ of state s at resolution i are calculated by applying the controls $u \in \mathcal{U}_i$ to s and checking if the resulting motion primitive is valid as defined in Chapter 2. If a successor has not been generated before, its g value and back pointer are initialized.

If we have found a cheaper path to a successor s', its g value and back pointer are updated. The successor is then inserted into the queues at all resolutions i with $s' \in \mathcal{S}_i$, unless it was already expanded at that resolution. Because searching for and removing a node from a queue would be computationally expensive, we simply insert s' with the new key and ensure that a node is only expanded at a given resolution once, even if it is inserted into the queue multiple times. We achieve this by cleaning up the queues at the start of `ChooseQueue`(), discarding the top elements until we reach a node which has not yet been expanded at that resolution. If s' is in the goal region and its g value is smaller than the cost of our current best path, we also update *goal*.

4.3 Goal Queue

Due to the limited control set $\mathcal{U}_n$ as well as the restricted state space $\mathcal{S}_n$ at a given resolution, searches often can't reach the goal pose on a given resolution level, particularly at coarser resolutions. We already discussed that we extend the goal state to a goal region; however, to make this useful for coarse resolutions, the goal region would need to be extended to an impractical size. Instead, we implement a special "goal queue" to reach the goal with anchor resolution motion primitives once we are close to the goal.

When generating a new node, we check if its h value is smaller or equal to a predefined value d_{goal}; if it is, we consider that node to be close to the goal. The threshold is defined based on the maximum cost C_{max} of an anchor level motion primitive and a configurable weight w_{close}:

$$d_{goal} := w_{close} C_{max} = w_{close}((3u_0^2 + \rho_0)\tau_0). \tag{4.4}$$

We choose $w_{close} = 3$. When using coarser resolutions, it may be necessary to increase w_{close} or define d_{goal} differently.

When an expansion updates the g value of a node that is close to the goal and has not been expanded at anchor level yet, we add that node to the goal queue, which is sorted exactly like the anchor queue.

If the goal queue is not empty, we expand from it instead of a queue chosen by `ChooseQueue`() in lines 11-25 in Alg. 1. We remove the top element from the goal queue, expand the node at anchor resolution, and mark it as closed at anchor resolution. We then increment a counter indicating how many goal queue expansions we have performed since the last regular expansion. If the counter exceeds a configurable limit, we clear all remaining entries in the goal queue. Finally, if the goal queue is empty, we reset the counter to 0.

4.4 Queue Choice Functions

The choice of which resolution level should be expanded next significantly influences the performance of the algorithm, both in terms of solution quality as well as planning time. Therefore, we implement and test several queue choice policies.

As mentioned in Sec. 4.2, we require that the queue chosen by our queue choice function fulfills a suboptimality condition to guarantee bounded suboptimality for the solution returned by the algorithm. Specifically, a queue OPEN_i may not be expanded if the top element's key is larger than w_2 times the key of the anchor queue's top element. We say a resolution level or queue is valid or a valid choice

if it does not violate the suboptimality condition, and the queue is not empty.

Round Robin (RR, RR-a, wRR): The simplest policy is Round Robin (RR), cycling through all resolution levels and expanding them if they are valid. Starting at the last expanded resolution, we increase the resolution by one - wrapping around to anchor resolution if we have reached the coarsest level - until we find a valid resolution. However, because the anchor search is an optimal A* search, it makes progress at a significantly lower rate than searches where the key is a weighted sum, assuming the heuristic function underestimates the real costs. The effect is compounded by the fact that anchor resolution motion primitives cover the shortest amount of distance. For these reasons, it is advisable to only expand at anchor resolution if required by the suboptimality condition. Therefore, we implement a Round Robin policy that excludes anchor resolution, unless no non-anchor resolution is valid. In that case we default to expanding at anchor level (RR-a). We also experiment with a weighted Round Robin (wRR) approach where resolution level i would be expanded 2^i times before switching to the next resolution level, assuming the queue is valid. The intention is to favor coarse resolution searches which make progress towards the goal quickly without eliminating the path cost improvements gained from finer resolutions. However, our experiments show no noticeable improvement over RR-a.

Coarsest First (CF): If the search space is mostly free, it is to be expected that a coarse resolution search requires significantly less expansions than finer searches to find a path to the goal, though that path is likely to have increased costs compared to the optimal path. For this reason we test a Coarsest First (CF) policy, expanding at the coarsest valid resolution. This policy may produce significantly more expensive trajectories than alternatives if the coarsest resolution is insufficient to approximate the optimal trajectory, for example if the optimal trajectory moves through a gap between obstacles that is not covered by a motion primitive at the coarsest resolution. Issues also arise from the fact that the goal region is likely not reachable using the coarsest resolution alone; this problem is hopefully counteracted by the Goal Queue introduced in Sec. 4.3.

