
Executing Multi-Agent Path Plans with Uncertainty



Yihao Liu

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2025

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

19/05/2025

.....

Date

[Handwritten signature]

Yihao Liu

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

19/05/2025

Date

.....

Prof. Xueyan Tang

Authorship Attribution Statement

This thesis contains material from a paper accepted at conferences in which I am listed as an author.

Chapter 3 is published as Liu, Yihao, Xueyan Tang, Wentong Cai, and Jingning Li. "Multi-Agent Path Execution with Uncertainty." In Proceedings of the International Symposium on Combinatorial Search, vol. 17, pp. 64-72. 2024.

The contributions of the co-authors are as follows:

- Prof Xueyan Tang provided the initial project direction.
- I co-designed the study with Prof Xueyan Tang, implemented all the code on my own and conducted experimental evaluations on a workstation at the College of Computing and Data Science, NTU.
- Prof Wentong Cai and Ms. Jingning Li provided some idea in the study.

19/05/2025

.....

Date

[Signature]

Yihao Liu

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Xueyan Tang, for his invaluable mentorship and steadfast support throughout my Ph.D. journey. His deep knowledge, patient guidance, and high standards have been instrumental in shaping my research and academic growth. I am especially grateful for the freedom and trust he gave me to explore my ideas, as well as the thoughtful feedback that always helped refine my work.

I am profoundly thankful to my parents and grandparents, whose unconditional love and encouragement have been a constant source of strength. Their sacrifices, understanding, and belief in my abilities made this journey possible, and I owe much of my perseverance to their quiet resilience and unwavering support.

To my partner, Shixin, thank you for being my anchor through the highs and lows of this long and challenging process. Your emotional support, patience, and thoughtful encouragement have meant more to me than words can express. I am truly grateful for your presence in my life.

¹Special thanks to the author of the L^AT_EX thesis template [\[1\]](#) used in this work.

“The Linux philosophy is ‘Laugh in the face of danger’. Oops. Wrong One. ‘Do it yourself’. Yes, that’s it.”

—Linus Torvalds

To my dear family

Abstract

The planning and execution of multi-agent systems is a fundamental problem in computer science, with broad applications in areas such as warehouse automation, autonomous vehicle coordination, and multi-robot exploration. Efficiently coordinating multiple agents to move without conflicts is crucial for improving safety, scalability, and operational efficiency in these systems. At the same time, the problem presents significant research challenges due to its computational complexity, dynamic environments, and the need to ensure robustness under uncertainty. In this thesis, I analyze problem formulations, conflicts, and objectives of various multi-agent problems and their extensions. I also study a number of Multi-Agent Path Finding (MAPF) solvers, including single agent path planning algorithms, suboptimal MAPF solvers and optimal MAPF solvers.

Several multi-agent problems are studied in the thesis. The first problem is Multi-Agent Path Execution with Uncertainty, where I study robust and effective execution of multi-agent path plans under uncertainty with conflict-freeness and deadlock-freeness. A feasibility-based coordination method is proposed to maintain reliable multi-agent operation under uncertain conditions. The second problem is Continuous-Time Multi-Agent Path Finding, where I design an efficient hierarchical suboptimal solver for the continuous-time MAPF problem. The method improves both solution quality and runtime by combining temporal path planning with large neighborhood search. The third problem is Robust Continuous-Time Multi-Agent Path Execution, which solves the first problem in continuous time with the help of the plan generated by the second problem. This approach allows smooth and conflict-free execution even when agents move asynchronously in continuous time. The continuous-time plans obtained from the second work are used as input trajectories for execution, providing a strong foundation for robust coordination.

I experimentally demonstrate the effectiveness and computational efficiency of the solutions to these problems. The proposed methods consistently outperform existing approaches in both runtime and solution quality across a wide range of benchmark instances. Future work will focus on lifelong task execution and integrating uncertainty-aware planning into large-scale multi-agent systems.

Contents

Acknowledgements	ix
Abstract	xiii
List of Figures	xix
List of Algorithms	xxii
Symbols and Acronyms	xxiii
1 Introduction	1
1.1 Background	1
1.2 Major Contributions	2
1.2.1 Multi-Agent Path Execution with Uncertainty	2
1.2.2 Continuous-Time Multi-Agent Path Finding	3
1.2.3 Robust Continuous-Time Multi-Agent Path Execution	3
1.3 Thesis Organization	4
2 Preliminaries & Literature Review	7
2.1 Multi-Agent Path Finding (MAPF) Problem	7
2.1.1 Problem Definition	7
2.1.2 Conflicts Definition	9
2.1.3 Objectives	10
2.1.4 Behaviors of Agents at Goal Locations	11
2.2 Solvers of MAPF Problem	11
2.2.1 Single Agent Path Finding Problem	12
2.2.1.1 A* Search	13
2.2.1.2 Space-time A* search	14
2.2.1.3 Safe Interval Path Planning (SIPP) Algorithm	16
2.2.1.4 Focal Search Algorithm	17
2.2.1.5 Better Heuristics with All Pair Shortest Path Algorithm	18
2.2.2 Optimal Solvers	21
2.2.2.1 Global A* Search	21
2.2.2.2 Increasing Cost Tree Search (ICTS)	23

2.2.2.3	Conflict Based Search (CBS)	24
2.2.2.4	Compilation-based Solvers	28
2.2.3	Suboptimal Solvers	29
2.2.3.1	Prioritized Planning (PP)	30
2.2.3.2	Prioritized-Based Search (PBS)	31
2.2.3.3	Priority Inheritance with Backtracking (PIBT)	32
2.2.3.4	Hierarchical Cooperative A* (HCA*)	33
2.2.3.5	Enhanced CBS (ECBS)	34
2.2.3.6	Explicit Estimation CBS (EECBS)	37
2.2.3.7	MAPF via Large Neighborhood Search (MAPF-LNS)	39
2.3	Extension of Multi-Agent Problems	40
2.3.1	Continuous-Time Multi-Agent Path Finding Problem	40
2.3.1.1	Problem Definition	41
2.3.1.2	Continuous-time Conflict Based Search (CCBS)	42
2.3.2	Multi-Agent Execution with Uncertainty	44
2.3.2.1	Time-Based Uncertainty	44
2.3.2.2	Probability-Based Uncertainty	46
2.3.2.3	Generalized Uncertainty	47
2.4	Summary	47
3	Multi-Agent Path Execution with Uncertainty	49
3.1	Problem Definition	50
3.2	Path Plan Feasibility Problem	51
3.2.1	Feasibility and Location Dependency Graph	51
3.2.2	Hardness of Path Plan Feasibility	55
3.2.3	Feasibility Test	60
3.2.3.1	A Basic Algorithm	60
3.2.3.2	Heuristic Feasibility Test	63
3.2.3.3	Check for Cycles in LDG	64
3.2.3.4	Group Agents in LDG	66
3.3	Coordinating Path Execution	69
3.3.1	Naive Approach	70
3.3.2	Semi-naive Approach	71
3.3.3	Heuristic Approach	71
3.4	Experimental Evaluation	73
3.4.1	Compare Different Versions of Feasibility Test	76
3.4.2	Compare Different Approaches in Coordinating Path Execution	76
3.4.3	Compare with Existing Methods	77
3.5	Summary	83
4	Continuous-Time Multi-Agent Path Finding	85
4.1	Continuous-Time Single Agent Shortest Path Problem with Dynamic Obstacles	86
4.1.1	SIPPS	87

4.1.1.1	Collisions	88
4.1.1.2	Safe Interval Table	89
4.1.1.3	Collision Avoidance Table	90
4.1.1.4	Search State in SIPPS	92
4.1.1.5	Main SIPPS Algorithm	93
4.1.1.6	Expanding Search State	95
4.1.1.7	Inserting State to Open List	95
4.1.2	Example Applications of SIPPS	98
4.1.2.1	Conflicted-Based Search (CBS)	98
4.1.2.2	Prioritized Planning (PP)	99
4.2	Continuous-Time MAPF via Large Neighborhood Search	99
4.2.1	Find Solution in a Neighborhood	100
4.2.2	Collision-Based Neighborhoods	101
4.2.3	Failure-Based Neighborhoods	102
4.2.4	Random Neighborhoods	104
4.2.5	Adaptive LNS (ALNS)	106
4.3	Experimental Evaluation	107
4.4	Summary	112
5	Robust Continuous-Time Multi-Agent Path Execution	113
5.1	Problem Definition	114
5.1.1	Feasibility and State Dependency Graph	115
5.1.2	Comparison with Existing Work	119
5.2	Feasibility Test	119
5.2.1	Feasibility Test Algorithm	120
5.2.2	Remove Redundant Conflicts	121
5.2.3	Construct Conflict Groups	127
5.3	Path Execution Solution	134
5.3.1	Offline Path Execution	134
5.3.2	Online Path Execution	136
5.4	Experimental Evaluation	138
5.4.1	Offline vs. Online Algorithm	140
5.4.2	Removing Redundant Conflicts and Constructing Conflict Groups	145
5.5	Summary	149
6	Conclusion & Future Work	151
6.1	Conclusion	151
6.2	Future Work	153
6.2.1	Lifelong Multi-Agent Path Execution with Uncertainty	153
6.2.2	Multi-Agent Path Finding with Uncertainty	154
6.2.3	Optimization of Suboptimal Multi-Agent Path Plans	155
	List of Author's Awards, Patents, and Publications	157

Bibliography**159**

List of Figures

3.1	A feasible path plan and its LDG.	54
3.2	An infeasible path plan and its LDG.	54
3.3	LDG of variables x_i and x_j	57
3.4	Grouping the agents in a LDG with its ADG.	68
3.5	A path plan (left) and the LDG of the residual path plan if all four agents are unblocked initially (right).	70
3.6	Maps from the Moving AI MAPF benchmark [2] and generated by us.	74
3.7	Comparison of different versions of feasibility test for 40 agents.	75
3.8	Comparison of naive, semi-naive and heuristic approaches for 30 agents.	78
3.9	Comparison of proposed and other methods for 30 agents.	81
3.10	Comparison of proposed and other methods for different numbers of agents with $k = 20$	84
4.1	An example of rectangle collision detection.	91
4.2	Maps from the Moving AI MAPF benchmark [2].	108
4.3	Illustration of 2^k -connected grids.	109
4.4	Results for 2^k -connected maps when $k = 2$	110
4.5	Results for 2^k -connected maps when $k = 3$	111
4.6	Results for 2^k -connected maps when $k = 4$	112
5.1	Some example conflicts.	115
5.2	An illustration of $\mathcal{E}_{\text{SDG}_1}$ in SDG.	116
5.3	Dependency edges for a node-node conflict.	117
5.4	Dependency edges for an edge-edge conflict.	117
5.5	Dependency edges for a node-edge conflict.	118
5.6	Dominance relationship of two conflicts between the same two agents.	122
5.7	Edge-edge conflicts induced from a node-node conflict.	125
5.8	Node-edge conflicts induced from a node-node conflict.	126
5.9	Edge-edge conflicts induced from a node-edge conflict.	126
5.10	Node-node conflicts that can be merged with a node-node conflict.	131
5.11	Node-edge conflicts that can be merged with a node-node conflict.	132
5.12	Node-node conflicts that can be merged with a node-edge conflict.	132
5.13	Node-edge conflicts that can be merged with a node-edge conflict.	132

5.14	Comparison of offline and online algorithms for 40 agents on 2^2 -connected maps.	141
5.15	Comparison of offline and online algorithms for 40 agents on 2^3 -connected maps.	142
5.16	Comparison of offline and online algorithms for 40 agents on 2^4 -connected maps.	142
5.17	Comparison of offline and online algorithms for different number of agents with $k = 20$ on 2^2 -connected maps.	143
5.18	Comparison of offline and online algorithms for different number of agents with $k = 20$ on 2^3 -connected maps.	144
5.19	Comparison of offline and online algorithms for different number of agents with $k = 20$ on 2^4 -connected maps.	144
5.20	Comparison of online algorithm and its optimized versions for 40 agents on 2^2 -connected maps.	146
5.21	Comparison of online algorithm and its optimized versions for 40 agents on 2^3 -connected maps.	147
5.22	Comparison of online algorithm and its optimized versions for 40 agents on 2^4 -connected maps.	148

List of Algorithms

1	Space-time $A^*(\mathcal{V}, \mathcal{E}, s, g, \mathcal{O})$	15
2	$CBS(\mathcal{V}, \mathcal{E}, \mathcal{A})$	26
3	$PP(\mathcal{V}, \mathcal{E}, \mathcal{A})$	31
4	Feasibility test of path plan $\mathcal{P}$	62
5	The for loop in heuristic feasibility test	63
6	Update C when $\langle v_{i,j}, v_{i',j'} \rangle$ is added	67
7	Naive approach to unblock agents	70
8	Identify agents that can definitely be unblocked or must be kept blocked	72
9	Heuristic approach to unblock agents	73
10	$buildSafeIntervalTable(\mathcal{V}, \mathcal{O}^h, \mathcal{O}^s)$	90
11	$buildCollisionAvoidanceTable(\mathcal{O}^h, \mathcal{O}^s)$	92
12	$findEarliestArrivalTimeWithoutCollision(CAT, \pi, low, high, type)$	93
13	Continuous-Time SIPPS($\mathcal{V}, \mathcal{E}, s, g, \mathcal{O}^h, \mathcal{O}^s$)	94
14	$ExpandState(n, OPEN, CLOSED, SIT, CAT)$	96
15	$InsertState(n, OPEN, CLOSED)$	98
16	$PP-SIPPS(\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{A}_n)$	100
17	$GenerateCollisionBasedNeighborhood(\mathcal{V}_c, \mathcal{E}_c, N)$	103

18	GenerateFailureNeighborhood($\mathcal{V}_c, \mathcal{E}_c, N$)	105
19	GenerateRandomNeighborhood($\mathcal{V}_c, \mathcal{E}_c, N$)	106
20	ALNS	107
21	FeasibilityTest($\mathcal{C}_{\text{unsettled}}$)	121
22	Detect Redundant Conflicts between Agents a_i and $a_{i'}$	125
23	Construct Conflict Groups (Iterative)	130
24	Construct Conflict Groups (Heuristic)	133
25	Offline path execution (of agent a_i)	135
26	Unblock agents in online path execution	138

Symbols and Acronyms

Symbols

v	a node in a graph
$\mathcal{V}$	a set of all nodes in a graph
e	an edge in a graph
$\mathcal{E}$	a set of all edges in a graph
$\mathcal{G}$	a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
a	an agent
M	the number of agents
$\mathcal{A}$	a set of agents $\{a_1, \dots, a_M\}$
t	a task
$\mathcal{T}$	a set of tasks $\{t_1, \dots, t_M\}$
s	the start node of an agent
g	the goal node of an agent
t	a timestep (or time instant)
δ	a time interval
π	an action of an agent
n	the number of actions in a path plan
p	a path plan of an agent, a sequence $\langle \pi_i, \dots, \pi_n \rangle$
$\mathcal{P}$	a set of path plans $\{p_1, \dots, p_M\}$
o	a dynamic obstacle
$\mathcal{O}$	a set of dynamic obstacles
c	a conflict
$\mathcal{C}$	a set of conflicts

σ	a search state
$g(\sigma)$	a cost function of the previous expanded path
$h(\sigma)$	a heuristic function that estimates the cost to expand the path to the goal
$f(\sigma)$	a cost heuristic function, usually $f(\sigma) = g(\sigma) + h(\sigma)$
$\epsilon(\sigma)$	an error function
w	cost weight
C	cost bound
d	the distance between two nodes
p	probability
U	unblocked agents
B	blocked agents
S_a	the shape of an agent
lr	learning rate

Acronyms

MAPF	Multi-agent Path Finding
PP	Prioritized Planning
PBS	Priority-Based Search
PIBT	Priority Inheritance with Backtracking
HCA*	Hierarchical Cooperative A*
ICTS	Increasing Cost Tree Search
CBS	Conflict-Based Search
ECBS	Enhanced Conflict-Based Search
EECBS	Explicit Estimation Conflict-Based Search
CCBS	Continuous-time Conflict Based Search
SIPP	Safe Interval Path planning
SIPPS	Safe Interval Path Planning with Soft Constraints
LNS	Large Neighborhood Search
CT	Constraint Tree
RT	Reservation Table
GT	Goal Table
ADG	Agent Dependency Graph

LDG	Local Dependency Graph
SDG	State Dependency Graph
SIT	Safe Interval Table
CAT	Collision Avoidance Table
SAT	Boolean Satisfiability Problem

Chapter 1

Introduction

1.1 Background

In computer science, the planning of multi-agent systems is a common problem in real-world applications, including warehouse management [3], aircraft towing [4] and mobile office service [5].

Autonomous guided vehicle (AGV) is a typical type of agent, which is widely used in warehouses to move objects. In earlier years, the functionalities of AGVs were limited to transporting large and heavy items that cannot be easily moved by humans. Nowadays, thanks to the rapid development of robotics, wireless communication, and microchips, AGVs are made much smaller, and they are able to do some more complicated work that can only be done by humans before.

Besides warehouse robots, multi-agent systems exist in many real-world scenarios. Robots can help human complete some dangerous tasks, such as mine exploration, rescuing and even outer-space exploration. In the past, such tasks only occupy one or several agents, but nowadays, a large number of agents can cooperate together and complete the tasks more efficiently.

It is known that the 21-st century is the age of Internet, and multi-agent systems occur in many online applications. For example, on a ride-hailing platform, the server is scheduling millions of cars at the same time to fulfill the demands of customers from a whole country. An annual report from a Chinese ride-hailing

platform DIDI shows that 21 million drivers and 450 million users used their service in a year [6].

In addition, many modern computer games contain a multi-agent system. For example, in a typical role play game, there are various enemies, allies, interactive objects and obstacles on the map. The strategy of how to plan the movement of the enemies and allies without conflicting with the objects and obstacles is the same of that in a real-world multi-agent application.

With the smaller and smarter agents, or agents in computer games, the task allocation, path planning and plan execution of the agents all face a new challenge, since the complexity of finding the optimal solutions of most of the problems above is NP-hard. When the number of agents in a single system keeps growing, the hardness to plan them together increases exponentially. The early studies are not suitable for the new problems, and in the recent decade, research on multi-agent systems becomes popular again.

1.2 Major Contributions

The major contributions of the thesis are the development of solutions to several extended multi-agent problems.

