



BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS
DEPARTMENT OF AUTOMATION AND APPLIED INFORMATICS



HUN-REN INSTITUTE FOR COMPUTER SCIENCE AND CONTROL
SYSTEMS AND CONTROL LABORATORY

ÁDÁM KOVÁCS

Rapid path planning in dynamic environments

SCIENTIFIC STUDENTS' ASSOCIATION REPORT

Advisor:

Dr. Tamás Péni
Senior research fellow
SZTAKI Systems and Control Lab

Supervisor:

Dr. Péter Pál Stumpf
Associate professor
BME VIK AUT

Dr. Roland Tóth
Senior research fellow / Associate Professor
SZTAKI Systems and Control Lab
Eindhoven University of Technology

Budapest, 2024.

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
1.1 Overview and classification of different approaches	2
1.2 Temporal A*	3
1.3 Safe Interval Path Planning	4
2 Jump Point Search algorithms	5
2.1 Preliminaries	5
2.2 Jump Point Search	7
2.3 Temporal Jump Point Search	8
2.3.1 Idea	8
2.3.2 Algorithm base	10
2.3.3 Pruning rules	11
2.3.4 Jumping rules	13
2.3.5 Path extraction	15
2.4 Development and optimization ideas	16
3 Trajectory generation	19
3.1 Discrete path optimization	19
3.2 Continuous trajectory fitting	22
3.3 Minimum Snap Trajectory	24
3.4 Summary and results	24
4 Simulation results	26
4.1 Two-dimensional benchmarks	27

4.2	Three-dimensional benchmarks	31
4.3	Trajectory generation	32
5	Summary	35
	Acknowledgements	35
	Bibliography	36

Kivonat

A drónok integrálása a polgári és kereskedelmi alkalmazásokba számos iparágat átalakított, beleértve a logisztikát, a mezőgazdaságot és a katasztrófaelhárítást, kihasználva kivételes mobilitásukat és sokoldalúságukat olyan feladatokhoz, mint a környezeti megfigyelés, az infrastruktúra ellenőrzés, a precíziós mezőgazdaság, a térképészet és földmérés. A pályatervezés elengedhetetlen a drónok számára, hogy autonóm módon és biztonságosan navigáljanak egy kiindulási pontból a célhelyre, ami különösen kihívást jelent a komplex és dinamikus környezetekben, ahol az akadályok váratlanul megjelenhetnek vagy mozgásba lendülhetnek. A földi járművekkel ellentétben a drónok háromdimenziós térben operálnak, ami bonyolult navigációs stratégiákat igényel, amelyek figyelembe veszik a statikus és mozgó akadályokat is. Mivel a drónok egyre inkább dinamikus környezetekben működnek, az adaptív és robusztus pályatervezés kulcsszerepet játszik hatékony alkalmazásukban. Ez a dolgozat elősegíti a drónok navigációjával kapcsolatos kutatásokat egy kifejezetten dinamikus környezetekre tervezett pályatervező és trajektória-generáló algoritmus kidolgozásával. A kidolgozott módszer a Temporal Jump Point Search (TJPS) algoritmusra épít, kiterjesztve azt a háromdimenziós térre, és gyorsítási technikákat implementálva a dinamikus akadályok által implikált nehézségek kezelésére. Ezenkívül egy többlépéses trajektória-generáló eljárást is tervezünk, hogy biztosítsuk a kapott pályák dinamikai megvalósíthatóságát és ütközésmentességét, anélkül, hogy előzetes környezeti adatfeldolgozásra lenne szükség. A szimulációs eredmények betekintést nyújtanak az algoritmus hatékonyságába a dinamikus térbeli navigáció valós összetettségének kezelésében.

Abstract

The integration of drones into civilian and commercial applications has transformed various industries including logistics, agriculture and emergency response, by leveraging their exceptional mobility and versatility for tasks such as environmental monitoring, infrastructure inspection, precision agriculture, disaster management and surveying. Pathfinding is essential for enabling UAVs to autonomously and safely navigate from a starting point to a target location, which becomes especially challenging in complex and dynamic environments where obstacles can appear or move unpredictably. Unlike ground vehicles, UAVs operate in three-dimensional space, requiring sophisticated navigation strategies that account for static and moving obstacles. As UAVs increasingly operate in settings with dynamic elements, adaptive and robust path planning has become crucial to their effective deployment. This thesis advances research in drone navigation by developing a pathfinding and trajectory generation algorithm specifically designed for dynamic environments. Our approach builds on the Temporal Jump Point Search (TJPS) algorithm, extending it to three-dimensional space and implementing speed-up techniques to handle the complexities introduced by dynamic obstacles. Additionally multi-step trajectory generation procedure is designed in order to ensure that the resulting paths are dynamically feasible and collision-free, without the need for prior grid map processing. Simulation and benchmark results provide insights into the algorithm's effectiveness in handling real-world complexities of dynamic drone navigation.

Chapter 1

Introduction

Autonomous drone navigation is a rapidly evolving field that addresses challenges in environmental perception, path planning, trajectory generation, and motion control. Among these, path planning plays a pivotal role in enabling drones to navigate dynamic environments efficiently and safely. In order to navigate complex and often unpredictable spaces, drones require real-time updates on obstacles and other moving entities. Traditional methods often involve a preprocessing step, where sensor data, such as point clouds from LiDAR sensors or images from cameras, is fused and interpreted. This approach relies heavily on maintaining an up-to-date planning environment that accounts for obstacle prediction and trajectory adjustment. While state-of-the-art algorithms like MADER (Multi-Agent Distributed, Efficient Routing) [16] employ convex optimization to plan multi-drone trajectories, they require extensive environmental preprocessing, which limits their applicability in real-time applications. [17]

Emerging machine learning approaches, particularly reinforcement learning (RL), hold promise for overcoming these limitations by learning from interaction with dynamic environments. Learning-based methods allow drones to adapt their trajectories based on real-time feedback, providing an advantage in scenarios where traditional methods struggle. However, current reinforcement learning models for path planning often lack a structured basis, which can restrict optimal learning. This thesis addresses this shortcoming by developing an optimal grid-based path planning method that serves as a foundation for RL training. Hybrid approaches in drone path planning combine traditional algorithms with machine learning or optimization techniques to leverage their respective strengths. This fusion enables the combination of the optimality and completeness of classical methods with the adaptability and learning capabilities inherent in modern approaches. Grid-based algorithms offer well-defined optimal solutions at the grid level, making them suitable for structured RL training. However, they are computationally intensive and not directly feasible for real-time navigation. By first training the RL model on an optimal grid-based algorithm, the model learns to generalize near-optimal paths directly from raw sensor data, thus reducing the dependency on a continuous preprocessing step and avoiding reliance on a graph-based planning structure.

The long-term objective of this research is to develop a reinforcement learning (RL) model capable of autonomous navigation, with the RL model relying on an optimal grid-based planning method to support its training process. This thesis focuses on identifying, designing, and testing an effective grid-based planning approach to achieve this goal. An essential step in realizing the proposed approach requires an optimal grid-based path planning algorithm that is capable of planning in dynamic environments and possibly considering weights during

the planning process. In order to address this requirement I propose to use the family of Jump Point Search algorithms detailed in chapter 2.

Following a discussion of the motivation behind this research, the current chapter provides a brief overview of the classification of different path planning approaches. It then reviews existing pathfinding algorithms that form the foundational basis for the further development of the proposed family of JPS algorithms. Chapter 2 introduces the Temporal Jump Point Search algorithm, detailing its core principles and functionality. To tailor the algorithm to our specific requirements, we extend its capabilities to operate in three-dimensional space, thereby enhancing its applicability to drone navigation. Additionally, we implement various speed-up techniques to improve its performance, ensuring that the modified algorithm can effectively handle the complexities of dynamic environments. Chapter 3 focuses on creating a dynamically feasible and collision-free trajectory for the drone, utilizing the path generated by the path planning algorithm. This is achieved through a multi-step procedure that operates without the need for grid map preprocessing. Chapter 4 presents and analyzes the benchmarks and simulation results, leading to conclusions regarding the effectiveness of the implemented path planning and trajectory generation algorithm.

1.1 Overview and classification of different approaches

Pathfinding algorithms play a fundamental role in numerous fields, from computer science and robotics to logistics and network routing. Their primary objective is to find the optimal or near-optimal path from a starting point to a goal point in a given environment, considering various constraints such as obstacles, traversal cost or time. Numerous algorithms have been developed to address distinct challenges, making it crucial to understand their classifications to select the best approach for a given application.

- **Sampling-Based vs. Optimization-Based**

One of the primary classifications of pathfinding algorithms is based on their underlying approach to exploration. Sampling-based algorithms, such as Rapidly-exploring Random Trees [10] construct a representation of the search space through random sampling and connect sampled points to form a network. These methods are particularly suitable for high-dimensional spaces or environments with complex obstacles. In contrast, optimization-based algorithms, like Dijkstra's [2] algorithm and A*[6], focus on finding the optimal path by evaluating the cost or distance between nodes in the graph. These algorithms often guarantee optimality but may struggle with large-scale or dynamic environments. [8]

- **Grid-based vs. Graph-based**

Grid-based algorithms discretize the environment into a grid of cells, often called an occupancy grid, where each cell represents a portion of the space. Gridmap-based algorithms can be computationally efficient for certain types of environments, especially when the environment is known and static. However, the grid resolution can significantly affect the computational complexity and accuracy of the pathfinding and trajectory generation process. Fine-grained grids provide more precise representations of the environment, but require more computational resources. Some gridmap-based algorithms are designed to handle dynamic environments where obstacles may move or change over time. On the other hand graph-based algorithms represent the environment as a graph, where nodes correspond to significant points or locations, and edges represent connections between these points. Graph-based algorithms often scale

better than grid-based methods, particularly in environments with sparse obstacles or complex geometry. They can adapt to the structure of the environment more dynamically, leading to more efficient path planning in various scenarios. [10] However, grid-based environment representation also holds a significant advantage, it allows the direct construction of the map from radar sensor or camera inputs. In contrast, with the graph-based representation, a preprocessing step with object detection is required to convert the input sensor data into an environment which can be used for planning.

- **Single-Agent vs. Multi-Agent**

Pathfinding algorithms can also be classified based on the number of agents or entities they consider during the search process. Single-agent algorithms, such as those mentioned earlier, aim to find the optimal path for a single entity navigating through the environment. Multi-agent algorithms, such as Cooperative A* and Multi-Agent Pathfinding [14] algorithms, account for the simultaneous movement of multiple agents, considering factors like collision avoidance and coordination.

Understanding these classifications provides a framework for comparing and selecting pathfinding algorithms based on the specific requirements and constraints of the application. In the following sections a the most important traditional path planning algorithms are detailed.

1.2 Temporal A*

Dijkstra's algorithm is a method for finding the shortest path between nodes in a weighted graph. The algorithm operates by maintaining a priority queue of nodes yet to be evaluated, continually expanding the frontier of explored nodes outward from the starting point. [2] At each step, it selects the node with the lowest tentative distance from the start and examines its neighboring nodes, updating their tentative distances if a shorter path is found. This process continues until the algorithm has evaluated every reachable node, yielding the shortest path from the starting node to all other nodes in the graph. Dijkstra's algorithm is particularly useful in scenarios such as navigation systems, network routing, and resource allocation, where finding optimal paths is essential.

Despite more sophisticated methods already existing Dijkstra's algorithm remains relevant because it computes the shortest path to every node, making it valuable in the context of training Reinforcement Learning models. Specifically, during the training phase, when new scenarios are generated, Dijkstra's algorithm is executed, and the resulting paths are stored. These precomputed paths can then be used to expedite the calculation of rewards during training. By referencing these stored results, the reward function can be computed more efficiently than if a faster planning algorithm, such as A*, were executed at each step of the training process.

The A* search algorithm, an improved variant of Dijkstra's algorithm is a widely-used optimal and complete pathfinding algorithm for weighted graphs. It belongs to the family of informed search algorithms, meaning it utilizes heuristic functions to guide its search towards the goal more efficiently than uninformed algorithms like Dijkstra's algorithm, however it only returns the shortest path between a specified source and destination node. [6]

In A*, each node is evaluated based on two cost metrics: the cost to reach that node from the start node (known as the "g-score") and an estimated cost to reach the goal node from that

node (known as the " h -score" or heuristic). The total cost of a node is the sum of these two scores, and A* prioritizes nodes with lower total costs, effectively biasing its search towards the most promising paths. Figure 1.1 shows a comparison between the set of nodes expanded by Dijkstra's algorithm and the A* algorithm respectively. The following visualization figures were generated with an open-source python library Motion Planning. [20]

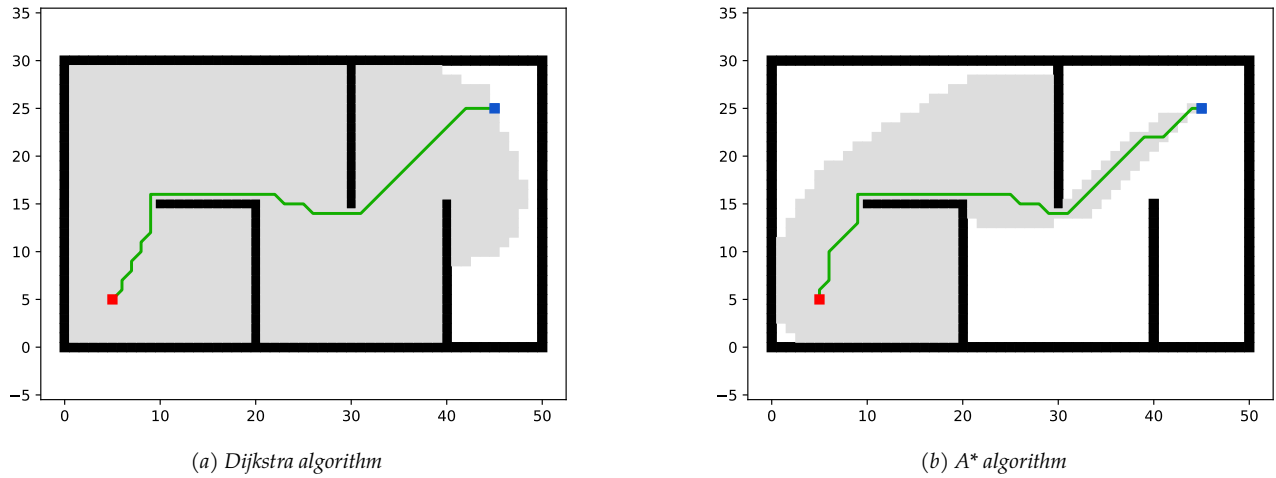


Figure 1.1: Algorithm efficiency comparison

The A* algorithm can be effectively extended to incorporate the temporal dimension in dynamic grid maps by simply adding a WAIT move to the set of possible actions. Although incorporating time introduces additional computational complexity, it does not equate to a full additional dimension, as time progresses only in one direction.

1.3 Safe Interval Path Planning

Safe Interval Path Planning (SIPP) is a pathfinding algorithm designed to generate collision-free paths for robotic agents in dynamic environments. It focuses on ensuring safety by considering both the current state of the environment and predictions of future movements of other agents or obstacles.

SIPP operates on a grid-based environment representation similar to other traditional pathfinding algorithms. It considers time as a continuous variable, allowing it to model the movement of agents. For each grid cell a set of safe intervals are calculated and maintained during planning, representing time intervals during which the agent can traverse the cell without colliding with obstacles or other agents. If a safe interval were extended one time step in either direction, it would result in collision. [13] Safe intervals are computed based on the current state of the environment and predictions of future movements of other agents and dynamic obstacles. The algorithm performs collision detection to determine if a given path segment between two states intersects with any obstacle in the environment. If a collision is detected, SIPP adjusts the safe intervals accordingly to avoid the collision. Optionally the algorithm checks whether the robot can feasibly traverse each interval without violating any motion constraints. While SIPP provides guarantees of collision-free paths, it may involve significant computational overhead due to collision checking and the need for continuous updating of safe interval sets. SIPP is suitable for applications where safety is paramount and the environment is predictable, such as autonomous navigation of mobile robots in warehouse environments with other dynamic agents and moving obstacles.