Smallest Heuristic First (SHF, SHF-a): The final queue choice function tested is Smallest Heuristic First (SHF), expanding from the valid queue with the smallest h value of its top element. The intuition behind this policy is that the search at this resolution is likely closest to finding a path to the goal, so SHF is expected to produce shorter search times. We also test a Smallest Heuristic First policy excluding anchor (SHF-a), expanding from the anchor queue only if no other resolutions

are valid.

4.5 Stagnation Detection

Algorithm 3 Stagnation Detection (Queue) (adjusted from [7])

```
1: procedure StagnationDetectQueue(i,h)
2:     for m = 0, 1, ..., σ_1 − 2 do
3:         Q_i(m) = Q_i(m + 1)
4:     Q_i(σ_1 − 1) = h
5:     bestOld = min_{0≤n<σ_2}{Q_i(n)}
6:     bestNew = min_{σ_2≤n<σ_1}{Q_i(n)}
7:     if bestNew > (bestOld + ε_1) then
8:         blocked_i = true
9:         blockH_i = bestOld
10:    else
11:        blocked_i = false
```

As mentioned in Sec. 4.4, a Smallest Heuristic First (SHF) queue choice function is expected to produce short search times. However, initial experiments instead showed that a search guided by SHF was actually slower on average than alternative searches using a Round Robin queue choice function.

Further investigation showed that this was largely caused by the search stagnating when it encountered a dead end or similarly, an obstacle like a flat wall orthogonal to the flight path. In any case where the search encounters a local minimum in the heuristic function, SHF ensures that the local minimum area is explored thoroughly on all resolution levels (except anchor) before a path out of the area is found.

To address this issue, we disable resolution levels temporarily if we detect that the search at that level is stagnating. This should result in the local minimum area only being explored completely at the coarsest resolution level, which requires the lowest number of expansions to do so. We implement and test two stagnation detection methods.

Count-based detection: The first method simply counts how many successive expansions at a given resolution do not result in at least one successor node having a h value at least ε_1 lower than that of the expanded node. If a certain number of successive expansions don't result in an improvement, we mark the resolution as blocked. If an expansion does result in an improvement, we unblock the resolution.

This is relevant because we do still expand at blocked resolutions to test if newly added nodes have allowed the search to leave the local minimum.

Queue-based detection: For our second detection method, we adapt the work of Kai Mi et al. [7], who detect stagnation in a single resolution search to toggle multiple heuristic functions. For each resolution level i, we use a first-in-first-out queue Q_i to save the h values of the last σ_1 expansions at resolution i. Alg. 3 shows the process, `StagnationDetectQueue`(i,h) being called after a node s with $h = h(s)$ was expanded at resolution i. After updating Q_i, we calculate the minimum of both the oldest σ_2 and the newest $(\sigma_1 - \sigma_2)$ saved h values. If the best new h value is larger than the best old value plus a threshold ε_1, we mark resolution i as blocked, and save the best old h value for unblocking. As in the first detection method, we unblock the resolution if we do not detect stagnation, which is used when we speculatively expand from a blocked queue.

At the beginning of `ChooseQueue`(), we unblock all resolution levels if the anchor resolution is blocked, or if all resolution levels above anchor are blocked. This prevents issues if the local minimum area cannot be escaped quickly enough to prevent blocking the resolution required for it. A blocked queue is skipped by the queue choice functions discussed in Sec. 4.4. However, `ChooseQueue`() may still return a blocked resolution in an effort to unblock it. After a fixed number of normal expansions, we expand at each blocked resolution once, assuming the queue is valid.

Discard ($g + h$, $g + wh$, *waypoint*): Lastly, we attempt to detect when we have escaped the local minimum, and discard states in the local minimum area that are still in the queues to prevent them being expanded unnecessarily due to their small key. This may also help unblocking queues faster once we have left the stagnation area, since we would otherwise have to wait until the periodic speculative expansion results in an expansion of a state outside the area.

When we insert a new node into a blocked non-anchor queue, we check whether or not the new node is behind the obstacle that was causing the stagnation. This step is only in effect if we use the second stagnation detection method. We assume we have found a way around the blocking obstacle if the h value of the newly inserted node is smaller than the previous best h value, which we saved in Line 9 in Alg. 3. In that case, we unblock the resolution, and discard nodes from the queue based on one of three rules, in an effort to eliminate the nodes in the stagnation area. Because of their small key, they would otherwise likely still be expanded even after a path around the obstacle was found. All methods discard the top nodes of the queue until they find a node which does not conform to their

discard rule, or they have discarded a maximum allowed number of nodes. They use a configurable discard threshold ε_2.

The first discard rule is based on the sum of g and h values, discarding nodes with a sum that is at least ε_2 smaller than that of the newly added node ($g + h$ discard). The second rule is similar but discards based on the keys of both nodes, that is, the weighted sum $g(s)+w_1h(s)$, instead ($g+wh$ discard). Both are based on the assumption that the best path around the obstacle will not be significantly less expensive than a path leading through the newly added node, using the currently known path to said node.