1.2.1 Multi-Agent Path Execution with Uncertainty

In real-world multi-agent applications, unexpected conditions can break the assumptions made in path planning and degrade the effectiveness of path execution. We need to carefully plan the paths for the agents and coordinate the execution of the plan. While path planning (often known as Multi-Agent Path Finding, MAPF) has been extensively studied [7], designing execution policies for planned paths has received less attention. Robust and effective execution is not only critical but also challenging because plan execution is often accompanied by uncertainties due to unforeseen circumstances [8, 9].

In this work, we study robust and effective execution of multi-agent path plans under uncertainty. To guarantee conflict-freeness and deadlock-freeness, we define

a feasibility problem to check whether the remaining portion of a path plan can be successfully executed. We prove that the problem is NP-complete and propose feasibility test algorithms. We then develop algorithms to coordinate the agents online and have as many of them as possible moving concurrently to maximize the effectiveness of execution. Furthermore, we experimentally demonstrate the path execution effectiveness and computational efficiency of our algorithms.

1.2.2 Continuous-Time Multi-Agent Path Finding

Classical MAPF problems [7] usually assume that time is discretized and an agent either waits at its current location or moves to an adjacent location in each timestep. Nevertheless, in practical applications, time flows continuously and move actions are tightly coupled with trajectories and do not always take uniform times to complete. Many other simplifying assumptions such as grid maps and unit size of agents in classical MAPF problems may also reduce the range of possible application scenarios.

In this work, we propose an efficient hierarchical suboptimal solver for the continuous-time MAPF problem based on Safe Interval Path Planning with Soft Constraints [10] at the low level and Large Neighborhood Search [10, 11] at the high level. The objective of the problem is to find a *feasible* plan while minimizing the sum-of-costs of all paths in the plan. Experimental evaluation shows that our solution can handle large-scale continuous-time MAPF problems effectively and efficiently.

1.2.3 Robust Continuous-Time Multi-Agent Path Execution

The Multi-Agent Path Execution with Uncertainty problem can be further extended to continuous-time basis with the paths generated by the solver we proposed in the above Continuous-Time Multi-Agent Path Finding problem.

In this work, we develop a robust continuous-time path execution framework to deal with unexpected delays arising from various reasons. The framework takes the planned paths of agents as input and coordinates the path execution in an online

fashion based on the evolving states of agents. We first define all possible conflicts between the agents, taking into account non-uniform action durations, volumes of individual agents and distinct trajectories of paths. We then construct a state dependency graph to model the conflicts and formulate a feasibility problem to infer whether all the agents can successfully reach their goal locations following their respective paths. Furthermore, we develop execution policies to guarantee that the path execution is conflict-free and deadlock-free. To optimize the effectiveness of path execution, our execution policies enable as many agents as possible to move towards their goal locations concurrently. Experimental results demonstrate the correctness, effectiveness and efficiency of our algorithms.

1.3 Thesis Organization

Chapter 1 introduces the background of multi-agent problems and highlights their relevance in real-world applications. It outlines the major contributions of this thesis, which are extensions of classical MAPF problems motivated by the growing need for more robust and scalable multi-agent solutions. The chapter also includes a summary of the thesis structure.

Chapter 2 presents the preliminaries of the multi-agent pathfinding (MAPF) problem, including its formal definition, conflict types, objective functions, and agent behaviors at goal locations. It then reviews relevant literature covering a range of MAPF solvers, from single-agent methods to optimal and suboptimal multi-agent approaches. The chapter concludes with a discussion on the recent advances and open challenges in extended MAPF problems, particularly in continuous-time MAPF and robust path execution under uncertainty, which are two areas central to this thesis.

Chapter 3, the first major contribution of this thesis, proposes an execution framework for handling generalized uncertainties in discrete-time settings, specifically unexpected delays in agent movements. The framework formulates a feasibility problem to determine whether all agents can reach their goals without conflicts. This problem is proven to be NP-complete, and corresponding deterministic and heuristic feasibility test algorithms are developed. Agent coordination is performed

online using these tests to enable maximal concurrency while ensuring conflict-freeness and deadlock-freeness. Several optimizations are also proposed to improve the efficiency and scalability of the feasibility and coordination algorithms.

Chapter 4, the second major contribution of this thesis, introduces a hierarchical suboptimal continuous-time MAPF solver. The solver combines Safe Interval Path Planning with Soft Constraints (SIPPS) at the low level with Large Neighborhood Search (LNS) at the high level. The SIPPS algorithm is compatible as a low level algorithm with other hierarchical MAPF solvers such as Prioritized Planning (PP) and Conflict-Based Search (CBS). The chapter also presents various neighborhood selection strategies to enhance solver performance across different instance types.

Chapter 5, the third major contribution of this thesis, extends the discrete-time execution framework to address continuous-time MAPF with uncertainty. This extension introduces new modeling challenges, necessitating significant revisions to both the feasibility test and the execution policy. Despite the optimizations developed in the discrete-time setting, the increased complexity arising from additional conflict types in continuous-time requires further enhancements. To address this, targeted optimizations are proposed, ensuring that the extended framework remains efficient and scalable for large MAPF instances.

Chapter 6 summarizes the key contributions of the thesis and discusses potential directions for future research beyond the current work.

Chapter 2

Preliminaries & Literature Review

This chapter provides the necessary background and literature review for the research presented in this thesis. It begins with fundamental concepts and definitions related to multi-agent systems and Multi-Agent Path Finding (MAPF), followed by a discussion of major extensions and algorithmic approaches. The chapter also reviews representative solvers, including optimal, suboptimal, and hierarchical methods, as well as execution and coordination frameworks from the literature. To help readers navigate through the diverse body of research, a brief taxonomy of related studies is presented, grouping existing works by their methodological characteristics—such as centralized versus decentralized control, discrete versus continuous time, and deterministic versus uncertainty-aware execution. Throughout the chapter, real-world applications such as warehouse automation, autonomous driving, and multi-robot delivery are used as illustrative examples to clarify the abstract concepts and demonstrate their practical relevance.

2.1 Multi-Agent Path Finding (MAPF) Problem

2.1.1 Problem Definition

In a classical multi-agent path finding problem, we consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which the nodes $\mathcal{V}$ represent the locations where agents can occupy and

wait, and the edges $\mathcal{E}$ represent the straight lines between the nodes which agents can traverse. In most applications of a classical MAPF problem, $\mathcal{G}$ is a square grid graph where the nodes $\mathcal{V}$ represent points in the plane with integer coordinates and the edges $\mathcal{E}$ connect adjacent nodes at a unit distance.

There are a set of M agents $\mathcal{A} = \{a_1, \dots, a_M\}$, each of them is abstracted as a point that does not have a geometric shape or size. Task allocation phase is not included in a classical MAPF problem so that every task t_i in a set of tasks $\mathcal{T}$ is pre-allocated to a distinct agent a_i in $\mathcal{A}$ (the number of tasks and agents are the same, M). For each task t_i , there is a start location $s_i \in V$ and a goal location $g_i \in V$ (and usually $s_i \neq g_i$). The start and goal locations of the agents are unique across the agents, which means that for any pair of agents a_i and a_j in $\mathcal{A}$, $s_i \neq s_j$ and $g_i \neq g_j$.

A typical example is a fleet of warehouse robots moving goods between storage racks and packing stations, where each robot must follow a planned route without colliding with others.

A classical MAPF problem is a discrete-time problem, where time is assumed to be discretized into timesteps. In every timestep, each agent is situated in one of the nodes $\mathcal{V}$ in the graph and can perform a single *wait* action or *move* action. Consider a timestep t . Assuming that the location of an agent a_i at timestep t is $v_{i,t} \in \mathcal{V}$, and the location of the agent at timestep $t + 1$ is $v_{i,t+1} \in \mathcal{V}$, an action π can be defined by $\langle v_{i,t}, v_{i,t+1}, t \rangle$. In a *wait* action, the agent remains stationary and stays at node $v_{i,t}$ ($v_{i,t} = v_{i,t+1}$) at timestep $t + 1$, without making any movement. In a *move* action, the agent transitions to an adjacent node via an edge $e \in \mathcal{E}$ connecting nodes $v_{i,t}$ and $v_{i,t+1}$ during timestep t . During the transitions, the inertial constraints are usually ignored, which means that the agents can stop and accelerate instantaneously.

A path p_i of an agent a_i can be expressed by a sequence of actions $\langle \pi_{i,1}, \dots, \pi_{i,n_i} \rangle$, where n_i is the number of actions on the path. The path p_i is feasible if and only if for every timestep $0 \leq t < n_i$, either a *wait* action or a *move* action can be successfully executed by agent a_i (the agent is at its corresponding node $v_{i,t}$ in its action at timestep t). a_i is assumed to be situated in its start location of its allocated task before the plan execution ($v_{i,0} = s_i$), and in the goal location of its

allocated task after the plan execution ($v_{i,n_i} = g_i$). The cost of path p_i can be defined by $|p_i| = n_i$.

2.1.2 Conflicts Definition

The goal of a MAPF problem is to find a solution which is consisted of a path plan $\mathcal{P} = \{p_1, \dots, p_M\}$, which is a collection of paths for all agents $\mathcal{A}$. A path plan is *feasible* (or the MAPF solution is *valid*) if and only if there is no *conflict* between any two single-agent plans (p_i and p_j). The definition of *conflict* varies for different MAPF problems, and some common definitions of conflicts are listed below.

Node conflict Two agents cannot occupy the same node at the same timestep. For example, suppose $v_{i,t} \in p_i$ and $v_{j,t} \in p_j$, if $v_{i,t} = v_{j,t}$, a node conflict occurs.

Edge conflict Two agents cannot traverse the same edge at the same timestep. For example, suppose $v_{i,t}, v_{i,t+1} \in p_i$ and $v_{j,t}, v_{j,t+1} \in p_j$, if the edge $(v_{i,t}, v_{i,t+1})$ and the edge $(v_{j,t}, v_{j,t+1})$ are the same (undirected) edge, an edge conflict occurs. There are some variations of edge conflict, i.e., agents traverse the same edge at the same timestep in different directions; or in contrast, traverse in the same direction.

Following conflict One agent cannot occupy a node at a timestep if the node is occupied by another agent in the previous timestep. For example, suppose $v_{i,t} \in p_i$ and $v_{j,t+1} \in p_j$, if $v_{i,t} = v_{j,t+1}$, it means that agent a_j is following agent a_i and a following conflict occurs.

Swapping conflict A special case of edge conflict and following conflict, where the locations of two agents are swapped in a single timestep. In most problems, it is included in the case of edge conflict.

Cycle conflict A special case of following conflict, where in a same timestep, every agent in a set of two or more agents tries to occupy a node which is occupied by another agent in the set in the previous time step and forms a “cycle” pattern. Swapping conflict is a special case of cycle conflict when only two agents are involved.

There are some interesting relations between these types of conflicts. Forbidding node conflicts implies that edge conflicts in the same direction are also forbidden. Forbidding node conflicts and following conflicts is usually the most constrained restriction in MAPF problems, since it implies that all other conflicts listed above are also forbidden.

When defining a MAPF problem, one should specify which types of conflicts are considered in the problem. For example, in a warehouse environment, two robots attempting to occupy the same storage cell at the same time represent a node conflict, while two robots moving through the same narrow aisle in opposite directions at the same time illustrate an edge conflict. Most prior work on MAPF forbid node conflicts, and a large portion of them also forbid following conflicts.

2.1.3 Objectives

There are various definitions of objectives in MAPF problems, but most of them are finding a *feasible* plan while minimizing a global cumulative cost function.

One widely used cost function is *sum-of-costs* (Eq. 2.1), which is the sum of the timesteps of individual paths of the agents, or the sum of the timesteps when each agent reaches its goal location. *Sum-of-costs* is usually used in search-based MAPF algorithms [2], and it is also known as *flowtime* in some problems. An alternative definition, average sum-of costs, is defined by dividing sum-of-costs by the number of agents M .

$$\text{soc} = \sum_{a_i \in \mathcal{A}} |p_i| \quad (2.1)$$

Another cost function often used is *makespan* (Eq. 2.2), which means the timestep when all agents reach their goal locations. It depends on the timestep when the last agent reaches its goal location. *Makespan* is usually used in compilation-based MAPF algorithms [2].

$$\text{makespan} = \max_{a_i \in \mathcal{A}} |p_i| \quad (2.2)$$

Although *sum-of-costs* and *makespan* are two most commonly used cost functions in MAPF problems, there also exist some other cost functions. For example, we can define the sum of the timesteps that the agents are executing *wait* actions, or the

sum of timesteps that the agents are not in their goal nodes as cost functions. In different MAPF applications, the most suitable cost function can also be different, and this may be one reason that some MAPF algorithms can work on more than one definition of cost function.

2.1.4 Behaviors of Agents at Goal Locations

In most MAPF problems, agents can reach their goal locations at different timesteps, especially when *sum-of-costs* is used as the cost function. The behaviors of agents after they reach their goal locations vary in different problems. The following definitions are the most commonly used in classical MAPF problems.

One behavior is called *staying at goal*, which means that agent a_i remains at its goal node indefinitely after it has executed all actions in its path p_i . When timestep $t > n_i$, the location of a_i is always g_i . The infinite waiting agent can cause node conflicts with other agents that have not executed all actions, and should be considered in the path planning.

Another behavior is called *disappearing at goal*, which means that agent a_i disappears after it has executed all actions in its path p_i . The agent can be completely ignored after timestep $t = n_i$ so that the corresponding MAPF problem is simplified comparing to the problems with the *staying at goal* agent behavior.

Most classical MAPF problems use the *staying at goal* agent behavior, while some recent work start to consider the disappearing at goal agent behavior [12].

2.2 Solvers of MAPF Problem

The first attempt to solve the MAPF problem is to find a solution in a search space of global states. A local state of an agent a_i is represented by a pair consisting of a node v and a timestep t , denoted as $\sigma_i = (v, t)$. The global state is a collection of the local states of all agents, $\{\sigma_i \mid a_i \in \mathcal{A}\}$. As can be seen, the search space of global states in a MAPF problem is exponential in M (number of agents), which makes this kind of solvers usually not time efficient.

Considering that the state space of a single agent a_i is only linear in the graph size $(n_i|\mathcal{V}|)$, in recent research, multi-agent path finding problems are usually solved in a hierarchical form, which means, a typical solver contains at least two layers: low level search and high level search. The low level search aims at finding the path for a single agent under some constraints, while the high level search considers a number of agents, tries to schedule them without conflicts with each other, and calls the low level search to plan the paths for them. The algorithms used in low level search and high level search are rarely the same since the objective of these two searches are usually different.

MAPF problems were shown to be NP-hard [13]. We can use solvers to find an optimal solution if the number of agents is relatively small, while suboptimal solvers are useful for a large number of agents. Some suboptimal solvers are developed from optimal solvers by sacrificing some optimality for a faster runtime [14].

In this section, I introduce a variety of search algorithms and techniques widely used in popular MAPF solvers recently.

2.2.1 Single Agent Path Finding Problem

Single agent path finding algorithms are indispensable in MAPF solvers since the global search algorithms are evolved from them and most high level search algorithms in hierarchical MAPF solvers rely on them to find the path for a single agent.

In most MAPF solvers, single agent path finding algorithms need to consider dynamic obstacles, so the state-of-the-art shortest path algorithms (e.g., Dijkstra’s algorithm, A* search algorithm) cannot be applied directly. These dynamic obstacles are usually generated by the other agents whose paths have been already planned before.

We first define three types of dynamic obstacles, which are node obstacles, edge obstacles and goal obstacles. A node obstacle $(v, t, t + 1)$ indicates that the node $v \in \mathcal{V}$ is occupied from timestep t to $t + 1$. It occurs when another agent a_j executes a *wait* action $\pi = \langle v_{j,t}, v_{j,t+1}, t \rangle$ where $v = v_{j,t} = v_{j,t+1}$. An edge obstacle $(e, t, t + 1)$ indicates that the edge $e \in \mathcal{E}$ is occupied from timestep t to $t + 1$. It

occurs when another agent a_j executes a *move* action $\pi = \langle v_{j,t}, v_{j,t+1}, t \rangle$ where e is an edge is between $v_{j,t}$ and $v_{j,t+1}$. A goal obstacle (v, t, ∞) indicates that the node $v \in \mathcal{V}$ is occupied from timestep t onward infinitely. It only occurs when the MAPF problem uses the *staying at goal* agent behavior, where v is a goal node of another agent, and the agent arrives at its goal node at timestep t .

Consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with a start node $s \in \mathcal{V}$, a goal node $g \in \mathcal{V}$, and a finite set of obstacles $\mathcal{O}$, the objective of the single agent path finding problem is to find a path $p = \langle \pi_1, \pi_2, \dots, \pi_n \rangle$, which represents a sequence of n actions, that avoids conflicts with any obstacles while minimizing the cost (usually path length $|p| = n$).

2.2.1.1 A* Search

A* search is one of the most famous shortest path algorithms. It is well known for its simplicity of implementation, efficiency in execution and optimality of solution.

The procedure of A* search can be explained as a state expansion. Define each search state as a pair $\sigma = \langle v, i \rangle$, where v is the location of the agent, and i means the number of nodes on the path from the start node s and node v (excluding s). Then the initial state of the agent is $\langle s, 0 \rangle$ and the final state of the agent is $\langle g, n \rangle$, where n is the number of nodes on the path from s to g . In each iteration of the algorithm, A* tries to expand a state $\langle v, i \rangle$ based on a cost function $f(\sigma)$, which can be expressed as

$$f(\sigma) = g(\sigma) + h(\sigma), \quad (2.3)$$

where $g(\sigma)$ is the cost of the previously expanded path ($s \rightarrow v$) of the state σ and $h(\sigma)$ is a heuristic function that estimates the cost to expand the path to the goal node ($v \rightarrow g$). Though the heuristic function is problem-specific, we usually set $g(\sigma) = i$ and $h(\sigma)$ as the Manhattan distance between v and g on a grid graph, which is normally used in a classical MAPF problem. The heuristic function should be always no less than the actual cost of the path, so that A* is guaranteed to find an optimal solution (a path with the least cost) in such condition.

The time complexity of A* search mostly depends on the quality of the heuristic function. In the worst case, the search space is exponential in the depth of the

solution, but with a good heuristic, A* can prune away many branches in the search tree.

