MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Beckert Adam**	Personal ID number:	**478386**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Multi-agent Path Finding with kinodynamic constraints

Master’s thesis title in Czech:

Multi-agentní plánování s kinodynamickými omezeními

Guidelines:

Multi-Agent Path Finding is an NP-hard task widely studied by both the robotics and artificial intelligence communities. Especially in recent years, many approaches have been proposed that differ in the quality of the solution found and the computational requirements. One of the methods is the MAPF-LNS2 metaheuristic, which generates an initial solution and subsequently improves it iteratively. In each iteration, a part of the solution is deleted and then the solution is repaired. The aim of the work is to modify the MAPF-LNS2 algorithm in such a way that it takes into account the kinodynamic properties of robots. It will be done in the following steps.
1. Familiarize yourself with the MAPF-LNS2 method [1] and its freely available implementation [2].
2. Get acquainted with the Safe Interval Path Planning (SIPP) method [3], its variant taking into account the kinodynamic properties of robots (SIPP-IP) [4] and the freely available implementation of SIPP-IP [5].
3. Design and implement a modification of MAPF-LNS2 taking into account kinodynamic properties of robots.
4. Experimentally verify the properties of the developed method and compare it with the SIPP-IP method.
5. Describe the proposed method and discuss obtained experimental results.

Bibliography / sources:

[1] https://ojs.aaai.org/index.php/AAAI/article/view/21266
[2] https://github.com/Jiaoyang-Li/MAPF-LNS2
[3] https://www.cs.cmu.edu/~maxim/files/sipp_icra11.pdf
[4] https://arxiv.org/pdf/2302.00776.pdf
[5] https://github.com/pathplanning/sipp-ip

Name and workplace of master’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **09.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

RNDr. Miroslav Kulich, Ph.D.		prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Multiagent path planning with kinematic constraints

Adam Beckert

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Supervisor–specialist: Ing. David Zahrádka
Field of study: Open informatics
Subfield: Artificial intelligence
May 2024

Acknowledgements

I would like to express my sincere gratitude to my supervisor, RNDr. Miroslav Kulich, Ph.D., for his great leadership, profound knowledge, and tireless patience with me throughout the whole year.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. May 2024

Abstract

This thesis addresses the complex problem of Multi-agent path finding in environments with dynamic objects and kinematic constraints of robots, a challenge central to the field of autonomous systems. We introduce "Safe Interval Path Planning with Soft Constraints and Interval Projection" (SIPPS-IP), a novel algorithm that combines elements of two existing algorithms to enhance the coordination of multiple agents in these environments. We then combine the SIPPS-IP with adaptive LNS, to find a solution to our problem. This integration ensures efficient, collision-free navigation while adhering to the kinematic limitations of the robots. We validate the efficacy and correctness of SIPPS-IP through extensive simulations in varied and dynamic scenarios.

Keywords: MAPF, dynamic objects, kinematic constraints, autonomous systems, Safe Interval Path Planning (SIPP), soft constraints, SIPPS-IP, adaptive LNS, collision-free navigation, autonomous navigation, autonomous logistics

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Katedra kybernetiky

Abstrakt

Tato diplomová práce se zabývá multiagentním plánováním v prostředích s dynamickými objekty a kinematickými omezeními robotů, což je výzva, která je klíčová v oblasti autonomních systémů. Představujeme "Safe Interval Path Planning with Soft Constraints and Interval Projection" (SIPPS-IP), nový algoritmus, který kombinuje prvky dvou stávajících algoritmů za účelem zlepšení koordinace více agentů v těchto prostředích. Poté kombinujeme SIPPS-IP s adaptivním LNS, abychom našli řešení pro tento problém. Tato integrace zajišťuje efektivní a bezkolizní navigaci při dodržení kinematických omezení robotů. Účinnost a správnost SIPPS-IP ověřujeme prostřednictvím rozsáhlých simulací v různých scénářích.

Klíčová slova: MAPF, dynamické objekty, kinematická omezení, autonomní systémy, SIPP, SIPP-IP, SIPPS-IP, adaptivní LNS, navigace bez kolizí, autonomní navigace, autonomní logistika

Překlad názvu: Multi-agentní plánování s kinodynamickými omezeními

Contents

Figures

Tables

Chapter 1

Introduction

Multi-Agent Path Finding (MAPF) is a complex problem crucial to autonomous systems. This chapter outlines the challenges in MAPF, especially those involving kinematic constraints, and sets the stage for integrating pathfinding algorithms with re-planning methods.

1.1 Background

MAPF takes part in robotics, computer science, and artificial intelligence, playing an important role in fields like logistics, vehicle navigation, and general robotics. MAPF focuses on navigating multiple agents through a shared space efficiently and without collisions. The problem can be approached in various ways depending on the simplifications made regarding the search space and the agents' mobility.

The performance of MAPF algorithms is critical, particularly in complex and dynamic environments. Efficient path planning affects resource management, cost efficiency, and system capacity. Enhancing MAPF algorithms can significantly improve the performance of autonomous systems. This thesis aims to improve MAPF solutions by focusing on scenarios involving kinematic constraints and dynamic conditions.

1.2 Objective

This thesis addresses the challenge of MAPF with kinematic constraints, an area rarely covered by traditional MAPF algorithms. As we consider non-holomic robots, these constraints involve agents' physical limitations, such as varying speeds, acceleration and deceleration, orientation, and navigation. The primary goal is to refine MAPF algorithms to account for these kinematic aspects, ensuring safe, efficient, and realistic navigation.

We propose a new solution called Safe Interval Path Planning with Soft Constraints and Interval Projection (SIPPS-IP) to deal with this issue. This solution aims to integrate realistic kinematic constraints into the pathfinding process, bridging the gap between theoretical models of Safe Interval Path Planning (SIPP) and practical applications in real-world environments.

Our approach builds on existing methods that extend the original SIPP algorithm to handle kinematic constraints such as acceleration, deceleration, and collisions in between agents. By combining these it allows us to find a solution in reasonable time, by finding solution with collisions which are then iteratively optimized ending in a collision-free plan.

1.3 Research Contributions

This research makes several key contributions to the field of MAPF and algorithm development:

- **Introduction of SIPPS-IP:** Developed and introduced the SIPPS-IP algorithm, which improves upon the existing SIPP-IP algorithm by incorporating interval projections and better handling kinematic constraints.
- **Implementation and Evaluation:** Conducted a comprehensive evaluation using established benchmarks. Compared to the SIPP-IP in MAPF, which has not been shown so far.
- **Implementation:** Successfully implemented the SIPPS-IP and SIPP-IP algorithm into the MAPF codebase, where it could be further tested.

- **Handling Kinematic Constraints:** Addressed the challenge of kinematic constraints in MAPF, providing solutions that are applicable to real-world scenarios where agent movements are subject to physical constraints.
- **Heuristic Development:** Developed new heuristics for SIPPS-IP that considers node expansion limit, contributing to more efficient pathfinding.

1.4 Scope and Limitations

This research focuses on integrating SIPPS-IP and adaptive LNS algorithms within dynamic, grid-based environments. The scope includes:

- **Agents and Dynamic Obstacles:** The study will consider scenarios involving multiple agents and dynamic, hard obstacles within the environment.
- **Environment Modeling:** The environment is represented as a grid, where each agent can have one of four orientations and move at speeds ranging from zero to a defined maximum speed. Benchmarks include environments of various sizes, from simple to maze-like configurations.

The study has certain limitations and assumptions:

- **Kinematic Constraints:** Agents are assumed to follow predefined kinematic constraints like speed limits and acceleration/deceleration capabilities, with all agents limited by the same constraints.
- **Predictability:** There is an assumption of predictability in agent behavior and environmental conditions, meaning no uncertainties such as delays in robot actions.
- **Scalability:** The focus is on smaller-scale scenarios involving dozens of agents, acknowledging potential limitations in scalability for very large or dense environments.

Chapter 2

Related Work

This chapter presents a short overview of the literature on MAPF, with a particular focus on approaches that incorporate kinematic constraints and collision handling, as these are central to the thesis. The most important papers, SIPP-IP [AY23] and MAPF-LNS2 [LCH$^+$22], for this thesis are only discussed here from a broad perspective and in the theoretical part Sections 4.4, 4.2 and 4.3, are described in detail.

We do not formulate this section as an overview of all approaches, but only cover a few broadly, because there exists a recent overview of MAPF [GLL$^+$23], so we would like to direct a reader there for a more comprehensive and detailed overview of the various methods used in solving MAPF problem.

2.1 A review of graph-based multi-agent pathfinding solvers: From classical to beyond classical

As MAPF encompasses multiple sub-tasks it is critical to identify different solutions for various problems, and possibly utilize their advantages to our issue. The paper [GLL$^+$23] discusses various MAPF solvers, ranging from classical approaches like optimal, bounded sub-optimal, and unbounded sub-optimal solvers. These advanced solvers address real-world complexities such as task assignment, heterogeneous agents, execution delay, mechanical failures, and kinematic constraints. The review also highlights the transition from classical MAPF approaches, which often involve simplified motion models and discretized environments, to more sophisticated methods that attempt to align more closely with real-world scenarios in a trade-off for the computational time.

The paper underscores the importance of adaptability, computational efficiency, and multi-objective optimization in MAPF solvers and points out current challenges and directions for future research. It also establishes foundational concepts and definitions used across the field.

2.2 SIPP: Safe Interval Path Planning for Dynamic Environments

The paper [PL11] introduces the Safe Interval Path Planning (SIPP) algorithm, a significant advancement in path-finding algorithm that addresses dynamic obstacles in real-time planning scenarios. SIPP is built on the concept of 'safe intervals', periods during which a robot can occupy a space without colliding with any moving obstacles. This approach allows the planner to efficiently handle dynamic environments by reducing the computational complexity typically associated with time dimension in path planning. The algorithm is described in detail in Section 4.1.3.

2.3 Multi-Agent Pathfinding with Continuous Time

The paper [AYS+22] extends path planning algorithm so it can work in a continuous time framework, and adapts the conflict based search based on this. It uses problem formulation MAPF_R, where the subscript R denotes real-valued non-uniform edge weights, therefore it does not rely on discretization of state space.

The main parts this paper introduces are:

- **Continuous-Time Algorithms**: The introduction of Continuous-time Conflict-Based Search (CCBS) and Satisfiability Modulo Theories-CCBS (SMT-CCBS) allows to move from discrete to continuous time modeling.

- **Handling of Complex Movements**: These algorithms are specifically designed to handle scenarios with variable action durations, a feature not typically accommodated in traditional MAPF algorithms.
- **Enhanced Path Optimization**: By operating in a continuous domain, these algorithms optimize path planning with a finer granularity.

The two algorithms CCBS - Continuous-time Conflict-Based Search and SMT_CCBS - SAT Modulo Theory CCBS are the two algorithms developed and they both utilize collision detection mechanisms, that is used in further path planning CSIPP (Constrained SIPP).

This paper showed promise and could have been applied to our problem, but as several comparisons have demonstrated, other suboptimal solver methods perform better, especially on a larger scale. An example of this is seen in Figure 2.1.

2.4 Multi-Agent Path Finding with Kinematic Constraints

The paper [HKC+16] introduces *MAPF-POST*, a novel approach that uses a simple temporal network to post-process the output of a MAPF solver. This approach ensures that the resultant plan can be executed on robots with non-holonomic constraints, taking into account their maximum translational and rotational velocities and providing a guaranteed safety distance between them.

The MAPF-POST approach works in three main steps:

- **Initial MAPF Solution** -First, a traditional MAPF solver is used to find an initial set of collision-free paths. These solvers are capable of handling hundreds of agents in cluttered environments efficiently. However, the resultant paths do not consider the kinematic constraints.
- **Temporal Plan Graph (TPG) Construction** - The output from the MAPF solver is converted into a Temporal Plan Graph (TPG), which is a directed acyclic graph representing the sequence of events where each event corresponds to an agent entering a location. Temporal precedences

are enforced between events, ensuring that an agent enters locations in the specified order and maintaining the sequence in which different agents enter the same location.

- **Simple Temporal Network (STN) Transformation** - The TPG is then transformed into a Simple Temporal Network (STN). Each edge in the STN represents a temporal constraint between events, annotated with bounds that correspond to the kinematic constraints of the robots. Specifically, the lower bound is determined by the minimum time required to move between locations based on the maximum velocity limits. This transformation ensures that the resultant plan execution schedule maintains a guaranteed safety distance between agents and accommodates slack to absorb imperfect executions.

The results show that MAPF-POST is capable of generating execution schedules in polynomial time, ensuring safe and efficient navigation for multiple robots. The approach has been validated through extensive simulations and real-world experiments, highlighting its potential for applications in warehouse automation, airport ground traffic management, and more.

While this closely relates to our thesis, we decided to go with a different approach. Here the solution is found with a simplified solver and the results are transformed to account for kinematic constraints, meaning that the path planning is not aware of this transformation, therefore the quality of the solution is not guaranteed.

2.5 Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery

The work [MHK$^+$19] deals with a variation of the Multi-Agent Pickup and Delivery (MAPD) problem, a more specialized MAPF, which makes the problem more difficult. The authors expand on Token Passing method which replans agents based on considering other agent paths in replanning. Previously the path finding algorithm was space-time A*, however, they introduced a new algorithm Safe Interval Path Planning with Reservation Tables (SIPPwRT). It handles continuous forward movements and utilizes a reservation table, where it maintains efficiently safe intervals for cells.

The reservation table with SIPP is also present in our work as they have also been utilized in MAPF-LNS2, and the reservation table is described in Section 4.1.1.

2.6 Safe Interval Path Planning With Kinodynamic Constraints

The Safe Interval Path Planning with Interval Projection (SIPP-IP) algorithm, as described in the paper [AY23], extends the SIPP algorithm by incorporating interval projection. This enhancement is created to be able to work with kinematic constraints. As this algorithm is used for our solution, we describe it in more detail in Section 4.3

2.7 MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search

The paper [LCH+22] presents the MAPF-LNS2 algorithm, it introduces SIPPS, which allows collisions between agents. These collisions are temporary as Large neighborhood search (LNS) iteratively optimizes this solution until we get to a collision-free plan.

- **Large Neighborhood Search**: The core innovation of MAPF-LNS2 is its use of a large neighborhood search (LNS) strategy. This approach enables the algorithm to efficiently handle collisions in a set of paths by iteratively replanning for a subset of agents.
- **Enhanced Collision Handling**: MAPF-LNS2 is designed to reduce collisions and achieve collision-free paths more efficiently. It starts with a set of paths that contain collisions and systematically reduces these through iterative replanning.
- **Dominance of SIPPS Nodes**: introduces weak and strong dominance on SIPPS Nodes to limit the number of nodes to be expanded.

MAPF-LNS2 represents a significant step forward in MAPF as it outperforms the previous algorithms significantly, both in terms of speed and success rate. The Figure 2.1 shows a comparison on the benchmark (on all 33 maps) [SSF+19a], against other MAPF solvers: PPS, EECBS, EECBS*, PP^R, where it outperforms all of them and with the 5-minute time limit it can solve 80 % of problems, compared to the second best 63 %.

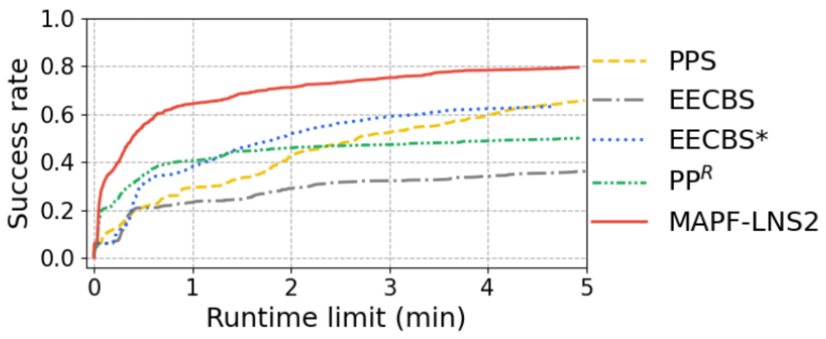


Figure 2.1: Success rates of LNS2 compared to PPS, EECBS, EECBS*, PP^R. Source: [LCH+22]

This work is the starting point for our approach and both the adaptation of SIPP and LNS are discussed in detail in the Theoretical Part.

Part I

Theoretical Part

Chapter 3

Problem specification

MAPF is a critical area of study that poses complex challenges, especially when agents must interact within dynamic environments. This chapter dives into the core aspects of MAPF, setting the stage for a deeper exploration of how agents navigate spaces crowded with obstacles and other agents.

Definition 3.1 (Classical MAPF)**.** The classical MAPF problem is defined by the following elements:

1. **Agent Set:** A set $A = \{1, 2, \ldots, k\}$ representing k agents.
2. **Graph:** An undirected graph $G = (V, E)$, where V denotes the node set and E denotes the edge set. Nodes represent possible agent locations, while edges represent the connections between them.
3. **Starting Locations:** A mapping $s : A \to V$ that assigns each agent to its starting point.
4. **Target Locations:** A mapping $t : A \to V$ that assigns each agent to its target point.

Time Assumption: The classical MAPF problem assumes discrete time steps, with each agent performing one action per time step. The actions include:

- **Wait Action:** The agent remains at its current node.

- **Move Action:** The agent moves to an adjacent node v', given that v' is different from the current node v and adjacent to it.

An action can be formally defined as a function $a : V \to V$, where $a(v) = v'$ represents moving from node v to node v', or remaining at node v if $v = v'$. Each action a also incurs a cost, denoted by $c(a)$, which quantifies the resources or effort required to perform the action. This cost can vary based on factors such as distance, time, or other relevant metrics, depending on the environments.

Plans: Agents follow sequences of actions to move from their starting to target locations. The sequence of actions performed by agent i is denoted by $\pi_i = (a_1, a_2, \ldots, a_n)$ and is known as a plan. If agent i starts at location $s(i)$ and reaches target $t(i)$ using plan π_i, it is considered a single-agent plan. The actions must follow temporal and spatial consistency, ensuring that each action starts immediately after the preceding one ends, and their respective vertices are the same.