Chapter 2

Jump Point Search algorithms

The family of Jump Point Search (JPS) algorithms is designed to enhance the performance of the classic A* algorithm on uniform cost grids, representing a significant advancement in pathfinding within grid-based environments. JPS algorithms exploit the inherent symmetry and structure of uniform-cost grids to significantly reduce the number of nodes evaluated during the search process. The original JPS algorithm, which focuses on uniform cost grids in static environments, is detailed in section 2.2 and has both two- and three-dimensional implementations available. [4] The Temporal Jump Point Search (TJPS) algorithm, described in section 2.3 incorporates a temporal dimension, enabling the algorithm to handle dynamic environments on uniform-cost grids. [7] TJPS is a relatively new approach and is currently limited to two-dimensional grids. To facilitate the application of the TJPS algorithm for pathfinding in unmanned aerial vehicles (UAVs), the following chapters presents the development, implementation, and testing of the three-dimensional expansion of the algorithm. Another limitation of the algorithm is its constant planning speed, which requires further attention and improvement.

2.1 Preliminaries

Gridmap A gridmap is a two or three-dimensional data structure containing square shaped grid cells often used in the context of pathfinding. The cells of a binary gridmap are marked as either traversable or non-traversable, while in case of non-binary gridmaps each cell has a non-negative cost assigned to it. In the context of grid maps, the *n-connected* property defines the permissible adjacency or connectivity between grid cells (or nodes), directly influencing pathfinding algorithms. In a 4-connected grid, each cell is connected only to its immediate horizontal and vertical neighbors, so during path finding only orthogonal moves are allowed. In an 8-connected grid, each cell connects to both its immediate and diagonal neighbors. Similarly, in three-dimensional grids, a 6-connected configuration indicates that only orthogonal moves are allowed, excluding diagonal connections. The TJPS algorithm works on 4 and 6-connected uniform-cost gridmaps in two and three dimensions respectively. A time-expanded gridmap adds a discretized temporal dimension to the gridmap, and in three dimensions it is denoted by $grid_{x,y,z}(t)_{\{0,0,0\}}^{\{N_x,N_y,N_z\}}$. Conceptually, gridmaps corresponding to each timestep are stacked similarly to an image sequence in order to form a time-expanded gridmap, which will be referred to as a grid sequence in short.

Node In a gridmap each cell is uniquely identified by its spatial coordinates and is denoted as $n_{x,y,z}$. Within the context of gridmaps a Node combines the spatial coordinates of a grid cell with a time index. The $n(t) = n_{x,y,z}(t)$ notation is used to refer to a node $n_{x,y,z}$ at timestep t . The neighbors of $n(t)$ are time-indexed cells $n'(t + 1)$ where n' represents a node adjacent to n (for simplicity, the spatial coordinates are not displayed).

A Move action displaces the current node (also referred to as an agent) within the spatial dimensions and advances the time index by exactly one timestep. A move is considered valid if both the origin and destination cells are traversable. Since binary grids are considered each move action has a cost of 1. In addition to orthogonal spatial moves, time-expanded gridmaps allow for another type of action: the WAIT action.. Waiting has a cost of 1 and it advances the agent only in the time dimension from $n(t)$ to $n(t + 1)$ Diagonal moves are excluded from consideration in time-expanded grids, as their non-unit duration and cost introduces problems regarding discretization in the temporal dimension.

In order to understand the algorithmic description of TJPS the following additions are necessary. Two nodes are equal if both their spatial and time indexes are equal. The plus sign operator between nodes and moves means that a move is applied to a node and a new node is created. During this operation both the spatial and temporal coordinates, the parent move and the cost-so-far is updated accordingly. Given that gridmaps have unit dimensions, a node's coordinate, index, or vector representation are effectively interchangeable in implementation. In the implementation, each node keeps track of the following parameters:

- c : spatial coordinates of the current cell
- t : time
- pm : parent move (the move that resulted in the current node)
- g : accumulated cost or cost-so-far
- h : heuristics or cost-to-go
- jp : start node of the current jump

Path, plan, problem instance and objective A path $path = [n_0(t), \dots, n_k(t + k)]$ is an ordered sequence of nodes. A path is valid if every location on the path is traversable and if there exists a valid move from each location on the path to the next. A plan is an ordered sequence of move actions $\vec{plan} = [m_1, \dots, m_k]$ which realizes the path by moving the agent in the time-expanded grid from start to destination. A problem instance is pair of a non-time indexed *start* and *goal* locations and the corresponding time-expanded grid sequence containing both static and dynamic obstacles. The objective of the planning is to find an optimal (w.r.t. to the total cost, or equivalently the total time) path in the time-expanded grid.

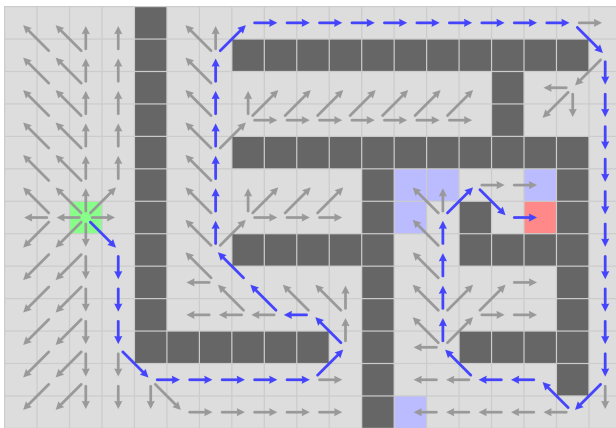
Acquiring the time-expanded gridmap A key distinction between grid-based and graph-based dynamic environments lies in the requirement for obstacle detection and trajectory tracking. In grid-based environments, such detection is not strictly necessary, as obstacles are not treated as individual entities. Grid-based dynamic planning approaches need the time-expanded grid map predicted into the future. Of course the production of this time-expanded grid map can be achieved by detecting the obstacles and then rendering them, but there are also newer methods employing neural networks, which can predict the entire dynamic occupancy grid into the future directly.

2.2 Jump Point Search

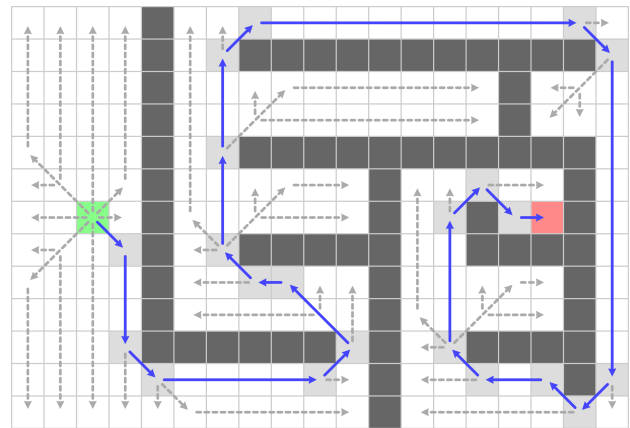
Jump Point Search is an optimized pathfinding algorithm designed to enhance the performance of the classic A* algorithm on uniform cost grids. JPS aims to accelerate the pathfinding process by efficiently pruning unnecessary nodes and exploring only the most promising directions. [4] The core innovation of the JPS algorithm is its pruning mechanism, which skips over multiple nodes by jumping directly to the most promising points, referred to as jump points. These points are critical turning points where the path changes direction or encounters an obstacle. By only evaluating these jump points rather than every single grid node JPS reduces the overall search space, thus speeding up the pathfinding process.

The algorithm reduces symmetries in the search procedure by means of graph pruning, eliminating certain nodes in the grid based on assumptions that can be made about the current node's neighbors, as long as certain conditions relating to the grid are satisfied. As a result, the algorithm can consider long jumps along straight lines in the grid, rather than the small steps from one grid cell to the next that ordinary A* algorithm considers. [4] The JPS algorithm is particularly effective in open spaces or environments with long, obstacle-free corridors. In sparse environments, where long jumps are possible, JPS is more than an order of magnitude faster than the classic A* algorithm [4] [5], while in dense environments it has about the same worst-case execution time. A visual understanding of the JPS algorithm can be attained by consulting the following website. [19]

Figure 2.1. shows a comparison of the path finding process of the A* and JPS algorithms. The start and destination cells are marked with green and red respectively, while obstacles are marked with dark grey in each figure. Cells marked with light gray belong to the CLOSED list, while blue cells are in the OPEN list. As it can be seen on Figure 2.1a in this complex scenario all nodes are expanded and added to the OPEN list and most of them are also evaluated before reaching the target. On Figure 2.1b dashed arrows mean that during the jumping and pruning process introduced for symmetry breaking the traversed cells are checked for being forced neighbors or natural successors, but they are not added to the OPEN list right away, hence reducing the priority queue operations by a huge margin.



(a) A* algorithm



(b) Jump Point Search Algorithm

Figure 2.1: Comparison of algorithms [19]

2.3 Temporal Jump Point Search

Temporal Jump Point Search (TJPS) is an extension of JPS that incorporates time as a dimension, allowing it to handle dynamic environments where obstacles may move over time. In addition to spatial coordinates, TJPS considers time as a dimension, allowing it to model the evolution of the environment over time. In such settings, obstacles can appear and disappear, potentially preventing the agent from reaching its target and introducing significant complexities to the search process. The temporal dimension increases the size of the search space, which often contains many symmetric paths that are indistinguishable except for the sequence of grid moves. To address these challenges TJPS identifies and breaks these symmetries, thereby enhancing performance.

Potential use cases include automated factory or logistical warehouse environments where the future trajectories of other agents in the environment are known, allowing for the generation of discrete occupancy grids, or scenarios where the trajectories of multiple dynamic agents are planned concurrently using the same algorithm. In the latter case, two types of algorithms can be distinguished: competitive or simultaneous algorithms, which plan agent trajectories in a quasi-parallel manner, and conformist or non-simultaneous algorithms, which plan paths individually while considering previously planned agent paths as dynamic obstacles. Another application area is computer games, where dynamic environments may change due to scripted or timed events. [7] In these environments although all dynamic obstacle trajectories are known, finding a time-optimal path is considerably more challenging than conventional grid-based pathfinding problems. The number of states within the environment increases because a temporal dimension is added, and the branching factor of the agent increases from 4 to 5 due to the possibility of a WAIT action. Additionally time progression is unidirectional, introducing further complications into the search process. [7]

Symmetry in pathfinding refers to paths within a time-expanded grid that have identical costs and start and goal locations, differing only in the order of moves. Temporal Jump Point Search aims to break both spatial and temporal symmetries in time-expanded and 4-connected grids, by introducing a new canonical ordering, where the order of moves are constrained. Figure 2.2b. illustrates an example where multiple equivalent paths exist between the start and goal cells. The only difference is in the sequence of actions. Figure 2.2b. shows a different type of equivalence which we call temporal symmetries. The spatially canonical path of Figure 2.2a is fixed, but a temporal obstacle is placed at cell *E1* at timesteps 7-9. In this example the optimal path from start to goal necessarily contains three WAIT actions, the canonical path only differs from the other equivalent paths in where the WAIT actions are taken. For any optimal time-expanded grid path there exists an equivalent path, with the same cost, which is canonically ordered. The ordering allows TJPS to prune symmetric path permutations per expanded node which can dramatically improve search performance. [7]

The following paragraphs contain some excerpts from the original TJPS article discussing the two-dimensional case, but have been modified and extended to discuss the three-dimensional extension of the algorithm. [7]

2.3.1 Idea

Temporal Jump Point Search introduces a new canonical ordering called Vertical-then-Horizontal-then-Wait (VHW). This canonical ordering generates paths with vertical and horizontal moves that appear as early as possible and wait moves, which are

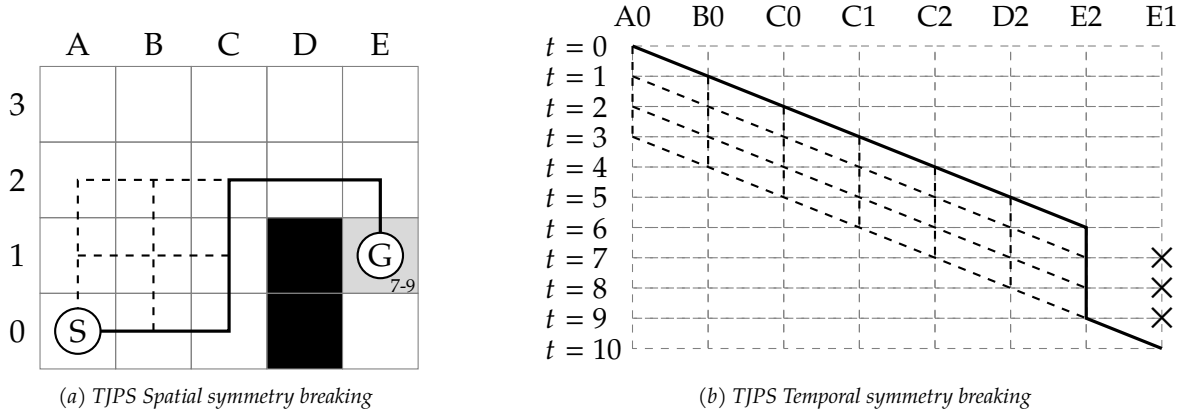


Figure 2.2: TJPS symmetry breaking [7]

taken as late as possible. The original article shows that for any optimal path in a time-expanded grid there exists an equivalent path, with the same cost, which is also VHW. [7] The original publication also establishes the optimality of the TJPS algorithm. Given that the fundamental concept remains consistent with our extended algorithm, this proof continues to be valid. [7]

In order to generalize the idea and extend TJPS to three-dimensional grids an XYZW canonical ordering is introduced to break spatial and temporal symmetries. The XYZ letters in the notation refer to the corresponding spatial coordinate axes and thus the spatial symmetry breaking, while W refers to the WAIT move and the temporal symmetry breaking. This ordering generates sub-paths in which the order of the moves is constrained according to the XYZW canonical ordering between every temporal jump point. TJPS restricts the generation of successor nodes to those where the current path can be continued in a canonical manner. The canonical property prevents the exploration of paths that are symmetric permutations of one another. [7]

Definition: XYZW canonical path A path is XYZW canonical if its corresponding plan satisfies the ordering property and the backtrack-free property.

The ordering property expresses that in the plan associated with the path there is no subsequent pair of move actions m_i and m_{i+1} which can be swapped to generate a new valid path from starting point to destination. Cases:

- m_i : Move along the Y axis, m_{i+1} : Move along the X axis
- m_i : Move along the Z axis, m_{i+1} : Move along the Y axis
- m_i : Move along the W axis, m_{i+1} : Move along the Z axis

Backtrack-free property: There exists in the plan no subsequent pair of actions with $m_i \neq m_{i+1}$ which could be replaced with a pair of subsequent WAIT actions to generate a new valid path from start to destination.

Every optimal path can be transformed into an equivalent path which is XYZW canonical, by reordering the moves. [7] Figure 2.2a shows an example where the ordering property breaks spatial symmetries in two-dimensions, and Figure 2.2b shows an example of the temporal symmetry breaking. Notice that WAIT moves are only utilized after all possible spatial moves have been used.

2.3.2 Algorithm base

At its core, the TJPS algorithm operates similarly to the traditional A* algorithm, utilizing a priority queue to manage the nodes that need to be evaluated. The algorithm begins by initializing the OPEN list with the *Start* node and the CLOSED set as empty. As the algorithm progresses, it repeatedly selects the *Node* with the lowest total cost from the OPEN list. Algorithm 1 presents the core of the TJPS algorithm.