Most A* search implementations utilize a priority queue, the OPEN list, to store all generated nodes and expand the node with the minimum estimated cost $f(\sigma)$. In each iteration, the state $\sigma = (v, i)$ with the lowest $f(\sigma)$ value is removed from the OPEN list, and new search states are generated for the neighboring nodes of v based on the heuristic function, assigning appropriate f and g values before inserting them into the OPEN list for further exploration. Expanded states are stored in a separate list, the CLOSED list, indicating that they have already been processed. The search concludes when the goal node is selected and removed, allowing the shortest path to be traced. If all nodes are removed without reaching the goal, no valid path exists between s and g .

A* search efficiently finds the shortest path between s and g on a graph $\mathcal{G}$ with static obstacles, but it cannot handle the dynamic obstacles defined in the single-agent path finding problem because it does not account for the time dimension in its search.

2.2.1.2 Space-time A* search

Space-time A* search [15] defines each search state as $\sigma = \langle v, t \rangle$ where t represents the arriving timestep of the state to tackle the dynamic obstacles. The minimum estimated cost $f(\sigma)$ in a state of space-time A* measures the minimal timesteps (or number of actions) required to reach the goal node g , which can be larger than the cost of a normal A* search because *wait* actions can be added into the path. When we expand a state (v, t) , we can generate one more state $(v, t + 1)$ to represent a *wait* action.

Algorithm 1 shows the procedure of the space-time A* search algorithm. Firstly, the root search state σ_{root} is initialized with the start node s and timestep 0 and added into the OPEN list (lines 1-2). In each iteration, the state $\sigma = (v, t)$ with minimal f -value is expanded and added into the CLOSED list (lines 3-6). If the state consisting of the goal node g is expanded, the shortest path from s to g without conflicts with any obstacles in $\mathcal{O}$ is successfully found (lines 7-8). Otherwise, we generate some new states with the adjacent nodes of v plus v itself, and timestep

Algorithm 1: Space-time A*($\mathcal{V}, \mathcal{E}, s, g, \mathcal{O}$)

```

1  $\sigma_{root} \leftarrow \text{State}(v, 0);$ 
2  $OPEN \leftarrow \{\sigma_{root}\}, CLOSED \leftarrow \emptyset;$ 
3 while  $OPEN \neq \emptyset$  do
4    $\sigma \leftarrow$  remove the state with minimal  $f$ -value from  $OPEN$ ;
5    $CLOSED \leftarrow CLOSED \cup \{\sigma\};$ 
6    $(v, t) \leftarrow \sigma;$ 
7   if  $v = g$  then
8     return  $\text{ExtractPath}(\sigma);$ 
9   foreach  $v' \in \{v' \mid \langle v, v' \rangle \in \mathcal{E}\} \cup \{v\}$  do
10     $\sigma' \leftarrow \text{State}(v', t + 1);$ 
11    if  $\sigma'$  does not conflict with any obstacle in  $\mathcal{O}$  then
12       $OPEN \leftarrow OPEN \cup \{\sigma'\};$ 
13 return No solution;

```

$t + 1$ as the arriving timestep (lines 9-10). For each generated state σ' , if it does not conflict with any obstacle in $\mathcal{O}$, we calculate its minimum estimated cost $f(\sigma')$ and add it into the OPEN list for future expansion (lines 11-12). If no shortest path is found after all states in the OPEN list are expanded, the single agent pathfinding problem has no solution (line 13).

When a new state σ' is generated from the expanded state σ (line 10), we define σ to be the parent state of σ' . The $\text{ExtractPath}(\sigma)$ function (line 8) constructs the shortest path by recursively finding the parent state of σ until the root state σ_{root} is found. The reversed sequence of the found states can be transformed into a sequence of actions p , which is the solution of the single agent pathfinding problem.

In line 11 of Algorithm 1, the conflict between a state $\sigma = \langle v, t \rangle$ and all three types of dynamic obstacles in $\mathcal{O}$ can be handled, each with a different method.

Node obstacles can be stored in a reservation table (RT), which is a two-dimensional boolean array of size $|\mathcal{V}| \times t_{max}$, where t_{max} is the maximum timestep of all obstacles. We set $\text{RT}[v][t]$ to true if there exists a node obstacle $(v, t, t + 1)$ in $\mathcal{O}$, or false otherwise. When space-time A* expands the state σ and generates a new state $\sigma' = (v', t + 1)$, σ' is skipped if $\text{RT}[v'][t + 1]$ is true (occupied by a node obstacle). The reservation table is typically implemented using a sparse array for space and time efficiency, as most entries in the table are false.

Edge obstacles can be handled using different approaches depending on the specific definition of conflicts. If following conflicts are considered in the problem, we can simply set both $RT[v][t]$ and $RT[v][t + 1]$ to true if there exists a node obstacle $(v, t, t + 1)$ in $\mathcal{O}$. This approach prevents following conflicts, thereby naturally eliminating all edge conflicts. As a result, edge obstacles do not require special handling since their corresponding edge conflicts can be disregarded. Otherwise, if following is allowed, an edge reservation table of size $|\mathcal{E}| \times t_{\max}$ can be initialized, allowing edge obstacles to be handled in the same manner as node obstacles.

Similarly, goal obstacles can be stored in a goal table (GT), which is a one-dimensional integer array of size $|\mathcal{V}|$. Each entry $GT[v]$ is set to t if there exists a goal obstacle (v, t, ∞) in $\mathcal{O}$, or ∞ otherwise. When generating new states, $\sigma' = \langle v', t + 1 \rangle$ is skipped if $GT[v'] \leq t + 1$.

Space-time A* search can effectively find an optimal solution to the single-agent path finding problem, but the additional time dimension expands the search space, leading to longer search times. To better accommodate the MAPF problem, especially for large instances with a sizable graph $\mathcal{G}$ and numerous agents M , further improvements are necessary.

2.2.1.3 Safe Interval Path Planning (SIPP) Algorithm

Safe Interval Path Planning (SIPP) algorithm is a fast variant of space-time A* search that extends each state's timestep to a safe time interval [16]. A safe time interval $\delta = [t, t')$ at a given node v means that no dynamic obstacles exist within this period at the node. In a safe time interval δ at the node v , SIPP prioritizes the state σ based on the earliest arrival time $t_0 \in [t, t')$, which is the minimum timestep at which the agent reaches node v within $[t, t')$ without conflicting with any dynamic obstacles. To get such states, SIPP employs a “wait and move” strategy, generate states, where it waits for the minimal necessary time in each safe time interval and reaches the next safe interval as early as possible. A state σ' with an arrival time later than t_0 within the same safe interval $[t, t')$ can be pruned without compromising optimality because the safe interval of any new state generated by σ' is a subset of the safe interval of a new state generated by σ .

The efficiency of SIPP is significantly improved compared to space-time A* for several reasons. First, SIPP effectively reduces the search space since the number of safe time intervals is much smaller than the number of timesteps. Empirically, it is rare for a single node to be conflicted by many dynamic obstacles. If a node is affected by N dynamic obstacles, SIPP generates only $N + 1$ safe time intervals. Second, unnecessary search states within the same time interval at a node can be pruned, further reducing the number of state expansions during the search. Since this pruning process does not compromise optimality, SIPP remains an optimal solver for the single-agent pathfinding problem.

2.2.1.4 Focal Search Algorithm

Focal search is a suboptimal variant of A* search where bounded relaxation is used. It can also be applied to space-time A* search and SIPP because it only modifies the heuristics (cost function). While it still uses an OPEN list to store the f -values of the search states in increasing order, the difference is that it maintains another FOCAL list which contains all states whose f -values are no larger than a value. This value can be a constant cost bound defined before the search, or it can be dynamically defined by a factor w times the minimum $f(\sigma)$ value in the OPEN list [17].

Let $f_{min} = \min\{f(\sigma) \mid \sigma \in OPEN\}$, we can write

$$FOCAL = \{\sigma \in OPEN \mid f(\sigma) \leq wf_{min}\}, \quad (2.4)$$

where $w \geq 1$ is a user specified sub-optimality bound; or

$$FOCAL = \{\sigma \in OPEN : f(\sigma) \leq C\}, \quad (2.5)$$

where $C \geq 0$ is a user specified cost bound.

In each iteration, the A* search (or SIPP) always selects the node with the minimal f -value, which is also the first node in the OPEN list. This is the optimal strategy, but it may take too long time to find the best solution in the search. In many suboptimal MAPF solvers, we do not need to obtain the best solution in the low level search. Instead, we only need to find a comparably good solution of the single

agent pathfinding problem by expanding a state in the FOCAL list, which may not be the state with the minimum f -value. We can control how good the solution is by setting the sub-optimality bound w , and we can also set an upper bound to the search by setting the cost bound C .

Focal search can find a suboptimal solution faster than A^* search because it can expand any state in the FOCAL list. Note that the runtime is also dependent on which state in the FOCAL list is chosen to expand. A priority function h_{FOCAL} should be defined to choose a state in the FOCAL list to expand in each iteration.

With the sub-optimality bound w defined before, we have wA^* which uses the priority function

$$h_{FOCAL}(\sigma) = g(\sigma) + wh(\sigma). \quad (2.6)$$

With the cost bound C defined before, we have Potential Search (PS) which uses the priority function

$$h_{FOCAL}(\sigma) = \frac{C - g(\sigma)}{h(\sigma)}. \quad (2.7)$$

Then, in each iteration, we select the state with the minimal h_{FOCAL} value from the FOCAL list for expansion. If the FOCAL list is empty, we fall back to selecting the state with the minimal f -value from the OPEN list as a backup. This ensures that the search process continues even when no promising states are available in the FOCAL list.

2.2.1.5 Better Heuristics with All Pair Shortest Path Algorithm

As mentioned in the A^* search, the quality of the heuristic function $h(\sigma)$ affects much on the time complexity of the search. We need to keep it admissible which means it never overestimates the cost of reaching the goal.

In a grid graph, we usually use the Manhattan distance of two nodes as the heuristic function. Suppose the coordinate of the source node is (x_1, y_1) , and the coordinate of the goal node is (x_2, y_2) , we define the heuristic function as

$$h(\sigma) = |x_1 - x_2| + |y_1 - y_2|. \quad (2.8)$$

This heuristic function is easy to compute and also has a good quality if the grid is not a maze and the number of obstacles is small compared to the number of available nodes. However, in a more complex map, sometimes it is not very efficient to use the simple heuristic function based on the Manhattan distance. Instead, we can pre-calculate the all pair shortest paths between all available nodes before the path planning and use the shortest path length between any two nodes as the heuristic.

The best known all pair shortest path algorithms is the Floyd-Warshall algorithm, which can compute all possible paths in the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ between each pair of nodes in $\mathcal{O}(|\mathcal{V}|^3)$ comparisons. It incrementally improves the estimate on the shortest path between any two nodes until the estimate is optimal. One of the main advantages of the Floyd-Warshall algorithm is its simplicity and ease of implementation.

Suppose there are two nodes v_i and v_j , we define a function $d(v_i, v_j, k)$ where $1 \leq k \leq |\mathcal{V}|$ so that $d(v_i, v_j, k)$ is the shortest distance between v_i and v_j when only nodes in $\{v_1, v_2, \dots, v_k\}$ are used as intermediate nodes in the shortest path. Then the shortest distance between v_i and v_j can be expressed by $d(v_i, v_j, |\mathcal{V}|)$.

For $k = 0$, we can set $d(v_i, v_j, 0) = 0$ if $v_i = v_j$ because the distance between the same node is zero; we can set $d(v_i, v_j, 0) = w(v_i, v_j)$ if there is an edge (v_i, v_j) between v_i and v_j and $w(v_i, v_j)$ is the weight of the edge; otherwise, we can set $d(v_i, v_j, 0) = \infty$ because we cannot find a path between (v_i, v_j) without any other intermediate node.

Dynamic programming can be used to find $d(v_i, v_j, k + 1)$ if we have $d(v_i, v_j, k)$ for any pair of v_i and v_j . If the path of $d(v_i, v_j, k + 1)$ does not go through v_{k+1} , we have $d(v_i, v_j, k + 1) = d(v_i, v_j, k)$; if the path of $d(v_i, v_j, k + 1)$ goes through v_{k+1} , we have $d(v_i, v_j, k + 1) = d(v_i, v_{k+1}, k) + d(v_{k+1}, v_j, k)$. At least one of the condition holds, so

$$d(v_i, v_j, k + 1) = \min\{d(v_i, v_j, k), d(v_i, v_{k+1}, k) + d(v_{k+1}, v_j, k)\} \quad (2.9)$$

for any pair of v_i and v_j .

In each iteration, we calculate $d(v_i, v_j, k + 1)$ for $\mathcal{O}(|\mathcal{V}|^2)$ times, and there are $\mathcal{O}(|\mathcal{V}|)$ iterations ($1 \leq k \leq |\mathcal{V}|$), so the time complexity of the algorithm is $\mathcal{O}(|\mathcal{V}|^3)$. The algorithm is well-suited for MAPF problems on maps of normal size, but as $|\mathcal{V}|$ increases, its efficiency declines, making it less effective for very large maps. For example, the *orz900d* map from the Moving AI MAPF benchmark [2] has a size of 1491×656 with 96,603 reachable nodes, leading to approximately 9×10^{14} calculation steps with Floyd-Warshall algorithm, which is infeasible for a modern PC.

Fortunately, there exists another all-pairs shortest path algorithm, Johnson's algorithm, which combines Bellman-Ford algorithm and Dijkstra's algorithm with a time complexity of $\mathcal{O}(|\mathcal{V}||\mathcal{E}| + |\mathcal{V}|^2 \log |\mathcal{V}|)$. It is generally more efficient than the Floyd-Warshall algorithm when the number of edges $|\mathcal{E}|$ is much smaller than $|\mathcal{V}|^2$ for a large number of nodes $|\mathcal{V}|$. This condition holds in most MAPF problems because they are typically defined on a 2^k -connected grid graph where k is a small number starting from 2, or on a roadmap where each node is connected by only a limited number of roads.

Dijkstra's algorithm is a special form of A* search if we set the heuristic function $h(\sigma) = 0$. If there are negative edge weights in the graph, $h(\sigma)$ may overestimate the distance, which may compromise the optimality of the A* search. Therefore, Dijkstra's algorithm can only find the shortest paths in graphs where all edge weights are non-negative, which is true for MAPF problems. The time complexity of Dijkstra's algorithm is $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)$ if a Fibonacci heap is used to implement the priority queue (the OPEN list). This is also the upper bound of the time complexity of A* search.

Bellman-Ford algorithm is similar to Dijkstra's algorithm (or A* search), while during the state expansion, it does not expand the edges of the minimal cost node in the OPEN list, but simply expands all edges in each iteration. Since there are $|\mathcal{V}|$ nodes to iterate, the time complexity of Bellman-Ford algorithm is $\mathcal{O}(|\mathcal{V}||\mathcal{E}|)$. One useful feature of Bellman-Ford algorithm is that it can detect negative cycles, which cannot be achieved by most other shortest path algorithms. If we expand the edges one more time after $|\mathcal{V}|$ iterations and the shortest distance between two nodes decreases, a negative cycle is detected.

The basic idea of Johnson’s algorithm is to apply Dijkstra’s algorithm $|\mathcal{V}|$ times to find all-pairs shortest paths. In the original version of Johnson’s algorithm, the Bellman-Ford algorithm is first used to ensure that no negative cycle exists in the graph. If a negative cycle exists, the graph cannot be transformed into one with non-negative edge weights, which is a necessary condition for Dijkstra’s algorithm to work. However, this step can be skipped in MAPF problems since negative edge weights are not present. The time complexity of Johnson’s algorithm is $\mathcal{O}(|\mathcal{V}||\mathcal{E}| + |\mathcal{V}|^2 \log |\mathcal{V}|)$ because it involves running Dijkstra’s algorithm $|\mathcal{V}|$ times. This time complexity remains unchanged regardless of whether Bellman-Ford algorithm with a complexity of $\mathcal{O}(|\mathcal{V}||\mathcal{E}|)$ is used.

2.2.2 Optimal Solvers

In this section, we will introduce some optimal MAPF solvers that guarantee global minimization of the objective (e.g., sum-of-costs, makespan). Different optimal MAPF solvers exhibit varying performance across different problem instances; a solver that performs efficiently on one category of problems may not achieve the same efficiency on others. While these solvers ensure both optimality and completeness of the solution, they often struggle to handle large instances, such as those involving larger or more complex maps or a higher number of agents.

2.2.2.1 Global A* Search

The straight-forward global A* search algorithm of the MAPF problem is to define a state space of placing all agents into different nodes in the graph. The state transitions are formed by non-conflicting actions (including *move* and *wait* actions) of all agents [14].

Suppose the search is conducted on a grid graph where each node is connected to at most four other nodes. The branching factor of the search is 5, as the agent can either move to one of its neighboring nodes or remain stationary during each state transition. Now, consider that all M agents are within the same state space; the branching factor for the global A* search becomes at most 5^M in a grid graph, which grows rapidly as M increases. Although the actual branching factor may

be smaller than 5^M due to potential conflicts between agents, especially when the agent density in the graph is high, it remains computationally challenging to find a solution.

The heuristic function used in a global A* search can be the sum of the heuristic functions of all individual agents. For a single agent, the heuristic function can be calculated as the length of the optimal path from its current location to its goal location, ignoring the presence of all other agents. This path length can be efficiently obtained from the result of a pre-calculated all-pairs shortest path [18].

Many enhancements have been studied to speed up the global A* search algorithm, including Independence Detection (ID), Conflict Avoidance Table (CAT) and Operator Decomposition (OD) [18].

We can reduce the number of agents in a problem using Independence Detection (ID), which can provide an exponential speedup over the original global A* search. The key idea is to divide agents into groups, where each agent belongs to exactly one group, and then find an optimal solution for each group separately. If the solutions can be combined without introducing conflicts, these groups are considered independent, effectively breaking the problem into smaller, more manageable instances. Specifically, each agent is initially placed into its own group, and an optimal solution is computed for each group independently. Conflicts between any two groups are then identified, and if conflicts are detected, the groups are merged into a single group. This process continues until no conflicts exist between any groups. Independence Detection reduces the runtime to that of the largest sub-problem, making it highly effective when agents remain separated into smaller groups. However, if all agents are ultimately merged into one group, ID offers no benefit. This concept can be applied to enhance the efficiency of various MAPF solvers.

The Conflict Avoidance Table (CAT) is a dynamic lookup table that records the location and timestep of every agent. When multiple search states in the OPEN list of the global A* search have the same f -value, ties can be broken by referencing the CAT. Specifically, the search state with fewer entries in the CAT is preferred, as it indicates a lower likelihood of conflicts with other agents. This strategy helps guide the search toward conflict-free solutions more efficiently.