Because we detect the first time we reach a node that is significantly behind the obstacle, it is likely that we have found a suboptimal path to this node. The previous discard rules are expected to discard nodes which could lead to a better path to the newly discovered node. In our initial testing, this results in significantly increased solution costs. To address this, we implement a new discard rule based on the estimated path costs to the new node. Nodes are discard candidates if the sum of their g value and the estimated path cost to the newly added node is at least ε_2 smaller than the g value of the new node. We additionally require that a discarded node has a larger h value than the new node (*waypoint* discard).

Because we do not discard nodes from the anchor queue nor expand from queues that violate the suboptimality criterion, none of these actions compromise the suboptimality guarantee.

5 Evaluation

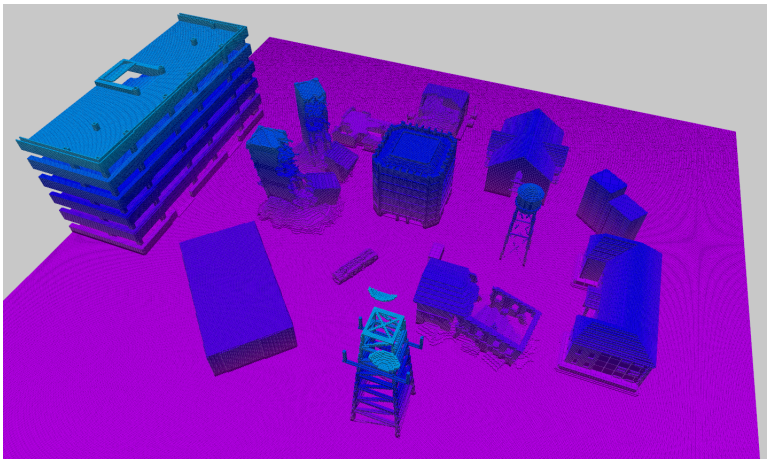

Figure 5.1: The evaluation environment.

To evaluate our work, we run searches on a set of 500 pairs of start- and goal states, set in the simulated outdoor environment also used by Schleich and Behnke [8], which contains several obstacles with varied shapes (see Fig. 5.1). We generate random pairs of start and goal states with positions within the map boundaries, velocity components of the start states conforming to the velocity limits, and 0 velocity for the goal state. We then run A* search with a heuristic weight of 2.0 on these pairs to determine if a path from start to goal exists at anchor level. If the search finds a path or reaches its timeout point of one million expansions before terminating, we add the pair to our set of search pairs. While we can not say for certain that a path exists if the search terminates due to a timeout, we include these pairs to determine if our method can find a path. Of the resulting set of 500 pairs, the weighted A* search found a path for 97.6% of the pairs. We use a different test set to compare our work to that of Schleich and Behnke [8], because that algorithm requires the start position to be at the center of the map. For this set, the distance along the x- and y-axes from the start to the goal state may not exceed 64 meters, and the map is shifted to make the start position the new center. Otherwise, the test set is generated in the same way.

Our map has a size of $128 \times 128m$ and we allow a flight altitude of 0-$10m$. The

UAV may not exceed a velocity of $4.0m/s$ or acceleration of $2.0m/s^2$ along individual axes. We consider any position that is closer than $1m$ to an obstacle to be blocked, any that is further away to be free. We allow up to one million expansions before we terminate the algorithm. For all Multi-Resolution A* searches, we use a keyweight $w_1 = 2.0$ and a suboptimality bound $w_2 = 2.0$.

To compare different design choices of our proposed method against each other, we use the number of expansions required to find a path as a measure of computation speed, because they are more precise than actual time measurements and not influenced by external factors. Towards the end of the chapter we include measured times to compare our approach to others that may not require the same amount of time for an expansion. We noticed that required times for identical tasks varied, with the first execution requiring significantly longer than all following executions. This effect appears to stem from additional time being required when more nodes are added to a queue than its previous capacity. It does not happen if the same search was performed before, regardless of how many other searches have been performed in between, unless the entire program is restarted. We attempted to combat this by reserving memory space for the queues before starting the execution, however, this had no effect. To eliminate this effect in the evaluation, we perform each search twice, and use the time required by the second search.

5.1 Goal Queue:

Resolution levels	using Goal Queue	success rate (%)	average # of expansions	average path cost
2	no	60.0	55493.8	198.974
2	yes	**99.4**	**2623.4**	**196.526**
3	no	**99.0**	2493.6	209.993
3	yes	**99.0**	**1591.2**	**208.184**
4	no	**99.0**	3234.9	214.601
4	yes	**99.0**	**1673.8**	**213.235**

Table 5.1: Goal Queue evaluation: Comparing searches using and not using goal queue with different numbers of resolution levels.