Algorithm 1 Temporal Jump Point Search: main function

Input: $grid_{x,y,z}(t)_{\{0,0,0\}}^{\{N_x,N_y,N_z\}}$: time expanded grid, $Start_{\{t=0,x,y,z\}}$, $Goal_{\{t=k,x,y,z\}}$

```

1:  $OPEN \leftarrow Start$  ▷ Priority queue
2:  $CLOSED \leftarrow \emptyset$ 
3: while Node in OPEN do
4:   if Node in CLOSED then
5:     continue
6:   if Node == Goal then
7:      $CLOSED \leftarrow CLOSED \cup Node$ 
8:     return  $extractPath(CLOSED)$ 
9:    $OPEN \leftarrow OPEN \cup getSuccessors(Node)$ 
10:   $CLOSED \leftarrow CLOSED \cup Node$ 
11: return None
```

In each iteration of the processing while loop nodes are systematically evaluated to determine the optimal path. Within each iteration of this loop, the algorithm executes the following steps:

- *Node* selection: The *Node* with the lowest total cost, calculated as the sum of the *g*-cost and *h*-cost, is popped from the OPEN list.
- CLOSED set check: A verification is performed in order to determine if *Node* has already been evaluated. If the *Node* is found in CLOSED, it is skipped and the execution continues.
- Goal check: If the current *Node* equals the *Goal* node, then *Node* is added to the CLOSED set and the algorithm reconstructs the path from the CLOSED set and terminates.
- Successor generation: The algorithm retrieves the successors of the current *Node* with the *getSuccessors* method outlined in Algorithm 3.

After the *getSuccessors* method is executed and the returned jump points are added to the OPEN the current *Node* is added to the CLOSED set. If the OPEN list empties before finding the goal the algorithm returns with *None*, since it could not find a path.

2.3.3 Pruning rules

Neighbor pruning rules are applied during node expansion in order to reduce the set of possible moves. These rules aim to eliminate moves that would fail to preserve the XYZW canonical ordering. The pruning rules are visualized in Figure 2.3.

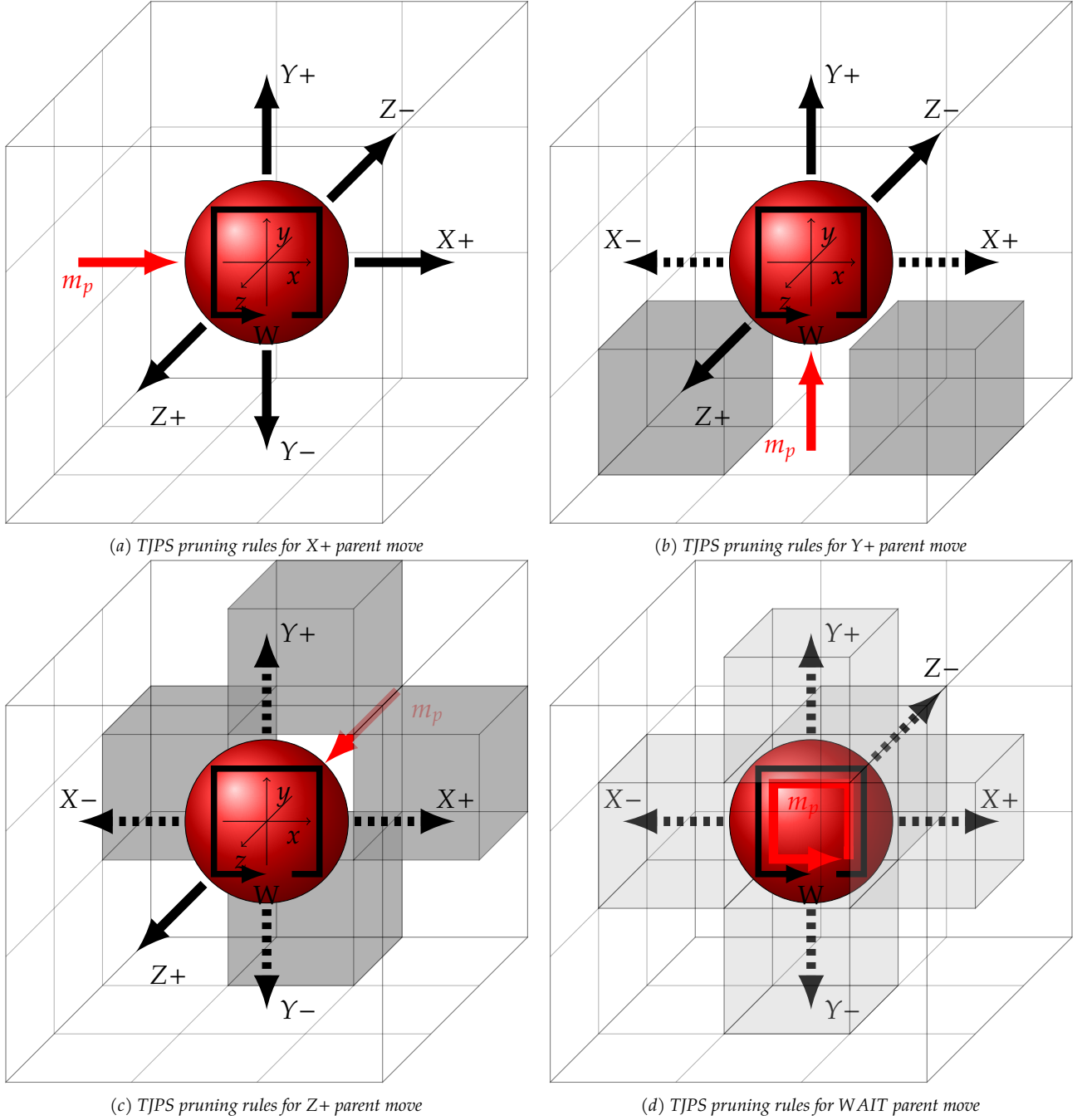


Figure 2.3: TJPS pruning rules

The `getCanonicalMoves` method, outlined in Algorithm 2 determines the set of possible moves based on the current *Node* and its corresponding parent move m_p . In contrast to the original implementation this function also verifies whether the current *Node* is a forced node.

Figure 2.3 illustrates a three-dimensional XYZ coordinate system. At the origin the red sphere represents the investigated agent or in other words the current *Node*. The red arrow, labeled m_p indicates the parent move of the agent, reflecting the step taken by the agent along the arrow in the previous iteration. The black arrows represent the possible moves that can be

Algorithm 2 Temporal Jump Point Search: getCanonicalMoves function

Input: $n(t)$: current node, m_p : parent move

```
1: Forced = False
2: if  $m_p == X+$  then
3:    $M = \{X+, Y+, Y-, Z+, Z-, W\}$ 
4: if  $m_p == X-$  then
5:    $M = \{X-, Y+, Y-, Z+, Z-, W\}$ 
6: if  $m_p == Y+$  then
7:    $M = \{Y+, Z+, Z-, W\}$ 
8:   if  $n(t, x+1, y-1, z)$  is obstacle and  $n(t, x+1, y, z)$  is not obstacle then
9:      $M \leftarrow M \cup X+$ ; Forced = True
10:  if  $n(t, x-1, y-1, z)$  is obstacle and  $n(t, x-1, y, z)$  is not obstacle then
11:     $M \leftarrow M \cup X-$ ; Forced = True
12: if  $m_p == Y-$  then
13:    $M = \{Y-, Z+, Z-, W\}$ 
14:   if  $n(t, x+1, y+1, z)$  is obstacle and  $n(t, x+1, y, z)$  is not obstacle then
15:      $M \leftarrow M \cup X+$ ; Forced = True
16:   if  $n(t, x-1, y+1, z)$  is obstacle and  $n(t, x-1, y, z)$  is not obstacle then
17:      $M \leftarrow M \cup X-$ ; Forced = True
18: if  $m_p == Z+$  then
19:    $M = \{Z+, W\}$ 
20:   if  $n(t, x+1, y, z-1)$  is obstacle and  $n(t, x+1, y, z)$  is not obstacle then
21:      $M \leftarrow M \cup X+$ ; Forced = True
22:   if  $n(t, x-1, y, z-1)$  is obstacle and  $n(t, x-1, y, z)$  is not obstacle then
23:      $M \leftarrow M \cup X-$ ; Forced = True
24:   if  $n(t, x, y+1, z-1)$  is obstacle and  $n(t, x, y+1, z)$  is not obstacle then
25:      $M \leftarrow M \cup Y+$ ; Forced = True
26:   if  $n(t, x, y-1, z-1)$  is obstacle and  $n(t, x, y-1, z)$  is not obstacle then
27:      $M \leftarrow M \cup Y-$ ; Forced = True
28: if  $m_p == Z-$  then
29:    $M = \{Z-, W\}$ 
30:   if  $n(t, x+1, y, z+1)$  is obstacle and  $n(t, x+1, y, z)$  is not obstacle then
31:      $M \leftarrow M \cup X+$ ; Forced = True
32:   if  $n(t, x-1, y, z+1)$  is obstacle and  $n(t, x-1, y, z)$  is not obstacle then
33:      $M \leftarrow M \cup X-$ ; Forced = True
34:   if  $n(t, x, y+1, z+1)$  is obstacle and  $n(t, x, y+1, z+1)$  is not obstacle then
35:      $M \leftarrow M \cup Y+$ ; Forced = True
36:   if  $n(t, x, y-1, z+1)$  is obstacle and  $n(t, x, y-1, z)$  is not obstacle then
37:      $M \leftarrow M \cup Y-$ ; Forced = True
38: if  $m_p == W$  then
39:    $M = \{W\}$ 
40:   if  $n(t, x+1, y, z)$  is obstacle and  $n(t+1, x+1, y, z)$  is not obstacle then
41:      $M \leftarrow M \cup X+$ ; Forced = True
42:   if  $n(t, x-1, y, z)$  is obstacle and  $n(t+1, x-1, y, z)$  is not obstacle then
43:      $M \leftarrow M \cup X-$ ; Forced = True
44:   if  $n(t, x, y+1, z)$  is obstacle and  $n(t+1, x, y+1, z)$  is not obstacle then
45:      $M \leftarrow M \cup Y+$ ; Forced = True
46:   if  $n(t, x, y-1, z)$  is obstacle and  $n(t+1, x, y-1, z)$  is not obstacle then
47:      $M \leftarrow M \cup Y-$ ; Forced = True
48:   if  $n(t, x, y, z+1)$  is obstacle and  $n(t+1, x, y, z+1)$  is not obstacle then
49:      $M \leftarrow M \cup Z+$ ; Forced = True
50:   if  $n(t, x, y, z-1)$  is obstacle and  $n(t+1, x, y, z-1)$  is not obstacle then
51:      $M \leftarrow M \cup Z-$ ; Forced = True
52: if  $m_p == \text{None}$  then
53:    $M = \{X+, X-, Y+, Y-, Z+, Z-, W\}$ 
54: for  $m_n$  in  $M$  do
55:    $n(t+1) = n(t) + m_n$ 
56:   if  $n_n(t+1)$  is obstacle then
57:      $M \leftarrow M \setminus \{m_n\}$ 
58: return  $M, \text{Forced}$ 
```

► Start cell

taken by the agent in order to preserve the XYZW canonical property of the path. Dark gray cubes represent obstacles that will also be present in the next step, while light gray cubes denote obstacles currently present that will be cleared in the subsequent step.

The first three subfigures in Figure 2.3 illustrate the spatial pruning mechanism employed by TJPS. In Figure 2.3a the parent move of the agent is $X+$. According to the XYZW ordering only the $X-$ move is pruned by the canonical ordering since it violates the backtrack-free property. The $X-$ move is pruned because the agent could have simply waited to reach the same location at an equivalent cost. All possible moves are included in the set of moves M except $X-$, as it is described in Algorithm 2 at line 2 and 3.

In Figure 2.3b the agent's parent move is $Y+$. According to the XYZW canonical ordering moves parallel to the X axis are pruned, because they violate the ordering property, and the move $Y-$ is also pruned for violating the backtrack-free property. This figure also presents obstacles adjacent to the agent's path. If an obstacle is located at $(1,-1,0)$ relative to the agent (obstacle on the right in the figure) the $X+$ move cannot be pruned, as the equivalent path that would satisfy the canonical ordering is obstructed. In such scenarios the $X+$ move is considered *forced* to be generated. A *Node* is considered *forced* if a corresponding forced move is identified and has to be generated in order to maintain the completeness of the search. It is important to note that the presence of both obstacles is not required for the *Node* to be considered a forced node. Consequently, the aforementioned rule must also be applied in other directions preceding the examined move in the canonical ordering as detailed in Algorithm 2 from lines 7 to 11.

In Figure 2.3c the agent's parent move is $Z+$. According to the XYZW ordering all moves are pruned except for the $Z+$ and *WAIT* moves. If obstacles are present at locations marked with dark cubes in Figure 2.3c then moves represented with dashed arrows are forced to be generated on a per direction basis, as it is shown in Algorithm 2 from lines 19 to 27.

Figure 2.3d illustrates the temporal pruning, where the agent's parent move is the *WAIT* move. In accordance with the XYZW ordering all moves are pruned except for the *WAIT* move. The light cubes in the figure represent obstacles that will be cleared in the next timestep. In such cases, forced moves must be generated in every direction where a clearing obstacle is present, as described in Algorithm 2 from lines 39 to 51. Note that if there are no clearing obstacles present around the agent, all moves are pruned except for the *WAIT* move, effectively reducing the branching factor.

In Algorithm 2, there exists a scenario where the parent move is *None*, which can only occur in the case of the *Start* node, thus in this case all moves are permitted. At the end of the algorithm after applying the corresponding pruning rules and identifying forced moves it is checked whether applying the moves currently in the set M would lead to a collision. If a move m_n results in collision from the current *Node*, it is excluded from the set of possible moves M .

2.3.4 Jumping rules

Pruning rules can improve the search performance by reducing the number of successor nodes generated at each expansion step for the current node. Additionally, pruning rules improve efficiency within the set of candidate successors for which we have already established there exists a XYZW canonical continuation of the path. When applied to such candidate successors, pruning rules can often reduce the branching factor to zero or one. Instead of adding such nodes to the OPEN list they are immediately expanded and the pruning rules are reapplied

recursively. This recursive process continues until either the next move is invalid indicating a dead-end branch or until the recursion yields a node which is then returned as a successor. Successors yielded by the recursion are called temporal jump points. [7]

The previously described principles are realized in Algorithm 3. The `getSuccessors` method is responsible for expanding the temporal jump points extracted from the OPEN list. Initially the

Algorithm 3 Temporal Jump Point Search: `getSuccessors` function

Input: $n(t)$: current node, m_p : parent move, g :goal

- 1: $S \leftarrow \emptyset$
 - 2: $M \leftarrow \text{getCanonicalMoves}(n(t), m_p)$
 - 3: **for all** $m_n \in M$ **do**
 - 4: $n(t+1) \leftarrow n(t) + m_n$
 - 5: $S \leftarrow S \cup \text{jump}(n(t+1), g)$
 - 6: **return** S
-

set of successor nodes S is initialized as an empty set. Subsequently, the `getCanonicalMoves` function is invoked to gather all possible moves M that can lead to a canonical continuation of the path from the current *Node*. For each potential move m_n a next node is generated by applying the move to the current *Node*, then a jump operation is executed and the jump points returned by the jump are added to the set of successors. Note that in all presented algorithms m_p corresponds to the parent move of the current *Node* which is stored within the *Node* and it is not always displayed separately.

Definition: Temporal Jump Point A time-indexed node $n(t)$ is generated by applying the move m_p at location p during timestep $(t-1)$, such that $p(t-1) + m_p = n(t)$. The node $n(t)$ along with parent move m_p is defined as a temporal jump point if: $n(t)$ is the target, or if $n(t)$ has a valid next move $m_n \neq m_p$ such that:

- the plan $p(t-1) + m_p + m_n$ is valid
- the plan $p(t-1) + m_n + m_p$ is invalid
- m_n comes before m_p in the XYZW canonical ordering

The jump method — detailed in Algorithm 4 — implements the core of the recursive jumping procedure. In the initial step, the set of canonical moves M (assigned to the current *Node* and its corresponding parent motion) is obtained using the `getCanonicalMoves` method.