Operator Decomposition (OD) is a technique designed to avoid generating surplus nodes during the A* search. Surplus nodes are those with costs greater than the optimal cost of the problem [19], but they cannot be precisely identified during the search. OD addresses this issue by introducing intermediate nodes that are only partially expanded during the search process. These intermediate nodes are not fully developed into regular nodes once the solution is found, thereby significantly reducing the number of non-intermediate surplus nodes and enhancing search efficiency.

2.2.2.2 Increasing Cost Tree Search (ICTS)

The global A* search constructs a search tree that explores all possible locations of the agents simultaneously. In contrast, increasing cost tree search (ICTS) takes a conceptually different approach by building the solution from individual paths, one for each agent. Instead of expanding all agents' states together, it incrementally constructs valid paths for agents, resolving conflicts as they arise [20]. This concept is later adopted by many other solvers such as Conflict-Based Search (CBS) [21] and Priority-Based Search (PBS) [12].

A new search structure called the Increasing Cost Tree (ICT) is introduced, where each node in the ICT contains a vector representing the costs of all individual paths for the agents. Each node corresponds to all possible solutions where the costs of individual paths match those specified in the vector. This structure allows systematic exploration of solutions by incrementally increasing the costs of individual paths until a valid, conflict-free solution is found.

First, a low level search algorithm is run to find the optimal shortest path for each individual agent, ignoring the presence of all other agents. The resulting vector of costs from this initial search serves as the root node of the Increasing Cost Tree. A child node is generated by increasing the cost of one of the agents by a single unit. If individual paths can be found such that there are no conflicts between these paths and their costs match those specified in an ICT state, that node is considered a goal node of the search tree.

The high level search algorithm explores the ICT using a breadth-first search, and for each node, the low level search algorithm checks whether it meets the goal

condition. For each agent, all possible individual paths corresponding to the costs in the node's cost vector are enumerated. The algorithm then iterates through all possible combinations of these individual paths until a valid, conflict-free solution is found.

The number of different paths with the same cost of an agent can be exponential, so they cannot be simply saved with fundamental data structures. A compact data structure called multi-value decision diagram (MDD) [22] can be used to store all paths. Moreover, 2-agent MDD can be built to further improve the performance.

Another strategy to optimize the exponential search space in the low level search is to apply pairwise pruning enhancement. In many cases, the low level search can be avoided by considering a sub-problem involving only a pair of agents. If the sub-problem for any two agents fails, it implies that the global MAPF problem must also fail. Therefore, when a failure is detected in the sub-problem, the low level search can be skipped, and the cost of the failing agent is increased directly. This approach effectively reduces unnecessary computations and accelerates the search process.

2.2.2.3 Conflict Based Search (CBS)

The Conflict-Based Search (CBS) algorithm is an optimal solver for MAPF problems [21]. The high level algorithm conducts a search on a tree structured around conflicts between agents, while the low level algorithm employs techniques discussed earlier (Section 2.2.1) to find the shortest path for each individual agent. Compared to applying A* search directly on the global state space where all agents are considered simultaneously, CBS typically examines significantly fewer states, improving efficiency without compromising optimality.

Similar to the concept of ICTS, CBS divides an MAPF problem into single-agent path finding problems. The difference is that CBS usually contains a large number (exponential to M) of small problems which are relatively simple to solve.

Given a set of M agents $\mathcal{A} = \{a_1, \dots, a_M\}$, we define a constraint by (a_i, v, t) , where agent a_i cannot occupy the node v at timestep t . This constraint also implies that there exists a dynamic node obstacle $(v, t, t + 1)$ in the single agent pathfinding

problem for agent a_i . Similarly, we define a conflict by (a_i, a_j, v, t) where agents a_i and a_j occupy the same node v at timestep t .

A consistent path for agent a_i is a path that satisfies all of its assigned constraints. A consistent solution of the MAPF problem is one where the paths of all agents in $\mathcal{A}$ are consistent with their respective constraints. A valid solution, however, is a solution where there are no conflicts between any of the agents' paths. It is important to note that consistency and validity are distinct concepts: a consistent solution can be invalid if the individual paths satisfy their constraints but still conflict with one another.

CBS maintains a set of constraints and utilizes single-agent pathfinding algorithms (low level search) to find consistent paths for individual agents. Once these consistent paths are found, forming a consistent solution under the current constraints, CBS checks for conflicts within the solution. If conflicts are identified, additional constraints are added to the set, and the low level search algorithms are invoked again to find updated consistent paths that adhere to the new constraints. This process repeats until no conflicts remain in the consistent solution, at which point it becomes a valid solution, and the problem is successfully solved.

CBS can be implemented using a binary tree of search states called Constraint Tree (CT). Each search state σ (CT state) in the tree contains a set of constraints, and a consistent solution $\mathcal{P}$. The heuristic (f -value) of σ is defined by

$$f(\sigma) = \sum_{a_i \in \mathcal{A}} |p_i| \quad (\text{sum-of-costs}), \quad (2.10)$$

$$\text{or } f(\sigma) = \max_{a_i \in \mathcal{A}} |p_i| \quad (\text{makespan}), \quad (2.11)$$

where the heuristics for sum-of-costs and makespan objectives are different, and each $p_i \in \mathcal{P}$ is the individual path of agent a_i in the consistent solution.

The high level search algorithm performs a best-first search on the constraint tree, aiming to minimize the objective (sum-of-costs or makespan) of the solution. This is achieved using an OPEN list, which functions similarly to the OPEN list in an A* search, sorting CT states based on their f -values. If the solution in a CT state is valid, meaning that all agents reach their goal locations without any conflicts, then the state is considered a goal state. Otherwise, if at least one conflict is found

Algorithm 2: CBS($\mathcal{V}, \mathcal{E}, \mathcal{A}$)

```

1 foreach  $a_i \in \mathcal{A}$  do
2    $p_i \leftarrow \text{SingleAgentPathPlanning}(\mathcal{V}, \mathcal{E}, s_i, g_i, \emptyset)$ ;
3  $\mathcal{P} \leftarrow \{p_i \mid a_i \in \mathcal{A}\}$ ;
4  $\sigma_{root} \leftarrow \text{CTState}(\emptyset, \mathcal{P})$ ;
5  $OPEN \leftarrow \{\sigma_{root}\}$ ;
6 while  $OPEN \neq \emptyset$  do
7    $\sigma \leftarrow \text{pop the CTState with the lowest cost from } OPEN$ ;
8    $(constraints, \mathcal{P}) \leftarrow \sigma$ ;
9    $conflict \leftarrow \text{find the first conflict } (a_i, a_j, v, t) \text{ in } \mathcal{P}$ ;
10  if  $conflict \text{ does not exist}$  then
11    return  $\mathcal{P}$ ;
12  foreach agent  $a_i$  in  $conflict$  do
13     $constraints' \leftarrow constraints \cup \{(a_i, v, t)\}$ ;
14     $\mathcal{O} \leftarrow \text{convert } constraints' \text{ into dynamic obstacles}$ ;
15     $p'_i \leftarrow \text{SingleAgentPathPlanning}(\mathcal{V}, \mathcal{E}, s_i, g_i, \mathcal{O})$ ;
16     $\mathcal{P}' \leftarrow (\mathcal{P} \setminus \{p_i\}) \cup \{p'_i\}$ ;
17     $\sigma' \leftarrow \text{CTState}(constraints', \mathcal{P}')$ ;
18    Insert  $\sigma'$  into  $OPEN$ ;
19 return No solution;

```

in the solution of a CT state, it is defined as a non-goal state, and the conflict must be resolved by further expanding the search tree.

The details of the CBS algorithm are presented in Algorithm 2. Initially, individual paths $\mathcal{P}$ are computed using an optimal single-agent pathfinding algorithm without considering dynamic obstacles (lines 1-3). A root CT state σ_{root} is then created with an empty set of constraints and the individual paths $\mathcal{P}$, and it is added to the OPEN list (lines 4-5). In each iteration, the CT state σ with the lowest cost in the OPEN list is expanded, following a process similar to A* search (lines 7-8). Validation is performed by simulating the agents $\mathcal{A}$ with the paths $\mathcal{P}$. If all agents successfully reach their goal locations without any conflict, σ is identified as the goal CT state, and $\mathcal{P}$ constitutes a valid solution (lines 10-11). If a conflict is detected, it must be resolved to continue the search (lines 12-18).

Given a non-goal CT state where a conflict (a_i, a_j, v, t) is detected in its solution, we need to introduce constraints in the child states of this CT state to prevent the same conflict from occurring in future low level searches. Since the solution can be valid if at most one of the agents a_i or a_j occupies node v at timestep t , we

create two child CT states, each containing a new constraint targeting one of the agents (line 13). The left child state resolves the conflict by adding the constraint (a_i, v, t) , prohibiting agent a_i from occupying node v at time t . The right child state resolves the conflict by adding the constraint (a_j, v, t) , prohibiting agent a_j from occupying node v at time t . In each branch, the path of the corresponding agent is replanned using a single-agent pathfinding algorithm (line 15), where the constraints are converted into dynamic node obstacles (line 14). This conversion process is straightforward: for a given agent a_i , the set of dynamic obstacles is defined as $\mathcal{O} = \{(v, t, t + 1) \mid (a_i, v, t) \in \text{constraints}\}$. This transformation ensures that the agent avoids violating constraints during replanning. By systematically branching and replanning, CBS effectively explores conflict-free solutions through incremental constraint addition and resolution.

Note that the original version of CBS can only handle node conflicts, but it can be easily extended to handle following conflicts, as discussed in the Space-time A* search in Section 2.2.1. By generalizing the definitions of conflicts, edge conflicts can also be addressed by converting them into dynamic obstacles. An edge conflict is defined by (a_i, a_j, e, t) , where two agents a_i and a_j attempt to traverse the same edge e between two nodes at timestep t . During conflict resolution, this conflict can be addressed in one of two ways: either by converting it into two separate dynamic node obstacles for agents a_i and a_j , each with a different node, or by introducing a single dynamic edge obstacle. With these extensions, CBS can effectively handle most types of conflicts in MAPF problems.

If a conflict involves more than two agents, there are two strategies to address it. The first strategy is to replace the binary CT with an n -ary tree, allowing the creation of more than two child states. If the conflict involves k agents, we can add $k - 1$ constraints, one for each agent, to resolve the conflict. This approach directly addresses all agents involved in the conflict simultaneously. The second strategy is to resolve the conflict incrementally by addressing only the first two agents involved. If a conflict on the same node v and the same timestep t persists in the solution of a child state with the rest agents, it can be addressed at that state. This incremental approach ensures that the conflict will eventually be resolved, as each level of the tree introduces a constraint for one of the agents.

Since the search is organized within the Constraint Tree, it is unnecessary to store all constraints explicitly in each CT state. Instead, each CT state only needs to record the newly added constraint relative to its parent state. By backtracking through the search tree using the parent states, the complete set of constraints for any given CT state can be reconstructed. During initialization, the root state contains no constraints, so both the constraint set and the parent of the root state can be set to null. This approach reduces memory usage and simplifies the structure of each CT state.

In recent works, various CBS enhancements have been studied to improve its efficiency. These include Bypassing Conflicts (BC) [23], Prioritizing Conflicts (PC) [24], Symmetry Reasoning (SR), Disjoint Splitting (DS) [25], and Improved Heuristics [26, 27]. These enhancements help to further improve the efficiency of CBS by reducing the search space, improving conflict resolution strategies, and enhancing heuristic accuracy. Many of these techniques can also be applied to other CBS-based algorithms.

2.2.2.4 Compilation-based Solvers

Compilation-based solvers focus on transforming the MAPF problem into NP-hard target formalisms that can be efficiently solved using existing algorithms. These formalisms include constraint satisfaction problems (CSP) [28], mixed integer linear programming (MIP) [29, 30], boolean satisfiability (SAT) [31], answer set programming (ASP) [32], and multicommodity flow [33]. By converting MAPF into these well-studied frameworks, powerful general-purpose solvers can be applied to find optimal solutions.

Most compilation-based solvers are similar to the CBS framework as they operate on the principle of incrementally resolving conflicts (laziness). From a compilation perspective, the CBS algorithm can be viewed as a lazy approach that attempts to solve an under-specified problem, relying on the possibility of finding a valid solution even with an incomplete specification [34]. Instead of anticipating all potential conflicts from the beginning, CBS progressively refines the problem by introducing constraints to the problem specification in different search branches only when conflicts are detected. In contrast, compilation-based approaches do not rely on conflict detection for branching; instead, branching is determined by the

solver of the target formalism. This dynamic process gradually builds the encoding of the MAPF problem, which may eventually result in a complete specification of the problem.

Compilation-based solvers can achieve better computation time on the small problem instance compared to search-based solvers (e.g., CBS) [29], but they have some notable disadvantages. First, these solvers are generally designed for classical MAPF problems, and their objectives are usually focused on minimizing the makespan. Adapting them to handle other objectives, such as the sum-of-costs, as well as incorporating complex agent behaviors, dynamic goals, or non-grid environments, can be challenging and may require substantial modifications to the compilation process. Second, the transformation or compilation of the MAPF problem into target formalisms can be computationally expensive. This initial overhead becomes particularly significant when applying these solvers to large problem instances, where the size of the encoding may grow rapidly, adversely affecting performance.

2.2.3 Suboptimal Solvers

Although a well-designed optimal MAPF solver can typically scale to handle around one hundred agents, the NP-hard nature of the MAPF problem means that optimal solvers struggle to efficiently solve instances with a larger number of agents.

Many suboptimal MAPF solvers are based on modifications or relaxations of optimal MAPF solvers and aim to find a solution quickly, allowing the cost of the returned solution to be close to the optimal cost.

Optimal solvers guarantee minimum travel cost, which is important in logistics systems where energy efficiency is critical, while suboptimal solvers favour faster computation suitable for real-time coordination.

There are two types of suboptimal MAPF solvers: unbounded solvers and bounded solvers.

Unbounded suboptimal MAPF solvers include priority-based and rule-based algorithms. These methods are typically very fast and can be applied to large problem

instances but cannot guarantee algorithm completeness or the quality of the solutions found.

Bounded suboptimal MAPF solvers are usually developed from search-based algorithms and guarantee that the solution cost is within a sub-optimality bound [35–37]. Suppose the optimal cost of a MAPF problem is C , then the solution given by a bounded suboptimal solver is within wC , where the sub-optimality bound factor w is a constant. However, they are still exponential-time algorithms with limited scalability, meaning they may run out of time or memory when solving difficult instances, even with a large sub-optimality bound.

2.2.3.1 Prioritized Planning (PP)

The Prioritized Planning (PP) algorithm plans the paths for individual agents one by one in the order of their priorities, while avoiding any conflict with the prior planned paths [38]. It is an unbounded and incomplete suboptimal MAPF solver. Any single agent pathfinding algorithm that deals with dynamic obstacles can be used as the low level search of PP.

In Algorithm 3, we begin by sorting each agent in $\mathcal{A}$ by a distinct and random priority (line 1). The priorities are usually generated by a random shuffling of the agents and assigning incremental priority one by one, but they can also be taken from the input of the user. The set of dynamic obstacles $\mathcal{O}$ is empty initially since no path is planned for any agent (line 2). We proceed by planning paths for agents in order of priority, from highest to lowest (line 3). If the single agent pathfinding algorithm fails to find a solution for an agent’s path, we immediately return “no solution” (lines 4-6). Otherwise, the computed path is converted into dynamic obstacles and added to $\mathcal{O}$ (lines 7-8). The conversion process is similar to CBS, where each constraint can be converted into a dynamic obstacle. Once paths for all agents are successfully planned, they are returned as the final solution (line 9).

However, due to its incompleteness, Prioritized Planning (PP) may fail to find a solution in some instances, as the dynamic obstacles formed by previously planned paths can render future paths unsolvable, even when a valid solution exists but is inconsistent with the assigned priorities.

Algorithm 3: $\text{PP}(\mathcal{V}, \mathcal{E}, \mathcal{A})$

```

1  $\mathcal{A}_p \leftarrow$  sort the agents in  $\mathcal{A}$  by their priorities;
2  $\mathcal{O} \leftarrow \emptyset$ ;
3 foreach agent  $a_i \in \mathcal{A}_p$  do
4    $p_i \leftarrow \text{SingleAgentPathFinding}(\mathcal{V}, \mathcal{E}, s_i, g_i, \mathcal{O})$ ;
5   if  $p_i$  does not exist then
6     return No solution;
7    $\mathcal{O}_i \leftarrow$  convert  $p_i$  into obstacles;
8    $\mathcal{O} \leftarrow \mathcal{O} \cup \mathcal{O}_i$ ;
9 return  $\{p_i \mid a_i \in \mathcal{A}\}$ ;
```

To address this issue, Prioritized Planning with Random Restarts (PP^R) restarts PP with a random priority ordering when a solution cannot be found using the current priorities (on line 6). While PP^R improves the likelihood of finding a solution by resetting the priority orderings, it still cannot guarantee completeness. This limitation arises from two main reasons: (1) PP^R may not randomly generate a set of priorities that leads to a solution within a reasonable amount of time, and (2) some valid solutions may require dynamic or adaptive prioritization, where the priorities of the agents are not fixed but instead change during the planning process.

2.2.3.2 Prioritized-Based Search (PBS)

Prioritized-Based Search (PBS) [12] is a two-level prioritized planning algorithm adapted from search-based solvers such as CBS. It is an unbounded suboptimal algorithm that ensures completeness and demonstrates better optimality compared to the original PP algorithm.

In PBS, the high-level search performs a depth-first search to dynamically construct a priority ordering, building a priority tree that is incrementally expanded as the search progresses. When a conflict is encountered, PBS assigns higher priority to the agents involved in the conflict and continues the search by deepening the tree one agent at a time. It explores alternative branches only if a valid solution cannot be found in the current branch, thereby incrementally constructing a partial priority ordering until a conflict-free solution is identified.

During the low-level search of PBS for an agent a_i , only dynamic obstacles generated by agents with higher priority than a_i are considered. The low-level search computes an optimal path for agent a_i that avoids conflicts with the paths of higher-priority agents. This strategy effectively reduces potential conflicts during subsequent high-level searches, improving the overall efficiency of the algorithm.