Solution: A valid solution to the MAPF problem is a set of k single-agent plans (one per agent), ensuring that the plans are collision-free. Once an agent reaches its target, it may remain at that location or disappear.

The cost of a plan, denoted by $c(\pi_i)$, is the sum of the costs of the individual actions within the plan. **The cost of the overall solution** can be evaluated using different metrics. The makespan is the maximum time taken by any agent to complete its plan. This metric emphasizes the time efficiency of the slowest agent. Alternatively, the Sum of Costs (SOC) is the total sum of the costs of all agents' plans, reflecting the overall resource usage of the solution. A valid solution to the MAPF problem aims to minimize these costs while ensuring that all plans are collision-free. The definition follows the one from paper [SSF$^+$19b].

3.1 Approaches to Solving MAPF

While there are different approaches to solving the MAPF problem, we focus on Prioritized Planning (a decoupled approach). Other approaches include coupled planning, where all agents are planned together in a single search process.

3.1.1 Prioritized Planning

Prioritized Planning is a sequential approach used to solve the MAPF problem by giving each agent a specific priority. The idea is to plan paths for agents one by one in the order of their assigned priority. The agents with higher priorities plan their paths first, and subsequent agents plan theirs while avoiding collisions with paths already planned.

The solution is found such that agents are sorted based on their priority, which is usually assigned randomly[1]. Each agent plans its path from its starting point to its target using a single-agent pathfinding algorithm while treating the paths of higher-priority agents as dynamic obstacles to avoid them.

Advantages: The approach is simple and easy to implement, requiring only a good pathfinding algorithm and effective priority assignment. Being a decoupled approach, it can still yield partial solutions, meaning that even if a complete solution isn't found, it can at least provide plans for a subset of the agents.

Limitations: The solution may not be optimal, and agents with lower priorities might end up with infeasible paths, even if simpler alternatives are available.

Despite these limitations, Prioritized Planning provides a practical balance between computation speed and solution quality. In this work, we will focus on this method as a basis for developing robust solutions.

3.1.2 Typical MAPF example

We start our journey into the world of MAPF with Figure 3.1, which illustrates a typical scenario in an 8 by 8 grid, where eight agents successfully navigate without conflict, picturing an ideal case of MAPF. The paths here lead from the starting position (green circle), denoted as s_i, and goal positions g_i (red circle). The plan is optimal as for each agent we have the shortest path, meaning that we cannot even simplify the solution further.

[1]The priorities are typically assigned randomly, but alternative initial priorities can be established based on heuristic values specific to each agent. An interesting approach to learning these priorities is discussed in Paper [ZLH$^+$22].

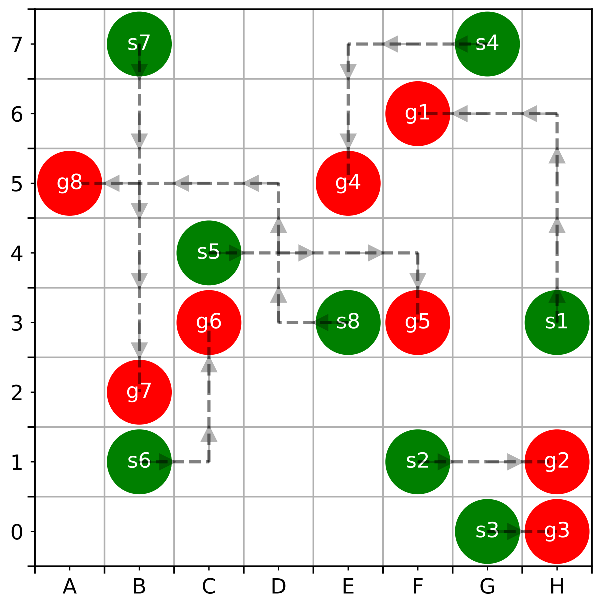


Figure 3.1: Example of regular MAPF with 8 agents

The chapter progresses to introduce practical examples that gradually reveal more complex interactions. These scenarios show different interactions between dynamic objects—agents and obstacles—that influence pathfinding. As we move forward, we will discuss how kinematic constraints affect agent movement.

Special attention is given to the pathfinding algorithm SIPP (Safe Interval Path Planning) and its extensions SIPP-IP and SIPPS, which are designed to address the intricacies of dynamic pathfinding. These algorithms are the foundation of our approach in creating SIPPS-IP.

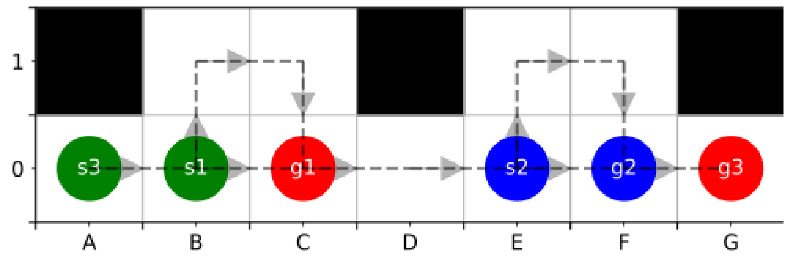
3.1.3 Collision-Prone MAPF Scenario

In the preceding section, we introduced the concept of MAPF using an example with eight agents navigating without collisions (see Figure 3.1). However, such ideal conditions are rare in practical applications. Real-world scenarios often involve dynamic interactions where agents' paths cross, leading to potential conflicts or collisions. Understanding these challenges is crucial for applying MAPF to complex environments like robotics, autonomous navigation, and logistics.

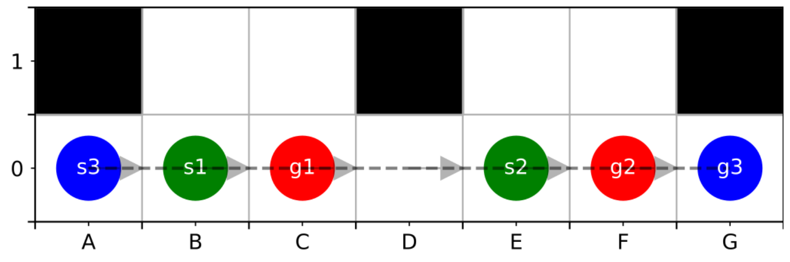
Example with Collisions

To illustrate the complexity introduced by collisions, consider the scenario depicted in Figure 3.2. This example contrasts two different planning outcomes based on the sequence in which agents plan their routes:

- **Collision-Free Scenario (Figure 3.2a)** Here, Agent S3 is planned first, followed by Agents S1 and S2. This sequence allows S3 to navigate through the space unobstructed by the others, who adjust their paths to accommodate S3. This example demonstrates how prioritizing one agent's path over others can lead to a collision-free outcome.
- **Collision Scenario (Figure 3.2b)** Conversely, when Agent S3 is planned last, its path conflicts with the routes of Agents S1 and S2, who have already established their paths. This arrangement leads to two collisions involving S3, highlighting how the order of planning significantly affects the feasibility of the final paths.



(a) : Collision avoided scenario



(b) : Collision forced scenario

Figure 3.2: Two scenarios of MAPF with 3 agents.

This comparison shows the crucial role of the order of agents in which we plan the paths of the agents. The order in which agents are scheduled can dramatically influence the effectiveness and feasibility of the solution. By analyzing collision scenarios, we can not only identify potential pitfalls in MAPF implementations but also explore strategic approaches for optimizing agent sequencing. Note that to get from the collision scenario in Fig 3.2 to collision-free we would have to re-plan all 3 agents together and in order, where S3 is the first one planned, otherwise the path will be obstructed by the other agents.

3.2 MAPF with Kinematic Constraints

The traditional MAPF problem is significantly limited in considering kinematic constraints, which reflect the physical capabilities and limitations of agents, such as their velocities, accelerations, orientation, and ability to maneuver in a given space.

Kinematic constraints introduce a layer of complexity to MAPF by imposing restrictions based on the physical properties of agents. These constraints affect how agents can start, stop, and change their velocities, requiring more sophisticated solutions than standard pathfinding algorithms provide. We define these constraints formally as follows:

Definition 3.2 (Kinematic Constraints)**.** A kinematic constraint for an agent covers limitations on its ability to modify its state of motion, encapsulated by parameters such as maximum speed $v_{\max}$, maximum acceleration $a_{\max}$, maximum deceleration $a_{\min}$, and maximum angular velocity $\omega_{\max}$. The kinematic state of an agent at any specific time is represented as a tuple comprising its position, linear velocity, and angular velocity (x, v, ω), where x denotes position, v represents linear velocity, and ω denotes angular velocity.

Definition 3.3 (Configuration)**.** The configuration of an agent in the context of MAPF with kinematic constraints represents all possible kinematic states that the agent can assume within the operational environment. This is formally modeled as a graph $G = (V, E)$, where each vertex $v \in V$ corresponds to a kinematic state of the agent, capturing both position and velocity. Each edge $(u, v) \in E$ within this graph signifies a feasible kinematic transition from state u to state v, adhering to the agent's kinematic constraints.

While configurations can be various for the purposes of this thesis, the configuration of an agent is defined as a three-tuple encompassing location, speed, and orientation, which will be denoted as (l, s, θ). Here, l represents the specific location of the agent within the environment. The speed s is from the domain $< 0, 1 >$, where 0 indicates a stop (zero velocity), and 1 indicates maximum speed. The orientation θ takes values in $\{1, 2, 3, 4\}$, corresponding to the four cardinal directions: North, South, West, and East, respectively. Throughout this thesis, when referring to a 'node', it is to be understood as this specific configuration tuple unless specified, (l, s, θ), representing a discrete state within the state space graph $G = (V, E)$.

These configurations are simplifications that are used together with the actions as motion primitives. They use the kinematic constraints to efficiently represent all reasonable movements of the robot. More information on motion primitives can be found in Section 4.3.2.

Path Representation with Time Intervals

Addressing the pathfinding problem with kinematic constraints requires that each agent's path be detailed not only in terms of spatial traversal but also in terms of the temporal occupation of each location. We represent the path of an agent as a sequence of tuples, where each tuple consists of a configuration and a corresponding time interval during which the agent occupies that location.

For simplicity, consider an agent moving through two cells at a constant speed, taking 10 timesteps per cell. This results in a path representation as follows:

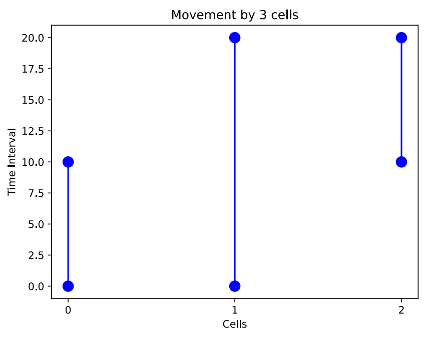
$$\{(\text{cell}_0, [0, 10)), (\text{cell}_1, [0, 20)), (\text{cell}_2, [10, 20])\}$$

In this model, as the agent begins moving, it immediately starts to partially occupy the next cell while still occupying the previous one. This results in the middle cell being occupied for the entire duration of the movement: initially shared with cell_0 for the first 10 timesteps and then with cell_2 for the next 10 timesteps.

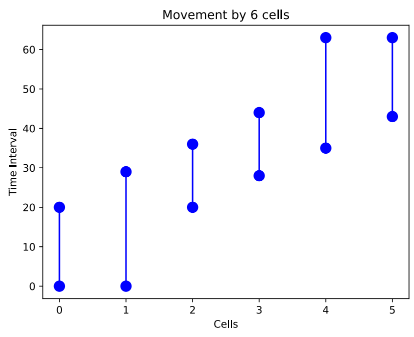
Figures 3.3b and 3.3c illustrate the agent's capability to accelerate to maximum speed. The first case demonstrates starting from speed 0, accelerating to maximum, and then decelerating back to speed 0. The latter case shows acceleration from speed 0 continuing until the final cell is reached.

Definition of Path

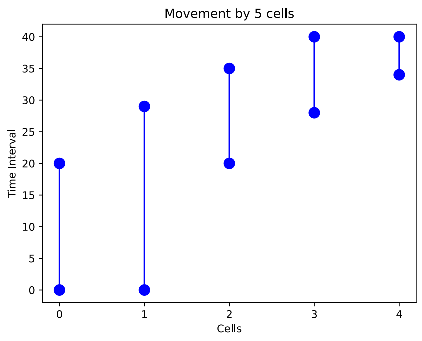
Definition 3.4 (Path). Let $G = (V, E)$ be a state-space graph where V represents the configurations of an agent and E represents the feasible transitions (edges) between these configurations under kinematic constraints.



(a) : Simple 2-cell movement



(b) : 5 cell movement primitive



(c) : Only accelerating primitive

Figure 3.3: Example of primitives

A timed path π for agent a can be formally defined as a sequence of tuples:

$$\pi = \{(e_1, t_1), (e_2, t_2), \dots, (e_L, t_L)\}$$

where:

- $e_j = (v_{j-1}, v_j) \in E$ is an edge representing the motion from configuration v_{j-1} to v_j, adhering to the agent's kinematic constraints.
- $t_j = [lb_j, ub_j) \in \mathbb{Z}^+ \times \mathbb{Z}^+$ is the time interval during which the agent occupies the configurations v_{j-1} and v_j. Here, lb_j and ub_j denote the lower and upper bounds of the time interval, respectively.

The path is feasible if and only if for all consecutive edges e_j and e_{j+1}, the following conditions hold:

- $ub_j = lb_{j+1}$ ensuring temporal continuity and consistency.

- The transitions respect the kinematic constraints imposed by the agent's capabilities, such as speed limits, acceleration bounds, and angular velocities.

Cost of the Path

Definition 3.5 (Cost of a Timed Path)**.** Let $\pi_i = \{(e_1, t_1), (e_2, t_2), \ldots, (e_L, t_L)\}$ represent the timed path of agent a_i, where $e_j = (v_{j-1}, v_j)$ are edges representing the transitions under kinematic constraints from configuration v_{j-1} to configuration v_j and t_j are the respective time intervals of these transitions. The cost of the path, denoted as $C(\pi_i)$, is defined by the total time span from the start of the first transition to the completion of the last transition. Formally, the cost is calculated as follows:

$$C(\pi_i) \;=\; ub_L - lb_1$$

where:

- ub_L is the end timestep of the last transition in the path, indicating when this transition is completed.
- lb_1 is the starting timestep of the first transition in the path, marking the commencement of the agent's journey.

This calculation ensures that the cost reflects the total duration from the initial movement to the conclusion of the last activity, encompassing any intermediate waiting times inherently between transitions.

Path example. Let's consider a simple scenario where an agent has to find a path from cell [5,2] to [0,0], such that there is an obstacle at a goal location at an interval [60,103). In Figure 3.5 the intervals in the lattice show intervals when the agent occupies the given cell. In the bottom image, we can see that the path consists of 4 actions each labeled with a different color. The blue one is moving by 5 cells, initially, the intervals are quite wide, in the midpoint of this motion much shorter as the agent picks up speed and again widens as the agent slows down. The red action is turning to the left, which takes 10 timesteps. Next, we are facing the goal and we can reach it, however, if we take immediately 2 steps to we will end up in a collision with an obstacle, therefore the yellow action is waiting until we can perform this motion. Lastly, the 2-step motion is the green action, with which we reach the goal position immediately after the dynamic obstacle leaves the cell (black interval).

The path is then represented as in the grid, where for each cell we have the corresponding interval so that it can be used for planning of other agents.

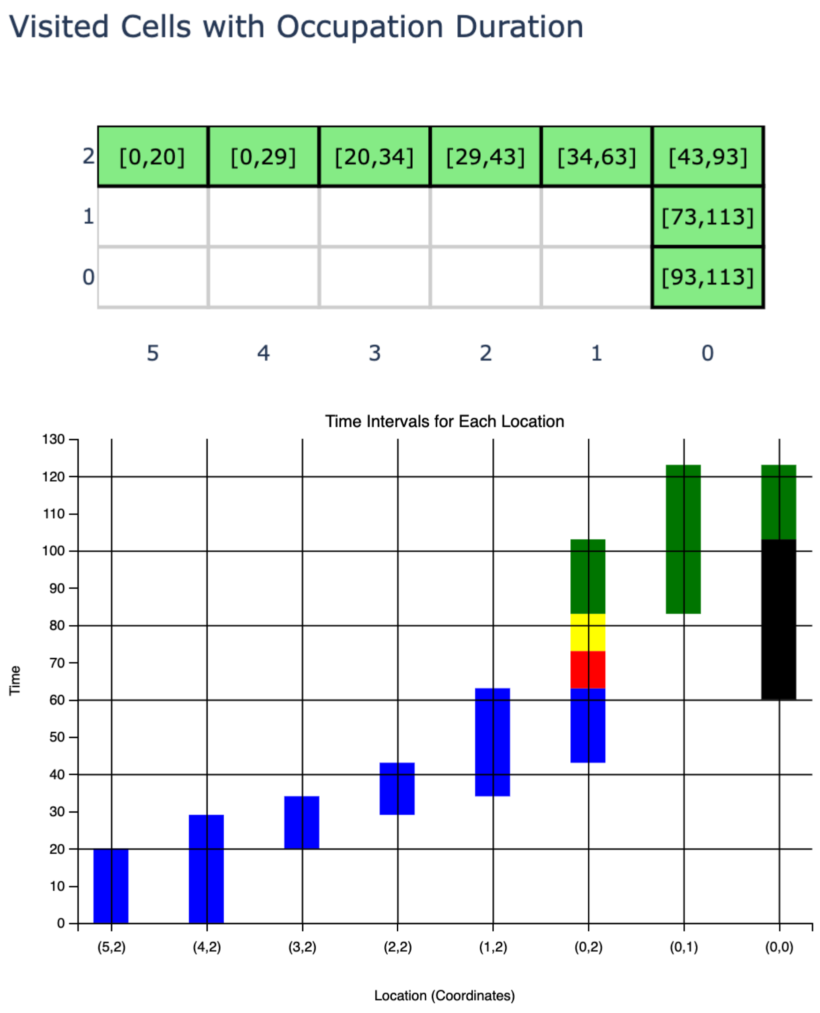


Figure 3.5: Intervals of the path

3.2.1 Collisions

Firstly, we need to identify two types of objects that we deal with in this framework: soft objects (agents) and hard objects (dynamic obstacles), the set of agents denoted by A (or O^s) and the set of hard objects as H (or O^h), with O describing the set of all objects. Paths for both types of objects are defined as described in the previous section.