According to the definition a node $n(t)$ is a temporal jump point if it is either the goal node or has a forced neighbor. A forced neighbor can only be locally reached by going against the canonical ordering, since the preferred plan is invalid. [7] Upon detecting a temporal jump point the recursion terminates and the current node is returned. The set s represents the collection of successor nodes generated during the jump operation.

In order to preserve the XYZW canonical ordering, the canonical continuations in M are processed in a specific order during recursion. Nodes that can be reached with a WAIT action are prioritized first, followed by spatial moves in reverse order Z, Y and X. This ordering ensures that the recursion only proceeds along paths that are XYZW canonical. This approach means that, before each move in the Z direction, the algorithm explores if the path can navigate around obstacles via a series of move actions in the X and Y planes. Similarly, before any move in the Y direction, the algorithm checks if the path can bend around obstacles with moves in the X direction. [7] This ordering is implemented in lines 5 to 16 of Algorithm 4.

Algorithm 4 Temporal Jump Point Search: jump function

Input: $n(t)$: current node, g :goal

```
1:  $M \leftarrow \text{getCanonicalMoves}(n(t), m_p)$ 
2: if  $n(t) == g$  or  $\exists m_n \in M | m_n$  is forced then
3:   return  $(n(t), m_p)$ 
4:  $S \leftarrow \emptyset$ 
5: if WAIT  $\in M$  and  $t \leq$  timestep of last dynamic obstacle disappear then
6:    $n(t+1) \leftarrow n(t) + \text{WAIT}$ 
7:    $S \leftarrow S \cup \text{jump}(n(t+1), m_n)$ 
8: for all  $m_n \in M | m_n$  is a move along Z do
9:    $n(t+1) \leftarrow n(t) + m_n$ 
10:   $S \leftarrow S \cup \text{jump}(n(t+1), m_n)$ 
11: for all  $m_n \in M | m_n$  is a move along Y do
12:    $n(t+1) \leftarrow n(t) + m_n$ 
13:    $S \leftarrow S \cup \text{jump}(n(t+1), m_n)$ 
14: for all  $m_n \in M | m_n$  is a move along X do
15:    $n(t+1) \leftarrow n(t) + m_n$ 
16:    $S \leftarrow S \cup \text{jump}(n(t+1), m_n)$ 
17: return  $S$ 
```

The algorithm incorporates another condition and an associated constant, the timestep of last dynamic object disappear. After the last dynamic obstacle has cleared from the grid, further waiting becomes unnecessary, as the environment has become static, if a valid path has not been discovered by this time, it implies either that no path exists or that further waiting and recursion would lead to a dead-end. In scenarios where the algorithm operates within a constantly dynamic environment with persistent dynamic obstacles in the search space, a moving time window can be utilized, governed with a predefined MAX_WAIT parameter. This time counter starts from the current node's last timestep before any WAIT move is taken. This way the algorithm may never discover path's if the MAX_WAIT parameter is not sufficiently large, but the computation time can be accelerated greatly by limiting the recursion depth associated with WAIT actions.

2.3.5 Path extraction

The algorithm takes as input a time-expanded grid that represents the dynamic environment, along with the start and goal coordinates for planning. While TJPS outputs a sequence of temporal jump points with (x, y, z) and *time* coordinates, it does not generate an immediately executable path. However from the sequence of temporal jump points each step of the path can be reconstructed using the canonical ordering rules. In order to gain the extracted path which is a step-by-step sequence of neighboring nodes we compute the difference between each pair of adjacent temporal jump points along the X, Y, Z and T dimensions. Then, create an XYZT canonical sub-path by following a sequence of dX, dY, dZ spatial moves, and dT wait steps. In other words to derive the complete canonical path we simply take the difference of every adjacent temporal jump point and divide it to motion primitives parallel to coordinate axes which are then ordered according to the canonical ordering. This ensures that every sub-path between two temporal jump points follows the canonical ordering, guaranteeing both validity and optimality. The path extraction process is detailed in Algorithm 5.

Algorithm 5 Temporal Jump Point Search: `extractPath` function

Input: $JP_list = [Start_{\{t=0,x,y,z\}}, \dots, n_{\{t,x,y,z\}}, \dots, Goal_{\{t=k,x,y,z\}}]$

```
1:  $t = 0$ 
2:  $path[t] \leftarrow Start$ 
3: for  $\{i = 0; i < (len(JP\_list) - 1); i++\}$  do
4:    $n_s = JP\_list[i]$ 
5:    $n_g = JP\_list[i + 1]$ 
6:   while  $n_s.t < n_g.t$  do
7:      $d = n_g.(x, y, z) - n_s.(x, y, z)$ 
8:      $t \leftarrow t + 1$ 
9:     if  $d.x \neq 0$  then
10:       $m \leftarrow (sgn(d), 0, 0)$ 
11:       $s \leftarrow s + m$ 
12:     else if  $d.y \neq 0$  then
13:       $m \leftarrow (0, sgn(d), 0)$ 
14:       $s \leftarrow s + m$ 
15:     else if  $d.z \neq 0$  then
16:       $m \leftarrow (0, 0, sgn(d))$ 
17:       $s \leftarrow s + m$ 
18:     else
19:        $s.t \leftarrow s.t + 1$ 
20:      $path[t] \leftarrow s.(x, y, z)$ 
```

2.4 Development and optimization ideas

Multi-step planning with corridors Due to the algorithm’s suboptimal scalability in large search spaces, planning speed can be enhanced by implementing a multi-step planning procedure. In this approach, an optimal path is initially determined on a lower-resolution grid derived from the original grid. Subsequently, the search space of the original resolution map is constrained by creating a corridor alongside the previously identified path. This method of search space reduction can effectively decrease overall planning time, resulting in an almost linear relationship with respect to grid size, as demonstrated in section 4.1.

To generate the bounded grid sequence, the following algorithm is employed. The primary input parameter is the diameter of the corridor to be constructed around the path planned on the reduced grid sequence. In the first step, a fully occupied grid sequence is created with the same dimensions as the original grid sequence. In the second step we iterate through every path point and subtract a static ellipsoid shaped obstacle with the given diameter from the fully occupied grid sequence. It is important to note that generating numerous static obstacles may slow down the calculation; therefore, static ellipsoids are subtracted only when the distance from the previous one along the path exceeds one-third of the given diameter. After creating and subtracting this ellipsoids from the fully occupied grid sequence, it is added to the original grid sequence frame by frame and optionally clipped. Essentially, the original grid sequence and the corridor grid sequences are combined using a logical OR operation if we consider the grid sequence as a binary occupancy grid.

Multi-step planning presents an additional application in sequential multi-agent pathfinding. In scenarios where multiple agents must be dispatched from a starting area to a destination area while avoiding collisions with one another and other dynamic obstacles in the environment, this approach proves beneficial. After the optimal path is computed on the

reduced grid, and the bounded grid sequence is created, it can be utilized to plan for multiple agents more efficiently than if each agent were to be planned individually using the original resolution gridmap.

Jump limits Jump limits can be applied within the jumping mechanism in order to prevent excessively long, inefficient jumps resulting in the algorithm scanning uninteresting areas. By using prior knowledge of the environment — such as dimensions, feature and dynamic obstacle sizes — these limits can be adjusted so that the algorithm does not jump too far from the originating node. Completeness and optimality remains intact, as when the jump limit is reached, the currently scanned node is added to the OPEN list. Even in an unfortunate case of limiting the jump headed in the correct direction towards the goal, the same node will be processed again in the next iteration after being popped from the OPEN list. Direction-specific jump limits are also possible, which can be especially useful in path planning for fixed-wing aircraft, as they allow limiting the height component of jumps.

In order to incorporate jump limits into the algorithm, the origin node of a jump, denoted by *jump_start* is stored within the *getSuccessors* function (described in Algorithm 3) prior to calling the *jump* method (described in Algorithm 4). Within the *jump* method, at the line corresponding to the direction where a jump limit is desired, the algorithm checks if the current node $n(t + 1)$ lies within the *jump_limit* radius of *jump_start*. If this condition is met, the algorithm continues jumping in the specified direction; otherwise, it adds the current node to the OPEN list.

Non-uniform cost environment A significant limitation of the (Temporal) Jump Point Search algorithm is its reliance on uniform-cost grids. This constrain enables the application of jumping and pruning rules, which contribute to the algorithm’s efficiency. The challenge, however, lies in modifying the algorithm to handle weighted grids while maintaining both optimality and high execution speed. Incorporating weighted grids would be advantageous, as it would allow the algorithm to avoid potentially hazardous areas or minimize the time spent within them.

When applying the basic JPS algorithm to weighted grids, a key issue arises: the assumptions regarding path symmetry, along with the derived jumping and pruning rules, are no longer valid. To address this, the creators of the original JPS and TJPS algorithms developed an adaptation for weighted grids known as Weighted JPS [1]. In this variant, the pruning rules are exchanged with an online Dijkstra-based approach that examines the current node’s 3x3 dimensional neighborhood tile. This modified approach prunes any moves leading to nodes where a more cost-effective path exists directly from the parent node. In other words, if a lower-cost path bypassing the current node (from parent node to next node) is available, the move generating that next node is pruned from the set of possible moves. The jumping rules are also modified: rather than jumping until a forced node is identified according to the previously described very specific rules or the jump is terminated, the algorithm now jumps until the 3x3 neighborhood of the current node contains more than one weight value. In other words, the jump continues only until a change in terrain is encountered.

Compared to the A* algorithm, which natively supports weighted grids, the WJPS algorithm introduces an additional overhead due to the need to compute an online Dijkstra search at each step within a jump operation. Furthermore, upon encountering a terrain change, the algorithm generates significantly more jump points, resulting in the repeated scanning of the same regions during the evaluation of these generated jump points. This repeated scanning imposes a greater computational demand than the online Dijkstra calculation itself. In their

original article [1], the authors propose a solution to address the repeated scanning issue; however, it involves the use of either online caching or a pre-generated cache. Applying these rules dynamically during execution is challenging, even in two-dimensional scenarios, not to mention the 3-dimensional case. The authors' benchmarks indicate that WJPS outperforms baseline A* only when caching is applied. My own Python implementation confirms these findings, leading to the decision not to pursue a 3D extension of the WJPS algorithm.

Incorporating a temporal dimension into the WJPS algorithm presents a significant, if not impossible challenge, primarily because the online Dijkstra-based pruning rules would also have to take time into account, imposing a computational burden. Additionally, the presence of dynamic obstacle boundaries — already imposing a considerable issue in terms of re-scanning even without movement — would generate forced jump points at every boundary position and for each time step. This would lead to repeated searching of the same areas to a far greater extent than required by the Temporal A* algorithm. Consequently, optimal path planning on temporal weighted grids is best suited to the Temporal A* algorithm. However, if optimality is not required and a sub-optimal solution is acceptable for a predefined set of problems, an alternative approach can be developed.

One of our goals is to identify or create an algorithm capable of efficient path planning in three-dimensional environments featuring static terrain, a limited number of expansive radar fields and numerous dynamic obstacles. In this context, the weight assigned to a radar field corresponds to the probability of detection. While radar fields are traversable, prolonged presence within them raises the likelihood of detection.

The proposed approach involves modeling weighted radar fields as layered structures with occupancy values that decrease in incremental steps from the center outward. By limiting the number of layers, multiple planning iterations can be executed sequentially with varying threshold values. Applying a threshold effectively converts the grid to a binary representation with flexible boundaries, allowing the path obtained from sequential planning to traverse only through the outermost layer required to achieve an obstacle-free path. Once a path is identified, it can be refined in a post-optimization step, detailed later in section 3.1. The purpose of this post-optimization step is to adjust the path points in the direction of the negative gradient of the discrete local potential field, thereby guiding the path away from obstacles and regions with higher occupancy values. While this method sacrifices optimality, it remains computationally efficient and can successfully identify a path even in scenarios where the standard TJPS algorithm would fail to find a path.

Incorporating this concept into the planning algorithm does not require any special modifications; it simply requires the introduction of a threshold value within the obstacle function utilized in the previous algorithms to determine whether a cell is classified as an obstacle at a given timestep or not.

Chapter 3

Trajectory generation

The objective of this chapter is to present the complexity of continuous trajectory generation within the context of grid based path planning algorithms, particularly focusing on the transition from discrete to feasible continuous paths while considering static and dynamic obstacles. The aim is to propose a methodological framework that addresses the challenges inherent in this process, resulting in the development of a streamlined algorithm suitable for trajectory generation, especially in dynamic environments, while ensuring collision avoidance and adherence to specified constraints.

Before delving deeper into the topic, it's crucial to distinguish between a path and a trajectory: A path denotes a sequence of points or states detailing the route an object traverses from one location to another, typically representing the intended or planned course. On the other hand, a trajectory is a specific instance of motion, describing the actual movement of an object through space and time. It includes not only the path followed but also information about the object's velocity, acceleration, and other dynamic properties over time. Given the focus of this thesis on algorithms discrete in both time and space dimensions, it becomes essential to clarify another distinction. A time-indexed path implies that each path point is temporally constrained, necessitating consideration during trajectory generation.

The proposed framework applies a three step process to complete the task. The first step is Discrete path optimization, which make the discrete path points more suitable for collision free continuous path fitting. The purpose of the second step is to fit a collision free continuous trajectory to the improved discrete path. In the third step the drone's dynamic properties are finally taken into consideration and a dynamically feasible path is generated with the Minimum snap trajectory method.

3.1 Discrete path optimization

Thus far, all examined grid-based path planning algorithms have yielded discrete paths. While algorithms tailored exclusively for static maps provide only the discrete path points, those designed for dynamic scenarios additionally provide the time indices alongside the path points due to the planning being conducted with constant velocity in consideration.

Algorithms that solely rely on orthogonal moves for planning tend to generate a path with a choppy trajectory, which isn't optimal for direct trajectory fitting. Adding diagonal moves to the planning process presents another issue: corner skipping, as real-world agents of finite size cannot travel diagonally. Moreover, due to their symmetry-breaking jumping and

pruning rules, all JPS algorithms inherently tend to plan paths that pass obstacles closely, adding further complexity to the process. In case of static grid based path planning algorithms velocity can be optimized without any special constraints, since the environment is static, but in case of dynamic environments the path can not be modified this freely, because the original planning was done with constant velocity in mind. Thus modifying these trajectory points may lead to collision with obstacles. Before fitting a continuous path to the given path points a local optimization has to be done to mitigate the before mentioned problems. The basic idea is to shift the path points away from obstacles where this is possible.

Prior to shaping a continuous path from the given path points, a local optimization step is imperative to alleviate the aforementioned challenges. The fundamental concept involves displacing the path points away from obstacles wherever feasible. To enhance the feasibility of generating trajectories from integer path points, I suggest employing a negative gradient path-shifting algorithm leveraging the surrounding potential field of the weighted grid, presented in Algorithm 6. The core concept involves calculating the discrete potential field of the environment around each discrete path point, considering the grid weights. Subsequently, each path point undergoes a shift, the direction of which is determined by the normalized negative gradient and the shift is limited so that no collision will occur.

The Discrete Path Optimization algorithm computes a weight gradient for each individual point along the path. This is achieved by summing the weights of neighboring grid cells, each multiplied by its respective normalized direction vector. The neighboring grid which is considered during the gradient calculation is a cube with a given radius. During the normalization step we divide by the square of the direction vector's length in order to reduce the impact of neighboring cells farther away from the original point we aim to shift. Additionally, the total weight for each neighbor grid is also computed, as the normalized direction vectors of neighboring grid cells do not sum up to one at the grid edges, so normalizing with sum of the weight gradient vector would result in an incorrect calculation. After gaining the point's weight gradient the shifted path point is calculated by subtracting the weight gradient.

The right choice of the shift factor parameter guarantees that the drone remains within its original hypercube at all times, if with a given sized drone the shifted path points land outside the point's original hypercube then the shift amount can be clipped to prevent collisions as a simple solution. The term hypercube refers to the added time dimension in case of temporal planning algorithms. In the case of a temporal path planning algorithm, the path points shifted with Algorithm 6 must remain within their original hypercube if force clipping is enabled.