We start by evaluating the effects of the Goal Queue (Sec. 4.3). We limit our comparison to searches using one Queue Choice Function for simplicity, and because experiments over smaller test sets showed similar results regardless of the

Queue Choice Function used. We choose a Round-Robin policy because it is the most basic policy, and because it is expected to benefit less from the Goal Queue than other policies like Coarsest-First. Therefore, if the Goal Queue shows significant improvements for Round-Robin, it can be assumed to be generally useful. We exclude anchor level from the Round-Robin selection because this is significantly more effective, as we shall see when evaluating Queue Choice Functions (Sec. 5.2). We allow for two Goal Queue expansions before clearing the queue. As detailed in Sec. 4.3, we consider a state to be close to the goal and add it to the goal queue if its h value is smaller or equal to $w_{close} = 3$ times the maximum cost of an anchor level motion primitive. To calculate the average number of expansions and path costs, we only consider the 297 start/goal pairs for which all searches found a path.

Tab. 5.1 compares the statistics of searches using the Goal Queue with those that do not, using 2, 3, and 4 resolution levels including anchor. It is easy to see that using the Goal Queue provides strictly better results in all cases, requiring significantly less expansions to find a slightly cheaper path, and succeeding in finding a path in at least the same number of cases. Similar results have been observed with other queue choice policies and parameters as well. Therefore, we will be using the Goal Queue in all further evaluations.

5.2 Queue Choice Functions

We will now compare the performance of the six options for `ChooseQueue`() proposed in Sec. 4.4. Tab. 5.2 shows the results of our tests using two, three and four resolution levels including anchor level. The averages for number of expansions and trajectory costs are calculated over those pairs where all searches with the same number of resolution levels found a path. At two resolution levels that were 492 of the 500 pairs; at three and four resolution levels 497 pairs resulted in successful paths with all queue choice functions.

When only one resolution level above anchor exists, the three functions excluding anchor level (RR-a, wRR and SHF-a) as well as CF are all equivalent, which explains why they perform identically.

The basic Round-Robin (RR) function always results in significantly longer searches than its variants excluding anchor level (RR-a and wRR), while only providing a minor improvement to path costs. As discussed in Sec. 4.4, this is because expansions at anchor resolution advance the search less than expansions at higher resolutions, both because anchor search is equivalent to optimal A* search and because anchor level motion primitives cover less distance. Because Round-Robin excluding anchor (RR-a) and weighted Round-Robin (wRR) perform similarly, we

Resolution levels	Queue Choice Function	success rate (%)	average # of expansions	average path cost
2	RR	98.4	21800.3	361.5
2	RR-a/wRR	**99.8**	**4360.3**	367.9
2	CF/SHF-a	**99.8**	**4360.3**	367.9
2	SHF	**99.8**	4798.1	**366.0**
3	RR	**99.6**	5386.3	389.1
3	RR-a	99.4	2771.8	393.0
3	wRR	99.4	2447.0	394.6
3	CF	99.4	**2366.7**	422.4
3	SHF	99.4	11863.4	**374.9**
3	SHF-a	99.4	13609.4	376.1
4	RR	**99.6**	5603.9	399.8
4	RR-a	99.4	**3482.1**	405.0
4	wRR	99.4	3536.6	411.0
4	CF	99.4	3626.7	475.4
4	SHF	99.4	10157.3	**379.8**
4	SHF-a	99.4	11794.2	382.3

Table 5.2: Queue Choice Function evaluation: Comparing Round-Robin (RR), Round-Robin excluding anchor (RR-a), weighted Round-Robin (wRR), Coarsest-First (CF), Smallest-Heuristic-First (SHF) and Smallest-Heuristic-First excluding anchor (SHF-a) with different numbers of resolution levels.

conclude that the additional complexity of wRR is not justified.

As expected, the paths produced by the Coarsest First have the highest costs. While at three levels of resolution CF does indeed require the smallest number of expansions, at four levels RR-a and wRR are actually faster. This may be due to suboptimal settings for the Goal Queue, or simply because Level-4 motion primitives are not flexible enough to plan an efficient path mainly on this level in the given environment.

Contrary to expectations, both variants of Smallest Heuristic First are significantly slower than all alternatives when using more than two resolution levels, though they produce the cheapest paths as well. A closer look at individual searches shows that this is at least partially caused by the algorithm getting "stuck" in local minima. When the search encounters a dead end or an obstacle that extends orthogonal to the straight path to the goal, SHF ensures that most nodes at all resolution levels in the local minimum are expanded before a path around the obstacle can be found. To overcome this problem, we implement Stagnation

Detection, which is evaluated in the next section.

In general, it is notable that average path costs increase when adding coarser resolution levels. This is not surprising, since the less flexible motion primitives at the added levels almost by necessity introduce suboptimalities in the path. However, the required number of expansions does not strictly decrease as coarser resolutions are added; in fact, all three Round-Robin policies as well as Coarsest-First require more expansions with four resolution levels than with three. A likely reason is that areas get explored multiple times due to overlapping motion primitives at different resolution levels.