Definition 3.6 (Collision and Number of Collisions)**.** A collision occurs between two distinct objects, an agent $a_i \in A$ and another object $a_j \in O$ where $a_i \neq a_j$ if their paths lead to occupying the same location at overlapping time intervals. Formally, a collision between two paths $\pi_i = \{(e_{i,1}, t_{i,1}), \dots, (e_{i,L}, t_{i,L})\}$ and $\pi_j = \{(e_{j,1}, t_{j,1}), \dots, (e_{j,M}, t_{j,M})\}$ is defined if there exists at least one location p such that $p \in \{v_{i,k}, v_{i,k-1}\} \land p \in \{v_{j,l}, v_{j,l-1}\}$ for some k and l, where $v_{i,k}$ and $v_{j,l}$ are endpoints of the edges $e_{i,k} = (v_{i,k-1}, v_{i,k})$ and $e_{j,l} = (v_{j,l-1}, v_{j,l})$, and the time intervals $t_{i,k} = [lb_{i,k}, ub_{i,k})$ and $t_{j,l} = [lb_{j,l}, ub_{j,l})$ overlap:

$$[lb_{i,k}, ub_{i,k}) \cap [lb_{j,l}, ub_{j,l}) \neq \emptyset.$$

We distinguish between two types of collisions:

- **Vertex Collision:** A vertex collision occurs if two agents meet at the same location at the same time. Formally, a vertex collision between paths π_i and π_j occurs if there exists a location v and times $t_{i,k}$ and $t_{j,l}$ such that:

$$v = v_{i,k} = v_{j,l} \quad \text{and} \quad [lb_{i,k}, ub_{i,k}) \cap [lb_{j,l}, ub_{j,l}) \neq \emptyset.$$

- **Edge Collision:** An edge collision occurs if two agents traverse the same edge in opposite directions at the same time. Formally, an edge collision between paths π_i and π_j occurs if there exist edges $e_{i,k}$ and $e_{j,l}$ such that:

$$e_{i,k} = (v_{i,k-1}, v_{i,k}) \quad \text{and} \quad e_{j,l} = (v_{j,l-1}, v_{j,l}),$$

  and the agents traverse these edges in opposite directions with overlapping time intervals:

$$v_{i,k} = v_{j,l-1} \quad \text{and} \quad v_{i,k-1} = v_{j,l} \quad \text{and} \quad [lb_{i,k}, ub_{i,k}) \cap [lb_{j,l}, ub_{j,l}) \neq \emptyset.$$

The number of collisions between paths π_i and π_j, denoted as $c(\pi_i, \pi_j)$, is given by the sum of all colliding edges between the paths:

$$\begin{aligned} c(\pi_i, \pi_j) = \sum_{k=1}^{L} \sum_{l=1}^{M} \mathbf{1}\Big(& (\{v_{i,k}, v_{i-1,k}\} \cap \{v_{j,l}, v_{j-1,l}\} \neq \emptyset) \\ & \land ([lb_{i,k}, ub_{i,k}) \cap [lb_{j,l}, ub_{j,l}) \neq \emptyset) \Big) \end{aligned} \tag{3.1}$$

where $\mathbf{1}(\text{condition})$ is the indicator function that evaluates to 1 if the condition is true and 0 otherwise.

Total Number of Collisions for an Agent

Now that we have collisions between two paths we can extend it to apply to the whole set of agents and dynamic obstacles.

Definition 3.7 (Total Number of Collisions)**.** The total number of collisions c_i for agent a_i with the set of all other agents O is defined as:

$$c_i = \sum_{o_j \neq a_i, o_j \in O} c(\pi_i, \pi_j)$$

where $c(\pi_i, \pi_j)$ is the number of colliding edges between paths as defined previously and π_i and π_j are paths of the objects.

To quantify the total number of collisions among all agents in the set A, we can define:

The total number of collisions among all agents in A is given by:

$$c_{\text{total}} = \frac{1}{2} \sum_{i=1}^{|A|} \sum_{j \neq i, j=1}^{|A|} c(\pi_i, \pi_j) + \sum_{i=1}^{|A|} \sum_{j=1}^{|H|} c(\pi_i, \pi_j)$$

Since we do not plan the dynamic obstacles, it is expected that those are collision-free and even if they were not, it should not affect the total number of collisions as we would not be able to change it[2].

Example Collisions. Consider a scenario Figure 3.6 with three agents where:

- Agent a_1's path is $\pi_1 = \{((1,2),[0,2)),((2,3),[2,4)),((3,4),[2,4))\}$.
- Agent a_2 has $\pi_2 = \{((2,1),[0,2)),((1,2),[2,3))\}$.
- Dynamic obstacle a_3's path: $\pi_3 = \{((4,3),[0,3)),((3,4),[3,5))\}$

[2]The factor of 1/2 is used to ensure that each pair of collisions is only counted once, as $c(\pi_i, \pi_j)$ and $c(\pi_j, \pi_i)$ would otherwise both contribute to the count for each pair of agents i and j.

Each tuple in the paths of the agents and the dynamic obstacle now accurately represents the transitions between locations and the specific intervals for these transitions. This update ensures that the paths reflect the actual movement dynamics, including the beginning and end of each transition, crucial for determining potential collisions based on shared locations and overlapping time intervals.

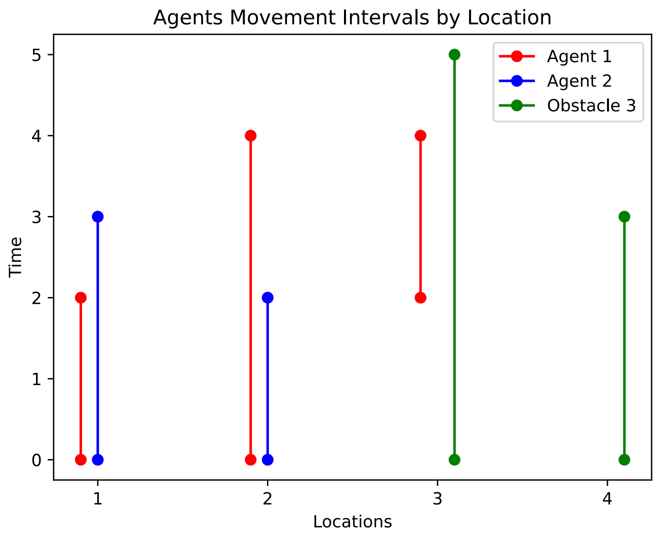


Figure 3.6: Example of collisions for 3 objects

In this scenario, agent a_1 experiences:

- Two soft-collisions with agent a_2 at locations 1 and 2, for 2 timesteps
- One hard-collision with obstacle a_3 at location 3 during the overlapping interval $[2, 4)$ interval.

While we distinguish between the two types of collisions, if we talk about number of collisions we mean either sum of soft-collisions and hard-collisions or just the sum of soft-collisions, if it is obvious. (the obstacles are already accounted in the state-space) So in this context we can say that agent 1 has 3 collisions, and agent 2 has 2 soft-collisions with agent 1.

3.2.2 Collision-Free Paths

Collision-free paths are ones where agents and obstacles do not share the same space at overlapping time intervals. We can formalize this concept using the collision metrics defined previously.

Definition 3.8 (Collision-Free Paths)**.** A set of paths for agents in the set A and obstacles in the set H is considered collision-free if the total number of collisions c_{total} among all pairs of agents, and between agents and obstacles, is zero. Formally, this is defined as:

$$c_{\text{total}} = 0$$

This implies that:

$$\forall i, j \in A, i \neq j : C(\pi_i, \pi_j) = 0 \quad \text{and} \quad \forall i \in A, \forall o \in H : C(\pi_i, \pi_o) = 0$$

3.2.3 Definition of MAPF with Kinematic Constraints

Let us consider a set of dynamic obstacles H and a set of agents $A = \{a_1, a_2, ..., a_n\}$, each with unique starting s_i and target positions g_i on a graph $G = (V, E)$ representing the environment. Each agent a_i has associated kinematic constraints that define its movement capabilities. The task is to find paths π_i for all agents such that:

- π_i is a valid path in G from s_i to g_i,
- All paths are collision-free,
- All paths are feasible under the kinematic constraints of agents.

Optimization Objective - Sum of costs (SOC)

In our MAPF problem, we aim to minimize the total cost for all agents to reach their respective goals, thereby optimizing the sum of the costs of all agents.

So the optimization objective, that we are interested in is:

$$\min \sum_{i=1}^{n} (cost(\pi_i))$$

with respect to the MAPF with Kinematic Constraints formulation. [3]

[3]The MAPF can have other various objectives, such as minimizing the makespan, computation time, or total success rate on the subset, neither of which will be discussed here, however their solutions can be found in MAPF overview [GLL+23].

3.2.4 Sources of Complexity in MAPF

MAPF presents several intrinsic challenges that significantly contribute to its computational complexity. The task involves coordinating multiple agents to ensure that each reaches its destination efficiently without collisions, under the constraint of minimizing the sum of costs. Below, we outline the key factors that complicate MAPF:

- **High-Dimensional Search Space:** The search space for MAPF is exponentially large due to the number of agents. Each agent introduces additional dimensions to the problem's state space, representing its possible locations at each time step. The growth of the state space is exponential to the number of agents, which severely complicates the search process for feasible and optimal paths.
- **Inter-Agent Collision Avoidance:** One of the central complexities in MAPF is ensuring that no two agents collide. Each agent must be aware of not only static obstacles within the environment but also the dynamic trajectories of other agents. Planning paths that prevent collisions involves considering the current and future states of multiple agents, which adds a significant layer of complexity, especially as the number of agents increases.
- **Dynamic Environments:** In many practical applications, the agents operate in environments where obstacles or layout configurations may change dynamically. Adapting to such changes in real-time further complicates the pathfinding problem as it requires agents to continuously update their paths in response to the environment's evolution.
- **Kinematic Constraints:** imposing limitations on the agents' movement capabilities such as maximum speed, acceleration, and turning capabilities. These constraints require that the path-planning algorithms consider not only the position but also the velocity and orientation of agents at each step. Unlike usual MAPF formulations where transitions are well-defined and often just relate to neighboring locations in space, we have to consider continuous movements. This is later simplified to motion primitives. 4.3.2

3.2.5 Why MAPF is NP-hard

A problem is classified as NP-hard if solving it is at least as difficult as the hardest problems in NP (nondeterministic polynomial time). An NP-hard problem does not necessarily have a known algorithm that can solve it in polynomial time, and it is as difficult as any problem to which an NP problem can be reduced in polynomial time.

Reduction from Known NP-hard Problems: MAPF can be reduced from well-known NP-hard problems such as the "3-SAT" problem [SFSB16]. These reductions show that solving MAPF is at least as difficult as these classical NP-hard problems. Specifically, the complexity in MAPF arises because ensuring that multiple agents navigate from their start positions to their destinations without collision involves solving multiple, interdependent pathfinding problems simultaneously. Each agent's path potentially affects every other's, creating a problem structure similar to solving multiple "Vertex Disjoint Paths Problems" concurrently.

Computational Intractability: Optimal solvers for MAPF, which aim to find collision-free paths with the minimal possible cost (makespan or sum-of-costs), face the challenge of searching through an exponentially growing state space as the number of agents increases. This growth results in a combinatorial explosion of potential solutions, making the search computationally intractable, especially in environments densely populated with agents. The complexity is further compounded by the need to account for dynamic changes in the environment and agent interactions, which continually alter the feasible solution space.

These factors collectively underscore the inherent difficulty in devising efficient, optimal algorithms for MAPF and justify its classification as an NP-hard problem.

Chapter 4

Solution Approach

The solution, that we propose, involves integrating techniques for solving the MAPF problem with pathfinding methods that address kinodynamic constraints. Specifically, we leverage MAPF-LNS2, which incorporates the SIPPS algorithm for pathfinding alongside an effective re-planning strategy known as LNS2 (adaptive LNS). Additionally, we utilize SIPP-IP, a method focused on path planning with kinematic constraints. As both approaches are based on SIPP, we have devised a unified strategy that merges these methodologies. This integrated solution maintains the same re-planning process as MAPF-LNS2, while introducing a novel algorithm, SIPPS-IP, for enhanced pathfinding.

This chapter is divided into two main sections. The first section describes the pathfinding methods for agents, specifically focusing on how SIPPS and SIPP-IP handle the challenges of navigating agents through environments where obstacles and conditions can change. The second section explains finding the solution to the MAPF problem. This part introduces the MAPF-LNS2 approach and how we find a solution.

4.1 Pathfinding with Dynamic Objects

Classical pathfinding algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's algorithm are designed primarily for static environments [RN16]. In such settings, these algorithms perform optimally by assuming that obstacles and paths remain constant throughout the navigation process. However, this assumption limits their applicability in dynamic environments where the state of obstacles and paths can change unpredictably over time. These changes necessitate algorithms that can adapt to evolving conditions to maintain path validity and optimality.

To effectively address these limitations in dynamic scenarios, advanced algorithms such as Space-Time A* (STA*) [Sil05] and Safe Interval Path Planning (SIPP) [PL11] have been developed. STA* incorporates time as an explicit dimension in its search strategy, allowing the algorithm to consider both spatial and temporal variables in pathfinding. This integration enables STA* to anticipate and adapt to changes in the environment, enhancing its utility in dynamic settings. Conversely, SIPP on the other hand works splits time dimension into intervals, which are then used instead during the search, thus reducing computational overhead.

4.1.1 Space-Time A* Algorithm

Space-Time A$*$ (STA$*$) extends the classic A* search algorithm by incorporating a time dimension into the search space resulting in a three-dimensional grid. Each node in this space-time grid represents a state defined by its spatial coordinates and a specific time step. This section summarizes a document[1] [Sil20].

Definition 4.1 (Space-Time Node)**.** A *space-time node* in STA* is defined as a tuple (p, t), where $p = (x, y)$ are the spatial coordinates on the grid and t represents the time step at which the node is evaluated.

Node Expansion. Node expansion considers not only the spatial neighbors of the current node but also how these neighbors change over time. The algorithm evaluates potential paths based on both their spatial and temporal viability.

1. **Temporal Consistency:** Ensures that transitions between nodes are possible not only spatially but also at appropriate times.
2. **Dynamic Obstacle Avoidance:** Each node must be checked against the trajectories of moving obstacles to avoid collisions at the next step.

[1]The online document (https://www.davidsilver.uk/wp-content/uploads/2020/03/coop-path-AIWisdom.pdf) describes the Space-Time A* more in-depth really well concepts used in planning, I recommend the reader to follow there to get more information there.

Heuristic Function

The heuristic function in STA* is adapted to estimate the cost from a node to the goal considering both distance and time. It typically combines a spatial heuristic, such as the Manhattan distance, with a temporal component that estimates the minimal time required to reach the goal safely.

$$h((x,y),t) \;=\; D((x,y),g) + T((x,y),t) \tag{4.1}$$

Here, $D((x,y),g)$ represents the spatial distance from the current position (x,y) to the goal g, and $T((x,y),t)$ denotes the estimated time delay as a function of time t.

The f-value in STA* is the total estimated cost from the start node to the goal, passing through the given node. It combines the actual cost from the start to the current node (g-value) and the heuristic estimate of the cost from the current node to the goal (h-value). Mathematically, it is expressed as:

$$f(n) \;=\; g(n) + h(n)$$

In this equation, $g(n)$ is the cost from the start node to the current node n, typically the timestep we enter the current node, and $h(n)$ as define earlier $(n = ((x,y),t))$.

Estimating the time delay $T((x,y),t)$ accurately in a space-time context is challenging. Unlike the spatial distance, which can be relatively straightforward to calculate using methods like the Manhattan distance, estimating the time delay must account for various dynamic factors such as potential future conflicts with other agents, varying speeds, and the need to synchronize movements. This complexity makes the time delay estimate highly non-trivial and prone to inaccuracies, further complicating the search process and potentially leading to suboptimal paths. Various algorithms, such as SIPPS and SIPP-IP, have different approaches to dealing with this challenge, but some may choose to simplify or ignore the time delay component, as the heuristic remains admissible.

Algorithm

The STA* is very similar to traditional A*, therefore the main part of the Algorithm 1 is basically identical to the A*. The algorithm begins by initializing the cost of the start state s_{start} to zero (line 1). The OPEN list, which stores nodes to be explored, is initially empty (line 2). The start state is inserted into the OPEN list with an f-value equal to its heuristic estimate $h(\text{start})$ (line 3).

The main loop of the algorithm runs until the goal state s_{goal} is expanded (line 4). Within the loop, the algorithm selects the node s with the smallest f-value from the OPEN list (line 5) and generates its successors using the *getSuccessors* function (line 6).

For each successor s', the algorithm checks if it has been visited before (line 8). If not, the initial cost $g(s')$ and $f(s')$ are set to infinity (line 9). If the newly computed cost to reach s' via s is lower than the previously known cost $g(s')$, the costs $g(s')$ and $f(s')$ are updated (lines 11-13), and s' is re-inserted into the OPEN list with its new f-value (line 14).

Algorithm 1 space-time A*

```
1:  $g(s_{\text{start}}) \leftarrow 0$
2:  OPEN $\leftarrow \emptyset$
3:  **insert** $s_{\text{start}}$ into OPEN with $f(s_{\text{start}}) = h(\text{start})$
4:  **while** $s_{\text{goal}}$ is not expanded **do**
5:      $s \leftarrow$ remove node with the smallest $f$-value from OPEN
6:      successors $\leftarrow$ getSuccessors($s$)
7:      **for** each $s'$ in successors **do**
8:          **if** $s'$ was not visited before **then**
9:              $f(s') \leftarrow g(s') \leftarrow \infty$
10:         **if** $g(s') > g(s) + c(s, s')$ **then**
11:             $g(s') \leftarrow g(s) + c(s, s')$
12:             $f(s') \leftarrow g(s') + h(s')$
13:             **insert** $s'$ into OPEN with $f(s')$
```

The function *getSuccessors*, shown in Algorithm 2, generates valid successor states for a given state s. The function starts by initializing an empty set of successors (line 2) and retrieves the current time and node from the state s (lines 3-4). For each possible action, it computes the resulting next node and the time required to perform the action (lines 5-6).