If by default force clipping is disabled we use the calculated gradient as a direction guidance, and shift the original path point in steps until a collision occurs with the given drone size or a hard limit is reached. In this case the final shifted point is determined by the last location where a collision did not occur or a hard limit imposed on the maximum shift amplitude has not been reached. The `detectCollision` function also takes *obs_threshold* as an argument, which will be important later when the TJPS algorithm is expanded to handle non binary maps. The algorithm can be improved with a local velocity optimization by also observing the temporal dimension.

In visual terms, the algorithm mentioned above guides the agent away from obstacles within a limited distance, which is proportional to the neighboring obstacle weights and considers their direction. For instance, if the initial discrete path runs parallel to a wall, the entire path is adjusted to move away from the wall. However, if the agent needs to navigate between walls or through a narrow corridor, the resulting weighted gradient becomes zero. Consequently,

Algorithm 6 Discrete Path Optimization

Input:

<i>grid_seq</i> ,	▷ time expanded grid sequence
<i>path</i> ,	▷ extracted path from planning algorithm
<i>dronesize</i> ,	▷ diameter of the UAV in grid dimensions
<i>shift_factor</i>	▷ shift limiting multiplier in grid dimensions
<i>radius</i> ,	▷ gradient calculation box radius
<i>forceclip</i> ,	▷ force clip shift into original cell
<i>obs_threshold</i>	

```
1: function DISCRETEPATHOPTIMIZATION
2:   path_shifted  $\leftarrow$  []
3:   for point, grid in path, grid_seq do                                ▷ time indexing synced
4:     grid_neighbours = getNeighbourGrid(point, radius)
5:     neighbours_cnt = getCellNum(grid_neighbours)
6:     weight_grad  $\leftarrow$  0
7:     weight_total  $\leftarrow$  0
8:     for neighbour in grid_neighbours do
9:       direction  $\leftarrow$  neighbour - point
10:      if direction  $\neq$  0 then
11:        weight_grad  $\leftarrow$  weight_grad +  $\frac{\text{direction}}{\| \text{direction} \|^2} \cdot \text{grid}[\text{neighbour}]$ 
12:        weight_total  $\leftarrow$  weight_total + grid[neighbour]
13:      weight_grad  $\leftarrow$  weight_grad  $\cdot \frac{\text{weight\_total}}{\text{neighbours\_cnt}} \cdot \text{shift\_factor}$ 
14:      point_shifted = point - weight_grad
15:      if forceclip then
16:        point_shifted = clip(point_shifted) ▷ Clip to grid cell considering agent size
17:      else
18:        n_steps = round( $\| \text{weight\_grad} \| / (\text{dronesize} / 2)$ )
19:        len_steps =  $\| \text{weight\_grad} \| / \text{n\_steps}$                                 ▷ check for 0 divisor
20:        weight_grad_unit_neg =  $-\frac{\text{weight\_grad}}{\| \text{weight\_grad} \|}$ 
21:        for step in range(n_steps) do
22:          point_check = point + (step + 1)  $\cdot$  weight_grad_unit_neg  $\cdot$  len_steps
23:          collision = detectCollision(grid, point_check, dronesize, obs_threshold)
24:          if collision or  $\| \text{point} - \text{point\_check} \| \geq 2$  then                ▷ safety limit
25:            point_shifted = point + step  $\cdot$  weight_grad_unit_neg  $\cdot$  len_steps
26:            break
27:      path_shifted.append(point_shifted)
28:  return path_shifted
```

the original discrete path remains unaltered, enabling the drone to pass through the center of the cell.

3.2 Continuous trajectory fitting

In this section our main goal is to plan a continuous path based on the previously optimized discrete path points. All of our previous steps yielded collision free path points, so it can be stated that linearly connecting these lines, would result in a collision free path, but this simple solution is not really suitable for a dynamic vehicle, it has to be smoothed out. Several possible solutions can be employed to solve the aforementioned problem.

The first idea is to only smooth out the corners where the drone makes a turn in a way that the resulting local curvature does not loll out from the original grid cube the path point is included in. This method can be accomplished by simply fitting Bezier curves inside each cube along the planned path. This algorithm results in collision free path by design, so not other constraints are needed to take into consideration. This method would result in a really choppy continuous path which would be deliberately not optimal for a dynamic trajectory.

Another idea is that collisions avoidance could be taken into consideration with Minimum snap Trajectory generation algorithm discussed in the section 3.3, but then the obstacle avoidance had to be expressed as an optimization problem. This would require preprocessing the discrete map to find separating hyperplanes / construct convex to work with convex nonlinear optimisation. Since the aim of this thesis is mainly focused on grid based path planning, we just use a simple solution which satisfies the requirements, but is not state of the art. A possible implementation of the described method can be found in the FASTER Algorithm [17] [16]

Another possible idea is to fit Basis-Splines or B-Splines to the path points. B-Splines are a good fit to use as the basis of Minimum Snap Trajectory, since they are G^2 continuous meaning that the that the acceleration along the spline is continuous. B-Splines also does not go through the original path points which is useful because a dynamically smoother and hence faster path can be generated, but on the other hand collision avoidance is not guaranteed by design.

For the B-spline fitting implementation I used the scipy package's `splprep` function, which utilizes the FITPACK package written in the FORTRAN programming language. [3] The FITPACK package typically uses a least squares fitting algorithm to compute the B-spline representation of the input data. The following listing gives a brief overview on the steps involved in B-Spline fitting.

- **Data Input:** Input a set of 2D or 3D data points, which serve as the basis for the curve fitting process.
- **Degree Determination:** Select an appropriate degree for the B-spline. This degree determines the order of the polynomial segments constituting the spline and significantly influences its flexibility and smoothness. In case of `splprep` the default degree is 3.
- **Knot Vector Initialization:** Generate a knot vector that defines the parameter values where each polynomial segment of the spline begins and ends. The knot vector must adhere to certain conditions based on the chosen spline degree.

- **Control Point Setup:** Initialize the control points, cardinal elements that specify the shape of the resulting B-spline curve. Typically, the number of control points equals the number of input data points.
- **Optimization Procedure:** Employ an optimization algorithm, such as least squares fitting, to iteratively adjust the control points in order to minimize the deviation between the fitted spline and the input data points. This optimization process involves solving a system of equations to determine the optimal control point positions.
- **Output:** Obtain the fitted B-spline curve, represented by the adjusted control points and the knot vector.
- **Evaluation:** Evaluate the fitted B-spline at additional points to obtain a smooth curve representation.

In summary, the B-spline fitting procedure encompasses initializing the knot vector and control points, determining an appropriate spline degree, optimizing control point positions to approximate the input data, and retrieving the resulting fitted B-spline curve for analysis and visualization.

When it comes to the smoothing parameter, it affects the balance between the fidelity of the spline to the input data points and the smoothness of the resulting curve. The following paragraph explains how the smoothing parameter is typically taken into consideration within the FITPACK algorithms.

The FITPACK algorithms aim to minimize an objective function that represents the discrepancy between the spline and the input data points. This objective function typically consists of two terms: a data fidelity term and a smoothing term. The Data Fidelity term penalizes the deviation of the spline from the input data points. A smaller value of s puts more emphasis on minimizing this term, resulting in a spline that closely interpolates the input data points. The Smoothing term penalizes the curvature of the spline. A larger value of s puts more emphasis on minimizing this term, resulting in a smoother spline that may not necessarily pass through the input data points. The smoothing parameter s serves as a balance between these two competing objectives. By adjusting the value of s , the trade-off between fidelity to the input data and smoothness of the resulting spline can be controlled. The FITPACK algorithms typically use optimization techniques, such as least squares minimization, to find the values of the spline parameters that minimize the objective function. The value of s is used to weigh the contributions of the data fidelity term and the smoothing term in this optimization process.

I propose to calculate the best possible s value which does not result in a collision by using the bisection method described in Algorithm 7 on a function which calculates the number of collisions along a path generated or fitted with an input s parameter.

The collision detection along a path works in linear time, and with reasonable initial parameter choosing the bisection method also converges really fast, 5 iterations are usually enough to find the best possible smoothness value with unit tolerance. After the largest smoothness parameter is gained while still maintaining a collision free path, the continuous path is interpolated with that s parameter, and these continuous path points are given as input to the Minimum Snap Trajectory algorithm detailed in section 3.3.

Algorithm 7 Bisection Method

```
1: function BISECTIONMETHOD(func, a, b, tolerance, max_iter)
2:   if func(a) · func(b) ≥ 0 then
3:     return None                                ▶ The function does not change sign over the interval
4:   iter_count ← 0
5:   while (b − a)/2 > tolerance and iter_count < max_iter do
6:     c ← (a + b)/2
7:     if func(c) = 0 then
8:       return c
9:     else if func(c) · func(a) < 0 then
10:      b ← c
11:    else
12:      a ← c
13:    iter_count ← iter_count + 1
14:  return (a + b)/2
```

3.3 Minimum Snap Trajectory

The Minimum Snap Trajectory [11] method is a technique used in robotics, particularly in the field of motion planning and control for autonomous vehicles such as drones. It focuses on generating smooth and dynamically feasible trajectories for such vehicles to follow, taking into account various constraints and objectives.

In essence, the method seeks to minimize the snap, or the rate of change of jerk, along the trajectory. Jerk is the rate of change of acceleration, and snap is the rate of change of jerk. Minimizing snap results in trajectories that are not only smooth but also exhibit smooth changes in acceleration, ensuring comfortable and efficient motion for the vehicle. The Minimum Snap Trajectory method typically formulates the trajectory generation as an optimization problem, where the objective is to find the trajectory that minimizes a cost function while satisfying constraints such as collision avoidance, dynamic feasibility, and possibly others like time and energy constraints.

This method is widely used in applications where smooth and precise motion is desired, such as in aerial robotics, autonomous vehicles, and robotics in general. Its effectiveness lies in its ability to produce trajectories that are both dynamically feasible and comfortable for the vehicle to execute. The python implementation I used for this project was implemented by Péter Antal at SZTAKI and it uses the Casadi Optimizer.

3.4 Summary and results

The purpose of this section is to provide a brief overview of the challenges involved in trajectory generation for grid based path planning algorithms and the steps of the proposed solution.

Grid-based path planning algorithms generate discrete paths not directly suitable for motion planning. Static environments and algorithms allow retrospective velocity optimization without special constraints, but temporal algorithms tailored for dynamic environments also including time indices require constant velocity planning to avoid collisions. Planning with only orthogonal moves results in choppy paths, while diagonal moves can cause the problem of corner skipping. Furthermore Jump Point Search algorithms tend to plan paths close to

obstacles by design adding complexity to the trajectory fitting process. Figure 3.1 visualizes the steps of the trajectory generation process.

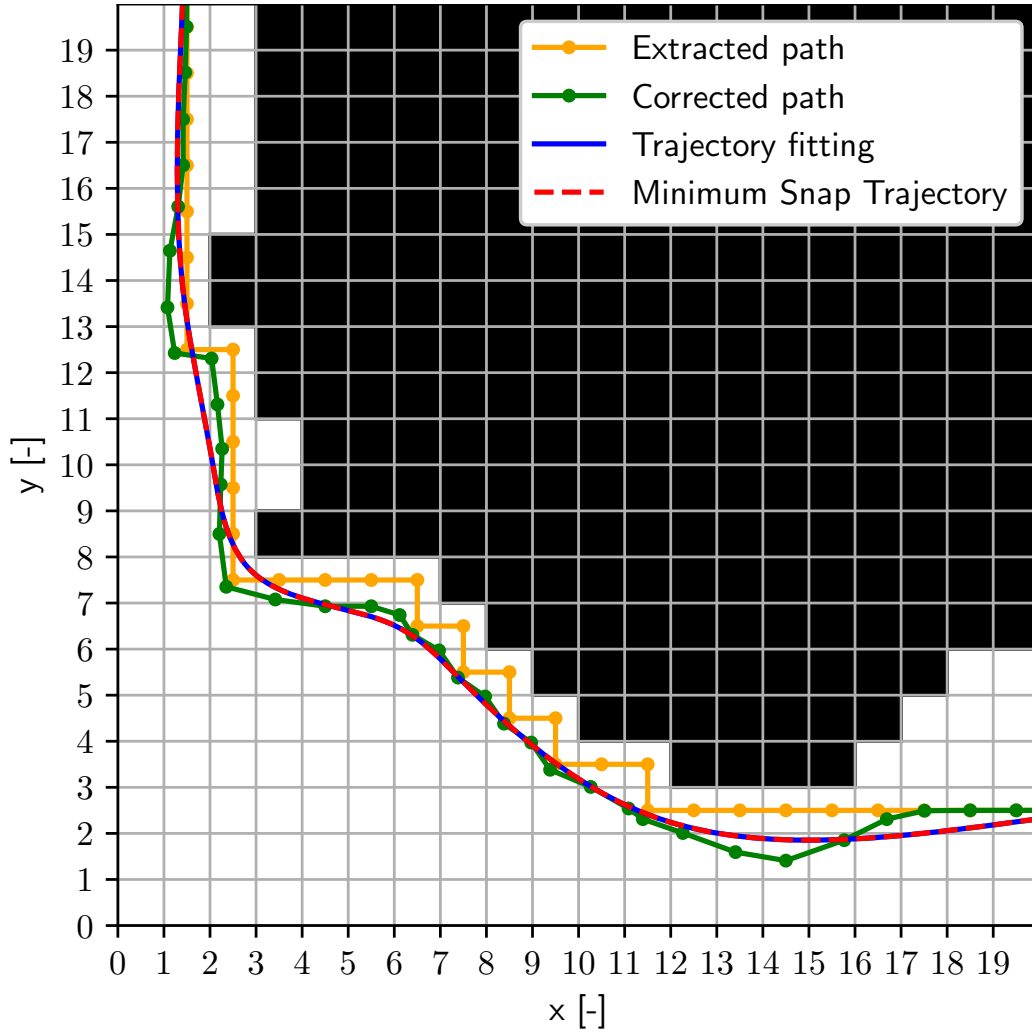


Figure 3.1: Trajectory generation process

1. Discrete Path Optimization

Before creating a continuous path, local optimization is applied to shift path points away from obstacles in order to be able to fit a smoother trajectory. This involves using a negative gradient path-shifting algorithm based on the potential field of the occupancy grid. The algorithm calculates a weight gradient for every point along the path, adjusting its position according to this gradient while also ensuring collision avoidance. This method facilitates navigation around obstacles, while also ensuring that in narrow corridors and obstacle-free areas the path remains unchanged.

2. Continuous Trajectory Fitting

A smooth and continuous path is generated from the adjusted discrete path points using B-spline fitting. The smoothing parameter is optimized with the bisection method to achieve maximum smoothness while maintaining collision avoidance.

3. Minimum Snap Trajectory

This method focuses on generating a smooth, dynamically feasible trajectory based on the previously fitted continuous trajectory.

Chapter 4

Simulation results

This chapter provides a benchmarking analysis of the TJPS algorithm, comparing its performance with that of the Temporal A* algorithm across various scenarios and problem sets. Additionally, the multi-step planning procedure and trajectory generation algorithm is also evaluated. The analysis begins with two-dimensional scenarios, testing empty maps with various resolutions populated only with dynamic obstacles, as well as on publicly available benchmark maps that also include dynamic obstacles. The analysis then extends to three-dimensional problems, exploring realistic scenarios that feature static terrain, radars, and dynamic obstacles.