5.3 Stagnation Detection

Resolution levels	Detection Mode	Discard Mode	success rate (%)	average # of expansions	average path cost
3	none	none	99.4	11863.2	**374.2**
3	Count	none	**99.6**	6129.6	384.5
3	Queue	none	**99.6**	**3026.7**	382.8
3	Queue	$g + h$	99.4	3200.6	382.5
3	Queue	$g + wh$	99.4	3042.1	382.8
3	Queue	*waypoint*	**99.6**	3199.5	383.2
4	none	none	99.4	10157.1	**379.2**
4	Count	none	99.4	7044.8	394.7
4	Queue	none	**99.6**	5304.8	388.6
4	Queue	$g + h$	99.4	**2616.5**	388.4
4	Queue	$g + wh$	99.4	3979.1	389.0
4	Queue	*waypoint*	**99.6**	3014.3	386.3

Table 5.3: Evaluation of Stagnation Detection options, using Smallest-Heuristic-First.

To evaluate the effectiveness of Stagnation Detection, we analyze the effects of the methods introduced in Sec. 4.5, that being Count- or Queue-based detection. For Queue-based detection, we also compare discarding nodes based on the sum $(g + h)$ or weighted sum $(g + wh)$ of their g and h values, based on the estimated distance to the node we detected as behind the obstacle (*waypoint*), and not discarding at all. All variants are tested with three and four resolution levels using SHF, since this Queue Choice Function motivates Stagnation Detection. Stagnation Detection has no relevant effect with only two resolution levels because all

queues are immediately unblocked if anchor resolution or all non-anchor resolutions are blocked.

We speculatively expand valid blocked resolution levels once after 20 regular expansions. For both Count- and Queue-based detection, we choose an improvement threshold $\varepsilon_1 = 0.5$. For Count-based detection, we block a resolution level after ten consecutive expansions that did not result in a significant improvement. Similarly, for Queue-based detection, we save the last $\sigma_1 = 20$ h-values, and consider $\sigma_2 = 10$ of them to be old values. We allow for up to 100 nodes to be discarded at once.

Tab. 5.3 shows the results of these experiments, averaging numbers of expansions and path costs over the 496 search pairs for which all searches found solutions. All variants of Stagnation Detection reduce the number of required expansions by more than 30% while slightly increasing path costs. However, searches utilizing Count-based detection still require significantly more expansions than those using Queue-based detection with or without discards, while also producing more expensive paths. At three resolution levels, discarding nodes does not seem to have a significant impact, but at four levels all three discard modes reduce the required number of expansions significantly, without impacting path costs very much. Using four resolution levels, discarding based on the key $g + wh$ produces more expensive paths while also requiring more expansions than both alternatives; $g + h$ discard results in the fastest searches here, while *waypoint* discard produces the cheapest paths. It should also be noted that *waypoint* discard has a higher success rate at both resolution levels, though the difference of 0.2% equates to only a single search.

We also evaluate the effect of Queue-based Stagnation Detection with *waypoint* based discard and the same parameters as above on other Queue Choice Functions, using three and four resolution levels. The results are shown in Tab. 5.4. Basic Round-Robin including anchor (RR) is not listed because Stagnation Detection does not improve on the large number of expansions it requires, making it nonviable. Once again, averages are formed over those pairs where all searches with the same number of resolution levels found a path, which is the case for 497 pairs both with three and four levels. It should also be noted that for all searches listed, the success rate increased from 99.4% to 99.6% when using stagnation detection.

When using three resolution levels, Stagnation Detection reduces the required number of expansions for all functions listed, though unsurprisingly the improvement for SHF and SHF-a is significantly larger than for the other options. Resulting path costs are also reduced for RR-a, wRR and especially CF, though they are increased for SHF and SHF-a.

When using four levels, Stagnation Detection is less beneficial for RR-a and

		Stagnation Detection		no Stagnation Detection	
Resolution levels	Queue Choice Function	average # of expansions	average cost	average # of expansions	average cost
3	RR-a	**1931.1**	**392.4**	2771.8	393.0
3	wRR	**2053.8**	**393.2**	2447.0	394.6
3	CF	**1847.0**	**401.9**	2366.7	422.4
3	SHF	**3202.3**	384.1	11863.4	**374.9**
3	SHF-a	**2514.5**	385.5	13609.3	**376.1**
4	RR-a	3812.3	406.4	**3482.1**	**405.0**
4	wRR	4142.3	**410.3**	**3536.6**	411.0
4	CF	3655.2	**424.4**	**3626.7**	475.4
4	SHF	**3014.8**	387.1	10157.3	**379.8**
4	SHF-a	**3975.0**	393.4	11794.2	**382.3**

Table 5.4: Effects of Queue-based Stagnation Detection using *waypoint* based discard with different Queue Choice Functions: Round-Robin excluding anchor (RR-a), weighted Round-Robin (wRR), Coarsest-First (CF), Smallest-Heuristic-First (SHF) and Smallest-Heuristic-First excluding anchor (SHF-a).

wRR, increasing the number of expansions required without significantly changing path costs. For Coarsest-First, path costs are once again significantly reduced, though number of expansions no longer changes significantly. Despite the improvements, the paths generated by CF remain significantly worse than all alternatives, and as noted in Sec. 5.2, CF does not produce especially fast searches when a larger number of resolution levels is involved.