The validity of the next state, combining the next node and time, is checked (line 7). If valid, a new state is created, incorporating the updated node and time (line 8). The costs g and f for the next state are calculated (lines 9-10), and the next state is added to the set of successors (line 11). Finally, the function returns the set of valid successors (line 13).

Algorithm 2 Function getSuccessors

```
1:  function GETSUCCESSORS($s$)
2:      $successors \leftarrow \emptyset$
3:      $current\_time \leftarrow s.time$
4:      $current\_node \leftarrow s.node$
5:      for each $action$ in $possible\_actions$ do
6:          $next\_node \leftarrow \text{apply}(action, current\_node)$
7:          $next\_time \leftarrow current\_time + \text{cost}(action)$
8:          if $\text{isValid}(next\_node, next\_time)$ then
9:              $next\_state \leftarrow \text{createState}(next\_node, next\_time)$
10:             $next\_state.g \leftarrow s.g + \text{cost}(action)$
11:             $next\_state.f \leftarrow next\_state.g + \text{h}(next\_state, s_{\text{goal}})$
12:             $successors \leftarrow successors \cup \{next\_state\}$
13:     return $successors$
```

Reservation Table

A reservation table is a crucial data structure in cooperative pathfinding. Its primary purpose is to prevent path conflicts among multiple units navigating in the same environment. Each cell in a space-time map is represented within this table, with each entry indicating whether the cell is available or reserved. Once a path is chosen by a unit, it marks the cells along this path in the reservation table. These marked cells act as transient obstacles, ensuring no other unit can occupy the same space at the same time, effectively reserving the trajectory for the moving unit.

The reservation table is not being presented in the pseudocode, but it affects the neighbors selected. The part where it is used is function isValid (line 8), where we check whether the action can be applied and we do not collide with another object. The implementation is straightforward yet effective. It involves marking each cell in the space-time map that corresponds to the planned path of a unit. This marking process creates a reservation for each step of the unit's path, blocking those cells for specific time steps. To manage the potentially large size of space-time maps (for example, 256 x 256 x 256), the data structure is optimized for sparsity. The reservation table, is typically implemented through the hash map, which allows efficient handling of reservations as they can be quite sparse.

4.1.2 Theoretical Properties

- **Completeness:** STA* is complete, meaning it will find a path if one exists, as long as the dynamics of the obstacles are predictable and the environment is discretely and accurately modeled.
- **Optimality:** The path found by STA* is optimal with respect to the defined cost function, provided that the heuristic used is admissible and consistent.

Even though STA* is quite a flexible and efficient algorithm, for the dynamic environments SIPP outperforms it in computation time and especially memory. The STA* is a precursor to the following algorithms and has quite a lot in common, we use this as an entry point and pseudo-codes of other algorithms will be compared to this one (marked by yellow color).

4.1.3 Safe Interval Path Planning (SIPP)

SIPP is an innovative approach designed for dynamic environments [PL11] where the agent must avoid collisions with moving obstacles. It extends the classical path planning techniques by incorporating the time dimension effectively, thus enabling the prediction and avoidance of potential collisions with dynamically moving objects. Compared to STA*, it splits the time into intervals and operates with them instead leading to faster search.

Safe Interval Table

The core concept of SIPP involves the use of a *Safe Interval Table*, which is a structured way to manage the safe intervals during which a robot can occupy certain configurations without collisions with dynamic obstacles. A safe interval for a configuration is defined as a continuous segment of time during which the configuration remains collision-free, meaning if extended by one timestep in either direction, the configuration would result in a collision.

Definition 4.2 (Safe Interval)**.** A *safe interval* $[t_{start}, t_{end})$ for a configuration is a time period during which the agent can safely remain in or pass through a configuration without colliding with any moving obstacles. The interval is bounded by the first safe time step t_{start} and the first unsafe time t_{end} immediately after the safe period.

In Figure 4.1 we have an object moving within 4 cells, it takes 1 timestep to move from one cell to the neighboring one and moves through cells in this order [1, 2, 3, 4, 4, 3, 2]. The overlap of intervals is caused by a continuous movement of the agent and by the space representation, therefore when the object starts at timestep 1 in cell 1 it occupies both cell 1 and 2 simultaneously until it is in the middle of cell 2.

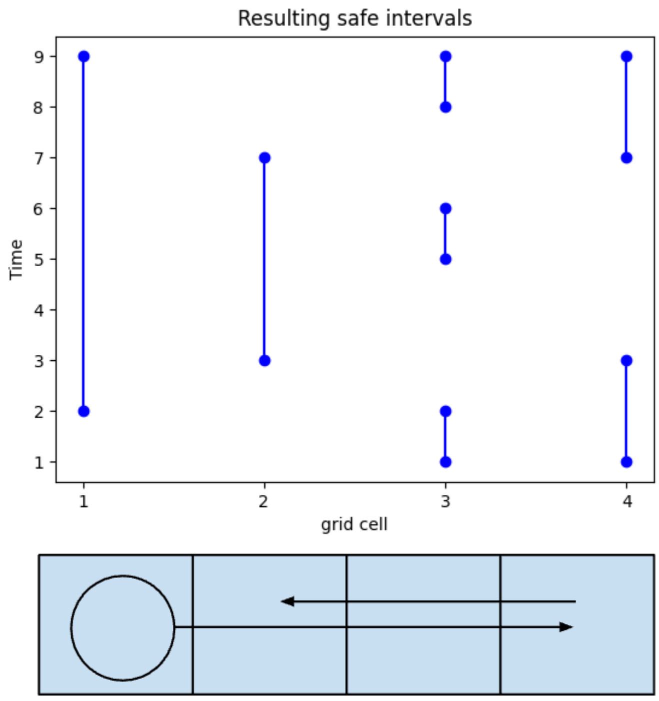


Figure 4.1: Example of safe intervals for 4 cells when dynamic object is added.

Graph Construction

The SIPP algorithm begins by constructing a timeline for each spatial configuration. This is done by iterating through each point along the trajectory of each dynamic obstacle and updating the timelines for all the configurations within collision distance of the point. So we end up with a safe interval table that already covers all dynamic obstacles and all we need to do is search for a path within these safe intervals[2].

[2]While we could use dynamic obstacles of various sizes and update the safe-intervals accordingly, for the evaluation we simply work with obstacles of same size as the robot. (occupying single space point)

SIPP Algorithm

SIPP modifies the traditional A* search algorithm by incorporating the safe intervals. The only difference between space-time A* is how we get the neighbors in `getSuccessors`, the main loop stays the same so we can reuse pseudo-code in Algorithm 1, however at the start of the algorithm we create the safe interval table, which was described earlier.

The function *getSuccessors*, as detailed in Algorithm 3, is responsible for generating valid successor states for a given state s. The function begins by initializing an empty set of successors (line 2). It then iterates over each possible motion m that can be applied to the current state s (line 3), M(s) provides all possible motions at the current state. For each movement m, the resulting configuration *cfg* is determined , and the time required to execute the movement m_{time} is calculated (line 4-5). The start and end times for this movement are then computed based on the current state's time and interval (lines 6-7).

Next, the function iterates over each safe interval i in the resulting configuration cfg (line 8). It checks if the interval i is valid by ensuring that its start time is not after the movement's end time and its end time is not before the movement's start time (lines 9-10). If the interval i is valid, the function computes the earliest arrival time t at the configuration cfg during the interval i without collisions (line 11). If no valid arrival time exists, the function skips to the next interval (lines 12-13), otherwise if a valid arrival time t is found, a new state s' is created with the configuration cfg, the interval i, and the arrival time t, this new state s' is then added to the set of successors ((lines 14-15)). After processing all possible movements and intervals, the function returns the set of valid successors (line 16).

Theoretical Guarantees

SIPP provides strong theoretical guarantees, including completeness and optimality under certain conditions. It ensures that if a path exists, the algorithm will find it, and the path will be optimal with respect to the shortest time to traverse while avoiding collisions. The guarantees are under the assumption that the heuristic function used is admissible and consistent.

Algorithm 3 getSuccessors

```
function GETSUCCESSORS($s$)
    successors $\leftarrow \emptyset$
    for each $m$ in $M(s)$ do
        $cfg \leftarrow$ configuration of $m$ applied to $s$
        $m_{\text{time}} \leftarrow$ time to execute $m$
        $start_t \leftarrow \text{time}(s) + m_{\text{time}}$
        $end_t \leftarrow \text{endTime}(\text{interval}(s)) + m_{\text{time}}$
        for each safe interval $i$ in $cfg$ do
            if $\text{startTime}(i) > end_t$ or $\text{endTime}(i) < start_t$ then
                continue
            $t \leftarrow$ earliest arrival time at $cfg$ during interval $i$ with no collisions
            if $t$ does not exist then
                continue
            $s' \leftarrow$ state of configuration $cfg$ with interval $i$ and time $t$
            insert $s'$ into successors
    return successors
```

4.2 SIPPS

SIPPS builds upon SIPP by focusing on soft collisions among agents. Although SIPPS permits some collisions in its initial planning phase, potentially leading to an infeasible plan, this approach is strategically chosen. It assumes that subsequent replanning sessions will more readily resolve these conflicts, and iteratively simplify the path towards feasible solutions.

With this in mind, safe intervals are defined a bit differently for SIPPS:

Definition 4.3 (Safe Interval)**.** A **safe interval** for a vertex in a graph is a continuous period during which the vertex remains *unobstructed by any hard obstacles*. It is represented as a tuple $[a, b)$ where a and b denote the start and end of the interval, respectively. Within this interval, the vertex may be intermittently affected by soft obstacles that do not entirely block passage but may influence path optimality. For each interval, we hold whether it has soft-collision or not.

SIPPS integrates the concept of safe intervals to navigate around both permanent and soft obstructions, calculating paths that either avoid or minimize interactions with these obstacles.

4.2.1 SIPPS Graph Node

Definition 4.4 (SIPPS Node). A **SIPPS node** n within the search tree of the SIPPS algorithm encompasses five primary elements:

- A vertex $n.v$, representing the current position within the graph.
- A safe interval $[n.lb, n.ub)$ where $n.lb$ is also known as the earliest arrival time, and this interval denotes the time during which the vertex $n.v$ is navigable without interference from hard obstacles.
- An index $n.id$ that correlates the node to a specific safe interval within the safe interval table $T[n.v]$, indicating that the node's safe interval is a subset of the id-th interval listed for vertex $n.v$.
- A Boolean flag $n.is\_goal$, which is set to false by default and indicates whether the node represents a goal state.
- $n.c_v \;=\; 1$ - if the safe interval includes soft obstacles and 0 otherwise
- $c_e \;=\; 1$ - if the edge leading to n is collision with agent in O^s and 0 otherwise.

The computational utility of a SIPPS node is derived from its f-value, which is the sum of its g-value and h-value:

- The g-value is defined as $g(n) \;=\; n.lb$, positioning the node based on the earliest feasible arrival time.
- The h-value is a heuristics estimate, calculated as a lower bound on the minimum travel time from $n.v$ to the goal vertex g.

Moreover, each SIPPS node maintains a c-value, which quantifies the underestimated number of soft collisions encountered along the path from the root node to node n. This value is computed as:

$$c(n) \;=\; c(n') + c_v + c_e,$$

where n' is the parent node.

The underestimation comes from the c_v, where we do not count how many soft-collisions occurred, but instead whether it happened within that interval.

The edge collisions (c_e) when the agents would "swap places" meaning that one agent would be leaving the first cell and entering the second cell, while the second agent would go in reverse order at the same time.

Moreover, we need this c-value for the priority queue, where nodes with the lowest c-value are selected first. This ensures that paths with fewer soft collisions are prioritized during the search process.

4.2.2 Main Algorithm of SIPPS

The Algorithm 4 begins by initializing the OPEN, CLOSED lists, the safe interval table T, the start state s_{start} is initialized with the first safe interval from the table T, and its cost $g(s_{\text{start}})$ is set to zero. The maximum time $T_{\max}$ is determined based on the presence of any hard obstacles at the goal. If such obstacles exist, $T_{\max}$ is set to one more than the maximum time associated with these obstacles (lines 1-8).

In each iteration of the main loop, the node s with the smallest f-value is removed from the OPEN list (line 12). If s is at the goal vertex g and its low interval value is at least $T_{\max}$ (line 13), the algorithm calculates the number of future soft collisions at the goal (line 14). If no future soft collisions are detected, the path is extracted and returned (lines 15-16). Otherwise, a copy of s and its cost is updated with the future collision count, this is then inserted into the OPEN list (lines 17-21). This new state is then inserted into the OPEN list (line 21).

The *expandSuccessors* function is called to generate potential successor nodes (line 23). Finally, the current node is moved to the CLOSED list to prevent re-exploration (line 24).

Node expansion in SIPPS

The *expandSuccessors* function, shown in Algorithm 5, is responsible for generating potential successor nodes for the given node n. The function begins by iterating over each possible edge from the current node's vertex $n.v$ to a neighboring vertex v, identified by id (line 2). For each safe interval (lb, ub) associated with vertex v in the safe interval table $T[v][id]$ (line 3), the function calculates the earliest arrival time *low* at v within the interval

Algorithm 4 SIPPS

```
1: OPEN $\leftarrow \emptyset$
2: CLOSED $\leftarrow \emptyset$
3: $T \leftarrow$ buildSafeIntervalTable$(V, O^h, O^s)$
4: $s_{\text{start}} \leftarrow$ Node$(s, T[s][1], 1, \text{false})$
5: $g(s_{\text{start}}) \leftarrow 0$
6: $T_{\max} \leftarrow 0$
7: **if** $\exists t : (g, t) \in O^h$ **then**
8:     $T_{\max} \leftarrow \max\{t|(g, t) \in O^h\} + 1$
9: **insert** $s_{\text{start}}$ into OPEN with $f(s_{\text{start}}) = h(\text{start})$
10: **while** OPEN is not empty **do**
11:     $s \leftarrow$ OPEN.pop()
12:     **if** $s.v = g \wedge s.lb \geq T_{\max}$ **then**
13:         $c_{future} \leftarrow |\{(g, t) \in O^s|t > s.lb\}|$
14:         **if** $c_{future} = 0$ **then**
15:             **return** extractPath$(s)$
16:         $s' \leftarrow$ a copy of $s$
17:         $c(s') \leftarrow c(s) + c_{future}$
18:         $f(s') \leftarrow g(s') + h(s')$
19:         **insert** $s'$ into OPEN with $f(s')$
20:     $successors \leftarrow$ expandSuccessors$(s)$
21:     CLOSED.insert(n)
22: **return** "No Solution"
```

$[lb, ub)$, ensuring no collisions with hard obstacles O^h, if not found continue with next interval (lines 4-6).

Next, the function computes low', the earliest arrival time at v within the interval $[lb, ub)$ without colliding with both hard and soft obstacles $O^h \cup O^s$ (line 7). If low' exists and it is different from low, the function creates two new nodes: n_1 for the interval $[low, low')$ and n_2 for the interval $[low', ub)$ (lines 8-12). If low' does not exist or they are the same with low, a single new node n_3 is created for the entire interval $[low, ub)$ (line 14-15).

The function continues this process for all neighboring vertices and their associated safe intervals, ensuring that all valid successor nodes are generated and inserted into the successor list.

Algorithm 5 expandSuccessors (in original paper EXPANDNODE)

```
function EXPANDSUCCESSORS($n$)
    for each $(v, id)$ such that $(n.v, v) \in E$ do
        for each $[lb, ub)$ in $T[v][id]$ do
            $low \leftarrow t_{\text{earliest}}$ at $v$ within $[lb, ub)$ without colliding $O^h$;
            if $low$ does not exist then
                continue;
            $low' \leftarrow t_{\text{earliest}}$ at $v$ within $[lb, ub)$ without colliding $O$;
            if $low'$ exists $\wedge low' > low$ then
                $n_1 \leftarrow \text{Node}(v, [low, low'), id, \text{false})$;
                $n_2 \leftarrow \text{Node}(v, [low', ub), id, \text{false})$;
                insertNode($n_1$);
                insertNode($n_2$);
            else
                $n_3 \leftarrow \text{Node}(v, [low, ub), id, \text{false})$;
                insertNode($n_3$);
```

Node Insert

Nodes within the SIPPS framework are identified and compared based on their position, safe interval, and goal status:

Definition 4.5 (Node Identity). Two nodes n_1 and n_2 are said to have the same identity, denoted as $n_1 \sim n_2$, if and only if $n_1.v = n_2.v$, $n_1.id = n_2.id$, and $n_1.is\_goal = n_2.is\_goal$.

Definition 4.6 (Node Dominance). We denote node similarity, $n_1 \sim n_2$, if following conditions are true: $n_1.v = n_2.v, n_1.id = n_2.id, and n_1.is_goal = n_2.is_goal$ meaning that they are at the same position, of the same interval, and they both are or aren't goals.

A node n_1 weakly dominates another node n_2, denoted as $n_1 \succeq n_2$, if $n_1 \sim n_2$, the interval $[n_1.lb, n_1.ub)$ encompasses $[n_2.lb, n_2.ub)$, and $c(n_1) \leq c(n_2)$. Dominance implies that node n_1 can replace n_2 without loss of pathfinding completeness or correctness.

In practice, dominance is used to avoid redundant path evaluations in the path search:

- When a new node n is considered for insertion into the *OPEN* list, all nodes in *OPEN* and the *CLOSED* list that share the same identity with n are examined.

- If an existing node q dominates n, then n is not added to *OPEN* since its path is already represented by q.

- Conversely, if n dominates q, then q is removed from *OPEN* and *CLOSED*, as n provides a more optimal or equally optimal path with a shorter or same-length interval.

- If n and q have overlapping intervals but neither dominates the other, the search interval of the node with the higher starting point is adjusted to avoid overlap, thus preventing duplicate efforts and reducing computational overhead.