Obstacles A random obstacle is represented as a rasterized ellipsoid, with its diameter selected at random from a specified range. Both the velocity and direction of these obstacles are randomized as well. By default, these random obstacles follow a linear path, which may change randomly at each step based on a predetermined probability. Upon reaching the boundary of the map, the obstacle bounces back, maintaining a constant number of dynamic obstacles in the environment throughout the planning process. The environment may also incorporate preplanned obstacles, which are typically ellipsoidal with a defined diameter and follow trajectories determined by a temporal planning algorithm. This setup allows for multi-agent path planning. Preplanned obstacles halt upon reaching their destination, which may affect the count of actively moving dynamic obstacles. In the two-dimensional benchmarks, only fully randomized obstacles were used. The environment can handle obstacles of different shapes, such as rectangles or custom shapes; however, testing indicated that ellipsoid-shaped obstacles impose the greatest computational demand on the algorithm. This is due to their jagged boundaries when rasterized, particularly with random diameters, which complicates execution. The three-dimensional environment also presents radars with a layered structure. The occupancy value in case of radars represent the possibility of the aircraft being detected. The radar shape is derived from rotating a droplet-like shape around the Z axis. [9]

Technical details The current implementations are developed in Python, with all benchmarks conducted on an Intel Core i7 3.2 GHz processor with 32GB of RAM. The original TJPS study [7] includes a two-dimensional implementation of the TJPS algorithm in C, which, operates roughly an order of magnitude faster than the Python implementation evaluated here. This performance difference arises because the Python version represents the grid environment as a numpy array of floats in order to work with weighted grids, while the C implementation uses a binary representation of the environment stored in a contiguous memory segment. Consequently, the jump mechanism and the pruning rules can be

implemented through byte masking in C, significantly accelerating the process compared to Python’s if-else conditions. Additionally, the C implementation incorporates precomputed heuristics, as described in the paper. When benchmarking the original C implementation the authors used one cell sized dynamic obstacles, in contrast we use extensive obstacles with random diameters. The two-dimensional tests included here serve both to compare the algorithms themselves and to evaluate the execution speed differences between the C and Python implementations. Results indicate that, under identical map and problem instances, the C implementation achieves at least an order of magnitude faster execution speed. In case of the three-dimensional implementation I only had the chance to compare the Python implementations of the algorithms. To provide a fair comparison between TJPS and Temporal A* the cost of all obstacles were set to infinite during the benchmarks. The total path length displayed in the figures corresponds to the number of steps taken from the start to the goal.

4.1 Two-dimensional benchmarks

The initial benchmarks evaluate the algorithms on an empty static map with an increasing number of random dynamic obstacles, with both start and goal positions randomly selected. The benchmarks illustrated in Figure 4.1 was conducted on an empty map of size 50x50. Figure 4.1a displays the execution time as a function of the number of dynamic obstacles in the environment, while Figure 4.1b shows the execution time plotted against the total planned path length. The path length is plotted, as the Euclidean distance from the start to the goal may be misleading due to the presence of obstacles. Note that extreme outliers (values outside the range $[Q1 - 3 \cdot IQR, Q3 + 3 \cdot IQR]$) were filtered out, and are not shown on the plot. The results indicate that in relatively small-scale planning problems, the Temporal A* algorithm outperforms the TJPS, as the computational overhead of the jumping and pruning rules in the Python implementation of TJPS outweighs their advantages.

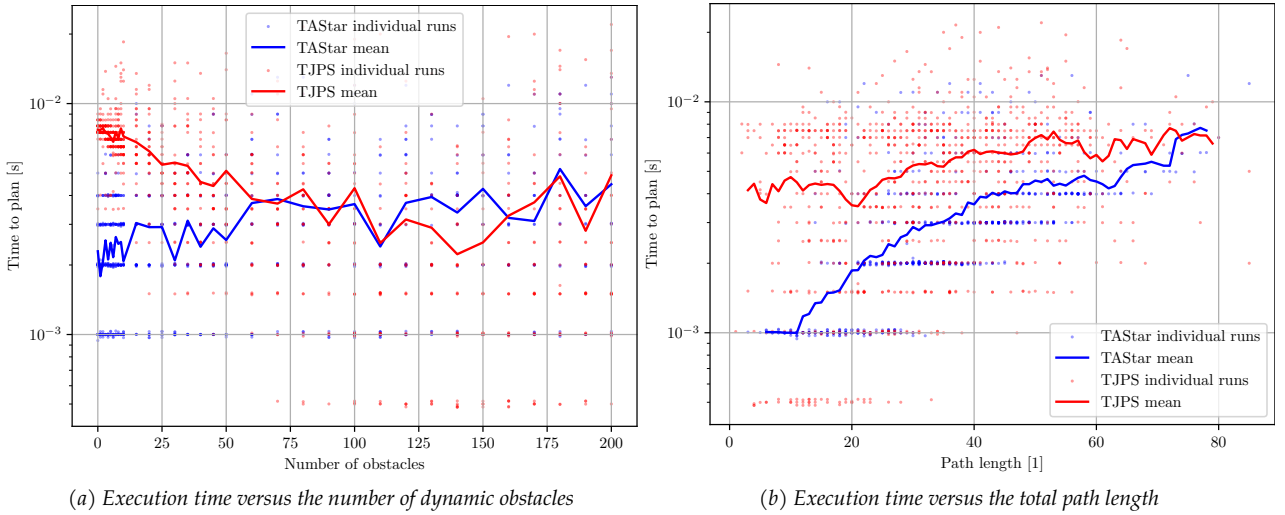


Figure 4.1: Empty map (50x50) with random dynamic obstacles, diameter range: (1,3)

Figure 4.2 presents the benchmark results for an empty static map with a side length of 200 units. Similar trends are observed as in the smaller grid case, however both algorithms exhibit an order of magnitude increase in execution time with the doubled grid size. Notably, the execution time of the TjStar algorithm—unlike that of TjPS—is strongly dependent on the number of dynamic obstacles, showing a positive correlation. In an empty map, TjPS performs suboptimally due to its reliance on generating temporal jump points in order to progress toward the goal. With relatively few static or dynamic obstacles, TjPS often scans the entire

map without producing jump points, resulting in a negative correlation with obstacle count. However, as the generation of more temporal jump points become feasible, the benefits of jumping and pruning rules start to offset their computational cost, though TJPS remains slower than TASTar in an empty grid setting. In contrast, according to the original C implementation, TJPS achieves faster execution in this scenario. It is worth noting that with the dynamic obstacle sizes specified in the figure descriptions, grid occupancy exceeds 70%, testing the algorithms under conditions more challenging than typical real-world scenarios.

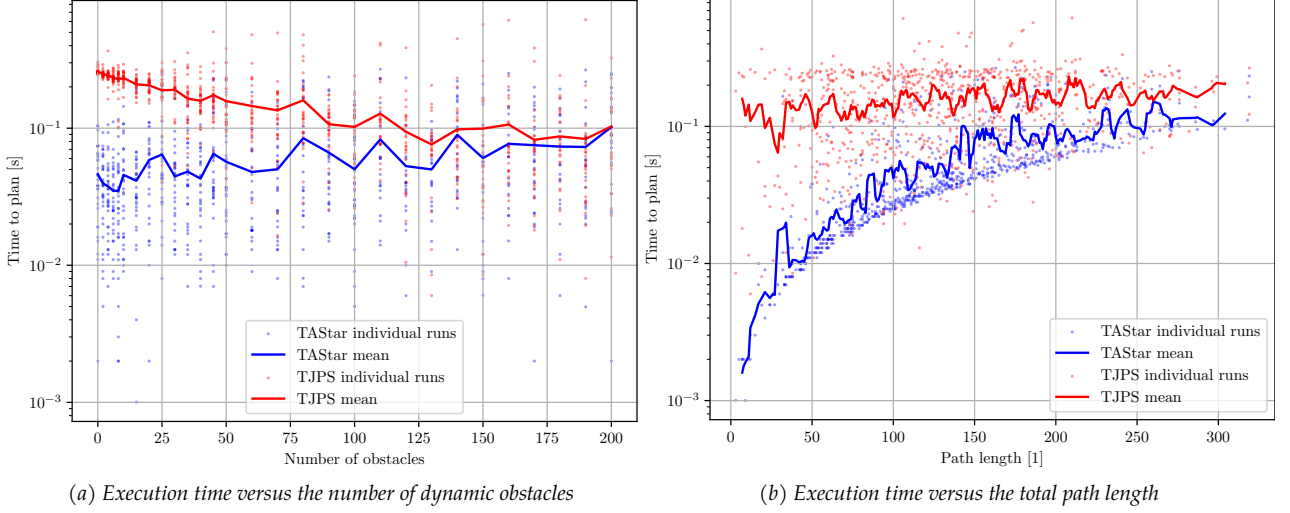


Figure 4.2: Empty map (200x200) with random dynamic obstacles, diameter range: (1,10)

The subsequent benchmarks were conducted using static maps featuring dynamic obstacles. The static maps are sourced from Sturtevant’s well-known and publicly available benchmark map set [15], while the path planning instances remain randomly generated. Unlike in the original TJPS article, these benchmarks include extensive dynamic obstacles, capable of fully blocking sub-paths for multiple timesteps.

The Berlin map, representing a city layout with both narrow streets and spacious intersections, provides an environment where the TJPS algorithm can efficiently identify temporal jump points. The benchmarks are shown in Figure 4.3. In the absence of obstacles, the Temporal A* algorithm remains faster than the Python implementation of TJPS as is can be seen in Figure 4.3b. However, as more and more dynamic obstacles are introduced, the performance hierarchy shifts. In a dense environment TJPS becomes an order of magnitude faster in terms of execution speed.

The 1ak303d map, shown in Figure 4.4, originates from a real game and presents a significant challenge due to narrow gateways that can be fully obstructed by extensive dynamic obstacles. From an algorithmic perspective, this map poses additional difficulties; if the start or goal points are positioned in the central open area, or at opposite ends of the map, the applied heuristics can mislead the agent, making this a complex problem worthy of detailed analysis. The benchmarking results for this scenario are presented in Figure 4.5.

The TJPS algorithm allows for a limitation on waiting times at each temporal jump point. Figure 4.4b illustrates a planning instance where waiting is disabled, here the agent moving with constant speed must make some detours through its journey since the narrow corridors leading to the destination are not always traversable. Incorporating waiting into the planning yields a considerably shorter, yet still collision-free path as shown in Figure 4.4a. The environment presented in Figure 4.4 contains 100 random dynamic obstacles, with static and dynamic obstacles represented in blue and green respectively. The agent is depicted as a red circle, while small dots along the path indicate one or more WAIT action taken by the agent.

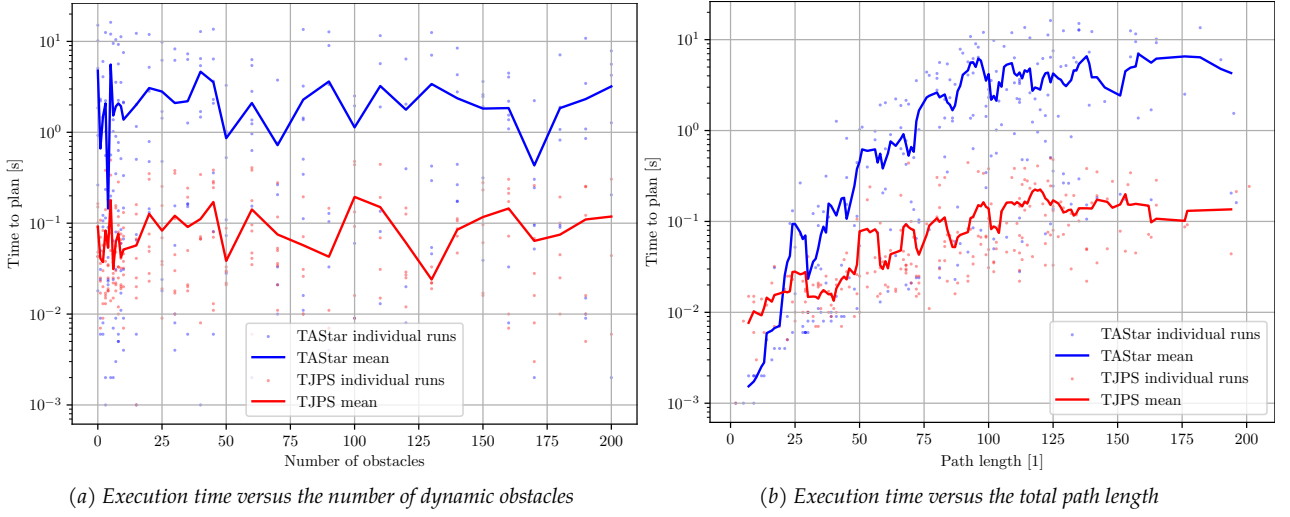


Figure 4.3: Berlin map (256x256) with random dynamic obstacles, diameter range: (1,10)

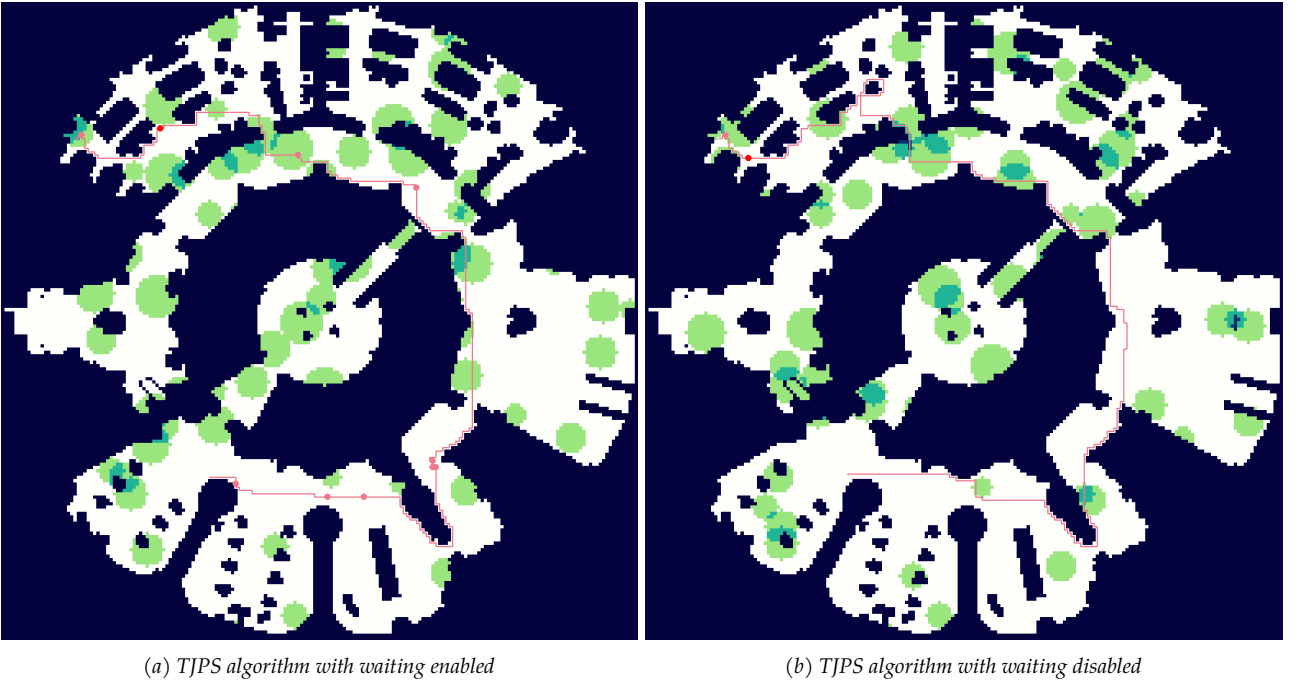


Figure 4.4: lak303d map (200x200) with 100 dynamic obstacles, diameter range: (6,14)

In the subsequent benchmark, we evaluate the multi-step planning method previously discussed in section 2.4. Given that grid-based planning algorithms do not scale effectively at high resolutions, the initial step involves planning on a grid with reduced resolution, followed by the construction of a tube or corridor to limit the search space of the original grid, thereby accelerating the search process. In this instance the resolution is exactly halved, which is considered good practice in order to mitigate programming pitfalls, although this is not a strictly defined requirement. Both the spatial and temporal dimensions are similarly reduced to preserve the original constant planning speed.