2.2.3.3 Priority Inheritance with Backtracking (PIBT)

Priority Inheritance with Backtracking (PIBT) [39] is another priority-based unbounded algorithm that repeatedly performs prioritized planning one by one timestep until it terminates. At each timestep, each agent first updates its unique priority and then sequentially determines its next location in decreasing order of priority, avoiding nodes that have been requested by higher-priority agents. PIBT can operate efficiently even in large-scale problems because the time complexity of a single iteration of PIBT in a classical MAPF problem is $\mathcal{O}(M \log M)$, where M is the number of agents.

PIBT ensures that every agent eventually reaches its goal node within a finite amount of time under the assumption that agents disappear upon reaching their goal nodes. However, PIBT may be incomplete when agents are required to remain at their goal nodes, a situation typically resolvable by other optimal or suboptimal MAPF solvers.

PIBT⁺ [39] is an extended version of PIBT that ensures completeness by invoking another existing MAPF solver with completeness after running PIBT for T_{min} iterations. Here, T_{min} is defined as the maximum distance of the individual agents' paths, assuming no other agents are present. Additionally, PIBT⁺ initializes the priorities of the agents based on the lengths of their individual paths, providing a chance that all agents reach their goals within T_{min} . This hybrid approach ensures that PIBT⁺ can provide a valid solution even when PIBT alone fails due to its incompleteness on some problems.

The original PIBT algorithm plans paths for all agents one timestep at a time, meaning it operates with a unit time window. A generalized version of PIBT, called Windowed PIBT (winPIBT) [40], extends the time window of PIBT to a configurable value, allowing agents to plan multiple steps ahead within each window

while ensuring reachability. It typically produces more efficient paths than PIBT, especially when the window size is appropriately chosen.

2.2.3.4 Hierarchical Cooperative A* (HCA*)

Cooperative A* (CA*) is an unbounded, search-based suboptimal solver for MAPF. It decomposes the problem into a sequence of single-agent A* searches in three-dimensional space-time [41]. After computing the path for each agent, the corresponding states (v, t) are stored as obstacles in a global reservation table, which marks those positions as unavailable for subsequent agents. This table, typically implemented as a sparse hash map, ensures that agents avoid conflicts during planning. However, this decoupled greedy strategy can fail to find a solution since earlier planned agents may block critical paths for later agents and prevent them from reaching their goals.

A key factor influencing CA*'s efficiency is the heuristic function. Simple heuristics like Manhattan distance can perform poorly in large-scale or constrained environments. While precomputed all-pair shortest paths offer optimal heuristics on static maps, they are unsuitable for dynamic or frequently changing maps. To improve the heuristics in CA*, Hierarchical Cooperative A* (HCA*) [41] introduces a hierarchy by omitting the reservation table and time dimension when calculating heuristics. It uses a Reverse Resumable A* (RRA*) search, which runs A* in reverse from the goal to the start. The search terminates as soon as a desired node is expanded, returning the optimal cost to the goal from that node. This allows on-demand heuristic computation on dynamic maps.

However, HCA* has several practical shortcomings. First, if agents adopt the *staying at goal* behavior, they can indefinitely occupy goal locations, potentially blocking other agents whose paths have not been planned, especially in narrow corridors. Second, the initial order in which agents are planned greatly affects solution quality. A good initial order of agents defined by priority may help improve the solution, but a better solution is to dynamically adjust the order during the planning. Third, the complete path of each agent is planned in a three-dimensional space-time search space, which is computationally expensive, especially in dense environments.

Windowed Hierarchical Cooperative A* (WHCA*) [41] addresses these issues by introducing a fixed planning window of w timesteps. Each agent plans only a partial path within this window and executes it before replanning in the next window. While the A* search still targets the final goal location by heuristic guidance from RRA*, only the first w timesteps of the path are recorded in the reservation table. This allows agents to benefit from strong heuristics while avoiding deep exploration of the whole search space.

To support this behavior, when the low level A* search reaches a state $\sigma = (v, w)$, indicating the agent arrives at node v after w timesteps, a special terminal edge is introduced from v directly to the goal node g of the agent. The cost of this edge equals the optimal distance from v to g calculated by RRA*. Rather than expanding neighbor nodes of v in a regular A* search, the search simply expands this terminal edge to finalize the path.

This mechanism also enables WHCA* to continue planning beyond the agent's goal location to address potential blocking by infinite waiting. Specifically, the agent's objective can be shifted from reaching the goal location to completing the window via the terminal edge, potentially vacating the goal location if necessary to accommodate other agents. If the agent has already reached its goal, WHCA* may allow it to remain there for as long as it does not block other agents, or move temporarily if it helps complete global coordination.

There are various enhancements of HCA* and WHCA* as well. For example, the window can be placed only around conflicts in WHCA* and the priority of the agents can be defined by the probability of whether they will be involved in a conflict [42]. The heuristics used in HCA* can also be optimized to reduce the size of the state space and the runtime of building the heuristics [28, 43].

2.2.3.5 Enhanced CBS (ECBS)

Enhanced CBS (ECBS) is a bounded-suboptimal variant of CBS that employs focal search with the same sub-optimality factor w for both high and low level searches, producing a bounded-suboptimal solution with a sub-optimality factor w . It balances solution quality and runtime by allowing a controlled relaxation of optimality. Although ECBS sacrifices some solution quality, it guarantees that the

cost of the solution is within w times the cost of the optimal solution. This trade-off enables significantly faster computations compared to optimal CBS, making ECBS more suitable for large-scale or time-sensitive MAPF problems.

In the low level search of ECBS, by recalling the focal search algorithm used in single-agent pathfinding, we define the FOCAL list as:

$$FOCAL_L = \{\sigma \in OPEN_L \mid f(\sigma) \leq wf_{min}\}, \quad (2.12)$$

where $OPEN_L$ is the OPEN list of the low level search, w is the sub-optimality factor and f_{min} is the minimum f -value among the search states in the OPEN list.

The heuristic function $h_c(\sigma)$ that sorts the CT states in the FOCAL list counts the number of pairs of agents that have at least one conflict between them, which is defined as

$$h_c(\sigma) = |\{(a_i, a_j) \mid a_i, a_j \in \mathcal{A}, a_i \neq a_j, p_i \text{ conflicts with } p_j\}|, \quad (2.13)$$

where $0 \leq h_c(\sigma) \leq \binom{M}{2}$.

In each iteration, the CT state σ with the minimum $h_c(\sigma)$ value in the FOCAL list is expanded. When a solution p_i is found by low level focal search for agent a_i , let f_{min}^i be the minimum f -value in the current OPEN list of the low level search, and f_{opt}^i be the cost of the optimal single agent pathfinding solution, then

$$f_{min}^i \leq f_{opt}^i \leq |p_i| \leq wf_{min}^i \quad (2.14)$$

because all f -values of the search states in the FOCAL list are no more than wf_{min} .

Thus, p_i is a bounded suboptimal solution for the single agent pathfinding problem with sub-optimality factor w , and f_{min}^i serves as a lower bound for the optimal solution's cost.

In the high level search of ECBS, we define the lower bound on the minimum sum-of-costs or makespan for the solution under the constraints of a CT state σ as

$$lb(\sigma) = \sum_{a_i \in \mathcal{A}} f_{min}^i \quad (\text{sum-of-costs}), \quad (2.15)$$

$$\text{or } lb(\sigma) = \min_{a_i \in \mathcal{A}} f_{min}^i \quad (\text{makespan}), \quad (2.16)$$

where each f_{min}^i is the lower bound of the cost of $p_i \in \mathcal{P}$ for CT state σ . $lb(\sigma)$ is used as the heuristic function in the OPEN list instead of the standard $f(\sigma)$ employed by CBS.

We then define the FOCAL list based on the minimum $lb(\sigma)$ of the CT states in the OPEN list as

$$lb_{OPEN} = \min_{\sigma \in OPEN_H} lb(\sigma), \quad (2.17)$$

$$FOCAL_H = \{\sigma \in OPEN_H \mid f(\sigma) \leq w \cdot lb_{OPEN}\}, \quad (2.18)$$

where $OPEN_H$ is the OPEN list of the high level search, and w is the same sub-optimality factor used in the low level search.

Let C^* represent the optimal sum-of-costs or makespan of the MAPF problem, ECBS demonstrates that lb_{OPEN} is also a lower bound of C^* , meaning

$$lb_{OPEN} \leq C^* \leq \sum_{a_i \in \mathcal{A}} p_i \leq w \cdot lb_{OPEN}, \quad (2.19)$$

where $\mathcal{P} = \{p_i \mid a_i \in \mathcal{A}\}$ is a bounded suboptimal solution to the MAPF problem with sub-optimality factor w .

The high-level focal search of ECBS uses the same heuristic function $h_c(\sigma)$ as the low-level focal search to sort the FOCAL list. This design implies that the focal search in the low level serves to accelerate the focal search in the high level. Since $h_c(\sigma)$ measures the number of conflicts, agents find paths with fewer conflicts during low level searches, which also contributes to reducing conflicts at the high level and improves the overall efficiency of ECBS.

2.2.3.6 Explicit Estimation CBS (EECBS)

Although ECBS can solve MAPF problems more efficiently in many problem instances, there are two notable drawbacks associated with the high-level focal search of ECBS [44]. First, ECBS only requires the f -value of the selected CT state σ in the FOCAL list to be within the sub-optimality bound $w \cdot lb_{OPEN}$. However, the f -values of the generated states from σ may exceed this sub-optimality bound. When expanding these states, ECBS can exhibit thrashing behavior, where the search becomes stuck in a local area of the Constraint Tree (CT) and fails to effectively explore other parts of the CT [45]. Second, the lower bound lb_{OPEN} rarely increases during the search. The search state corresponding to lb_{OPEN} typically has a high h_c value, representing a large number of unresolved conflicts. Since the FOCAL list is sorted by h_c values in ascending order, states with high h_c values are only expanded when the FOCAL list is nearly empty. Moreover, because many CT states share the same cost, the FOCAL list is rarely exhausted. As a result, states with the cost lb_{OPEN} are seldom expanded, causing the search progress to stall. This limitation becomes particularly problematic when the optimal solution is not within the initial sub-optimality bound, making it difficult for ECBS to find a solution within a reasonable amount of time.

Explicit Estimation CBS (EECBS) [44] resolves these drawbacks with Explicit Estimation Search (EES) with the sum-of-costs objective [46]. Each search state σ in EES has a more informed but potentially inadmissible cost function $\hat{f}(\sigma)$, which is defined as

$$\hat{f}(\sigma) = f(\sigma) + \hat{h}(\sigma), \quad (2.20)$$

where $f(\sigma)$ is the sum-of-costs of the current consistent plan $\mathcal{P}$ of σ (the cost heuristic function used in CBS), and $\hat{h}(\sigma)$ is the estimated cost-to-go function that contains an admissible cost heuristic $h(\sigma)$ and a distance-to-go heuristic $d(\sigma)$.

The error $\epsilon_d(\sigma)$ of the distance-to-go heuristic $d(\sigma)$, called one-step distance error, is defined as

$$\epsilon_d(\sigma) = d(bc(\sigma)) - (d(\sigma) - 1), \quad (2.21)$$

where $bc(\sigma)$ is the best child state of σ that has the smallest $\hat{f}(\sigma)$, breaking ties in favor of the smallest $d(\sigma)$.

Similarly, the error of the admissible cost function $f(\sigma)$, called one-step cost error, is defined as

$$\epsilon_h(\sigma) = h(bc(\sigma)) - (h(\sigma) - cost(\sigma, bc(\sigma))), \quad (2.22)$$

where $cost(\sigma, bc(\sigma))$ is the cost of moving from state σ to state $bc(\sigma)$.

Finally, the estimation of the true cost-to-go function, $\hat{h}(\sigma)$, is defined as

$$\hat{h}(\sigma) = h(\sigma) + \frac{d(\sigma)}{1 - \overline{\epsilon_d}(\sigma)} \cdot \overline{\epsilon_h}(\sigma), \quad (2.23)$$

where $\overline{\epsilon_d}$ and $\overline{\epsilon_h}$ are the running average of the one-step errors observed so far in EES [47].

Unlike focal search, EES maintains three lists of nodes: CLEANUP, OPEN, and FOCAL. CT states in the CLEANUP list are sorted by the heuristic $lb(\sigma)$, similar to the OPEN list in the high level focal search of ECBS. CT states in the OPEN list are sorted by the cost function $\hat{f}(\sigma)$ described above. The FOCAL list contains the states in the OPEN list that fit within the sub-optimality bound w , which is defined as

$$\hat{f}_{min} = \{\hat{f}(\sigma) \mid \sigma \in OPEN\}, \quad (2.24)$$

$$FOCAL = \{\sigma \in OPEN \mid \hat{f}(\sigma) \leq w \hat{f}_{min}\}. \quad (2.25)$$

In EECBS, the admissible cost and distance-to-go heuristic functions are set to

$$h(\sigma) = 0, \quad d(\sigma) = h_c(\sigma), \quad (2.26)$$

where $h_c(\sigma)$ is the same heuristic function of the FOCAL list used in ECBS.

The inadmissible cost function $\hat{f}(\sigma)$ of EECBS is then defined as

$$\hat{f}(\sigma) = f(\sigma) + \frac{h_c(\sigma)}{1 - \overline{\epsilon_d}(\sigma)} \cdot \overline{\epsilon_h}(\sigma). \quad (2.27)$$

In each iteration of EECBS, the CT state to be expanded is selected from one of three lists: CLEANUP, OPEN, and FOCAL. First, we define lb_{best} as

$$lb_{best} = \min_{\sigma \in CLEANUP} lb(\sigma), \quad (2.28)$$

which is similar to lb_{OPEN} used in the high-level focal search of ECBS. The sub-optimality bound of the optimal solution with sub-optimality factor w is given by $w \cdot lb_{best}$. The lower bound of the cost of the expanded state should not exceed this sub-optimality bound. The selection of the CT state for expansion proceeds as follows:

1. If the cost of the best state in the FOCAL list, σ_{FOCAL} , is within the bound, i.e.,

$$f(\sigma_{FOCAL}) \leq w \cdot lb_{best}, \quad (2.29)$$

then the best state from the FOCAL list is selected.

2. If the above condition is not satisfied but the cost of the best state in the OPEN list, σ_{OPEN} , is within the bound, i.e.,

$$f(\sigma_{OPEN}) \leq w \cdot lb_{best}, \quad (2.30)$$

then the best state from the OPEN list is selected.

3. Otherwise, if neither of the above conditions is satisfied, the best state from the CLEANUP list is selected. This state is the one whose lower bound of cost is equal to lb_{best} .

Some CBS improvements are also incorporated into EECBS, including Bypassing Conflicts (BC) [23], Prioritizing Conflicts (PC) [24], Symmetry Reasoning (SR), and Improved Heuristics [27]. EECBS addresses the two drawbacks of ECBS mentioned earlier and generally achieves a shorter runtime and higher success rate than ECBS across most problem instances. The performance improvement becomes more pronounced as the sub-optimality factor w decreases, indicating its enhanced efficiency in finding high-quality solutions near optimality.

2.2.3.7 MAPF via Large Neighborhood Search (MAPF-LNS)

MAPF via Large Neighborhood Search (MAPF-LNS) combines some of the above techniques and uses Large Neighborhood Search (LNS) as a heuristic for finding good solutions [11]. It starts with an initial solution found by a fast but not optimal MAPF solver such as Prioritized Planning (PP), deletes a neighborhood

(the paths of a certain set of agents) from the solution, and optimizes the paths in the neighborhood by replanning while fixing the remaining part of the solution. MAPF-LNS can tolerate conflicts in the initial solution and try to minimize the number of conflicts in the neighborhood. The reduced problem can be solved by any existing MAPF solver, which is simpler than the original one because only the paths in the neighborhood need to be replanned. The LNS procedure can be executed iteratively, since the new solution found by the previous LNS with a smaller number of conflicts can be used in the next iteration as the initial solution. MAPF-LNS is also an anytime algorithm, which means that it can generate the initial solution fast and returns the best solution it can find with an arbitrary time limit. Empirically it scales to very large numbers of agents and finds near-optimal solutions eventually within reasonable time.

The extended version of the algorithm, MAPF-LNS2, introduces a single agent pathfinding algorithm called Safe Interval Path Planning with Soft Constraints (SIPPS) to solve the reduced MAPF problem within a specified neighborhood. Additionally, new neighborhood selection methods are developed based on SIPPS to further enhance performance. In Chapter 4, I extend MAPF-LNS2 to a continuous-time MAPF algorithm, where more details can be found.

2.3 Extension of Multi-Agent Problems

In recent research, many new multi-agent related problems are defined as extensions to the classical MAPF problem. I will introduce some of them which are related to the research in the following chapters.

2.3.1 Continuous-Time Multi-Agent Path Finding Problem

Classical MAPF problems are discrete-time problems, typically based on the following assumptions: (1) time is discretized into timesteps; (2) the duration of every action is exactly one timestep; (3) at each timestep, every agent occupies exactly one node; and (4) the graph is a 4-connected grid graph. Most of the prior work

introduced in the previous section operates under these assumptions or a subset of them.

However, in real-world applications, these assumptions may not always hold. For example: (1) the duration of each action can vary because the lengths of the edges are not necessarily uniform, and the speeds of different agents can be configurable; (2) an agent may occupy more than one node simultaneously due to its physical size, which can exceed the distance between two adjacent nodes; and (3) the underlying graph could be a roadmap or a k -connected grid graph with diagonal edges or other complex connections between nodes. Addressing these discrepancies requires developing algorithms that are flexible and capable of handling continuous-time dynamics, varying edge lengths, and more complex graph structures.

2.3.1.1 Problem Definition

In Chapters 4 and 5, I work on continuous-time multi-agent problems. In these works, I consider the continuous-time problem for non-point translating agents in a 2-dimensional or 3-dimensional Euclidean space. The agents are constrained to a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which the nodes $\mathcal{V}$ represent the geometric locations where agents can occupy and wait, and the edges $\mathcal{E}$ represent the trajectories (such as straight lines) between the nodes which agents can traverse. We ignore the acceleration and deceleration of the agents, so they can start and stop moving instantaneously.