The *insertNode* function, shown in Algorithm 6, calculates the g, h, f, and c-values for the new node n (line 1). It then identifies all nodes in the *OPEN* and *CLOSED* lists that are similar to n (i.e., nodes that share the same position, interval, and goal status) (line 2). For each similar node q, the function checks if q dominates n or vice versa (lines 3-12). If q dominates n, the new node n is not added to the *OPEN* list, avoiding redundancy (lines 4-5). Conversely, if n dominates q, the existing node q is removed from both the *OPEN* and *CLOSED* lists, as n offers a more optimal or equally optimal path (lines 6-7). If n and q have overlapping intervals but neither dominates the other, their intervals are adjusted to eliminate overlap (lines 8-12). This process ensures that only the most promising and non-redundant nodes are retained, thereby optimizing the pathfinding efficiency. Finally, the adjusted new node n, is inserted into the *OPEN* list.

Algorithm 6 insertNode

```
1:  Compute $g$, $h$, $f$, and $c$-values of $n$;
2:  $\mathcal{N} \leftarrow \{q \in OPEN \cup CLOSED \mid q \sim n\}$;        ▷ Nodes identical to $n$
3:  for each $q \in \mathcal{N}$ do
4:      if $q.lb \leq n.lb$ & $c(q) \leq c(n)$ then
5:          return;                                    ▷ No need to generate $n$
6:      else if $n.lb \leq q.lb$ & $c(n) \leq c(q)$ then
7:          delete $q$ from $OPEN$ and $CLOSED$;                    ▷ Prune $q$
8:      else if $n.lb < q.ub$ & $q.lb < n.ub$ then
9:          if $n.lb < q.lb$ then
10:             $q.ub \leftarrow n.lb$;
11:         else
12:             $n.ub \leftarrow q.lb$;
13: insert $n$ into $OPEN$;
```

4.2.3 Heuristics used in SIPPS

Typically, most MAPF algorithms utilize the distance $d(n.v, g)$ — the length of the shortest path from a node $n.v$ to the goal g — as the heuristic value h for a node n during path planning. This distance is calculated during a preprocessing step and serves as an estimate to guide the search towards the goal efficiently.

Adaptation in SIPPS. However, in environments characterized by frequent and unpredictable changes, the standard heuristic requires adjustments:

- **Influence of Hard Obstacles**: The presence of hard obstacles can significantly alter the feasible travel time from $n.v$ to g, denoted as T, making it potentially much larger than $d(s, g)$.
- **Consideration of Soft Obstacles**: Similarly, $T' = \max\{t|(g, t) \in O^h \cup O^s\} + 1$, which represents a lower bound on the path's duration when there are no collisions, may also exceed $d(s, g)$. This factor needs to be incorporated into the heuristic calculation for non-goal nodes to ensure that paths avoiding soft collisions are favored where possible.

Given these factors, the heuristics h for a non-goal node n is computed as:

$$h(n) = \max\{d(n.v, g), T' - g(n)\}, \quad \text{if } c(n) = 0$$

$$h(n) = \max\{d(n.v, g), T - g(n)\}, \quad \text{if } c(n) \geq 1$$

where $c(n)$ represents the estimated number of soft collisions.

SIPPS is introduced in [LCH$^+$22] together with adaptive LNS (ALNS) which then helps to plan all the agents, this is described in Section 4.4

4.3 SIPP-IP

The main disadvantage of the standard (SIPP) in addressing complex scenarios is its inability to carry forward information about potential waiting actions from one node to the next. For example, when an agent arrives at a search node labeled as $(B, \text{vel} = 1), [0, \infty)$, the standard SIPP fails to recognize that the agent might wait at a previous node and then move anytime during that node's safe period. This is not an issue in simpler cases where agents can pause at any point, but it causes problems in more dynamic situations. To resolve this, an adaptation called Safe Interval Path Planning With Interval Projection (SIPP-IP) has been created [AY23]. In SIPP-IP, instead of using just the safe interval to identify a search node, we use a 'waiting interval'. This interval, which is part of the vertex's safe interval, contains all possible wait-and-move actions from the node's predecessor. This waiting interval is then extended to successors as nodes are expanded, ensuring that information about potential actions is consistently propagated throughout the search tree.

Mechanism of Interval Projection. The essence of interval projection in SIPP-IP lies in its ability to carry forward the wait-and-move capabilities of an agent through the pathfinding process. This is accomplished by projecting time intervals along the edges of the graph, ensuring that the temporal aspects of agent movement are maintained from one node to the next. Specifically, the projection operation takes a node $n = (v, [t_l, t_u))$, where the interval $[t_l, t_u)$ lies within a safe interval of v, and an edge $e = (v, v')$. The outcome is a set of time intervals $TI = \{t_i = [lb_i, ub_i)\}$, where:

- Each t_i is contained within a safe interval of the destination vertex v', ensuring that all resultant intervals are viable for the agent to enter.
- The intervals do not overlap, maintaining clear, distinct paths through the graph.
- For any valid transition along edge e that starts at any timestep within $[t_l, t_u)$, the ending timestep of this transition falls within one of the projected intervals, provided the transition does not lead to a collision.

Operational Benefits. The introduction of interval projection in SIPP-IP ensures that information about possible wait-and-move actions is not lost as the search progresses from the start node towards the goal. This approach not only enhances the completeness of the search—addressing scenarios that standard SIPP might fail to resolve—but also improves the algorithm's efficiency by reducing redundant calculations and maintaining a more compact search tree. The interval projection operation is especially pivotal in scenarios where agents have limited ability to halt, necessitating a careful consideration of movement dynamics over time. Section 4.5.1 provides a more detailed description with examples as well.

4.3.1 Algorithm

The main loop of the SIPP-IP (Algorithm 7) is more or less identical to that of SIPP. The only difference is that we set the starting node interval to be matching the safe interval (lines 3-6) when the velocity at the starting node is 0, which for our purpose it is every time. For the g-value, we use the lower bound of the node (line 14). The main differences are in how we acquire the neighbors in the function `getSuccessors`.

Algorithm 7 SIPP-IP algorithm

```
1:  function FINDPATH($v_{\text{start}}, t_{\text{start}}, v_{\text{goal}}, G(V, E), SI$)
2:      OPEN ← ∅, CLOSED ← ∅
3:      $ti \leftarrow [t_{\text{start}}, t_{\text{start}}]$
4:      if $v_{\text{start}}.\text{vel} = 0$ then
5:          $ti.t_{\text{ub}} \leftarrow$ upper bound of $SI(v_{\text{start}}, ti)$
6:      insert $s_{\text{start}}$ into OPEN with $f(s_{\text{start}}) = h(\text{start})$
7:      while OPEN not empty do
8:          $n \leftarrow$ remove node with the smallest $f$-value from OPEN
9:          if $n.v = v_{\text{goal}}$ then
10:             return $\pi \leftarrow$ ReconstructPath
11:         $succ \leftarrow \text{getSuccessors}(n)$
12:         for each $n'$ in $succ$ do
13:             if $n'$ was not visited before then
14:                 $f(n') \leftarrow n'.t_{\text{lb}} + h(n'.v)$
15:                 Add $n'$ to OPEN
16:         CLOSED.insert(n)
17:     return ∅
```

The *getSuccessors* function, shown in Algorithm 8, generates successor nodes for a given node n. It starts by initializing an empty set $SUCC$ to store the successors (line 2). For each edge $e = (n.v, v')$ in the set of available motions $M(n)$ (line 3), the function projects intervals from the current node n to the neighboring node v', using the safe interval table SI (line 4). If the velocity of v' is zero (line 5), meaning the agent can wait, the upper bound of each interval in v' is adjusted based on matching safe interval in SI (lines 6-8). Finally, the function inserts each valid interval t_i for v' into the successor set $SUCC$ (lines 9-10). The function returns the set of successors $SUCC$ (line 11).

The Algorithm 9 describes function *projectIntervals*, which projects safe intervals from the current node n through the edge $e = (n.v, v')$ using the safe interval table SI. It begins by initializing *time_ints* with the current interval and setting the time t to zero (lines 2-3). For each cell in e's cells, it initializes *new_ints* as an empty set and calculates the time difference Δ

Algorithm 8 SIPP-IP getSuccessors

```
1: function GETSUCCESSORS($n, SI$)
2:     $SUCC \leftarrow \emptyset$
3:     for each $e = (n.v, v')$ in $M(n)$ do
4:         $intrvls \leftarrow$ projectIntervals($n, e, SI$)
5:         if $v'$.vel $= 0$ then
6:             for each $t_i$ in $intrvls$ do
7:                 $t_i.t_{\text{ub}} \leftarrow$ upper bound of $SI(v', t_i)$
8:         for each $t_i$ in $intrvls$ do
9:             Insert $(v', t_i)$ to $SUCC$
10:    return $SUCC$
```

between the lower bound of the cell and the current time t, time t is updated to the cell's lower bound (lines 4-7).

Afterward, for each interval *ti* in *time_ints*, the function iterates over each safe interval *si* and computes the earliest and latest times t_{earliest} and t_{latest} for the interval to be valid (lines 8-11). If t_{earliest} is less than or equal to t_{latest}, the interval $[t_{\text{earliest}}, t_{\text{latest}}]$ is added to *new_ints* (lines 12-13). This way we project the current interval to the next cell in the motion, and at each cell we update *time_ints* to *new_ints* (line 14).

After iterating through all cells, the function initializes *succ* as an empty set (line 15). For each interval *ti* in *time_ints*, it retrieves the last cell for that motion, calculates the time adjustment Δ based on the cost of e and the last cell's lower bound (lines 16-18), and updates $ti.t_l$ and $ti.t_u$ accordingly (lines 19-20). The interval (v', ti) is then added to successors (line 22). In the end, yielding a set of available projected intervals for the given motion.

4.3.2 Motion primitives

The motion primitives, described in [PK11], provide a structured way to explore the state space by defining permissible transitions between states that account for the kinematic and dynamic constraints of the agent. Unlike traditional motion primitives, which may be densely packed in the state space and lead to redundancy, state lattice primitives ensure that each path leading to a region in the state space corresponds to a unique state value. This uniqueness is critical for optimizing the search process and improving the computational efficiency of planning algorithms.

Algorithm 9 SIPP-IP

```
1: function ProjectIntervals($n, e = (n.v, v'), SI$)
2:     $time\_ints \leftarrow \{[n.t_l, n.t_u]\}$
3:     $t \leftarrow 0$
4:     for each cell in $e$.cells consecutively do
5:         $new\_ints \leftarrow \emptyset$
6:         $\Delta \leftarrow lb^e_{\text{cell}} - t$
7:         $t \leftarrow lb^e_{\text{cell}}$
8:         for each $ti$ in $time\_ints$ do
9:             for each $si$ in SafeIntervals(cell) do
10:                $t_{\text{earliest}} \leftarrow \max(ti.t_l + \Delta, lb(si))$
11:                $t_{\text{latest}} \leftarrow \min(ti.t_u + \Delta, ub(si) - (ub_{\text{cell}} - lb^e_{\text{cell}}))$
12:                if $t_{\text{earliest}} \leq t_{\text{latest}}$ then
13:                    Insert $[t_{\text{earliest}}, t_{\text{latest}}]$ into $new\_ints$
14:        $time\_ints \leftarrow new\_ints$
15:    $succ \leftarrow \emptyset$
16:    for each $ti$ in $time\_ints$ do
17:        $last\_cell \leftarrow$ the last cell in $e$
18:        $\Delta \leftarrow \text{cost}(e) - lb^e_{\text{last cell}}$
19:        $ti.t_l \leftarrow ti.t_l + \Delta$
20:        $ti.t_u \leftarrow ti.t_u + \Delta$
21:        Insert $(v', ti)$ into $succ$
22:    return $succ$
```

Examples of primitives can be found in Figure 3.3. In SIPP-IP, these motion primitives are used to get the neighbors, which correspond to the last cell, while also ensuring that no collision happens during the transition. Because of using these primitives, we essentially allow only certain movements, that should be good enough for the agent's movements.

For the purpose of this thesis, we consider 5 types of motion primitives:

- **Turning**: When an agent is at a location with zero speed, it is allowed to turn. This motion primitive enables the agent to change its direction while remaining stationary.
- **Speed-Up**: This motion primitive describes the process of accelerating from a standing position (velocity = 0) to maximum speed. The duration of this action depends on the agent's maximum acceleration capability and typically requires several timesteps to achieve full speed.
- **Slow-Down**: This primitive involves decelerating from maximum speed to a complete stop. Similar to speeding up, the time required for this

action is governed by the agent's maximum deceleration rate and spans multiple timesteps.

- **Continue at Max Speed**: Once the agent reaches maximum velocity, it can maintain this speed and move to the next cell. This motion primitive ensures that the agent travels in a straight line at a constant speed.
- **Move n Cells**: This set of motion primitives allows the agent to move from a stopped position to another stopped position over a distance of n cells. It is designed to fill gaps of the previous motions. During this motion, the agent accelerates and decelerates without reaching maximum speed, but reaching the final cell as fast as possible.

It is important to note that except for the turning primitive, all actions involve straight-line motion. Motion primitives discretize the kinematic constraints into the most desirable motions. While there are many scenarios where we would not be able to find the path, such as a corridor requiring us to go at a constant speed of half the maximum speed, the situations are considered to be rare.

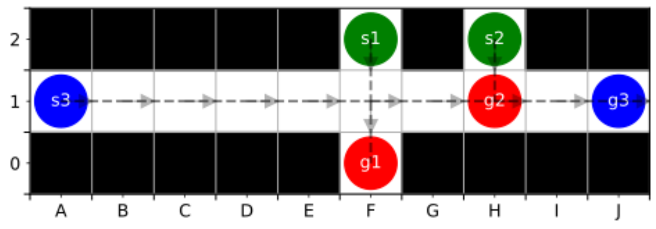


Figure 4.2: Kinematic constraints example similar to the one to the SIPP-IP paper

4.4 Optimizing the Solution

Assuming that we can find such a solution for agents using Prioritized planning, how do we make it better? A common technique is to improve specific parts of the solution, known as neighborhoods.

Large Neighborhood Search (LNS), originally proposed in [Sha98], is a well-known local search technique for enhancing solution quality in combinatorial optimization. From a given solution, it destroys part of it (a neighborhood) and repairs the solution. If the repaired solution is better, it replaces the old one. This process is repeated until a stopping criterion is met. In MAPF-LNS2, LNS iteratively selects a subset of agents and replans their paths [PR19].

Adaptive LNS (ALNS):. MAPF-LNS2 employs ALNS, a variant that uses multiple neighborhood selection methods, adapting based on their relative success. ALNS records the success of each neighborhood selection method and uses those metrics to prioritize future selections. Weights are adjusted iteratively to reflect the most effective methods.

Each neighborhood selection method i has an associated weight w_i representing its relative success in reducing the number of colliding pairs (CP). Initially, all w_i values are set to 1. At each iteration, a method i is selected with a probability $\frac{w_i}{\sum_j w_j}$ to generate a neighborhood and replan the paths. Once the paths are replanned, the weight w_i is updated as follows:

$$w_i \leftarrow \gamma \cdot \max\left(0, c^- - c^+\right) + (1-\gamma) \cdot w_i$$

where:

- c^- is the CP count before replanning.
- c^+ is the CP count after replanning.
- γ is a reaction factor, empirically set to 0.1, controlling the weight adjustment speed.

Neighborhood Selection. is critical for the success of LNS. MAPF-LNS2 employs three methods: Collision-Based Neighborhoods, Failure-Based Neighborhoods, and Random Neighborhoods. Neighborhoods are also parametrized by neighborhood size, which usually depends on how costly the replanning of agents is, but typically smaller neighborhoods (not exceeding 10) are chosen.

4.4.1 Collision graph

To be able to select collision-based neighborhoods we need to be able to represent the dependencies between the agents.

Definition 4.7. Let the current plan of agents is represented by $P = \{\pi_1, \ldots, \pi_n\}$, A as the set of agents, the neighborhood as A_s, and the neighborhood size $|A_s| = N$.

Collision graph $G_c = (V_c, E_c)$ is a graph, where V_c corresponds to index of agent from A, and $E_c = (i, j)|$ if $c(\pi_i, \pi_j) > 0$. The number of edges for a vertex is denoted as $deg(i)$.

Collision-Based Method. To select a neighborhood using collision-based methods, the process begins by selecting a random vertex v from the collision graph where $deg(v) > 0$ (indicating that the agent associated with v is involved in a collision). The algorithm identifies the largest connected component $G'_c \subseteq G_c$ containing the selected node v. From this point, there are two possible cases:

1. **Case 1:** If the connected component $|V'_c|$ contains less than or equal to N vertices, all agents associated with vertices in V'_c are placed into the neighborhood set A_s. Additional agents that might collide with any of the agents already in A_s are also added until $|A_s| = N$. At each iteration, a random agent from A_s performs a random walk starting from a random point on its path and stops when it collides with another agent, which is then added to A_s.
2. **Case 2:** Otherwise, if $|V'_c|$ contains more than N vertices, a random walk is performed on G'_c, starting from vertex v. The algorithm selects N vertices from G'_c and adds the corresponding agents to the neighborhood set A_s.

4.4.2 Failure-Based Method

The failure-based neighborhood method examines why collision-free paths could not be found for some agents in previous LNS iterations. In the Prioritized Planning approach, finding a path for an agent a_i that avoids conflicts with existing paths of agents. There are two primary scenarios where failures occur:

- **Scenario A:** Agent a_i is blocked by other agents who are already situated at their target vertices surrounding a_i. This makes all paths from s_i to g_i inaccessible due to the presence of these target obstacles.
- **Scenario B:** Agent a_i is "run over" by other agents' paths at (or near) s_i in the early time steps, leaving no way forward.

The failure-based method begins by selecting agent a_i from the set A proportionally to $deg(i)$ (number of agents a_i collides with) and adding it to As. It then gathers two sets of agents:

- As: Agents whose paths visit s_i.
- Ag: Agents whose target vertices are on a path from s_i to g_i.

The selection process follows these steps:

1. If $|As \cup Ag| = 0$, terminate and return As, as a_i can safely wait at s_i until other agents reach their targets, then proceed to g_i via the path p.
2. If $|As \cup Ag| < N - 1$, add all agents from As and Ag to As. Continue to add agents whose targets are visited by other agents in As until $|As| = N$.
3. Otherwise, follow this rule to add $N - 1$ agents to As:
   a. If $|As| = 0$, add $N - 1$ random agents from Ag to As.
   b. If $|Ag| \geq N - 1$, add the agent from As who first visits s_i, and then add $N - 2$ random agents from Ag.
   c. Otherwise, add all agents in Ag and fill the remaining slots in As from agents in ascending order of timesteps visiting s_i.