For this test the Paris static map was used and the results are shown on Figure 4.6. The original dimensions of the map are 256x256, while the reduced grid is has dimensions of 128x128. This map represents a city layout similar to the previously utilized Berlin map. During the generation of the bounded grid sequence a corridor with a diameter of 20 (measured in the original grid resolution) was used. The TJPS algorithm was used for planning. In the legend planning on the full resolution grid is marked with Original, Reduced corresponds to the planning on the grid with halved resolution, Bounded corresponds to planning on the grid

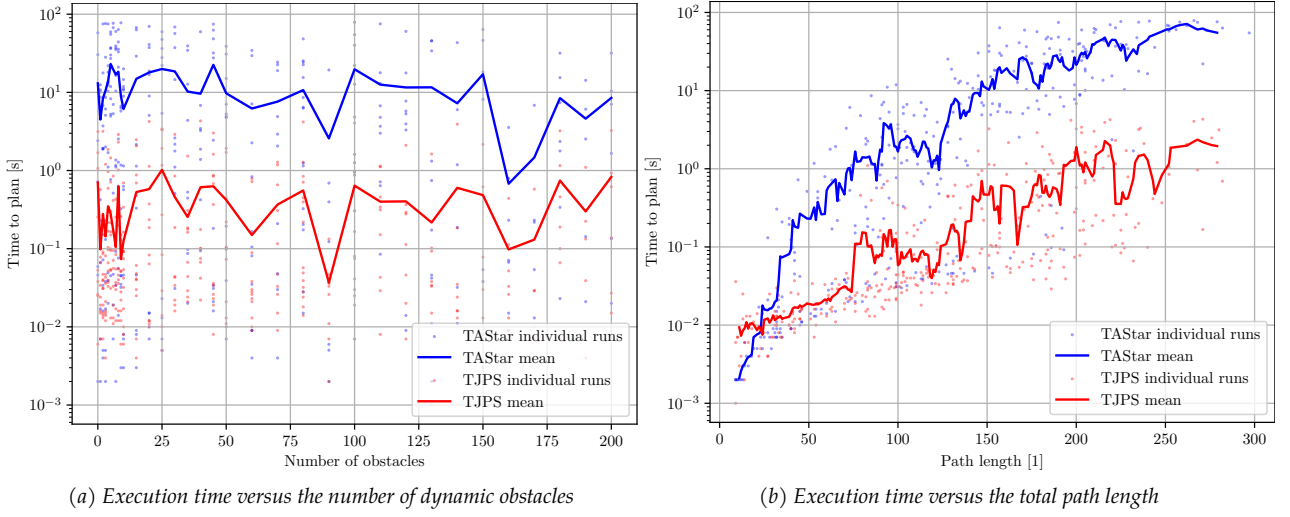


Figure 4.5: *lsk303d* static map (200x200) with ellipsoid shaped dynamic obstacles with random diameter in range (1,10)

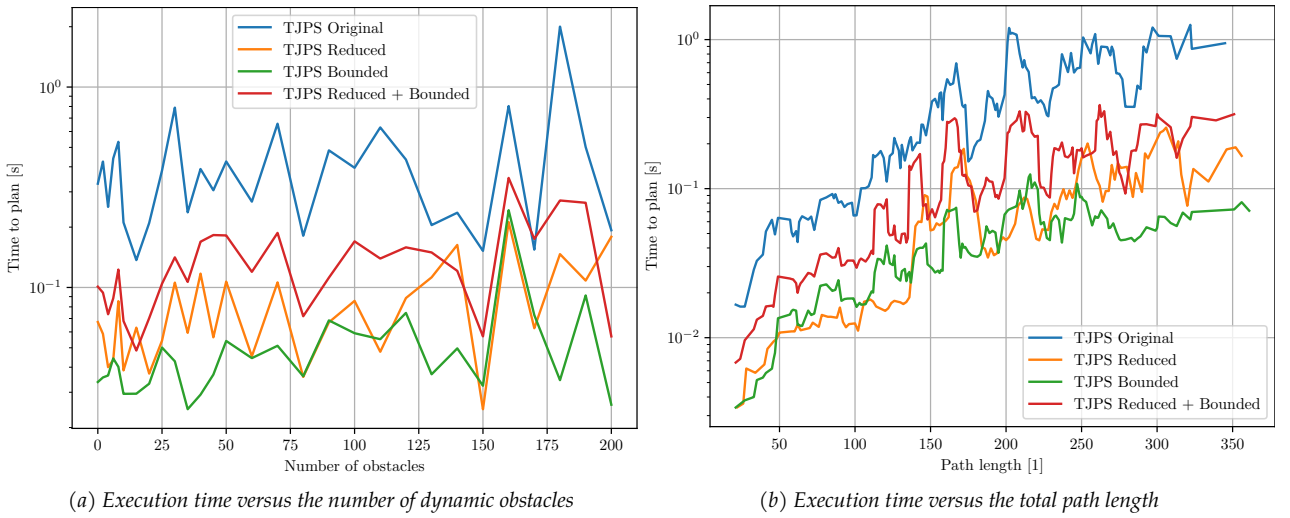


Figure 4.6: Multi-step planning on the *Paris* map (256x256) with dynamic obstacles, diameter range: (1,20)

bounded by a corridor with original resolution. The total planning time of the multi step method is also shown and can be compared to the Original case. It is important to note that the generation of the bounded grid sequence also requires time, which is not represented in the graph, but at the magnitude of the investigated dimensions it is negligible.

Another test was conducted to illustrate the capabilities of the multi-step planning approach. The Warehouse static map was employed, which, as its name suggests, depicts a warehouse environment characterized by repetitive, regularly arranged rectangular obstacles. In this context, the TJPS algorithm demonstrates significant advantages due to the implemented jumping rules. The results are presented in Figure 4.7.

As evidenced by both test instances, planning on the bounded grid sequence is the least computationally demanding part of the combined multi-step planning, indicating that this approach is advantageous for sequential multi-agent pathfinding when the agents share the same origin and destination areas.

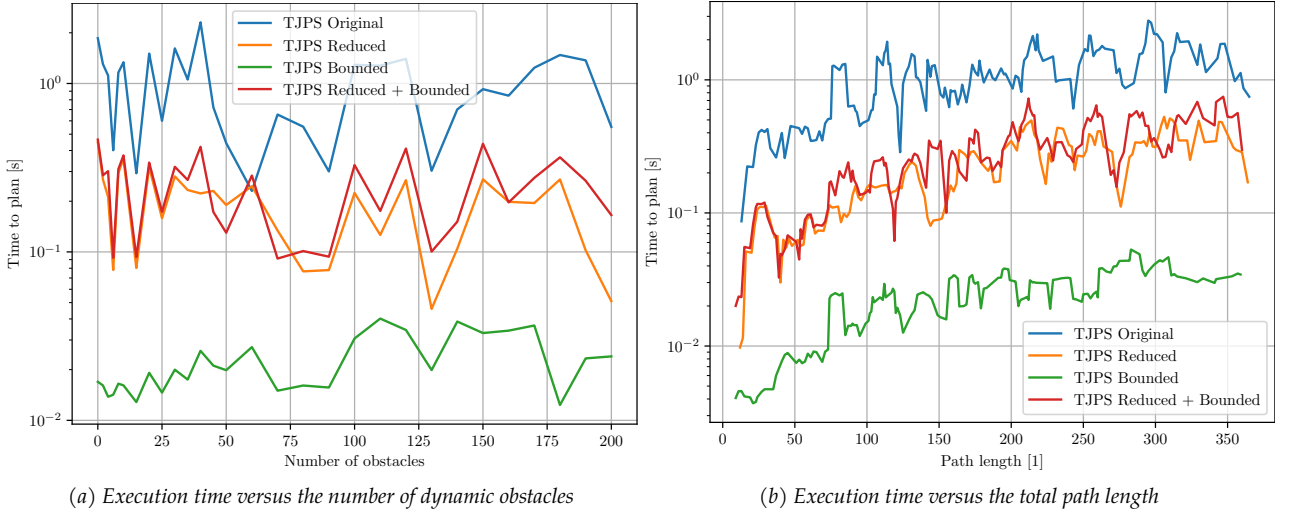


Figure 4.7: Multi-step planning on the Warehouse map (340x164) with dynamic obstacles, diameter range: (1,20)

4.2 Three-dimensional benchmarks

This section focuses on evaluating the three-dimensional TJPS algorithm introduced in the thesis. Empty scenarios are excluded from testing; instead, evaluation is conducted within environments that mirrors the original problem setup for which the planning algorithm was developed. This constructed scenario includes a static terrain map generated with the help of simplex noise, representing a valley with dimensions (64,128,16), along with designated radar zones that which we aim to steer clear of. [18][12] Additionally, an increasing number of randomly generated dynamic obstacles are introduced to the environment during testing. Figure 4.8 provides an illustration of the planning scenario. The RGB colors of the floating coordinate system above the map align with the XYZ axes of the environment. The color gradient in the radar indicates the layered structure described earlier, with each layer corresponding to a constant level of detectability risk. Note that the drones representing the agents are scaled up by a factor of 10 in all figures compared to their real dimensions.

In the first benchmark the problem instances (start and goal location pairs) are randomly selected from the set of free cells. The diameter of sequentially added dynamic obstacles is in the range of 1 to 10, resulting in a really dense and changing environment in order to present a greater challenge to the algorithms. Results are shown in Figure 4.9. In this benchmark, the Temporal A* algorithm demonstrated superior performance only in case of problem instances where the start and goal coordinates were relatively close. However, for all other problem instances, the TJPS implementation exhibited better performance, especially for longer paths.

In the second test, the three-dimensional performance of the multi-step planning method is tested. The scenario and radar placements remain unchanged; however, the diameters of dynamic obstacles are reduced to a range of 1 to 5. The problem instances are chosen to facilitate movement along the Y-axis, requiring the agent to navigate through the valley bordered by radar fields. These start and goal locations are randomly selected from a predefined ellipsoid located at opposite ends of the map, with its longest axis oriented perpendicularly to the valley and spanning the whole map in that direction, as illustrated in Figure 4.8. In this scenario, we plan for a sequence of five drones. Once the path for each drone is determined, they are introduced into the environment as dynamic obstacles. After reaching the target number of drones, additional random dynamic obstacles are incorporated. The results are presented in Figure 4.10. It can be seen that the original TJPS operates an

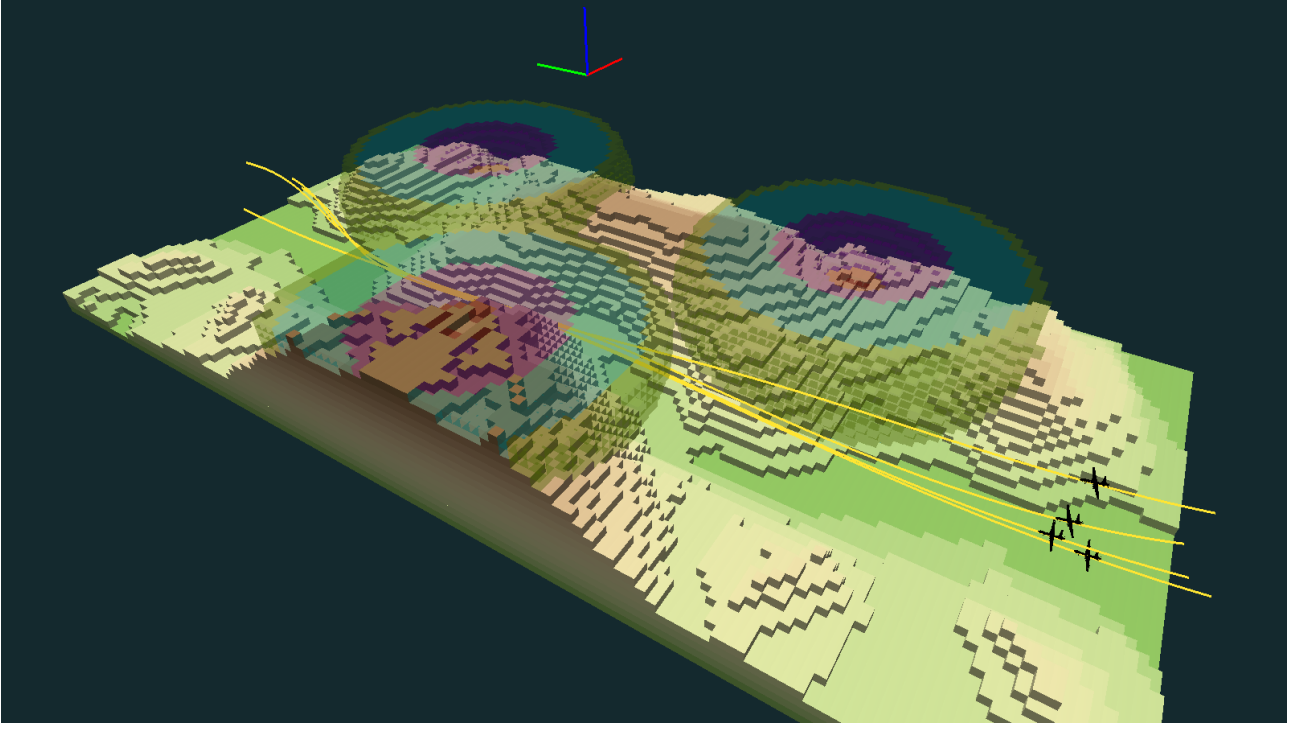


Figure 4.8: *RadarMap* scenario with 4 drones crossing the valley between radar fields

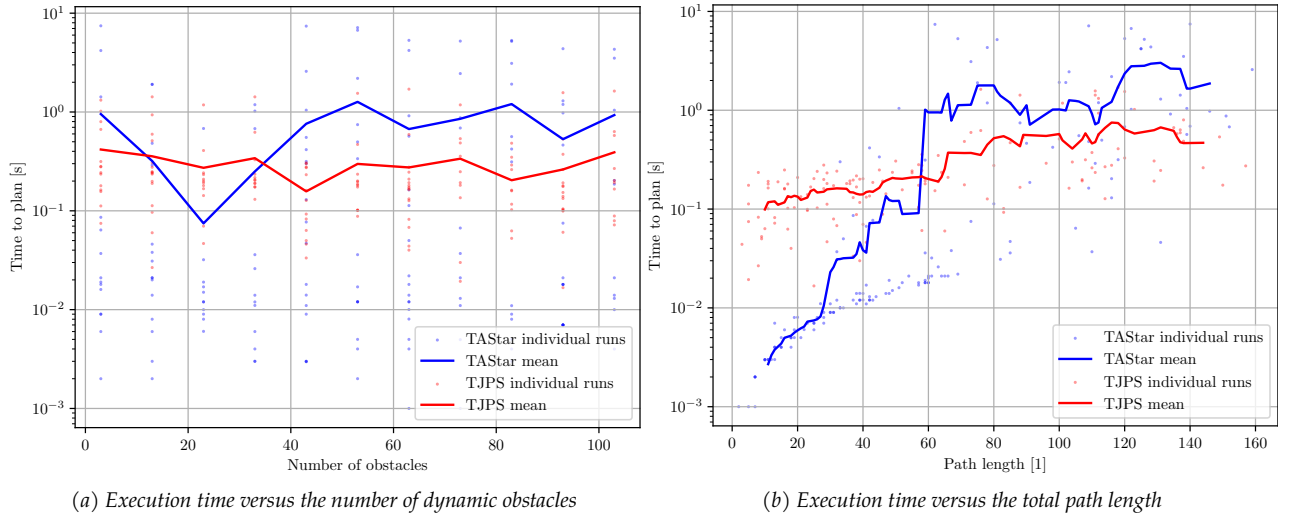


Figure 4.9: *RadarMap* environment with dynamic obstacles (1,10) and random problem instances

order of magnitude faster than the Temporal A* algorithm, and the incorporation of multi-step planning can enhance its speed by an additional order of magnitude.