There are a set of M agents $\mathcal{A} = \{a_1, \dots, a_M\}$, each of which has a given shape and size, and can move with a specified speed. Each agent a_i has a start node $s_i \in \mathcal{V}$ and a goal node $g_i \in \mathcal{V}$. At any time, an agent can be executing either a *wait* action $\pi = (v, t, t')$ or a *move* action $\pi = (e, t, t')$. We can also define $\pi = (x, t, t')$ where x is a node v or an edge e for generalization. When the agent executes a *wait* action, it stays at its current node v from time t to t' for a positive real number of duration $t' - t$. When the agent executes a *move* action, it travels from the current node to an adjacent node along an edge $e \in \mathcal{E}$ from time t to t' , where the duration $t_a(e) = t' - t$ is the length of the edge e divided by the speed of the agent a . We do not use two nodes to represent the edge e , as in the definition of the discrete-time problem, because there may be multiple edges between these two nodes corresponding to different trajectories. Selecting one of

these edges can result in different conflicts during the search since each edge has a different trajectory. Once starting to execute a *move* action, the agent has to continue executing the *move* action until it reaches the adjacent node. When the agent arrives at the adjacent node, it may then execute either a *wait* action to stay at the node or a *move* action to start moving along the next edge.

A path p_i of an agent a_i is a sequence of actions $\langle \pi_{i,1}, \dots, \pi_{i,n_i} \rangle$, where n_i is the number of actions on the path. If the agent starts moving from its start node s_i and executes all actions in the sequence, the agent will reach its goal node g_i and remain at the node indefinitely. The cost $|p_i|$ of the path is the sum of the durations of all the actions in the sequence, $\sum_{j=1}^n t(\pi_{i,j})$. A path plan $\mathcal{P} = \{p_1, \dots, p_m\}$ is a collection of paths for all agents. A *conflict* happens when the shapes of two agents overlap geometrically at the same time. A plan is *feasible* if it contains no conflict and *infeasible* otherwise.

These definitions are similar to those in classical discrete-time problems, but the main differences are the inclusion of action durations and consideration of geometric attributes. The objective of a continuous-time MAPF problem remains unchanged, with the most common objectives still being the minimization of sum-of-costs and makespan. Despite the shift from discrete-time to continuous-time formulations, these objective functions continue to be widely used due to their relevance in measuring effectiveness of the solutions. Currently, relatively little work has been done on continuous-time variations of existing MAPF problems.

2.3.1.2 Continuous-time Conflict Based Search (CCBS)

In the continuous-time MAPF problem, despite the conflicts between agents on waiting at their current nodes, conflicts can occur between agents traversing different edges, as well as between an agent waiting at a node and another agent moving along an edge. Continuous-time conflict-based search (CCBS) [48] solves the problem optimally by considering all types of conflicts (node-node, edge-edge and node-edge conflict) and generating corresponding constraints in the search states. The conflicts are detected and resolved with a geometry-aware conflict detection mechanism, which is agnostic to the CCBS algorithm.

Recall that in CBS, a (node) conflict between two agents a_i and a_j is defined by (a_i, a_j, v, t) , where agent a_i and a_j occupy the same node v at timestep t . In CCBS, the conflicts are defined by actions (*move* or *wait*), instead of nodes and timesteps in CBS. A CCBS conflict is simply defined by (π_i, π_j) , where $\pi_i = (x_i, t_i, t'_i)$ and $\pi_j = (x_j, t_j, t'_j)$ which means that there is a conflict between the agents if agent a_i executes π_i at time t_i and agent a_j executes π_j at time t_j .

When resolving the conflict (π_i, π_j) in CCBS, a CCBS constraint is generated for each agent a_i and a_j . Without loss of generality, the constraint (i, π_i, t_i, t_i^*) is generated for a_i , where π_i is the action of a_i in the conflict and $[t_i, t_i^*)$ is the unsafe interval computed from the conflict. Note that t_i^* is not necessary t'_i or t'_j defined in the actions, but the time interval from the starting time t_i of π_i to a maximal time t_i^* in which executing π_i will conflict with executing π_j .

In the low-level search of CCBS, SIPP is employed as the low-level algorithm, where all constraints are converted into dynamic obstacles. The conversion procedure differs depending on the type of action of the constraint. If the action is a *wait* action at the node v , the safe intervals of v are split accordingly. If the action is a *move* action from node v to v' , some of the generated states may conflict with the constraint. These conflicting states are discarded and not added to the OPEN list of the high-level search during state expansion. More details of the conversion procedure can be found in Chapter 4.

Experimental evaluation shows that conflict detection in CCBS becomes a bottleneck when solving continuous-time MAPF problems, resulting that CCBS struggles to solve problem instances with more than 20 agents. An extended version of CCBS [49] incorporates optimization techniques from CBS, including Disjoint Splitting (DS) [25], Prioritizing Conflicts (PC) [24], and Improved Heuristics [26, 27]. While these optimizations significantly improve the efficiency of CCBS, it still only scales to around 30 to 40 agents on various graphs.

This limitation suggests that optimal solvers for continuous-time MAPF problems may not be practical for real-world applications, where the number of agents often far exceeds these limits. Therefore, developing efficient suboptimal solvers is essential for handling large-scale continuous-time MAPF problems effectively.

2.3.2 Multi-Agent Execution with Uncertainty

In many multi-agent applications such as automated warehouse management and security patrolling, agents carry out tasks by moving from one location to another concurrently. To avoid collisions, we need to carefully plan the paths for the agents and coordinate the execution of the plan. While path planning (MAPF) has been extensively studied [7], designing execution policies for planned paths has received less attention. Robust and effective execution is not only critical but also challenging because plan execution is often accompanied by uncertainties due to unforeseen circumstances.

For example, when a robot breaks down, it may stay at its current location indefinitely; when there are humans ahead, a robot may have to stop to prevent colliding with them. These unexpected events are generally difficult to model or predict. Thus, it is hard to cater for them in the path planning before execution. On the other hand, if these events are not handled properly, they may adversely affect the plan execution of many agents in a cascading manner. When such events occur, one possible solution is to replan the paths for the agents. However, replanning is usually costly and may not even be possible for certain applications.

Some recent MAPF works have attempted to cater for uncertainty in plan execution. Two of the most common definitions of uncertainties are time-based uncertainty and probability-based uncertainty.

2.3.2.1 Time-Based Uncertainty

In a classical MAPF problem, the time taken by an agent to move through one edge in the graph is fixed and predefined before the planning (usually defined as one timestep per edge). However, in real life, many factors may cause the edge traversal to be non-deterministic. Some approaches have been studied to solve the problem by gathering statistics over all possible factors and try to model them, but it is usually very difficult to model all factors. For example, in an automated warehouse, where humans are responsible for packing and robots are responsible for delivering the packages, the time taken by a human to package one box is dependent on too many factors which are not feasible to model. But it is possible to define a time range, with a lower bound and an upper bound of the time spent on each

edge. This is the Multi-Agent Path Finding with Time Uncertainty (MAPF-TU) problem [50].

The problem can be solved by introducing the potential presence of an agent. If we have the lower bound and the upper bound of the possible time spent on each edge and a path of an agent, we can calculate whether the agent can present in each node in each timestep. In the A* search, we can prevent any potential conflict of the agents by preventing the potential presence of any two agents at one node at the same timestep.

On the agent level, we can also define a lower bound and an upper bound of the time it takes to finish moving on a path. The sum of lower bounds or upper bounds of the agents can both be defined as cost, which have different meanings. The sum of lower bounds (optimistic cost) represents the earliest time in which the agents can reach their goals; the sum of upper bounds (pessimistic cost) represents the latest time in which the agents can reach their goals.

Both A*+OD and CBS can be adapted to solve the problem. In A*+OD, the agent with the potential presence having the minimal optimistic bound is selected to avoid potential conflicts. In CBS, constraints can be added by potential conflicts instead of the real conflicts of agents.

Another MAPF problem that considers time uncertainty is called k -robust MAPF [51]. A k -robust plan means that each agent may be delayed up to k timesteps during plan execution and there will still be no conflicts between the agents.

k -robust MAPF can also be solved by Conflict-Based Search (CBS). When detecting k -robust conflicts in CBS, we can simulate the agents along their planned paths for the previous k timesteps and find whether there is any conflict in any of these timesteps. When resolving a k -robust conflict (a_i, a_j, v, t) , it means that there is a value $\Delta t \in \{0, \dots, k\}$ so that a_i occupies v at timestep t and a_j occupies v at timestep $t + \Delta t$. Two different constraints (a_i, v, t) and $(a_j, v, t + \Delta t)$ are generated for each agent to resolve the conflict.

The limitations of these works are that the plan may become invalid when the assumed delay bounds are violated and may be far from optimal in the cost objective if the bounds are loose.

2.3.2.2 Probability-Based Uncertainty

The MAPF with Uncertainty problem [52] and the p -robust MAPF problem [53] pursue plans that can be executed without conflicts with probabilistic guarantees. However, these works rely on the knowledge of the probability that an agent will be delayed when moving between adjacent locations, which is often hard to obtain in practice.

In another Multi-Agent Path Finding with Delay Probabilities (MAPF-DP) problem, uncertainty is defined by the delay probability when an agent moves through an edge. When an agent is going to execute a move action, it will stay in its current node with the delay probability p and traverse the edge otherwise (with probability $1 - p$) [54].

In this problem, we need to find a combination of a MAPF plan and a plan-execution policy with small average makespan, which is the average timestep when all agents reach their goal locations and stay there. If we set all delay probabilities to zero, the problem will become a classical MAPF problem.

The problem can be solved with a modified version of CBS. First, we change the total cost in the CBS search tree into approximate average makespan of the agents, which can be calculated based on the delay probabilities. If the delay probability when an agent moves on an edge is p , the estimated time it takes to pass the edge is $1/(1 - p)$. We can use this estimation as the path length in the low level A* or focal search and find collision free paths based on these estimations.

After the CBS search, a plan execution policy called MCP is designed to avoid conflicts by using a directed graph to capture precedence dependencies among agents, which originates from the Temporal Plan Graph (TPG) proposed by [55]. These graph concepts are extended to the Action Dependency Graph (ADG) by [8].

Sticking to a static ADG/TPG, however, can lead to ineffective execution when facing unexpected delays. Some recent studies [56–59] optimize the ADG/TPG by switching precedence dependencies during plan execution via mixed-integer programming or A* search etc. These approaches generally have high computational

overheads because they strive for a new plan optimizing the cost assuming no further delay, which nevertheless may not be necessary if further delays may occur in the future. Recently, the Bidirectional Temporal Plan Graph (BTPG) was introduced by [60] to initialize some switchable precedence dependencies before the plan execution and then facilitate the switching with a smaller computation time during plan execution.

2.3.2.3 Generalized Uncertainty

As can be seen, existing works on time-based uncertainty and probability-based uncertainty each have their own limitations. A common limitation is their reliance on the specific definition of uncertainty, which is typically derived from particular agent behaviors within real-world applications. Consequently, an algorithm designed for a problem with a predefined type of uncertainty cannot be easily adapted to another problem with a different definition of uncertainty. This is because the core logic of the algorithm is often tightly coupled with the underlying definition of uncertainty, making it inflexible and less generalizable across various scenarios.

An execution framework that can deal with generalized uncertainties in the form of unexpected delays in agent movements which can be simultaneously applied to various scenarios mentioned above is lacking.

2.4 Summary

This chapter reviewed the theoretical foundations and existing research on Multi-Agent Path Finding and its extensions. The definitions and formulations presented here form the basis for the problem settings addressed in later chapters. The review of existing solvers highlighted their strengths and limitations. While optimal solvers ensure solution quality, they struggle with scalability, and suboptimal or hierarchical methods trade optimality for efficiency. Furthermore, most existing approaches assume deterministic agent behavior and synchronized motion, which limits their applicability in uncertain or continuous-time environments. These gaps directly motivate the research in this thesis, which focuses on developing robust

execution frameworks and scalable solvers capable of handling uncertainty in both discrete-time and continuous-time multi-agent coordination.

Chapter 3

Multi-Agent Path Execution with Uncertainty

In many multi-agent applications such as automated warehouse management and security patrolling, agents carry out tasks by moving from one location to another concurrently. To avoid collisions, we need to carefully plan the paths for the agents and coordinate the execution of the plan. While path planning (often known as Multi-Agent Path Finding) has been extensively studied [2, 7, 61], designing execution policies for planned paths has received less attention. Some existing studies address certain aspects of plan execution [8, 9, 62], but they often assume idealized or synchronized motion and do not explicitly handle uncertainties arising from real-world conditions such as motion delays, sensing noise, or communication failures. As a result, current approaches lack general frameworks that can ensure robust, conflict-free, and deadlock-free coordination under uncertainty. This gap motivates the development of the execution framework proposed in this thesis.

For example, when a robot breaks down, it may stay at its current location indefinitely; when there are humans ahead, a robot may have to stop to prevent colliding with them. These unexpected events are generally difficult to model or predict. Thus, it is hard to cater for them in the path planning before execution. On the other hand, if these events are not handled properly, they may adversely affect the plan execution of many agents in a cascading manner. When such events occur, one possible solution is to replan the paths for the agents. However, replanning is usually costly and may not even be possible for certain applications.

In this chapter, we design an execution framework to deal with uncertainties in the form of unexpected delays in agent movements. We make two main contributions. First, we formulate a feasibility problem to check whether all agents can reach their goal locations following their respective paths. We prove that this problem is NP-complete and propose feasibility test algorithms. Second, we develop algorithms to coordinate agent movements in an online fashion and enable as many agents as possible to move concurrently while guaranteeing conflict-freeness and deadlock-freeness. Experimental results demonstrate the effectiveness and efficiency of our algorithms.

3.1 Problem Definition

Consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the nodes in $\mathcal{V}$ correspond to the locations in which agents can stay and the edges in $\mathcal{E}$ corresponds to the connections between nodes along which agents can move. There is a set of M agents $\{a_1, a_2, \dots, a_M\}$. Each agent a_i has a start location $s_i \in \mathcal{V}$, a goal location $g_i \in \mathcal{V}$, and a predefined path p_i from s_i to g_i expressed by a sequence of nodes $v_{i,1} \rightarrow v_{i,2} \rightarrow \dots \rightarrow v_{i,n_i}$ where each $\{v_{i,j}, v_{i,j+1}\} \in \mathcal{E}$, $v_{i,1} = s_i$, $v_{i,n_i} = g_i$ and n_i is the length of the path. To execute a path plan $\mathcal{P} = \{p_1, p_2, \dots, p_M\}$, each agent a_i moves from its start location s_i to its goal location g_i through the path p_i .

Assume time is discretized. In each timestep, an agent executes either a *wait* action or a *move* action. In a *wait* action, the agent stays at its current node until the next timestep. In a *move* action, the agent goes towards an adjacent node, but whether it can arrive at the adjacent node successfully before the next timestep is not guaranteed, which results in the *uncertainty* that we aim to deal with. If the agent fails to reach the adjacent node, it must continue executing the *move* action in the next timestep, i.e., the *move* action cannot be canceled before the agent arrives at the adjacent node.

We consider two classical types of conflicts: node conflicts where two agents stay at the same node at the same timestep; edge conflicts where two agents move along the same edge in opposite directions at the same timestep. In addition, we also consider following conflicts where an agent goes towards a node occupied by another agent in the previous timestep [2], because owing to uncertainty, if the latter agent

fails to move and the former agent succeeds in moving, they may collide at the node. When an agent executes a *wait* action, it occupies the current node where it stays. When an agent executes a *move* action, we consider it occupying both the current node and the adjacent node it is moving to, until the agent arrives at the adjacent node. In path execution, we impose the constraint that any two agents cannot occupy the same node concurrently. Such a constraint addresses all the aforesaid node, edge and following conflicts. We assume no violation of the constraint in the initial (resp. final) state, i.e., the start (resp. goal) locations of all agents are distinct. But it is possible for a start location s_i to be the same as a goal location g_j .

Given a path plan $\mathcal{P}$, we would like to coordinate the *move* and *wait* actions taken by agents during the execution to ensure that all agents eventually arrive at their goal locations. Since we do not assume any prior knowledge on the delays that may arise in the execution, it is difficult to define an objective function directly. Instead, we use the number of agents taking *move* actions as a measure of path execution effectiveness and aim to maximize this number at any time, which intuitively would help optimize classic cost objectives such as makespan or sum-of-costs.

3.2 Path Plan Feasibility Problem

Not all path plans can be successfully executed. A path plan $\mathcal{P}$ is said to be *feasible* if all agents can arrive at their goal locations following the paths; otherwise, it is *infeasible*. We start by investigating the feasibility of a given path plan. As shall be seen later, this is a building block for constructing an effective solution to our path execution problem.

3.2.1 Feasibility and Location Dependency Graph

If the paths in $\mathcal{P}$ do not share any common nodes, the plan is guaranteed to be feasible, as each agent can continuously perform *move* actions without any potential conflicts. If there exist common nodes among different paths, to execute the path plan, we need to decide the order for the agents to occupy common nodes, which can be modeled by a location dependency graph.

We refer to each node $v_{i,j}$ on an agent a_i 's path p_i as a *location state*.¹ The location dependency graph (LDG) is a directed graph $\mathcal{G}_{\text{LDG}} = (\mathcal{V}_{\text{LDG}}, \mathcal{E}_{\text{LDG}})$, where $\mathcal{V}_{\text{LDG}} = \{v_{i,j} \mid 1 \leq i \leq M, 1 \leq j \leq n_i\}$ is the set of all possible location states of the agents, and $\mathcal{E}_{\text{LDG}}$ specifies the precedence constraints for the agents to achieve their location states. $\mathcal{E}_{\text{LDG}}$ consists of four parts.

The first part of $\mathcal{E}_{\text{LDG}}$ is based on the path of each individual agent and includes the edges $\{\langle v_{i,j}, v_{i,j+1} \rangle \mid 1 \leq i \leq M, 1 \leq j < n_i\}$, meaning that agent a_i cannot achieve the location state $v_{i,j+1}$ before $v_{i,j}$.

The second part of $\mathcal{E}_{\text{LDG}}$ is based on the initial location states of the agents: for each agent a_i , if its initial location state $v_{i,1}$ shares the same node in the graph $\mathcal{G}$ with a non-initial location state $v_{i',j}$ of another agent $a_{i'}$, we add an edge $\langle v_{i,2}, v_{i',j} \rangle$ because $a_{i'}$ cannot achieve the location state $v_{i',j}$ before a_i moves away from its start location.²

The third part of $\mathcal{E}_{\text{LDG}}$ is based on the final location states of the agents: for each agent a_i , if its final location state v_{i,n_i} shares the same node in the graph $\mathcal{G}$ with a non-final location state $v_{i',j}$ of another agent $a_{i'}$, we add an edge $\langle v_{i',j+1}, v_{i,n_i} \rangle$ because $a_{i'}$ must pass the location state $v_{i',j}$ before a_i moves to its goal location. Otherwise, once a_i moves to its goal location and stays there infinitely, it will be impossible for $a_{i'}$ to achieve the location state $v_{i',j}$.