This selection rule prioritizes agents from Ag slightly over those from As since Scenario A is empirically more common than Scenario B [ČNKS15].

4.4.3 Random Neighborhoods

In the random neighborhood approach, N agents are selected randomly, with each agent a_i having a probability proportional to $deg(i) + 1$. The additional increment ensures that agents with no collisions also have a chance to be selected. This method provides a diverse set of neighborhoods to explore, allowing the algorithm to adaptively identify paths that minimize collisions.

4.5 Solution Idea - SIPPS-IP

Now that we have both SIPP-IP and SIPPS defined, we can proceed to our proposed solution: Safe Interval Path Planning with Soft Constraints and Interval Projection (SIPPS-IP). To find some solutions even if early collisions occur, we begin with SIPPS and incorporate the interval projection principles of SIPP-IP.

4.5.1 Differences between SIPPS and SIPP-IP

Despite their shared precursor algorithm, the main parts of the algorithms have a few differences, that do not make combining these algorithms easy. The pseudocode for SIPP-IP - Algorithm 7, and SIPPS - Algorithm 4.

Initialization

Initialization involves constructing the safe interval table and creating the root node. In SIPPS, the intervals are constructed differently due to the varying effects of soft and hard obstacles. As a result, the safe intervals are tailored specifically for SIPPS. For the root node, both algorithms establish the first safe interval aligning with the starting timestep. Although this interval might involve a collision, it aligns with the nature of the SIPPS algorithm, requiring no major adjustments. The value `T_max` remains consistent with SIPPS, following the nature of MAPF, where agents wait until all others reach their goals. This ensures feasible plans and is not directly relevant to SIPP-IP, which is focused on single-path planning.

Main Loop

In the main loop, agents must reach their goal after other agents have passed, maintaining the behavior set in the initialization stage and `T_max` management.

The goal condition differs slightly, in SIPPS, the future soft collisions are accounted for, and the node is reinserted into the open priority queue with the additional future collisions, meaning that if no better path exists, with fewer collisions, then this is the final goal node.

Neighbor expansion

The last part is handling the neighbors, both algorithms get their neighbors and then they expand them in terms of adding to the open list. This section is where the two algorithms differ the most.

The SIPP-IP works with the motion primitives and projecting the intervals. On the other hand, SIPPS works with weak and strong dominance of nodes as well as safe intervals with collisions. To be able to merge these two algorithms we need to be able to do these projections considering both the kinematic constraints, while also allowing to create nodes that have collisions with other agents.

Node Dominance

Both weak and strong dominance in SIPPS involves three key pieces of information: collisions, position, and interval. Although these elements are present here as well, the configuration now includes speed and orientation. For instance, consider two nodes that are similar but differ in speed. Their neighbors will vary significantly, so one node cannot eliminate the other. The same applies to orientation.

To apply SIPPS dominance, we must define the similarity between two nodes, denoted as $n_1 \sim n_2$. In our terms:

$$\begin{aligned} n_1 \sim n_2 \iff & (n_1.o = n_2.o \land n_1.v = n_2.v \land n_1.s = n_2.s) \\ & \land n_1.id = n_2.id \\ & \land n_1.is\_goal = n_2.is\_goal \end{aligned} \tag{4.2}$$

where we have simply added $n.o$ (orientation) and $n.s$ (speed) to represent the agent's state.

Interval Projection - applyPrimitive

Since we use motion primitives to represent agent movements, we use them to identify the neighboring nodes. Motion primitives are precomputed motions accessible using two key attributes: orientation and speed. Applying a motion primitive involves iterating through cells along the given direction and generating intervals that avoid collisions while leading to a final position. This process may result in multiple safe intervals, which account for obstacles along the primitive's path.

In SIPP-IP, only the final node was of interest, whereas here we also expand the intermediate positions. These positions serve as temporary nodes that store values and are automatically closed. They are not expandable because their velocities are not within the usual <0,1> range.

To illustrate this, consider a simplified example where an agent moves across four cells. Ignoring specific acceleration characteristics for now, Figure 4.3 demonstrates how intervals would appear if such a primitive were used.

In the diagram, we see different types of intervals. The blue intervals represent safe periods where no collisions occur within the scope of the current primitive. As the agent traverses through cells, extra time is required to transition between them. The red intervals, on the other hand, are the complement of the total safe interval and describe periods when an agent arriving at a cell would be collision-free but unable to reach the next cell in time. The green intervals represent valid periods in which an agent can perform the primitive, reaching the final cell successfully. Black boxes indicate dynamic obstacles.

The orange lines represent the projections of the green interval across the cells. The Algorithm 12 creates these intervals, where the forward pass generates intervals at the final location, and the backward pass projects corresponding intervals for the other cells.

Now, if we replace the dynamic obstacle (occupying the [9-10) interval) with an agent moving through cell 2 during the interval [5,10) (orange dots), as shown in Figure 4.4, SIPP-IP would treat this as an obstacle, thus not producing the first projected interval at all. In our approach, however, two intervals will still exist: the first with one collision and the second collision-free.

Notice that in this scenario, where cell 4 has a safe interval of [0-20], the

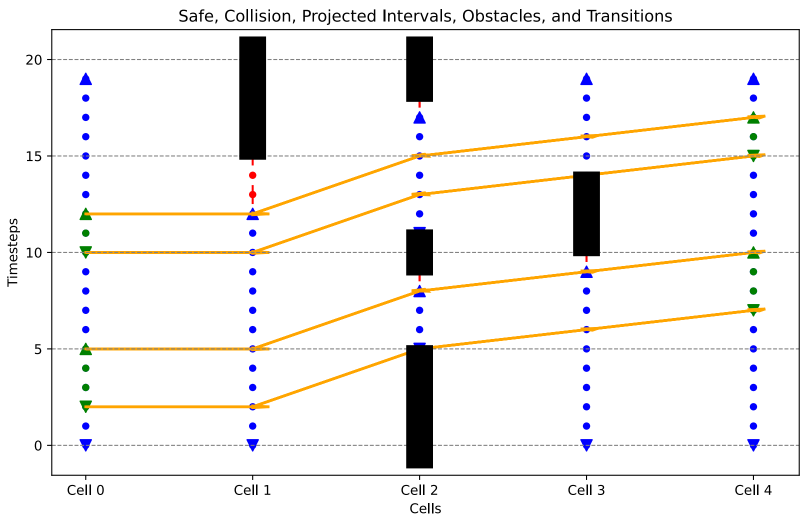


Figure 4.3: Projecting intervals

node could be dominated. However, since the first interval contains a collision while the second does not, they are treated separately. Conversely, if the intervals were reversed (the first interval being collision-free and the second having a collision), node n_1 (first interval) would weakly dominate node n_2 (second interval), because $n_1 \sim n_2$ and $n_1.lb \leq n_2.lb \wedge c(n_1) \leq c(n_2)$. If for the last cell, we had an obstacle in between those two intervals, the dominance would not occur once again, as $n_1 \not\sim n_2$, because of the same interval id condition[3].

SIPPS-IP Algorithm

In the main part of Algorithm 10, not much is changed from SIPPS algorithm. The initialization are combined trivially (lines 1-9), and goal handling is according to the conditions of SIPPS.

The *expandSuccessors* function, shown in Algorithm 11, generates successor nodes for a given node n. It starts by iterating over each potential successor (v, id) of $n.v$ (line 2), which are generated by the projecting intervals. Afterward, we handle the nodes in the same manner as the SIPPS, creating both low and low', which then affects, what nodes will be added to the OPEN list.

[3]For both of these examples, the final intervals would be shifted by a sweeping time, as we need to finish the motion (be at the middle of the cell) to start a new motion.

Algorithm 10 SIPPS-IP

```
1:  OPEN ← $\emptyset$
2:  CLOSED ← $\emptyset$
3:  $T$ ← buildSafeIntervalTable($V, O^h, O^s$)
4:  $s_{\text{start}}$ ← Node($s, T[s][1], 1, \text{false}$)
5:  $g(s_{\text{start}}) \leftarrow 0$
6:  $T_{\max} \leftarrow 0$
7:  if $\exists t : (g, t) \in O^h$ then
8:      $T_{\max} \leftarrow \max\{t | (g, t) \in O^h\} + 1$
9:  insert $s_{\text{start}}$ into OPEN with $f(s_{\text{start}}) = h(\text{start})$
10: while OPEN is not empty do
11:     $s$ ← OPEN.pop()
12:     if $s.v = g \wedge s.lb \geq T_{\max}$ then
13:         $c_{future} \leftarrow |\{(g, t) \in O^s | t > s.lb\}|$
14:         if $c_{future} = 0$ then
15:             return extractPath($s$)
16:         $s'$ ← a copy of $s$
17:         $c(s') \leftarrow c(s) + c_{future}$
18:         $f(s') \leftarrow g(s') + h(s')$
19:         insert $s'$ into OPEN with $f(s')$
20:     $successors$ ← expandSuccessors($s$)
21:     CLOSED.insert(n)
22: return "No Solution"
```

Algorithm 11 expandSuccessors

```
1:  function EXPANDSUCCESSORS($n$)
2:      for each $(v, id)$ such that $(n.v, v) \in getSuccessors(v, SI)$ do
3:          for each $[lb, ub)$ in $T[v][id]$ do
4:              $low \leftarrow t_{\text{earliest}}$ at $v$ within $[lb, ub)$
5:                          without colliding with $O^h$;
6:              if $low$ does not exist then
7:                  continue;
8:              $low' \leftarrow t_{\text{earliest}}$ at $v$ within $[lb, ub)$
9:                          without colliding with $O^h \cup O^s$;
10:             if $low'$ exists $\wedge low' > low$ then
11:                 $n_1$ ← Node($v, [low, low'), id, \text{false}$);
12:                 $n_2$ ← Node($v, [low', ub), id, \text{false}$);
13:                 insertNode($n_1$);
14:                 insertNode($n_2$);
15:             else
16:                 $n3$ ← Node($v, [low, ub), id, \text{false}$);
17:                 insertNode($n3$);
```

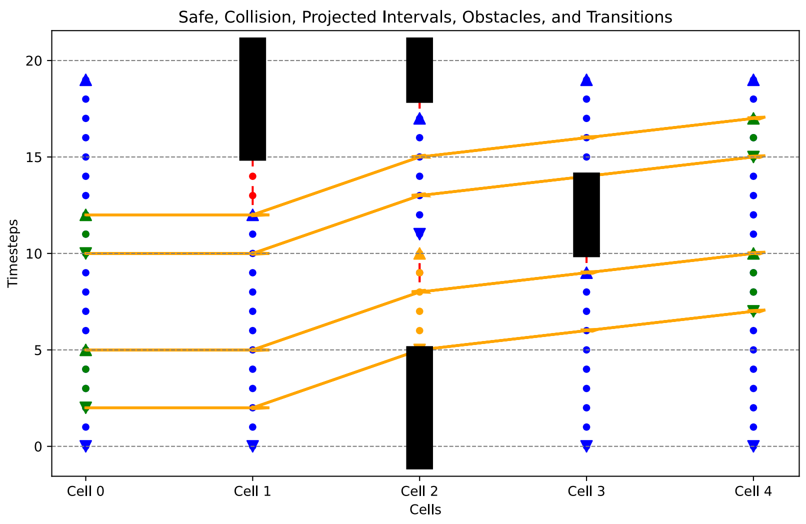


Figure 4.4: Projecting intervals with soft-collision

The *getSuccessors* function, shown in Algorithm 12, generates potential successor nodes for a given node n by considering all feasible primitive movements based on the current node's speed and orientation. The function initializes an empty set *neighbours* to store the successors (line 2). It then iterates over each primitive movement in $M(n.speed, n.orientation)$, using the *projectInterval* function to determine the valid intervals for each movement based on the safe intervals SI, the results are inserted into the *neighbours* set (lines 3-4). The function then returns the set of successors.

The *projectInterval* function, projects the safe intervals from the current node n through the primitive $p = (v, v')$ to determine valid successor states. It initializes an empty set *succ* to store the successors and performs a forward pass to compute the intervals *intrvls* using the *forwardPass* function (lines 2-3). For each interval *ti* obtained, we iterate over each move in primitive p, adjusting the intervals based on the time differences δ and the position shifts $move.d_{pos}$, these intermediate nodes are inserted into the CLOSED list (lines 4-9). Finally, the function adjusts the intervals for the last cell by sweeping time and inserts the valid successor states into *succ* (lines 10). The function returns the set of successors *succ*.

The *forwardPass* function, described in Algorithm 13, calculates the initial time intervals for the given node n moving through the primitive $p = (n.v, v')$. It starts with the initial interval $[n.t_{lb}, n.t_{ub}]$ and the corresponding safe interval collisions from $SI(n)$ (line 2). For each move in the primitive, the function updates the intervals by considering the time difference Δ and the earliest time t the move touches the new position (lines 5-8). It computes the

Algorithm 12 SIPPS-IP

```
1: function GETSUCCESSORS($n, SI$)
2:     $neighbours \leftarrow \emptyset$
3:     for each primitive in $M(n.speed, n.orientation)$ do
4:         neighbours.insert(projectInterval($n, primitive, SI$))
5:     return neighbours
1: function PROJECTINTERVAL($n, p = (v, v'), SI$)
2:     $succ \leftarrow \emptyset$
3:     $intrvls \leftarrow forwardPass(n, p, SI)$
4:     for $ti$ in $intrvls$ do
5:         for each move in p do
6:             $\delta \leftarrow move.touch - p_last.touch$
7:             $pos \leftarrow n.pos + move.d_{pos}$
8:             if move not last then
9:                 $CLOSED.insert([ti.lb + \delta, ti.ub + \delta], pos)$
10:        $succ.insert([ti.lb + p_{last}.swt, ti.ub + p_{last}.swt], v')$
11:    return $succ$
```

earliest and latest times for valid transitions (including intervals with soft collision) and adjusts the intervals accordingly (lines 10-12). If the interval is valid it is inserted to the new intervals (line 14). The function continues this process for all moves in the primitive p and iteratively updates the intervals (line 15). After the last cell we have all the projected intervals.

Reconstructing path

The paths we generate are similar to those in SIPP-IP, so reconstruction is done by iterating backward through the nodes. However, since SIPP-IP is designed for a single agent and doesn't require occupation intervals, we instead extract paths as intervals for each position. This involves iterating through the parents of the final node and aligning the times based on the lower bound and sweeping time.

The *reconstructPath* function, in Algorithm 14, reconstructs the path from the goal node to the start node by tracing back through each node's parent. The function starts by initializing an index and setting the current node n to the goal (lines 2-3). It then iterates through the nodes, counting the total number of distinct positions by checking if the current node's location differs from the previous location (lines 4-9). This total count is used to resize the path array to the correct length (line 10).

Algorithm 13 SIPPS-IP Forward

```
1:  function FORWARDPASS($n, p = (n.v, v'), SI$)
2:      $intrvls \leftarrow \{[n.t_{lb}, n.t_{ub}], SI(n).c\}$
3:      $t \leftarrow 0$
4:      for each move in p primitive do
5:          $new\_intrvls \leftarrow \emptyset$
6:          $\Delta \leftarrow move.touch - t$
7:          $t \leftarrow move.touch$
8:          $pos \leftarrow n.v + move.d_{pos}$
9:          for each $ti$ in $intrvls$ do
10:             for each $si$ in SI(pos) do
11:                 $t_{\text{earliest}} \leftarrow \max(ti.lb + \Delta, si.lb)$
12:                 $t_{\text{latest}} \leftarrow \min(ti.ub + \Delta, si.ub - move.swp)$
13:                 if $t_{\text{earliest}} \leq t_{\text{latest}}$ then
14:                     $new\_intrvls.insert([t_{\text{earliest}}, t_{\text{latest}}, si.c + ti.c])$
15:         $intrvls \leftarrow new\_intrvls$
16:     return $intrvls$
```

Next, the function iterates through the nodes again to fill the path array with the correct locations and their corresponding entry and wait times (lines 12-27). For each node, if the location is the same as the previous one, it accumulates the wait time (lines 14-17), however specific case when the entry and sweeping time do not align with the previous node needs to be adjusted for the wait action (lines 14-15). When the location changes, it updates the path array with the previous location and its associated time interval, then resets the entry time and wait time for the new location (lines 20-24). Finally, the last location and its time interval are updated in the path array, and the complete path is returned (lines 28-29).

4.5.2 Heuristics

As we follow the T_{max} usage and SIPP-IP is only using Manhattan distance, we take the approach defined in SIPPS. That is:

Given these factors, the heuristics h for a non-goal node n is computed as:

$$h(n) = \max\{d(n.v, g), T' - g(n)\}, \quad \text{if } c(n) = 0$$

$$h(n) = \max\{d(n.v, g), T - g(n)\}, \quad \text{if } c(n) \geq 1$$

where $c(n)$ represents the estimated number of soft collisions, T last timestep an hard obstacle occupies the position and $T' = \max\{t|(g,t) \in O^h \cup O^s\} + 1$

Algorithm 14 reconstructPath

```
procedure ReconstructPath(goal)
    index ← 0
    n ← goal
    while n do                                        ▷ total number positions
        if n.location $\neq$ location then
            index ← index + 1
            location ← n.location
        n ← n.parent
    resize(path, index)
    n ← goal
    wait_time, entry ← 0
    while n do
        if n.location == location then                          ▷ waiting
            if n.timestep + n.swt $\neq$ entry then
                wait_time ← wait_time + (entry - (n.timestep + n.swt))
            entry ← n.timestep
            wait_time ← wait_time + n.swt
        else                                          ▷ updating the path
            path[index] ← (location, (entry, entry + wait_time)
            index ← index - 1
            location ← n.location
            entry ← n.timestep
            wait_time ← n.swt
        n ← n.parent
    path[index].location ← (location, (entry, entry + wait_time)
    return path
```

4.6 Proposed Solutions

Now that we have all the SIPPS-IP defined and also underlying foundational concepts we can now introduce the solution proposed by this thesis.