4.3 Trajectory generation

Figure 4.11 illustrates the sequential stages of the trajectory generation algorithm. The red trajectory represents the initial path planned by TJPS. The blue trajectory reflects the output of the discrete path optimization process, which subtly shifts the path along the negative gradient direction while maintaining collision avoidance. The yellow trajectory shows the continuous B-spline fitted to the path. In this instance, the continuous trajectory fitting algorithm is configured to allow the trajectory to intersect the radar field's outer layer. Grid-based planning algorithms, such as TJPS, are known to generate paths optimally close to

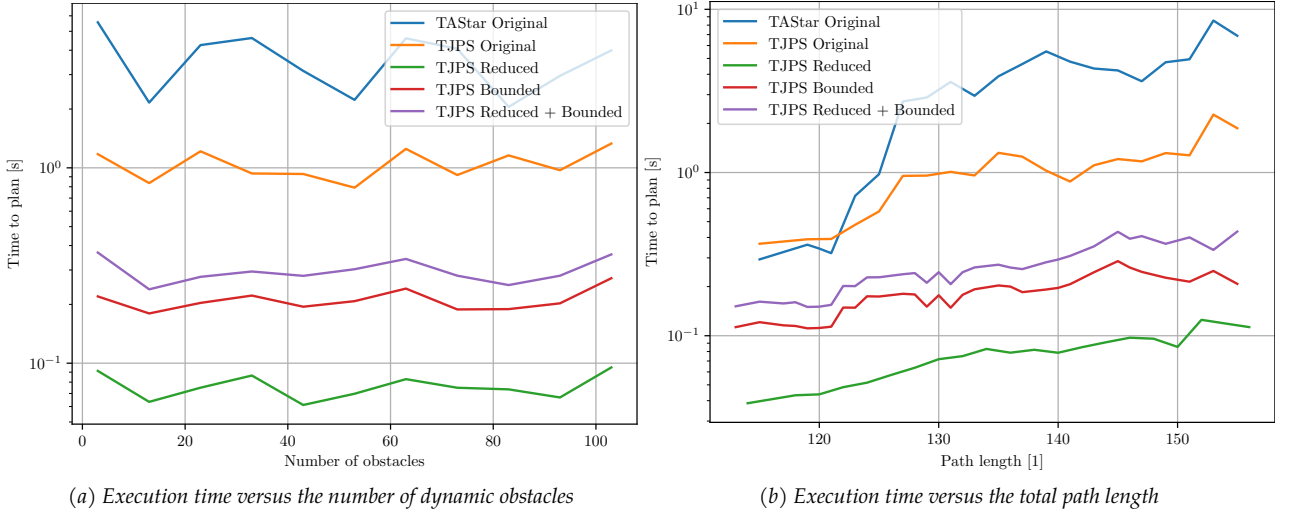


Figure 4.10: Multi-step planning in the *RadarMap* environment for 5 drones, with dynamic obstacles (1,5) and problem instances generated for the agent to cross the valley

obstacles, presenting a significant challenge in producing a collision-free, continuous trajectory from the extracted path. However, through the combination of the negative gradient-based discrete path optimization step and collision-free spline fitting, a smooth, executable trajectory can be achieved.

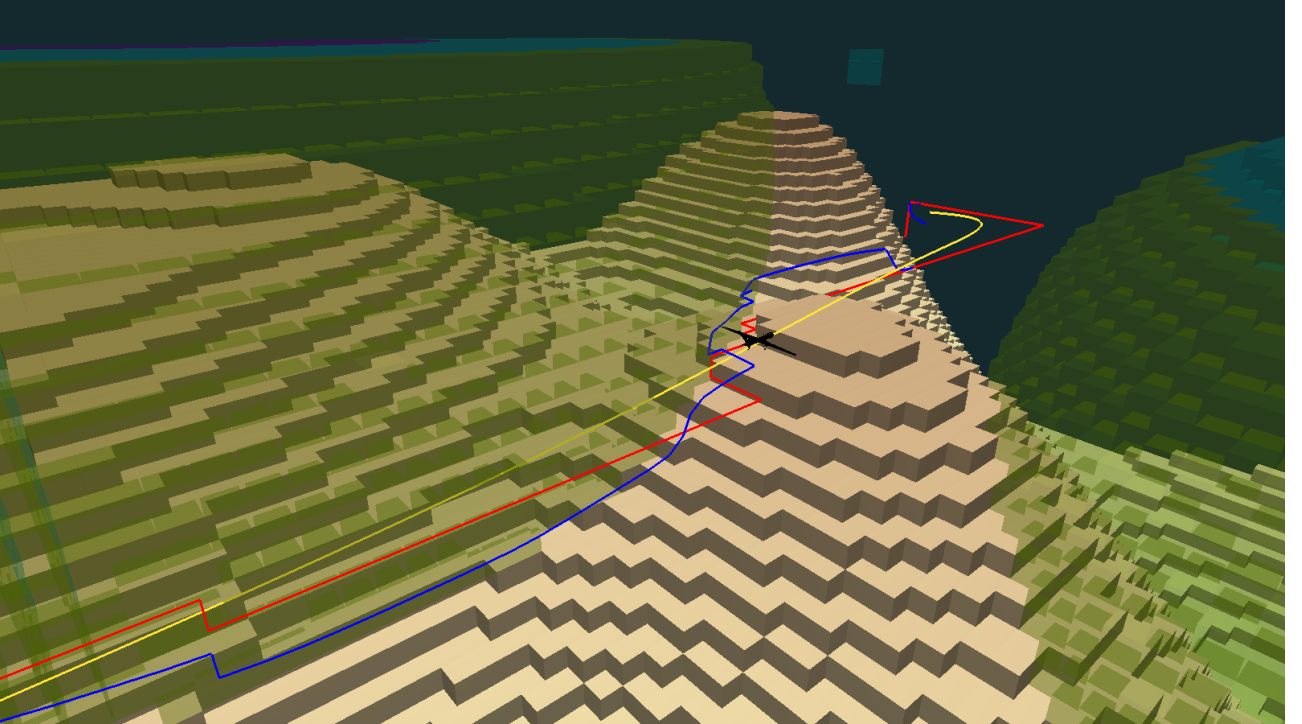


Figure 4.11: Visualization of trajectory generation

In the final scenario, all methods introduced in this thesis are applied. A higher resolution map (220x80x40) is used, generated with two-dimensional simplex noise. Multi-layered radars are positioned on the five tallest hills within the environment. The radar diameters are strategically chosen to prevent the agent from traversing the environment in a straight line. The result of the multi-step planning procedure is illustrated in Figure 4.12.

The multi-step planning procedure is employed to plan paths for three quadcopters. Initially, planning is conducted on a lower-resolution map, halving the resolution, followed by constructing a corridor with a diameter of 20 units. Subsequently, the planning is refined on

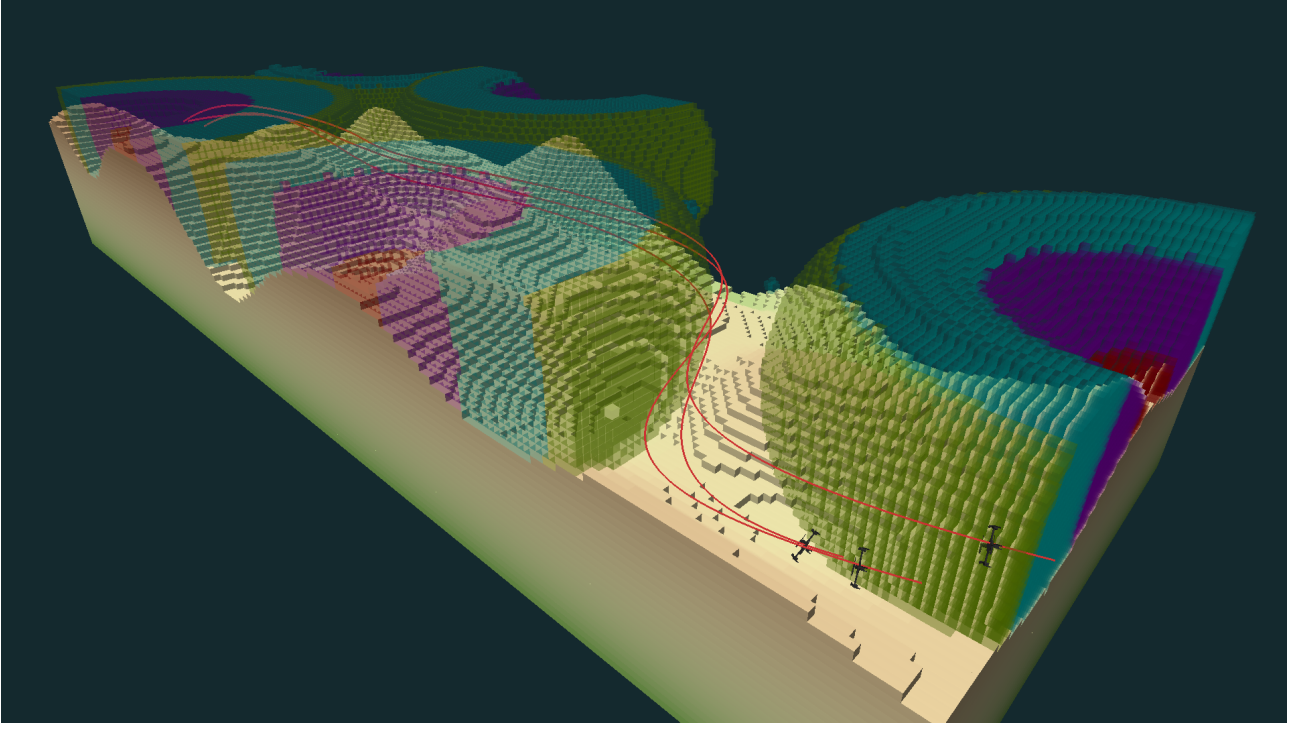


Figure 4.12: Path planning for multiple drones

the high-resolution map for all drones, with previous agent trajectories incorporated as dynamic obstacles. Once planning with the TJPS algorithm is completed, all trajectory generation steps are executed. The discrete path optimization step shifts the paths to avoid radar zones, allowing the continuous trajectory fitting algorithm to generate a smoother, collision-free path. In the final phase, the Minimum Snap Trajectory algorithm provides a dynamically feasible trajectory for the quadcopters, along with the necessary control inputs.

Reinforced by the testing results, we now summarize the advantages and disadvantages of the TJPS algorithm in comparison to traditional graph-based search approaches:

- The TJPS algorithm achieves optimality up to the grid resolution, similar to the A* algorithm, as it is based on a predefined cost measure. In contrast, traditional graph-based solutions typically yield approximate solutions.
- The TJPS algorithm exhibits robust scalability with an increasing number of dynamic obstacles, consistently identifying a valid path when one exists. This adaptability is critical for applications requiring real-time navigation through dynamic environments.
- Although the TJPS algorithm is approximately an order of magnitude faster than Temporal A* when comparing the C implementation, it does not scale as effectively on high-resolution grid maps as graph-based search algorithms. High-resolution requirements can lead to performance bottlenecks due to the increased computational complexity of managing fine-grained grids.
- Unlike graph-based search algorithms, grid-based algorithms do not produce intermediate solutions, which limits their adaptability in real-time applications. Consequently, the grid resolution must be carefully managed, or additional optimization techniques (such as planning on multiple grid resolution with corridors) must be applied to enable feasible online trajectory planning.

Chapter 5

Summary

Future development possibilities

Future development of the proposed navigation system could focus on several key advancements. Currently, the planning algorithm requires knowledge of all dynamic obstacle trajectories in advance to accurately plan a path to the destination. A possible improvement would involve training a neural network to predict the dynamic occupancy grid used in path planning. This would allow the system to predict the movement of obstacles, even without complete prior information, thereby enhancing adaptability in environments where obstacles' movements are unforeseeable.

Another promising direction is to employ reinforcement learning (RL) to explore whether a neural network could learn the task of path planning in dynamic environments more efficiently than traditional methods. An RL-based model might not only execute faster but could also address certain limitations of the current approach, such as the binary occupancy grid limitation. By allowing for weighted maps, RL could enable the planning algorithm to work with probabilistic maps, representing varying levels of obstacle likelihood rather than treating each cell as strictly occupied or free. Additionally, RL could incorporate uncertainties within dynamic obstacle trajectories, enhancing the system's robustness in real-world settings where obstacles do not follow perfectly predictable paths.

Finally, optimizing the discrete path generation and continuous trajectory fitting steps may further improve efficiency. Currently, these processes are designed to find a globally optimal solution for the entire path, which can be computationally intensive. Integrating these steps could allow for a more localized optimization approach, adjusting parameters in response to specific segments of the environment as needed, potentially reducing computation time and enabling real-time adaptability.

Acknowledgements

I would like to thank Tamás Péni and Roland Tóth at SZTAKI for their continuous support during my work. I would also like to thank Péter Stumpf at BME Department of Automation and Applied Informatics for his help and supervision.

Budapest, November 3, 2024

Ádám Kovács

Bibliography

- [1] Mark Carlson, Sajjad K. Moghadam, Daniel D. Harabor, Peter J. Stuckey, and Morteza Ebrahimi. Optimal pathfinding on weighted grid maps. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(10):12373–12380, Jun. 2023. DOI: 10.1609/aaai.v37i10.26458. URL <https://ojs.aaai.org/index.php/AAAI/article/view/26458>.
- [2] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- [3] Pearu Peterson Eric Jones, Travis Oliphant et al. Scipy: Open source scientific tools for python. <https://docs.scipy.org/doc/scipy/> (2024.05.02), 2001.
- [4] Daniel Harabor and Alban Grastien. Online Graph Pruning for Pathfinding On Grid Maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, volume 2, pages 1114–1119. Association for the Advancement of Artificial Intelligence (AAAI), 2011. ISBN 9781577355090. URL <http://www.aaai.org/Conferences/AAAI/aaai11.php>. AAAI Conference on Artificial Intelligence 2011, AAAI 2011 ; Conference date: 07-08-2011 Through 11-08-2011.
- [5] Daniel Harabor and Alban Grastien. Improving Jump Point Search. *Proceedings of the International Conference on Automated Planning and Scheduling*, 24(1):128–135, May 2014. DOI: 10.1609/icaps.v24i1.13633. URL <https://ojs.aaai.org/index.php/ICAPS/article/view/13633>.
- [6] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. DOI: 10.1109/tssc.1968.300136. URL <https://doi.org/10.1109/tssc.1968.300136>.
- [7] Shuli Hu, Daniel D. Harabor, Graeme Gange, Peter J. Stuckey, and Nathan R. Sturtevant. Jump Point Search with Temporal Obstacles. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):184–191, May 2021. DOI: 10.1609/icaps.v31i1.15961. URL <https://ojs.aaai.org/index.php/ICAPS/article/view/15961>.
- [8] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning, 2011.
- [9] Máté Koroknai. Radar field. <https://github.com/AIMotionLab-SZTAKI/AIMotionLab-Virtual/wiki/Radar-field> (2024.10.31), 2024.
- [10] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998. URL <https://api.semanticscholar.org/CorpusID:14744621>.
- [11] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011. DOI: 10.1109/ICRA.2011.5980409.
- [12] Amit Patel. Making maps with noise functions. <https://www.redblobgames.com/maps/terrain-from-noise/> (2024.05.01), 2022.
- [13] Mike Phillips and Maxim Likhachev. SIPP: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635, 2011. DOI: 10.1109/ICRA.2011.5980306.
- [14] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, pages 151–158, 2019.

- [15] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012. URL <https://www.movingai.com/benchmarks/index.html>.
- [16] Jesus Tordesillas and Jonathan P How. MADER: Trajectory Planner in Multi-Agent and Dynamic Environments. *IEEE Transactions on Robotics*, 2021.
- [17] Jesus Tordesillas, Brett T. Lopez, Michael Everett, and Jonathan P. How. Faster: Fast and safe trajectory planner for navigation in unknown environments. *IEEE Transactions on Robotics*, 38(2):922–938, 2022. DOI: 10.1109/TR0.2021.3100142.
- [18] Raouf Touti. Perlin noise: A procedural generation algorithm. <https://rtouti.github.io/graphics/perlin-noise-algorithm> (2024.05.01), 2023.
- [19] Nathan Witmer. A visual explanation of jump point search. <https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search/> (2024.04.17), May 2013.
- [20] Wu Maojia Yang Haodong. Motion planning and navigation of agv/amr (github repository). https://github.com/ai-winter/python_motion_planning (2024.04.18), 2023.