The above three parts of $\mathcal{E}_{\text{LDG}}$ are called *predetermined edges* because these orders are fixed in any path execution.

The last part of $\mathcal{E}_{\text{LDG}}$ is to resolve the order of two non-initial and non-final location states from different agents that share the same node in the graph $\mathcal{G}$. Recall that any two agents cannot occupy the same node simultaneously. Thus, for each pair of location states $v_{i,j}$ and $v_{i',j'}$ from two respective agents a_i and $a_{i'}$ sharing a common node, an edge $\langle v_{i,j+1}, v_{i',j'} \rangle$ can be formed if a_i achieves $v_{i,j}$ before $a_{i'}$ achieves $v_{i',j'}$, or an edge $\langle v_{i',j'+1}, v_{i,j} \rangle$ can be formed if $a_{i'}$ achieves $v_{i',j'}$ before a_i achieves $v_{i,j}$.

¹If the path of an agent contains a cycle, the agent has multiple location states referring to the same node in the graph $\mathcal{G}$. With node repetitions allowed, a path is usually called a walk in graph theory.

²We assume that the path of a_i includes at least two location states. If a_i 's path has only one location state, it has the same start and goal locations and does not move. It is impossible for $a_{i'}$ to pass through a_i 's location and the path plan is definitely infeasible.

We refer to these two edges as a pair of *unsettled edges*. To ensure conflict-free path execution, one of them must be chosen and added to the LDG. Note that by definition, different pairs of location states must have distinct unsettled edges. We refer to the LDG with predetermined edges only as a *partial* LDG, and refer to the LDG after choosing one edge from each pair of unsettled edges as a *complete* LDG. The following property is similar to what was stated in [56, 57].

Theorem 3.1. *A path plan is feasible if and only if there exists an acyclic complete LDG.*

Proof. If a complete LDG is acyclic, we can derive a topological ordering of all location states in the LDG. Note that the transition from one location state to the next location state of the same agent represents the execution of a *move* action by the agent. Thus, we can let the agents take *move* actions one at a time to achieve their location states following the topological ordering, and then all agents can arrive at their goal locations without conflicts or deadlocks. Hence, the path plan is feasible.

Conversely, if the path plan is feasible, there exists an execution that allows all agents to reach their goal locations and satisfies the constraints that agents never occupy the same node simultaneously. Thus, for each common node between the paths of two agents, the agents must achieve the corresponding location states sequentially in the execution. This implies that the execution must comply with one of the precedence constraints indicated by each pair of unsettled edges. By choosing all the unsettled edges that the execution complies with, we can construct a complete LDG that is acyclic because the execution is deadlock-free. $\square$

Figure 3.1 shows a path plan for two agents, where a_1 's path is marked in solid lines and a_2 's path is marked in dashed lines. In the LDG, predetermined edges are illustrated by solid lines, and unsettled edges are illustrated by dashed lines. As can be seen, there is only one pair of unsettled edges for the common node V_3 between the two paths. If the edge $\langle v_{2,3}, v_{1,3} \rangle$ is chosen, there is no cycle in the complete LDG. Thus, the path plan is feasible. To execute the path plan, the complete LDG indicates that a_1 should follow a_2 and be at least one node away from a_2 .

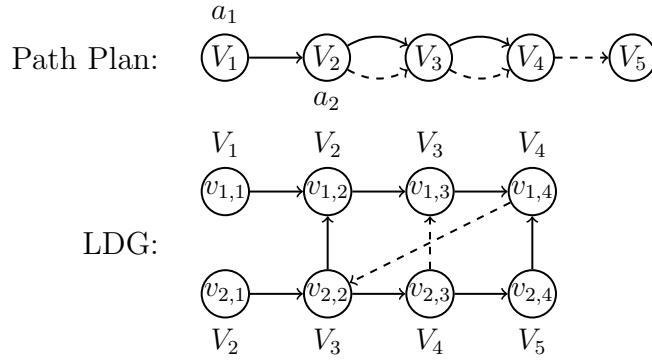


FIGURE 3.1: A feasible path plan and its LDG.

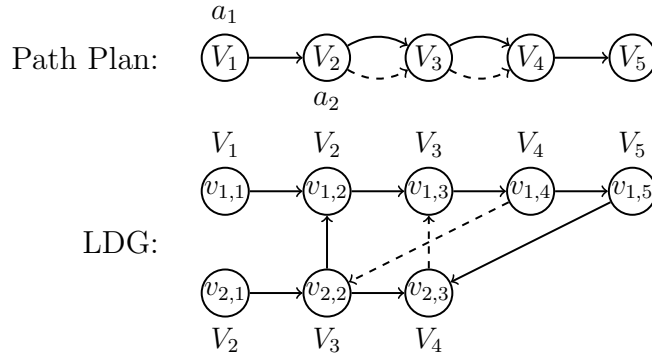


FIGURE 3.2: An infeasible path plan and its LDG.

Figure 3.2 shows another path plan. In the LDG, if the edge $\langle v_{2,3}, v_{1,3} \rangle$ is chosen, a cycle $v_{2,3} \rightarrow v_{1,3} \rightarrow v_{1,4} \rightarrow v_{1,5} \rightarrow v_{2,3}$ is formed. If the edge $\langle v_{1,4}, v_{2,2} \rangle$ is chosen, a cycle $v_{1,4} \rightarrow v_{2,2} \rightarrow v_{1,2} \rightarrow v_{1,3} \rightarrow v_{1,4}$ is formed. Since the complete LDG always has a cycle, the path plan is infeasible.

We remark that our LDG is conceptually similar to the alternative graph introduced for job-shop scheduling with blocking constraints [63]. In job-shop scheduling, different jobs may have operations that must be processed on the same machine, but each machine can process only one operation at a time. In the alternative graph, fixed arcs (similar to our predetermined edges) are created to model the specified order of operations in each job, and alternative arcs (similar to our unsettled edges) are created to model possible precedence relations between operations of different jobs sharing the same machine. However, a job initially does not occupy any machine and after a job finishes, it releases the machine of the last operation. Thus, the alternative graph does not include arcs corresponding to the second and third parts of $\mathcal{E}_{\text{LDG}}$ in our LDG.

3.2.2 Hardness of Path Plan Feasibility

Determining the feasibility of a path plan (named as the path plan feasibility problem) is computationally difficult.

Theorem 3.2. *The path plan feasibility problem is NP-complete.*

Obviously, the path plan feasibility problem is in NP. We reduce the *LSAT* (linear *SAT*) problem to the path plan feasibility problem to prove Theorem 3.2. The classical *SAT* (Boolean satisfiability) problem is to determine whether a formula can be made *true* by assigning appropriate logical values (*true*, *false*) to its variables. If there are at most three variables in each clause, it is called a *3-SAT* problem, which is known to be NP-complete [64]. The *LSAT* problem is a subclass of the *3-SAT* problem, where each clause intersects at most one other clause. Moreover, if two clauses intersect, they have exactly one literal in common. Recently, it has been shown that the *LSAT* problem is also NP-complete [65].

We reduce the *LSAT* problem to the path plan feasibility problem by constructing a path plan $\mathcal{P}$ from the variables and clauses of the *LSAT* formula, and show that determining whether it is possible to transform the partial LDG of $\mathcal{P}$ into an acyclic complete LDG is equivalent to evaluating the satisfiability of the *LSAT* formula.

For each variable x_i in the *LSAT* formula, we construct two *primary* agents a_i and $a_{\bar{i}}$, with paths $S_i \rightarrow V_i \rightarrow G_i$ and $S_{\bar{i}} \rightarrow V_i \rightarrow G_{\bar{i}}$ respectively. The location states of a_i and $a_{\bar{i}}$ are shown in the left dashed frame in the middle of Figure 3.3, where predetermined edges are marked by solid lines and unsettled edges are marked by dashed lines. Due to the common node V_i , there is a pair of unsettled edges $\langle v_{i,3}, v_{\bar{i},2} \rangle$ and $\langle v_{\bar{i},3}, v_{i,2} \rangle$. The choice between these two unsettled edges will be mapped to the assignment of the variable x_i 's value. If the edge $\langle v_{i,3}, v_{\bar{i},2} \rangle$ is chosen, x_i is assigned *true*; if the edge $\langle v_{\bar{i},3}, v_{i,2} \rangle$ is chosen, x_i is assigned *false*.

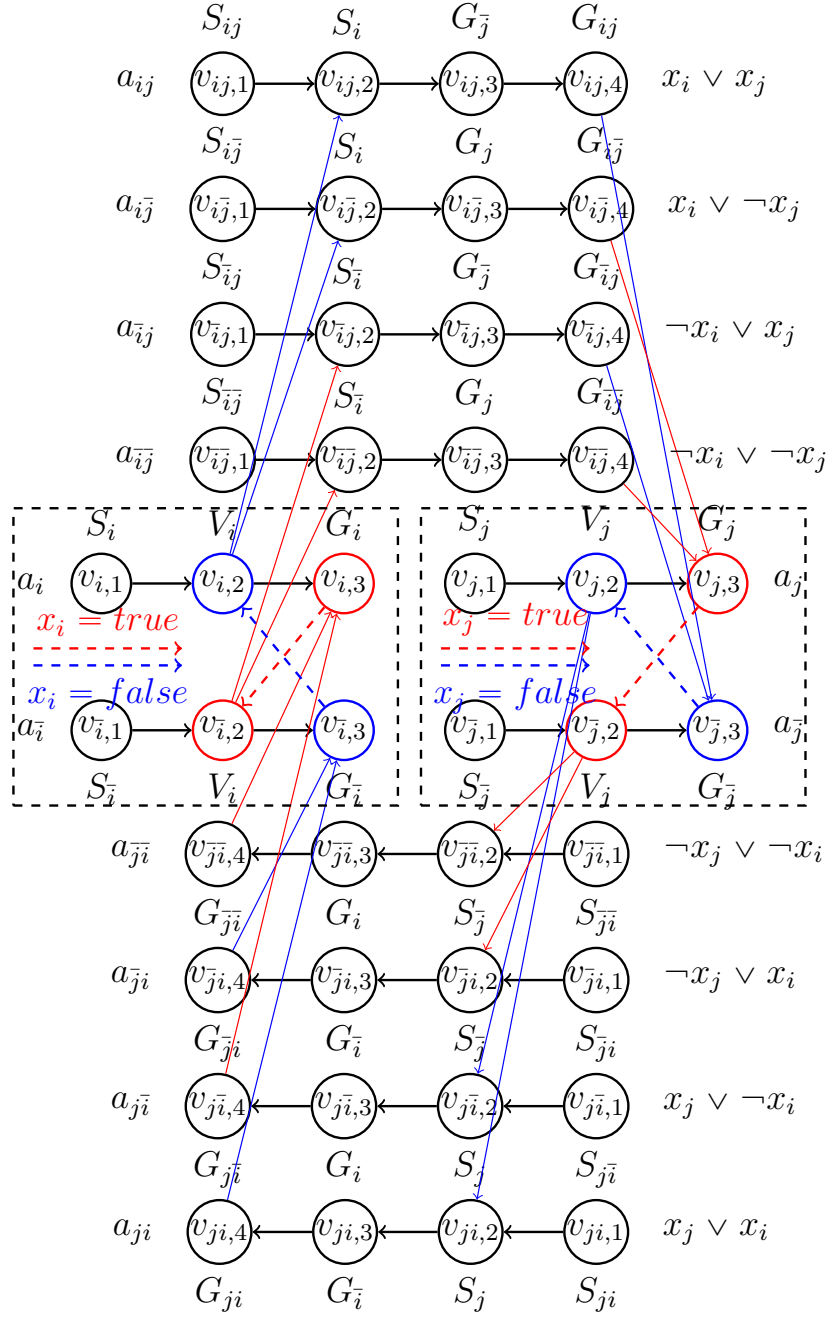
For each pair of variables x_i and x_j in the *LSAT* problem, we construct eight possible *auxiliary* agents a_{ij} , $a_{i\bar{j}}$, $a_{\bar{i}j}$, $a_{\bar{i}\bar{j}}$, a_{ji} , $a_{j\bar{i}}$, $a_{\bar{j}i}$, $a_{\bar{j}\bar{i}}$, and each agent is assigned a path of four nodes. The location states of these agents are shown in the upper and lower parts of Figure 3.3. The start and goal locations of the agents are all different. The pair of the second and third nodes in each path is one of the four combinations of a start location from $a_i/a_{\bar{i}}$ and a goal location from $a_j/a_{\bar{j}}$ or the

four combinations of a start location from $a_j/a_{\bar{j}}$ and a goal location from $a_i/a_{\bar{i}}$. Since the paths of auxiliary agents include only the start and goal locations of primary agents, only predetermined edges (solid lines) will be added to the LDG as shown in Figure 3.3.

Each auxiliary agent represents a possible clause of variables x_i and x_j . For example, the agent a_{ij} represents the clause $x_i \vee x_j$, since its path includes the start location S_i of a_i and the goal location $G_{\bar{j}}$ of $a_{\bar{j}}$. Note that a_{ij} 's path is created in this way because the clause $x_i \vee x_j$ evaluates to *false* only if both x_i and x_j are assigned *false*. Based on the mapping described above, assigning *false* to x_i and x_j implies that the unsettled edges $\langle v_{\bar{i},3}, v_{i,2} \rangle$ and $\langle v_{\bar{j},3}, v_{j,2} \rangle$ are chosen. Thus, a_{ij} 's path shares common nodes with location states $v_{i,1}$ and $v_{\bar{j},3}$, so that the predetermined edges form a path $v_{i,2} \rightarrow v_{ij,2} \rightarrow v_{ij,3} \rightarrow v_{ij,4} \rightarrow v_{\bar{j},3}$ (bold solid lines in Figure 3.3) to link the aforesaid unsettled edges in the LDG. Similarly, the agent a_{ji} represents the clause $x_j \vee x_i$ (which is equivalent to $x_i \vee x_j$). Hence, a_{ji} 's path shares common nodes with location states $v_{j,1}$ and $v_{\bar{i},3}$, so that the predetermined edges form another path $v_{j,2} \rightarrow v_{ji,2} \rightarrow v_{ji,3} \rightarrow v_{ji,4} \rightarrow v_{\bar{i},3}$ (bold solid lines in Figure 3.3) to link the aforesaid unsettled edges in the reverse direction.

If there are m boolean variables in the *LSAT* problem, $2m$ agents are constructed to represent the variables, and at most $4m(m-1)$ agents are constructed to represent the clauses. Thus, the path plan is constructed in polynomial time.

Given an *LSAT* formula, the two primary agents a_i and $a_{\bar{i}}$ for each variable x_i are always included in constructing the path plan. The auxiliary agents for each pair of variables are *selectively* added to the path plan according to the clauses in the *LSAT* formula. In an *LSAT* formula, the clauses can be 2-literal or 3-literal. We consider them separately. For a 2-literal clause involving variables x_i and x_j , we include the two auxiliary agents corresponding to the literals in the clause. For example, if the clause is $x_i \vee x_j$, we include the agents a_{ij} and a_{ji} . If both x_i and x_j are assigned *false*, the clause $x_i \vee x_j$ evaluates to *false* and hence the *LSAT* formula also evaluates to *false*. Accordingly, in the path plan feasibility problem, if the unsettled edges $\langle v_{\bar{i},3}, v_{i,2} \rangle$ and $\langle v_{\bar{j},3}, v_{j,2} \rangle$ are both chosen, a cycle ($v_{\bar{i},3} \rightarrow v_{i,2} \rightarrow v_{ij,2} \rightarrow v_{ij,3} \rightarrow v_{ij,4} \rightarrow v_{\bar{j},3} \rightarrow v_{j,2} \rightarrow v_{ji,2} \rightarrow v_{ji,3} \rightarrow v_{ji,4} \rightarrow v_{\bar{i},3}$) is formed in the LDG.

FIGURE 3.3: LDG of variables x_i and x_j .

For a 3-literal clause involving variables x_i , x_j and x_k ($i < j < k$), we include the three auxiliary agents corresponding to the literals in the clause. For example, if the clause is $x_i \vee x_j \vee x_k$, we include the agents a_{ij} , a_{jk} , a_{ki} . Similarly, if all of x_i , x_j and x_k are assigned *false*, the clause $x_i \vee x_j \vee x_k$ evaluates to *false* and hence the *LSAT* formula also evaluates to *false*. Accordingly, in the path plan feasibility problem, if the unsettled edges $\langle v_{i,3}, v_{i,2} \rangle$, $\langle v_{j,3}, v_{j,2} \rangle$ and $\langle v_{k,3}, v_{k,2} \rangle$ are all chosen, a cycle $(v_{i,3} \rightarrow v_{i,2} \rightarrow v_{ij,2} \rightarrow v_{ij,3} \rightarrow v_{ij,4} \rightarrow v_{j,3} \rightarrow v_{j,2} \rightarrow v_{jk,2} \rightarrow v_{jk,3} \rightarrow v_{jk,4} \rightarrow v_{k,3} \rightarrow v_{k,2} \rightarrow v_{ki,2} \rightarrow v_{ki,3} \rightarrow v_{ki,4} \rightarrow v_{i,3})$ is formed in the LDG.

What remains is to establish the equivalence between a satisfying assignment of the *LSAT* formula and an acyclic complete LDG of the path plan, or alternatively, the equivalence between an unsatisfying assignment and a cyclic complete LDG. First, the following property is quite obvious by the above construction.

Lemma 3.3. *If a variable assignment makes the LSAT formula evaluate to false, choosing the unsettled edges according to the assignments would give rise to a cycle in the complete LDG.*

Proof. If the *LSAT* formula evaluates to *false*, at least one clause evaluates to *false*. For each 2-literal or 3-literal clause in the *LSAT* formula, by the construction of the path plan (and the resulting LDG), if the clause evaluates to false, a cycle is formed in the complete LDG. Thus, if any clause evaluates to *false*, there must exist a cycle in the corresponding complete LDG. Hence, the lemma is proven. $\square$

Next, we prove that the reverse of the above property also holds based on the characteristics of the *LSAT* formula.