4.6.1 Combination of LNS + SIPP-IP

The simplest solution involves combining SIPP-IP with LNS. However, due to SIPP-IP's inability to handle collisions, it cannot leverage the main strengths of LNS2. Therefore combining these two is one approach, however, this turned out to lead to a failed solution even for smaller examples.

4.6.2 Reserved intervals

We begin with a comparison of LNS + SIPP-IP with and without reserved intervals as they both use the same algorithm. The difference is that for one of them, we define a path at the starting position so that other agents try to avoid it when planning. The interval that is chosen for the reservation is for the agent to leave the start from any configuration and leave at any configuration, so the agent has to be able to turn and also be able to use the slowest move action, Therefore, the interval upper bound for interval I_{ub} can be expressed as:

$$I_{ub} = 2 \times T_{\text{turn}} + \max(T_{\text{sweep},0})$$

where:

- T_{turn} represents the time required to complete a turning maneuver by 45 degrees.
- $T_{\text{sweep},i}$ denotes the sweep time of the i-th cell for the i-th motion primitive.
- $\max(T_{\text{sweep},0})$ is the maximum first step sweep time across all motion primitives.

In our case these values are $T_{\text{turn}} = 10$ and $\max(T_{\text{sweep},0}) = 30$

Thus we propose another solution, using a reservation system for agents, ensuring that each agent reserves its starting position and can leave in any direction. The reservation is enforced by temporarily assigning a path where each agent remains at the starting cell for a specified period, and during path planning, the currently active agent's path is removed and substituted by the new one. Although this might seem like we are making the solution worse, in the later phase (destroy-repair), the solution is iteratively improving, and the initial higher cost decreases soon by optimizing these initial reservations. We will describe this solution as *LNS + SIPP-IP with reservation intervals* (SIPP-IP-ri)

4.6.3 Combination of LNS2 + SIPPS-IP

Now finally the idea we have been converging to. Since SIPPS-IP, like SIPPS, allows and records collisions, we can use adaptive LNS with SIPPS-IP and hopefully achieve promising results as well. This application is straightforward because the outer algorithm (ALNS) remains the same, with the exception of using a different pathfinding algorithm.

Part II

Practical Part

Chapter 5

Implementation

The implementation of our project involved combining and extending two existing frameworks for MAPF. These frameworks are the MAPF-LNS2 and SIPP-IP, both of which are implemented in C++.

The source code for MAPF-LNS2 is available on GitHub and provides a robust foundation for local search techniques in path planning together with the implementation of several path planning algorithms. This can be accessed at MAPF-LNS2 Repository [Li21]. Similarly, the SIPP-IP algorithm, designed for safe interval path planning for individual agents, is hosted on GitHub and can be found at SIPP-IP Repository. This algorithm provides the necessary tools for managing dynamic obstacles and safe intervals [Yak21].

As the first repository has both parts (neighborhoods as well as path searching algorithms), we decided to use it as a starting point. The main additions that had to be implemented were algorithms SIPP-IP and SIPPS-IP. Our C++ implementation can be found in the attached files.

5.1 Implementation Details

The MAPF-LNS2 code base uses various structures to store paths, nodes, and constraints. The most important structures for SIPPS-IP in the codebase are:

- **Open List:**
  - Implemented as a pairing heap.
  - Uses a custom comparator for prioritizing nodes.
- **Focal List:**
  - Used as Open List for SIPPS-IP to sort nodes based on collisions first, also a pairing heap.
  - Uses a secondary custom comparator for tie-breaking.
- **Closed List:**
  - Implemented as a hash table.
  - Efficiently stores and retrieves nodes to prevent redundant processing.
- **Constraint Table:**
  - Manages constraints related to paths.
  - Interfaces with `PathTable` and `PathTableWC` (with collisions).
- **Reservation Table:**
  - Works with the `ConstraintTable`.
  - Ensures adherence to kinematic constraints by reserving space-time slots.

Pairing heap is a type of heap data structure that supports efficient priority queue operations. It consists of a collection of multiway trees, where each tree is a heap-ordered tree. This structure is used for both the OPEN list and the FOCAL list.

5.1.1 Motion primitives

The motion primitives used for this thesis deviate a bit from the implementation of [Yak21], by introducing Move by n cells. However as it is used for both algorithms SIPP-IP and SIPPS-IP, it should not change from the original paper [AY23], where they do not specify a set of motion primitives. These are all the actions available for all agents:

- **Turning**: The turning speed is 10 timesteps.

- **Speed-Up**: The agent occupies cells at these intervals (implementation follows [Yak21]), [[0, 20), [0, 29), [20, 35), [28, 40), [34, 40)], this primitive can be seen in Figure 3.3c).
- **Slow-Down**: This primitive mirrors the speed-up sequences but in reverse order.
- **Continue at Max Speed**: The agent maintains its maximum velocity for a duration of 5 timesteps, so the intervals are for cell 0 - [0, 5) and cell 1 - [0, 5].
- **Move n Cells**: Similarly to speed-up we define ranges for each cell, however, we end up with 0 velocity. The sequences for movement over various distances are:
  - **Move 1 cell**: $[[0,29)\,,[0,29)]$
  - **Move 2 cells**: $[[0,20)\,,[0,40)\,,[20,40)]$, Figure 3.3a
  - **Move 3 cells**: $[[0,20)\,,[0,30)\,,[20,50)\,,[29,50)]$
  - **Move 4 cells**: $[[0,20)\,,[0,29)\,,[20,38)\,,[28,67)\,,[38,67)]$
  - **Move 5 cells**: $[[0,20)\,,[0,29)\,,[28,44)\,,[34,63)\,,[43,63)]$, Fig 3.3b
  - **Move 6 cells**: $[[0,20)\,,[0,29)\,,[20,35)\,,[34,49)\,,[40,69)\,,[49,69)]$
  - **Move 7 cells**: $[[0,20)\,,[0,29)\,,[20,35)\,,[28,40)\,,[34,46)\,,[40,55)\,,[46,75)\,,[55,75)]$

5.2 Software and Tools

In our research, we developed a proof of concept implementation in Python first, which allowed us to identify potential issues early in the development process. The Python implementation was primarily based on pseudo code from the SIPP-IP and SIPPS papers.

We then explored the codebases from the research papers and learned to work with them, specifically the SIPP-IP Repository and the MAPF-LNS2 Repository. Based on these implementations, we adjusted the MAPF-LNS2 source code to implement the SIPP-IP and SIPPS-IP algorithms.

For the program development, I used two IDEs: Visual Studio Code (VSCode)[Mic15] for C++ and PyCharm [Jet10] for Python. C++ was used for the main algorithms, while Python was used for minor scripts, creating examples, or evaluating the data. Both IDEs had Github Copilot [Git21] extension enabled, during development.

5.3 Program Configuration

The program configuration involves several parameters that allow for fine-tuning the performance and behavior of the solution. Below is a list of these parameters along with a brief description of each:

- `-k [ -agentNum ] arg`: Number of agents (10, 25, 50)
- `-t [ -cutoffTime ] arg (=300)`: Cutoff time in seconds
- `-m [ -map ]`: input file for map
- `-a [ -agents ]`: input file for agents (scenes)
- `-solver arg (=LNS)`: Solver to be used (e.g., LNS - Large Neighborhood Search)
- `-sipp-ip arg (=1)`: Choose between SIPPS-IP (0) or SIPP-IP (1)
- `-seed arg (=0)`: Random seed for reproducibility
- `-c [ -constraintFile ] arg`: Input file for constraints
- `-initLNS arg (=1)`: Use LNS to find initial solutions if the initial solver fails
- `-neighborSize arg (=3)`: Size of the neighborhood for LNS
- `-maxIterations arg (=1000)`: Maximum number of iterations
- `-initAlgo arg (=PP)`: MAPF algorithm for finding the initial solution (PP - Priority Planning)
- `-reserved-intervals (=0)`: Whether we should add reserved intervals on starting positions
- `-replanAlgo arg (=PP)`: MAPF algorithm for replanning (PP - Priority Planning)
- `-destoryStrategy arg (=Adaptive)`: Heuristics for finding subgroups (e.g., Random, RandomWalk, Intersection, Adaptive)
- `-initDestroyStrategy arg (=Adaptive)`: Initial heuristics for finding subgroups (e.g., Target, Collision, Random, Adaptive)

Fixed parameters. We have decided to fix some of the parameters at these values `neighborSize` = 3, `cutoffTime` = 100, `maxIterations` = 1000. The values were selected this way to have a reasonable amount of experiments in our limited time[1].

To change to different solution approaches we need to adjust these parameters: `initLNS`, `maxIterations`, `sipp-ip`, `reserved-intervals` (reserved in table), our different setups can be seen in Table 5.1.

Approach	`initLNS`	`maxIterations`	`sipp-ip`	`reserved`
SIPP-IP	0	0	1	0
LNS+SIPP-IP	0	1000	1	0
LNS+SIPP-IP-RI	0	1000	1	1
LNS2+SIPPS-IP	1	1000	0	0

Table 5.1: Configuration of various MAPF approaches for the program.

5.4 Evaluation

The benchmarks used for evaluation were sourced from work [SSF+19b]. These benchmarks consist of open-source MAPF instances that include maps and scenes with several predefined agents. Although these benchmarks are not specifically designed for problems with kinematic constraints, they proved to be sufficient for our purposes. Additionally, we used scenes that were evenly distributed across the maps. Each map was evaluated using 20 different scenes to ensure the robustness of the results. More details about the benchmarks can be found at Moving AI MAPF Benchmarks[2].

5.5 Setup

The evaluation was conducted on the following hardware setup:

- **Model Name:** MacBook Pro

[1]Around 1500 results were used for evaluation, which took around 5 days of runtime, also excluding runs not used for evaluation as result of not interesting cases (no difference in number of solutions, or very little to no solutions)

[2]Website to access benchmarks is at https://movingai.com/benchmarks/mapf.html

- **Chip:** Apple M2
- **Total Number of Cores:** 8 (4 performance and 4 efficiency)
- **Memory:** 16 GB

The program was running single-threaded and it is important to add that the computer was actively used during the evaluation runs. This concurrent usage may have potentially impacted the performance and execution times of the algorithms being tested.

5.6 Results

5.6.1 Performance Metrics

To evaluate implemented algorithms, we utilized the following performance metrics:

- **Success Rate:** The percentage of instances where the algorithm successfully found a solution.
- **Sum of Costs:** The total sum of costs for all agents.
- **SOC Gap:** Comparison of SOC values between two methods.

Gap Analysis

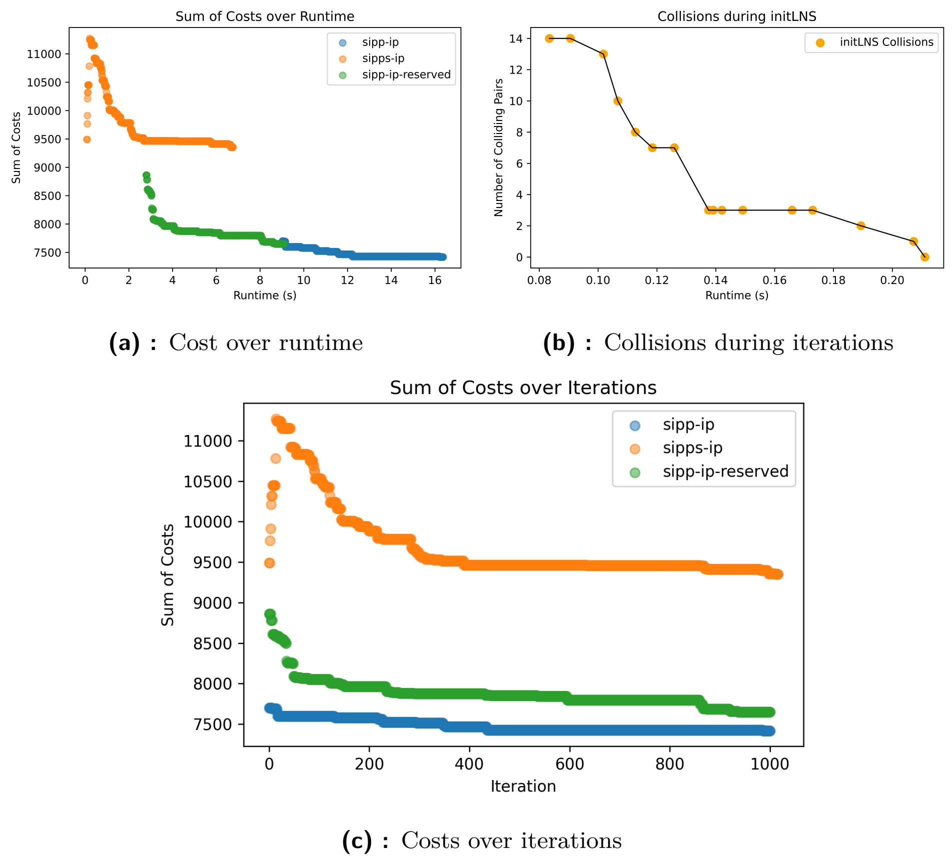
Gap analysis is a method used to quantify the relative difference in performance between two algorithms based on the SOCs. Let C_A represent the sum of costs associated with the first algorithm (main method) and C_B represent the sum of costs for the second algorithm (reference method). The gap is then defined as the normalized difference of the cost of the main method with the reference method, computed using the formula:

$$\mathrm{Gap}_{\mathrm{SOC}}(\mathrm{A},\mathrm{B}) = \frac{C_\mathrm{A} - C_\mathrm{B}}{C_\mathrm{B}} \tag{5.1}$$

This analysis provides a relative measure of how much more or less costly one algorithm is over the other. A positive gap value indicates that the first algorithm is more costly and the second is cheaper, whereas a negative value suggests the first one is more cost-effective.

5.6.2 Comparative Analysis

In Figure 5.1, we present a typical result of running LNS+SIPP-IP, LNS+SIPP-IP-RI, LNS2+SIPPS-IP methods on the empty-32-32 map with 50 agents and a 100-second time limit. Figure 5.1a shows real-time progression, while Figure 5.1c shows a solution cost for each iteration. Altogether we can see that the best plan was in the end found by SIPP-IP without reserved intervals, closely followed by SIPP-IP with reserved intervals, and lastly SIPPS-IP with a bit higher SOC.



(a) : Cost over runtime **(b) :** Collisions during iterations

(c) : Costs over iterations

Figure 5.1: Example run of methods on empty-32-32 map with 50 agents and scene 19.

Initially, SIPPS-IP increases its SOC until the dotted line, where the initial collision-free plan is found for it. The collisions during the initial LNS can be seen in Figure 5.1b, it starts with a high number of collisions and iteratively removes the collisions by repeating the search. After that, LNS is utilized to optimize the collision-free solution.

SIPP-IP on the other hand takes longer to find the initial solutions for reserved intervals, and even longer without reserved intervals, This is a result of the restart of the entire search, meaning that we scratch the current solution and replan the paths for all agents with different priorities (different permutation) in case SIPP-IP does not find a valid solution for current permutation.

5.6.3 Comparison of success rates

Figure 5.2 shows the success rates for seven different maps, each tested with varying numbers of agents. This comparison highlights the performance differences between our proposed solutions. The green bars represent LNS2+SIPPS-IP, the orange bars represent LNS+SIPP-IP-RI, and the blue bars represent LNS+SIPP-IP. The y-axis labels indicate the map and the corresponding number of agents (the last number). Each evaluation was conducted over 20 scenarios with a 100-second time limit. If there is a column missing for some scenario it means that the methods did not yield any results there.

The results show that LNS2+SIPPS-IP achieves the highest success rate in almost all cases. The only exception is maze-32-32-4 with 50 agents. Manual investigation shows, that for 4 scenes the last iterations have 1 collision, suggesting that LNS2 does not seem to be able to optimize the last collision, where I suspect that the neighborhood of 3 is too small for it to be able to resolve the conflict. In the other failed cases, the number of last collisions was around 5. I suspect, that the algorithm has not converged yet or it could be the same issue as before, unable to resolve the conflict with the current neighborhood size.

The LNS+SIPP-IP-RI performs better than LNS+SIPP-IP in all cases and even can find solutions for all scenarios, where LNS+SIPP-IP finds none (empty-32-32 with 100 agents). Suggesting that most collisions for LNS+SIPP-IP occur in fact at the starting positions.