SHF and SHF-a benefit from Stagnation Detection in a similar way at four resolution levels as they did at three, significantly reducing the number of expansions required at the cost of increased path costs. At three levels, SHF-a performs better, while at four levels SHF does.

In general, the fastest searches are achieved by using CF with three resolution levels and Stagnation Detection, followed by RR-a at the same settings. However, SHF and SHF-a can produce significantly better paths.

5.4 Comparison to A* at different Resolutions

To evaluate our approach as a whole and compare it to other approaches, we want to compare the resulting path costs to the optimal costs. Unfortunately, an optimal A* search is unable to find a path for many of our search pairs in a

reasonable time. Comparing only those pairs where we can generate an optimal trajectory would limit our sample size and likely also reduce the distance covered on average. We therefore compare our results to a bounded suboptimal A* search with a heuristic weight of 2.0, and compare the values created by that with the available optimal trajectories to estimate the difference between them.

Optimal A* search is only able to find a path for 52.4% of the pairs within one million expansions, while the suboptimal search is successful for 98% of the searches. Over the 262 pairs for which an optimal path was found, the average optimal costs were 91.796, while the average costs produced by the suboptimal search were 2.45% higher at 94.044.

We encounter a similar issue when comparing our Multi-Resolution A* approach to regular (suboptimal) A* search at coarser resolution levels — in many cases, those searches are unable to find a path, especially at very coarse resolutions. There are two primary reasons that we attempt to eliminate. First, rounding the start and goal positions to conform to the coarse resolution often results in one or both locations being inside an obstacle and therefore invalid. Second, due to the limited mobility granted by coarse resolution motion primitives and the fact that our start poses contain non-zero velocity components, the UAV often starts from a state which inevitably leads to a collision or to leaving the map, resulting in the search terminating with empty queues.

To solve these issues and generate a higher number of comparable searches, we use features of MRA to approximate A* search at coarser resolutions, by allowing certain anchor level motion primitives near the start and goal states. We call this approach virtual single resolution A* (vA*). Instead of rounding our positions to the coarse resolution grid, we round to our regular anchor resolution grid instead. We then expand that state at anchor level to a certain debth; specifically, we fully expand two anchor level motion primitives deep, performing a maximum of 28 expansions and generating a maximum of 729 nodes. We only insert nodes into the queue at the resolution we want to plan at, or the goal queue if they are close to the goal. By using the Goal Queue and goal region, we ensure that the goal can be found even if it is not included in the search resolution.

Tab. 5.5 shows how vA* performs compared to regular A*. Averages consider the pairs for which both A* and vA* found a path at a given planning resolution; thus, these values are only suitable for comparing searches at the same resolution. For Level-1 or Level-2, vA* has significantly higher success rates than A*, and also produces cheaper paths. This does come at the cost of some additional expansions. At Level-3, vA* actually performs worse than A*, possibly because the parameters for the initial anchor level expansion and the goal queue are insufficient to link the start and goal position to the coarse grid. However, because both methods have

resolution level	mode	success rate (%)	average # of expansions	average path cost
0	A*	97.6	41953.0	354.2
1	A*	77.0	**5299.4**	291.9
1	vA*	**94.0**	6595.7	**287.8**
2	A*	44.4	**435.7**	197.6
2	vA*	**65.8**	676.3	**193.0**
3	A*	**27.0**	**93.9**	**72.9**
3	vA*	22.2	1308.6	77.9

Table 5.5: Comparing A* search to virtual single resolution A* (vA*).

a success rate below 30%, we do not consider either method to be viable at this resolution and refrain from comparing it to other methods.

planning mode	success rate (%)	average plan time (ms)	average # of expansions	average path cost
A* Level-0	97.6	149.6	28275.7	**229.1**
vA* Level-1	94.0	16.5	2780.3	242.3
vA* Level-2	65.8	8.3	**785.6**	296.1
MRA RR-a Level-1	**99.8**	18.2	2867.7	240.7
MRA RR-a Level-2	99.6	**7.9**	1211.8	253.5
MRA CF Level-2	99.6	8.3	1179.9	258.9
MRA SHF-a Level-2	99.6	9.9	1524.4	248.1

Table 5.6: Comparison of MRA to virtual single resolution A* at different resolution levels; using Round-Robin excluding anchor (RR-a), Coarsest-First (CF) and Smallest-Heuristic-First excluding anchor (SHF-a) Queue Choice Functions for MRA. For MRA, Level-1 and Level-2 refer to the coarsest level used, while for (v)A* they refer to the primary planning level.

We will now compare our Multi-Resolution A* approach to A* and vA*. Tab. 5.6 shows the statistics comparing anchor level A*, vA* at Level-1 and Level-2, and multiple variations of MRA using two or three levels (including anchor), all with suboptimality weights of 2.0. For MRA we include the queue choice functions RR-a, CF and SHF-a, all using queue-based stagnation detection and *waypoint* discard. As mentioned earlier, at Level-1 these queue choice functions are all equivalent, so we only list one. The average and maximum numbers of expansions as well as the average path costs only consider the 323 pairs for which every search

found a trajectory.

As expected, anchor level A* produces the best trajectories. However, it also requires an order of magnitude more expansions than all alternatives. At Level-1, the performance of MRA and vA* is quite similar, requiring roughly twice as many expansions — and twice as much time — as Level-2 MRA but producing cheaper trajectories. However, vA* is unable to find a solution for 6% of the search pairs.

At Level-2, vA* requires significantly less expansions than MRA, however, it is not actually faster than MRA with RR-a or CF. It also produces by far the most expensive trajectories, and fails to find any solution for 34.2% of the pairs. For these reasons, MRA is likely a better choice in most scenarios.

The performance of all three MRA variants at level 2 is quite similar, with SHF-a producing the best trajectories but also requiring more time than the alternatives, and RR-a finding a solution the fastest, but producing a more expensive trajectory.

5.5 Comparison to Local Multiresolution State Lattices

planning mode	success rate (%)	average plan time (ms)	average # of expansions	average path cost
MRA RR-a res1	99.8	79.7	11407.1	255.9
MRA RR-a res2	**100.0**	31.4	5023.3	270.1
MRA CF res2	**100.0**	31.1	4931.5	282.1
MRA SHF-a res2	**100.0**	31.9	5185.5	263.8
LMRSL [8] $w_h = 1$ A*	65.4	-	-	-
LMRSL [8] $w_h = 1$ Level-A*	98.8	339.6	10900.6	**241.5**
LMRSL [8] $w_h = 2$ A*	99.0	74.4	2719.6	253.2
LMRSL [8] $w_h = 2$ Level-A*	99.2	**25.3**	**753.5**	269.3

Table 5.7: Comparison of MRA to Local Multiresolution State Lattices (LMRSL; Schleich and Behnke [8]): MRA using the Queue Choice Functions Round-Robin excluding anchor (RR-a), Coarsest-First (CF) and Smallest-Heuristic-First excluding anchor (SHF-a); LMRSL using and not using Level-A* with a heuristic weight of 1.0 and 2.0.

Finally, we compare our approach to the Local Multiresolution State Lattice approach developed by Schleich and Behnke [8]. Both approaches are similar in that they use the same formulation of motion primitives and similar definitions of resolution levels to accelerate searches by utilizing multiple resolutions. However,

our approach uses all resolution levels simultaneously, while Local Multiresolution uses only a single level at each position, based on the distance from the UAV. Our approach holds the advantage that we can plan at a finer resolution level anywhere on the map when necessary, however, in unexplored terrain, we usually do not have sufficient knowledge of the environment far from the UAV to necessitate or even enable such precise planning.

Because Local Multiresolution State Lattices centers the map on the UAV and we want to utilize the configuration introduced in [8], we can not plan trajectories covering a larger x- or y-distance than 64m. For this reason, we use a different test set for this evaluation, as mentioned at the beginning of the chapter. We compare the same MRA versions we used in Sec. 5.4 against four variants of Local Multiresolution State Lattices; using and not using the level-based expansion scheme (Level-A*) suggested in [8] each with a heuristic weight of 1.0 and 2.0.

Tab. 5.7 shows the results of the comparison, though we only show the low success rate for Local Multiresolution State Lattices not using Level-A* with a heuristic weight of 1.0. This is because we want to average planning times, number of expansions and path costs over a large number of pairs for which all searches found a path, and all other searches succeeded for 493 of the 500 pairs. Additionally, when limiting ourselves to those pairs for which this setting also found a path, we find that the average plan time was more than 75 times as long as for the next slowest approach. Thus, we can safely say that this variant is not relevant for the comparison.

Using a heuristic weight of 2.0, Local Multiresolution State Lattices requires significantly less expansions than MRA, however, without Level-A* it still requires more than twice as much time as MRA at Level-2. Because one expansion requires vastly different amounts of time between both methods, number of expansions is not a suitable metric for the required time here.