The overall success rates, over all the scenarios depicted in Figure 5.2, are shown in Table 5.2. Notably, both SIPP-IP and LNS+SIPP-IP share an

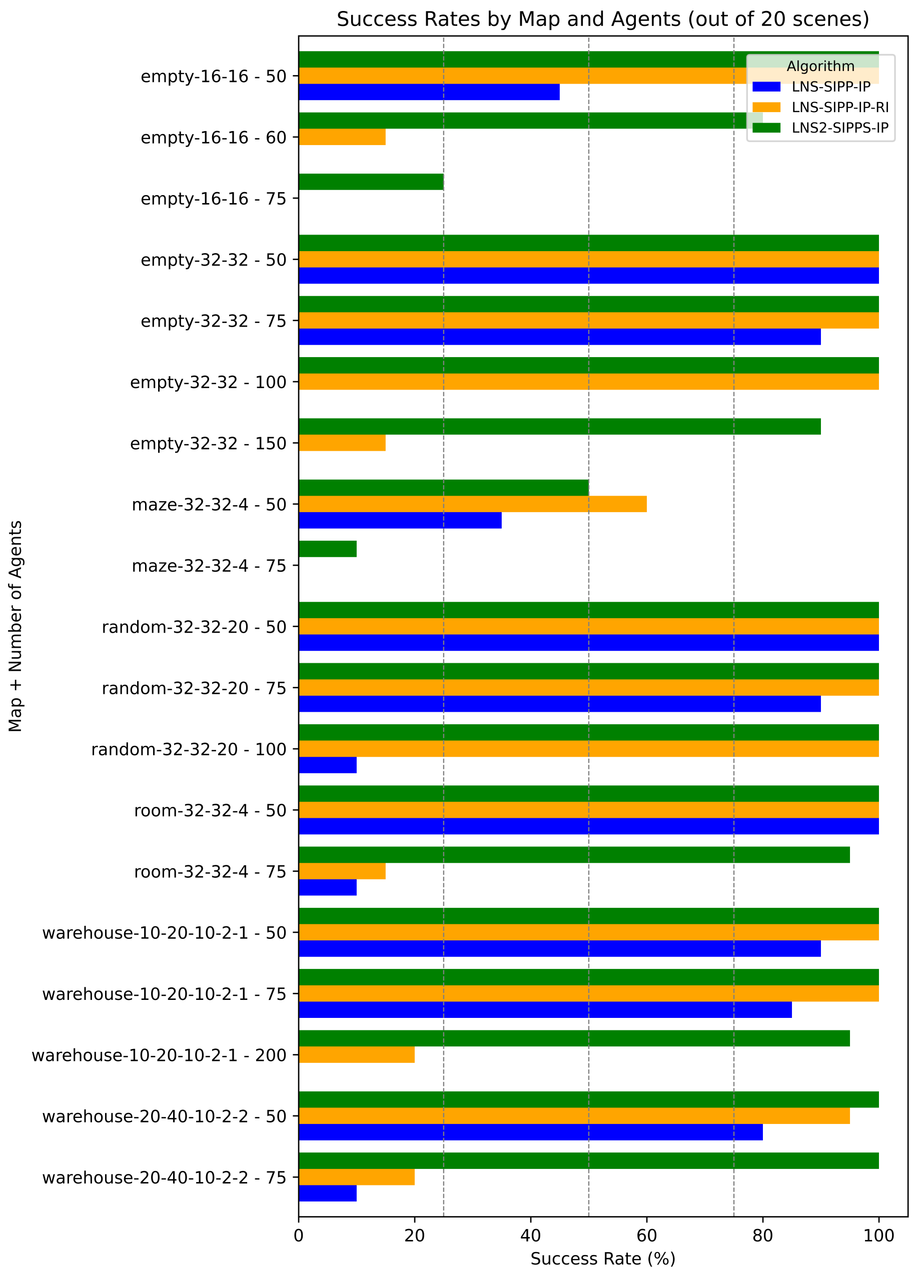


Figure 5.2: SOC gap analysis of individual maps between LNS+SIPP-IP-RI and LNS2+SIPPS-IP.

identical success rate of 44.47 %. This is because if LNS+SIPP-IP successfully finds a solution, then SIPP-IP must inherently do so too, the LNS part is only initiated after we find the initial solution the same way. Following these two is LNS+SIPP-IP-RI, which achieves a success rate of 65.26 %, and LNS2+SIPPS-IP 86.58 %, which clearly shows improvement in terms of finding more solutions, in these settings.

Algorithm	Total Success Rate (%)
SIPP-IP	44.47
LNS+SIPP-IP	44.47
LNS+SIPP-IP-RI	65.26
LNS2+SIPPS-IP	86.58

Table 5.2: Total Success Rates of Algorithms

Comparison of gap. To evaluate the enhancements to the discovered solution, we reference Figure 5.3, which compares the costs between SIPP-IP and LNS+SIPP-IP. Typically, the improvements are within a 0-10 % range, with LNS+SIPP-IP having lower SOC. The least significant improvement occurs on the warehouse map, which rarely exceeds 1 %.

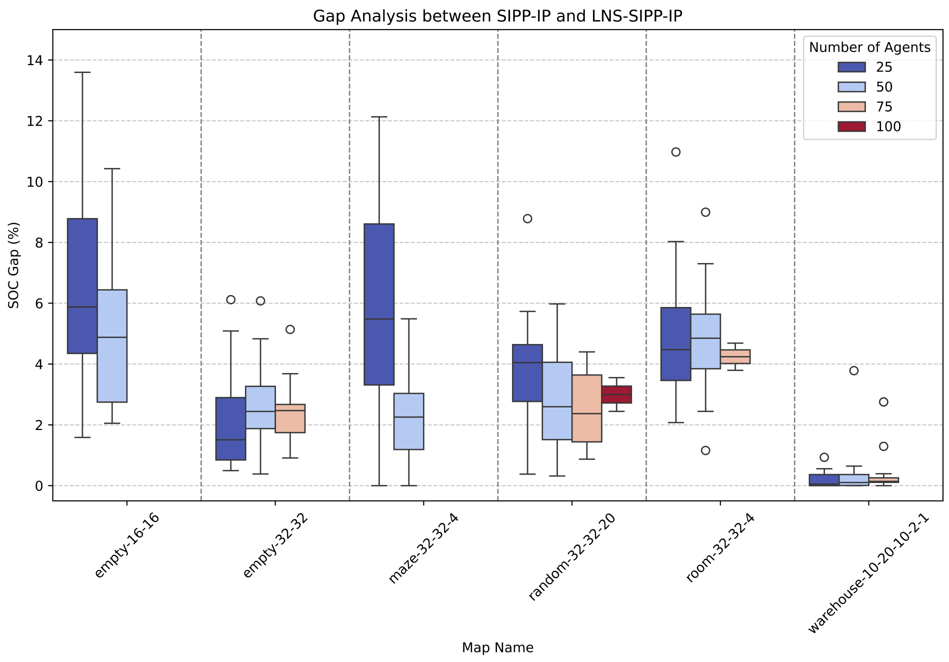


Figure 5.3: SOC gap analysis of individual maps SIPP-IP with reference algorithm LNS+SIPP-IP.

Comparing LNS+SIPP-IP and LNS+SIPP-IP-RI, Figure 5.4, gives us mean gap values closer to zero but favoring SIPP-IP a bit. This aligns with our assumption that we worsen the initial solution slightly at the beginning and iteratively optimize it.

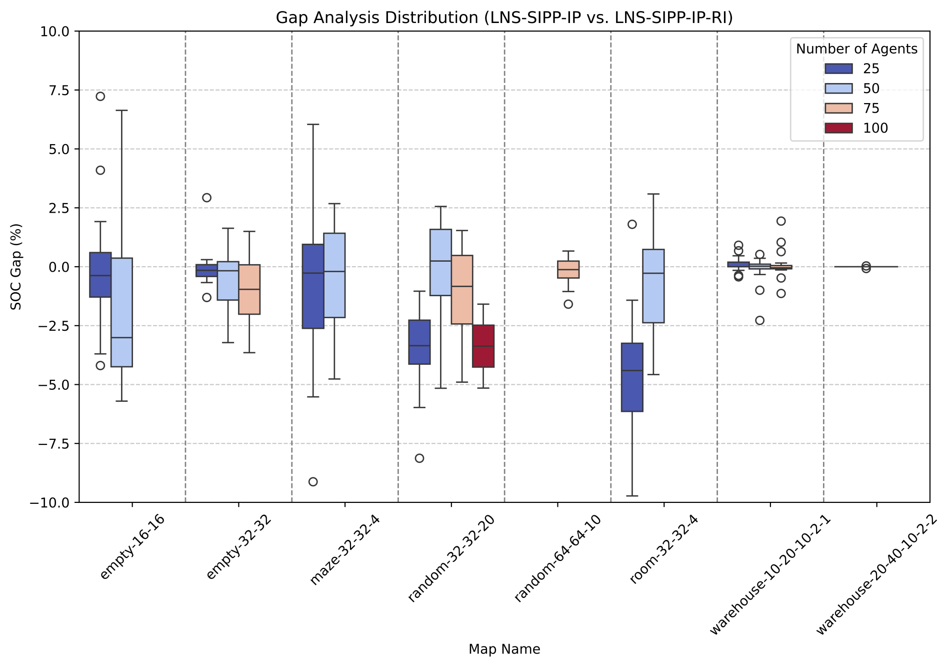


Figure 5.4: SOC gap analysis of individual maps of LNS+SIPP-IP with reference method LNS+SIPP-IP-RI

Figure 5.5 depicts a boxplot analysis of the SOC gap distribution across various densities of maps. The densities are calculated as $\frac{\#\text{ agents}}{\#\text{ free cells}}$, the number of agents divided by the number of free cells for the given map. At nearly 0 density, the ranges are close to zero, and with increasing density, the mean is approaching the 40 % gap. Afterward, the gap seems to decrease a bit to 30 % gap.

Please note that the gap analysis only includes cases where both algorithms have successfully found solutions. Although analyzing the differences in performance is important, finding a solution remains the primary goal, even if it results in a larger gap. This approach ensures that we prioritize effective outcomes over purely optimizing performance metrics.

5.6.4 Summary of results

The analysis reveals that the LNS2+SIPPS-IP plans are generally more costly compared to both LNS+SIPP-IP-RI and LNS+SIPP-IP. This increased

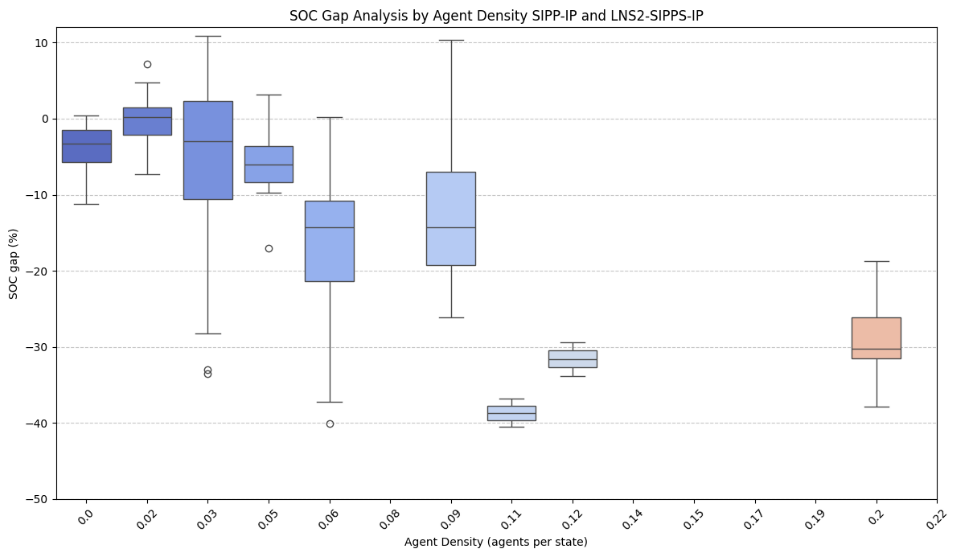


Figure 5.5: SOC gap by density with main method SIPP-IP and with reference method LNS2+SIPPS-IP.

expense could stem from insufficient solver iterations or the potential benefits of implementing early resets when we have early collisions. Despite the cost increase in LNS+SIPP-IP-RI compared to LNS+SIPP-IP due to interval reservation, this turns out to increase also quite significantly the number of found solutions. Even though LNS2+SIPPS-IP yields higher costs, it significantly increases the number of found solutions, especially in densely packed environments where the other methods fall short, which we consider a great result for our method.

5.7 Future Work

The findings of this study indicate that LNS+SIPPS-IP offers significant improvements over SIPP-IP for MAPF in terms of finding feasible solutions. However, the current implementation has several limitations that need to be addressed. Future research should focus on several areas to enhance the performance and applicability of SIPPS-IP:

- **Optimization of Interval Projections:** Reducing the redundancy in interval projections to minimize computational overhead and improve efficiency. As we did forward and backward passes, there should be a way to speed this, by projecting all primitives at once.

- **Handling Dynamic Obstacles:** Extending the evaluation and investigating how the approaches perform when dynamic obstacles are present.
- **Improved Heuristics:** Developing better heuristics considering factors such as orientation, speed, iteration, and runtime, and exploring state space optimizations.
- **Node dominance:** While we were utilizing node dominance, there is a possibility of optimizing nodes generated by motion primitives, such that they might be dominated by each other in certain cases, thus removing some branches early.
- **Intermediate nodes:** for the reconstruction of the path, we created intermediate nodes along the interval projection, this could be implemented more efficiently by storing which primitive was used and the intermediate steps would be created only from the knowledge of primitive in reconstructing the path. This could at least lower memory utilized.
- **Adjusting parameters of program:** Extending evaluation by adjusting parameters of the program for different algorithms - these could be runtime, neighborhood size, destroy strategy, or number of iterations.
- **Adaptive neighborhood size:** As we found cases where the LNS2-SIPPS-IP could not optimize collisions anymore, it might be interesting to increase the neighborhood size improving the adaptive LNS.
- **SIPPS-IP-RI:** Improving the LNS2+SIPPS-IP with reservation intervals for start position, the same way as we did for SIPP-IP.

Addressing these areas will not only enhance the performance of our methods but also broaden its applicability to more complex environments.

5.8 Conclusion

This thesis successfully developed and evaluated algorithms to solve the MAPF with kinematic constraints. We introduced LNS2+SIPPS-IP, LNS+SIPP-IP, and LNS+SIPP-IP-IR, which showed strengths in handling different environmental densities and agent counts.

Overall, our work demonstrated that integrating these algorithms finds a solution in dynamic environments. However, there is still room for improvement in the scalability and cost of the plan. Future research should focus on optimizing these algorithms for cost and scale. This research lays a strong foundation for future advancements in MAPF with kinematic constraints.

Appendices

Appendix A

Acronyms

ALNS	Adaptive LNS.
CCBS	Continuous-time Conflict-Based Search.
LNS	Large Neighbourhood Search.
MAPF	Multi agent path find.
MAPF-LNS2	Multi agent path find-Large Neighbourhood Search (LNS).
SIPP	Safe Interval Path Planning.
SIPP-IP	Safe Interval Path Planning with Interval Projection.
SIPP-IP-ri	Safe Interval Path Planning with Interval Projection and with Reserved Intervals.
SIPPS-IP	Safe Interval Path Planning with Soft Constraints and with Interval Projection.
SMT	Satisfiability Modulo Theories.
SMT-CCBS	Satisfiability Modulo Theories-CCBS.
SOC	Sum of Costs.
STA*	Space-Time A*.

Appendix B

Bibliography

[AY23] Zain Alabedeen Ali and Konstantin Yakovlev, *Safe interval path planning with kinodynamic constraints*, Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, 2023, pp. 12330–12337.

[AYS+22] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern, *Multi-agent pathfinding with continuous time*, Artificial Intelligence **305** (2022), 103662.

[ČNKS15] Michal Čáp, Peter Novák, Alexander Kleiner, and Martin Seleckỳ, *Prioritized planning algorithms for trajectory coordination of multiple mobile robots*, IEEE transactions on automation science and engineering **12** (2015), 835–849.

[Git21] GitHub, *Github copilot*, 2021, Accessed: 2024-05-22.

[GLL+23] Jianqi Gao, Yanjie Li, Xinyi Li, Kejian Yan, Ke Lin, and Xinyu Wu, *A review of graph-based multi-agent pathfinding solvers: From classical to beyond classical*, Knowledge-Based Systems (2023), 111121.

[HKC+16] Wolfgang Hönig, TK Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig, *Multi-agent path finding with kinematic constraints*, Proceedings of the International Conference on Automated Planning and Scheduling, vol. 26, 2016, pp. 477–485.

[Jet10] JetBrains, *Pycharm*, 2010, Accessed: 2024-05-22.

[LCH+22] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig, *Mapf-lns2: fast repairing for multi-agent path finding via* *large neighborhood search*, Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36, 2022, pp. 10256–10265.

[Li21] Jiaoyang Li, *Mapf-lns2: Multi-agent path finding with large neighborhood search*, `https://github.com/Jiaoyang-Li/MAPF-LNS2`, 2021, Accessed: 2024-05-12.

[MHK+19] Hang Ma, Wolfgang Hönig, TK Satish Kumar, Nora Ayanian, and Sven Koenig, *Lifelong path planning with kinematic constraints for multi-agent pickup and delivery*, Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 7651–7658.

[Mic15] Microsoft Corporation, *Visual studio code*, 2015, Accessed: 2024-05-22.

[PK11] Mihail Pivtoraiko and Alonzo Kelly, *Kinodynamic motion planning with state lattice motion primitives*, 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2011, pp. 2172–2179.

[PL11] Mike Phillips and Maxim Likhachev, *Sipp: Safe interval path planning for dynamic environments*, 2011 IEEE international conference on robotics and automation, IEEE, 2011, pp. 5628–5635.

[PR19] David Pisinger and Stefan Ropke, *Large neighborhood search*, Handbook of metaheuristics (2019), 99–127.

[RN16] Stuart J Russell and Peter Norvig, *Artificial intelligence: a modern approach*, Pearson, 2016.

[SFSB16] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski, *Efficient sat approach to multi-agent path finding under the sum of costs objective*, Proceedings of the twenty-second european conference on artificial intelligence, 2016, pp. 810–818.

[Sha98] Paul Shaw, *Using constraint programming and local search methods to solve vehicle routing problems*, International conference on principles and practice of constraint programming, Springer, 1998, pp. 417–431.

[Sil05] D Silver, *Collaborative pathfinding*, Proceedings of AIIDE (2005), 23–28.

[Sil20] David Silver, *Cooperative pathfinding*, 2020, Accessed: 2023-04-29.

[SSF+19a] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak, *Multi-agent pathfinding: Definitions, variants, and benchmarks*, Symposium on Combinatorial Search (SoCS) (2019), 151–158.

[SSF+19b] ———, *Multi-agent pathfinding: Definitions, variants, and benchmarks*, Symposium on Combinatorial Search (SoCS) (2019), 151–158.

[Yak21] Konstantin Yakovlev, *Sipp-ip: Safe interval path planning for individual agents*, `https://github.com/pathplanning/sipp-ip`, 2021, Accessed: 2024-05-12.

[ZLH+22] Shuyang Zhang, Jiaoyang Li, Taoan Huang, Sven Koenig, and Bistra Dilkina, *Learning a priority ordering for prioritized planning in multi-agent path finding*, Proceedings of the International Symposium on Combinatorial Search, vol. 15, 2022, pp. 208–216.

Appendix C

Attachment Content

- **evaluation_source_code/** – Contains evaluation scripts and script to run tests
- **input_data/** – Maps and scenes for evaluation
- **program_source_code/** – This directory contains developed program
- **latex/** – Source files for this thesis
- **results/** – Results from the program used for evaluation
- **supporting_scripts/** – Scripts to create examples or simple graphs