Unsurprisingly, the search with a heuristic weight of 1.0 produces the best paths, however, it is also much slower than all other options. MRA at Level-1 performs similar to Local Multiresolution State Lattices without Level-A* and with heuristic weight of 2.0, producing the next best paths but requiring more than twice as much time as the remaining alternatives. At Level-2, the time differences between MRA variants is negligible, though the path quality differs significantly. Local Multiresolution State Lattices with Level-A* and a heuristic weight of 2.0 is faster than MRA, however, MRA with Level-2 is still very fast, succeeds on more pairs, and in the case of SHF-a also produces cheaper trajectories.

At the current stage of development, Local Multiresolution State Lattices and our approach have similar performances. It is likely that both methods can still be improved upon.

6 Conclusion

In this thesis, we apply Multi-Resolution A* [2] to the search-based trajectory generation method proposed by Liu et al. [5] to quickly generate dynamically feasible second order trajectories for UAVs. We develop and compare six Queue Choice Functions to determine the resolution level for each expansion. We introduce the Goal Queue and Stagnation Detection as improvements to Multi-Resolution A* that are likely effective in many different settings outside the realm of trajectory generation.

We show that using Multi-Resolution A* results in significantly faster searches than A* search at anchor level and significantly better success rates than A* at coarser resolution levels. It also allows for more precise goal positions than A* at coarser resolutions.

We show that at this stage of development, our approach can produce trajectories of similar quality within similar time frames to the Local Multiresolution State Lattice approach proposed by Schleich and Behnke [8]. However, our approach is able to plan at a fine resolution regardless of distance from the start state, which may be useful for operations in partially known environments, including settings where multiple tasks must be performed in the same initially unknown environment.

Multi-Resolution A* can be used to accelerate the method proposed by Liu et al. [5] sufficiently for frequent replanning of second order trajectories. Unless a slight improvement of search times is more important than the quality of the generated trajectories, we advise using a Smallest Heuristic First excluding anchor (SHF-a) Queue Choice Function, since it produces cheaper trajectories than the slightly faster alternatives.

It is likely that the performance of our approach can be improved further, for example by using a different heuristic function. Additionally, it may be possible to use information from previous searches to accelerate replanning. One approach is to use the Smallest Heuristic First Queue Choice Function to choose resolution levels based on a heuristic informed by the previous search, separate from the admissible heuristic guiding the search. This would require the search direction to be inverted between each search. Because the motion primitives are symmetrical, this is not difficult. However, the heuristic function would need to be adjusted to

allow for goal states with non-zero velocities.

Finally, Liu et al. [6] extend their approach [5] to generate trajectories through gaps that are narrower than the UAV diameter, which is possible because flight attitude can be considered. Their method uses an initial lower dimensional search to guide the complex high dimensional search; using Multi-Resolution A* instead may improve performance.

List of Figures

List of Tables

Bibliography

[1] Reinis Cimurs and Il Hong Suh. "Time-optimized 3d path smoothing with kinematic constraints". In: *International journal of control, automation and systems* 18.5 (2020), pp. 1277–1287.

[2] Wei Du, Fahad Islam, and Maxim Likhachev. "Multi-resolution a*". In: *Thirteenth annual symposium on combinatorial search*. 2020.

[3] Jonathan Jamieson and James Biggs. "Near minimum-time trajectories for quadrotor uavs in complex environments". In: *2016 ieee/rsj international conference on intelligent robots and systems (iros)*. 2016, pp. 1550–1555.

[4] Sertac Karaman, Matthew R. Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. "Anytime motion planning using the rrt*". In: *2011 ieee international conference on robotics and automation*. 2011, pp. 1478–1483.

[5] Sikang Liu, Nikolay Atanasov, Kartik Mohta, and Vijay Kumar. "Search-based motion planning for quadrotors using linear quadratic minimum time control". In: *2017 ieee/rsj international conference on intelligent robots and systems (iros)*. IEEE. 2017, pp. 2872–2879.

[6] Sikang Liu, Kartik Mohta, Nikolay Atanasov, and Vijay Kumar. "Search-based motion planning for aggressive flight in se(3)". In: *Ieee robotics and automation letters* 3.3 (2018), pp. 2439–2446.

[7] Kai Mi, Jun Zheng, Yunkuan Wang, and Jianhua Hu. "A multi-heuristic a* algorithm based on stagnation detection for path planning of manipulators in cluttered environments". In: *Ieee access* 7 (2019), pp. 135870–135881.

[8] Daniel Schleich and Sven Behnke. "Search-based planning of dynamic mav trajectories using local multiresolution state lattices". In: *2021 ieee international conference on robotics and automation (icra)*. 2021, pp. 7865–7871.

[9] Zetian Zhang, Ruixiang Du, and Raghvendra V Cowlagi. "Randomized sampling-based trajectory optimization for uavs to satisfy linear temporal logic specifications". In: *Aerospace science and technology* 96 (2020), p. 105591.