Lemma 3.4. *If the choices of the unsettled edges lead to a cycle in the complete LDG, assigning the values to the variables according to the edge choices would make the LSAT formula evaluate to false.*

Proof. We examine the characteristics of a cycle in the complete LDG. By the construction of the path plan, from the path of any auxiliary agent, we can only go to the goal location state $v_{i,3}$ (or $v_{i,3}$) of a primary agent. If the unsettled edge incident to $v_{i,3}$ (or $v_{i,3}$) is not chosen in the complete LDG, we cannot go to any other location state and thus a cycle cannot be formed. If the unsettled edge incident to $v_{i,3}$ (or $v_{i,3}$) is chosen, we can go to the middle location state $v_{i,2}$ (or

$v_{i,2}$) of the other primary agent for the same variable in *LSAT*. From $v_{i,2}$ (or $v_{i,2}$), if we go to the goal location state $v_{i,3}$ (or $v_{i,3}$) of the same agent, we cannot further go to any other location state because the unsettled edge incident to $v_{i,3}$ (or $v_{i,3}$) must not be chosen (since only one edge from each pair of unsettled edges is chosen to form a complete LDG). Thus, to form a cycle, from $v_{i,2}$ (or $v_{i,2}$), we can only go to the path of an auxiliary agent. Hence, if a cycle is formed in a complete LDG, the cycle must involve the paths of the agents sequentially in the pattern of (an auxiliary agent, two primary agents for the same variable, an auxiliary agent, two primary agents for the same variable, $\dots$).

Recall that each auxiliary agent involved corresponds to a 2-literal clause. By the construction of the path plan, either (i) the 2-literal clause appears as a standalone clause in the *LSAT* formula, or (ii) together with the 2-literal clause of an adjacent auxiliary agent involved in the cycle, they form a 3-literal clause in the *LSAT* formula. We refer to this standalone 2-literal clause or this 3-literal clause as an *LSAT* clause involved. Note that an auxiliary agent cannot be mapped to two or more *LSAT* clauses (otherwise, these *LSAT* clauses have two literals in common, which contradicts the definition of the *LSAT* problem). Thus, each auxiliary agent involved in the cycle is mapped to exactly one *LSAT* clause.

By the aforesaid pattern of the cycle, the 2-literal clauses of two adjacent auxiliary agents involved must share a literal (of the variable represented by the two primary agents in between). This implies that if the *LSAT* clauses of two adjacent auxiliary agents are different, they must share a literal. As a result, if there are at least two *LSAT* clauses involved, every clause must share a literal with another clause at each end. If there are exactly two *LSAT* clauses involved, they must share two literals (such as $x_1 \vee x_2 \vee x_3$ and $x_3 \vee x_4 \vee x_1$), which contradicts the definition of the *LSAT* problem because two intersecting clauses must have exactly one literal in common. If there are three or more *LSAT* clauses involved, each of them must intersect two other clauses (such as $x_1 \vee x_2 \vee x_3$, $x_3 \vee x_4 \vee x_5$ and $x_5 \vee x_6 \vee x_1$), which also contradicts the definition of the *LSAT* problem because each clause can intersect at most one other clause. Therefore, there can only be one *LSAT* clause involved.

If the *LSAT* clause involved is a 2-literal clause (such as $x_i \vee x_j$), the cycle involves two auxiliary agents (such as a_{ij} , a_{ji}) and the primary agents for the two variables

in the 2-literal clause (such as $a_i, a_{\bar{i}}, a_j, a_{\bar{j}}$). By the variable assignment corresponding to the unsettled edges chosen in the complete LDG, the 2-literal clause must evaluate to *false*.

Similarly, if the *LSAT* clause involved is a 3-literal clause (such as $x_i \vee x_j \vee x_k$), the cycle involves three auxiliary agents (such as a_{ij}, a_{jk}, a_{ki}) and the primary agents for the three variables in the 3-literal clause (such as $a_i, a_{\bar{i}}, a_j, a_{\bar{j}}, a_k, a_{\bar{k}}$). By the variable assignment corresponding to the unsettled edges chosen in the complete LDG, the 3-literal clause must evaluate to *false*.

Finally, if one clause evaluates to *false*, the *LSAT* formula evaluates to *false*. Hence, the lemma is proven. $\square$

Note that there is a one-to-one correspondence between the variable assignments and the complete LDGs. In addition, an *LSAT* formula is satisfiable if and only if there exists a satisfying assignment. By Theorem 3.1, a path plan is feasible if and only if there exists an acyclic complete LDG. Therefore, by Lemmas 3.3 and 3.4, the *LSAT* problem and the path plan feasibility problem constructed are equivalent.

We note that the NP-completeness result of our path plan feasibility problem is similar to that of the flow control problem in store-and-forward packet switching networks [66]. However, there is an important difference. In the store-and-forward network, packets are removed from the network after reaching their destinations. This allows different packets to share the same destination. Indeed, the problem instances used for reductions in the NP-completeness proofs have packets sharing the same destination [66]. In contrast, we assume that the goal locations of all agents are distinct because agents will stay at their goal locations after reaching them. Hence, the proofs of [66] are not directly applicable to our problem.

3.2.3 Feasibility Test

3.2.3.1 A Basic Algorithm

Based on Theorem 3.1, we design a feasibility test (Algorithm 4). Suppose that the partial LDG is acyclic. The algorithm repeatedly scans through the unsettled edge

pairs ($\mathcal{E}_0$, line 1) and incrementally selects edges from them to produce complete LDGs. The edges selected, called *settled edges*, are recorded in the set $\mathcal{E}_{\text{settled}}$. The test involves a recursive function **FeasibilityTest** that checks whether there exists an acyclic complete LDG based on an input $\mathcal{E}_{\text{settled}}$ (line 5). The feasibility of a path plan is derived by a function call with an empty $\mathcal{E}_{\text{settled}}$ (line 3). The initial LDG in a function call is generated by adding the input $\mathcal{E}_{\text{settled}}$ to the partial LDG (line 6). Then, the function attempts to further select edges from the remaining unsettled edge pairs.

When examining an unsettled edge pair, there are three possible cases: (1) adding either unsettled edge will form a cycle in the LDG; (2) adding one unsettled edge will form a cycle in the LDG, but adding the other edge will not; (3) adding either unsettled edge will not form a cycle in the LDG. In case (1), the function concludes that no acyclic complete LDG exists based on the input $\mathcal{E}_{\text{settled}}$ (lines 19-20). In case (2), the function adds the edge not forming a cycle to $\mathcal{E}_{\text{settled}}$ and the LDG (lines 21-23). Both cases (1) and (2) cut the search space substantially and hence should be prioritized. Thus, in the first function call (by line 3), we skip lines 11-17 (due to line 10) and scan all unsettled edge pairs to find cases (1) and (2) (lines 18-23). After that, a recursive function call is made (line 24), in which we select an arbitrary unsettled edge pair $\langle e, e' \rangle$ and check if it falls in case (3) (lines 11-12).³ If so, we first add e to $\mathcal{E}_{\text{settled}}$ and make a recursive call to see if an acyclic complete LDG can be produced (lines 13-14). If this is so, the function returns a positive result (line 15). Otherwise, we remove e , add e' to $\mathcal{E}_{\text{settled}}$ and make another recursive call (lines 16-17). Only when both calls return negative results, the function concludes that no acyclic complete LDG exists. If the unsettled edge pair $\langle e, e' \rangle$ selected falls in case (1) or (2), we scan all unsettled edge pairs again to find cases (1) and (2) (lines 18-23), and then make another recursive call (line 24).

Theorem 3.5. *Algorithm 4 is sound and complete.*

Proof. Each recursive function call by lines 14 and 17 increases the number of settled edges $|\mathcal{E}_{\text{settled}}|$ by at least one. This is also true for each recursive function call by line 24 (except that in the first function call), because in order for lines 18-23 to be executed, the unsettled edge pair selected by line 11 must fall in case

³In our implementation, unsettled edge pairs are ordered by the 4-tuple of agent identifiers and node indexes involved on their paths. We choose the first edge pair in line 11.

Algorithm 4: Feasibility test of path plan $\mathcal{P}$

```

1  $\mathcal{E}_0 \leftarrow$  unsettled edge pairs in  $\mathcal{P}$ ;
2  $\text{LDG}_0 \leftarrow$  the initial partial LDG of  $\mathcal{P}$ ;
3 return  $\text{FeasibilityTest}(\emptyset)$ ;
4
5 Function  $\text{FeasibilityTest}(\mathcal{E}_{\text{settled}})$ :
6    $\text{LDG} \leftarrow$  add settled edges in  $\mathcal{E}_{\text{settled}}$  to  $\text{LDG}_0$ ;
7   if  $|\mathcal{E}_{\text{settled}}| = |\mathcal{E}_0|$  then
8     return feasible (with complete LDG);
9    $\mathcal{E}_{\text{unsettled}} \leftarrow$  remove unsettled edge pairs that contain settled edges in  $\mathcal{E}_{\text{settled}}$ 
     from  $\mathcal{E}_0$ ;
10  if this is not the first FeasibilityTest call then
11    Select a pair of edges  $\langle e, e' \rangle$  from  $\mathcal{E}_{\text{unsettled}}$ ;
12    if neither of  $e$  and  $e'$  forms a cycle in LDG then
13      Add  $e$  to  $\mathcal{E}_{\text{settled}}$ ;
14      if  $\text{FeasibilityTest}(\mathcal{E}_{\text{settled}})$  is feasible then
15        return feasible (with complete LDG);
16      Remove  $e$  from  $\mathcal{E}_{\text{settled}}$  and add  $e'$  to  $\mathcal{E}_{\text{settled}}$ ;
17      return  $\text{FeasibilityTest}(\mathcal{E}_{\text{settled}})$ ;
18  foreach pair of unsettled edges  $\langle e, e' \rangle$  in  $\mathcal{E}_{\text{unsettled}}$  do
19    if  $e$  and  $e'$  both form a cycle in LDG then
20      return infeasible (together with the two agents that  $e$  and  $e'$  involve);
21    else if one of  $e$  and  $e'$  forms a cycle in LDG then
22       $\bar{e} \leftarrow$  the edge not forming a cycle in LDG;
23      Add  $\bar{e}$  to LDG and  $\bar{e}$  to  $\mathcal{E}_{\text{settled}}$ ;
24  return  $\text{FeasibilityTest}(\mathcal{E}_{\text{settled}})$ ;

```

(1) or (2). The recursion terminates when $|\mathcal{E}_{\text{settled}}|$ reaches $|\mathcal{E}_0|$ (lines 7-8). Thus, the function always returns a result, so Algorithm 4 is complete.

We prove that Algorithm 4 is sound by backward induction on the function call. Note that the initial LDG generated by the input $\mathcal{E}_{\text{settled}}$ (line 6) is guaranteed to be acyclic in any function call.

1) Base case: when $|\mathcal{E}_{\text{settled}}| = |\mathcal{E}_0|$ (line 7), it means that there is no unsettled edge pair left. A positive result (feasible) is returned (line 8) because the LDG is complete and acyclic.

2) Induction: we show that the function result is correct for an input $\mathcal{E}_{\text{settled}}$, given the hypothesis that the result is correct for any input $\mathcal{E}'_{\text{settled}}$ such that $|\mathcal{E}_{\text{settled}}| < |\mathcal{E}'_{\text{settled}}|$. We examine all return clauses in the function except line 8, which is

covered in the base case. In line 20, selecting either edge from an unsettled edge pair forms a cycle in the LDG, so a negative result is returned. In line 24, all newly added settled edges by lines 21-23 have to be selected to avoid cycles, so the problems before and after adding these settled edges are equivalent. Since the recursive call is correct by the induction hypothesis, the result returned in line 24 is correct. In lines 13-17, the function joins the results of two recursive calls (correct by the induction hypothesis) with different selections of settled edges from the same unsettled edge pair. If both results are negative, a negative result is returned because selecting either edge cannot lead to an acyclic complete LDG. Otherwise, at least one acyclic complete LDG is returned so that the path plan is feasible. $\square$

3.2.3.2 Heuristic Feasibility Test

The feasibility test (Algorithm 4) has exponential time complexity in the worst case, as it may need to exhaustively check all possible complete LDGs. However, in practice, the test typically runs efficiently when the number of agents is moderate, as shall be demonstrated in our experiments. This efficiency arises because the algorithm first resolves the edge pairs in cases (1) and (2) before addressing those in case (3). In addition, the test terminates as soon as an acyclic complete LDG is found.

However, as the number of agents increases, it is not surprising that the computational time of Algorithm 4 grows fast. To address this issue, we design a faster heuristic feasibility test by replacing lines 11-17 in Algorithm 4 with a for-loop, as shown in Algorithm 5.

Algorithm 5: The for loop in heuristic feasibility test

```

1 foreach pair of unsettled edges  $\langle e, e' \rangle$  in  $\mathcal{E}_{\text{unsettled}}$  do
2   if  $e$  and  $e'$  both form a cycle in LDG then
3     return infeasible (together with the two agents that  $e$  and  $e'$  involve);
4   else if one of  $e$  and  $e'$  forms a cycle in LDG then
5      $\bar{e} \leftarrow$  the edge not forming a cycle in LDG;
6     Add  $\bar{e}$  to LDG and  $\bar{e}$  to  $\mathcal{E}_{\text{settled}}$ ;
7     break;
8   else /* neither of  $e$  and  $e'$  forms a cycle in LDG */
9     Add  $e$  to  $\mathcal{E}_{\text{settled}}$ ;

```

The for loop in Algorithm 5 is quite similar to lines 18-23 in Algorithm 4. We still need to handle the three possible cases when examining an unsettled edge pair. In case (1), if selecting either edge forms a cycle in the LDG, a negative result is returned (lines 2-3). In case (2), after adding the edge that does not form a cycle to $\mathcal{E}_{\text{settled}}$ and the LDG, the loop breaks (lines 4-7). Hence, at most one edge from case (2) is added during the loop. In case (3), if selecting either edge does not form a cycle, we only attempt to select one of them (lines 8-9), and the other edge will not be selected in the heuristic feasibility test.

The key difference between the heuristic feasibility test and the original one is that the heuristic approach continuously selects edges in case (3) whenever no case (2) is found. Additionally, the heuristic approach eliminates recursive calls to the `FeasibilityTest` function, significantly reducing the total number of function calls. This results in substantial improvement in computational speed. However, the heuristic feasibility test can produce false negative results. It is possible for the original feasibility test to return a feasible outcome, while the heuristic test deems it infeasible. This occurs because, in case (3), if the heuristic test selects one edge from an unsettled edge pair but constructing an acyclic complete LDG requires selecting the other edge, the test will fail to find the latter and hence return an infeasible outcome.

To avoid false negative results in our solution framework, we implement a two-level feasibility test. If the heuristic feasibility test returns an infeasible outcome, we then run the original feasibility test to validate the outcome. Our experiments show that the two-level feasibility test runs faster than the original feasibility test, particularly as the number of agents increases.

3.2.3.3 Check for Cycles in LDG

To check whether adding an unsettled edge $\langle v_{i,j}, v_{i',j'} \rangle$ will form a cycle in the LDG, we can simply run a breadth-first search to see if there is a path from $v_{i',j'}$ to $v_{i,j}$ before the edge is added. The time complexity of the search is $\mathcal{O}(|\mathcal{V}_{\text{LDG}}| + |\mathcal{E}_{\text{LDG}}|)$. Given that thousands of such checks are performed in a single feasibility test, especially as the number of agents increases and the LDG becomes larger, the computational efficiency of the test can be significantly impacted by the efficiency of these checks.

Exploiting the characteristic of the LDG that all location states of an agent form a linear structure based on the path of the agent (i.e., the first part of $\mathcal{E}_{\text{LDG}}$), we develop a more efficient algorithm to determine whether a path exists in the LDG. We use a 2-dimensional connection array $C[i, j][i'] = j^*$ to record the lowest-indexed location state v_{i', j^*} of agent $a_{i'}$, reachable from the location state $v_{i, j}$ of agent a_i . To check if a path already exists from $v_{i, j}$ to $v_{i', j'}$, we simply compare $C[i, j][i']$ with j' : if $C[i, j][i'] \leq j'$, a path exists; otherwise, it does not. Note that by definition, the connection array has the characteristic that $C[i, j][k] \leq C[i, x][k]$ for any i , $1 \leq j \leq x \leq n_i$ and k .

All entries in the connection array C are initially set to $+\infty$, with the exception that for each agent $a_i \in \mathcal{A}$ and $1 \leq j \leq n_i$, we set $C[i, j][i] = j$, as the lowest-indexed location state of a_i reachable from $v_{i, j}$ is itself. As edges are added to $\mathcal{E}_{\text{settled}}$, the entries in the connection array C are updated and their values can only decrease when updated. Note that the entries $C[i, j][i]$ never need to be updated because if they decrease, we have $C[i, j][i] < j$, i.e., from $v_{i, j}$, we can reach a lower-indexed location state of a_i , which means that a cycle is formed. When an unsettled edge $\langle v_{i, j}, v_{i', j'} \rangle$ is added to $\mathcal{E}_{\text{settled}}$ without forming any cycle, the connection array is updated as shown in Algorithm 6. If the lowest-indexed location state of $a_{i'}$ reachable from $v_{i, j}$ is already prior to $v_{i', j'}$, no update is needed since no new connection is formed in the LDG (lines 1-2). Otherwise, two updates are made to the connection array.

First, we update the connection array entries for each agent $a_k \in \mathcal{A} \setminus \{a_i\}$ reachable from agent a_i 's location states $v_{i, x}$ that are prior to or equal to $v_{i, j}$ (i.e., $1 \leq x \leq j$). If $C[i, x][k] > C[i', j'][k]$, since a path $v_{i, x} \rightarrow \dots \rightarrow v_{i, j} \rightarrow v_{i', j'} \rightarrow \dots \rightarrow v_{k, C[i', j'][k]}$ is now formed, we update the lowest-indexed location state of a_k reachable from $v_{i, x}$ to $C[i', j'][k]$ (lines 5-6). In addition, if there exists an x_0 where $1 < x_0 \leq j$ such that $C[i, x_0][k] \leq C[i', j'][k]$, we can infer that $C[i, x][k] \leq C[i, x_0][k] \leq C[i', j'][k]$ for each $x < x_0$. In this case, we can skip updating $C[i, x][k]$ for all $x < x_0$ (lines 7-8).

Second, we update the connection array entries reachable from agents other than a_i and $a_{i'}$. For all the location states $v_{k, x}$ ($1 \leq x \leq n_k$) of each agent $a_k \in \mathcal{A} \setminus \{a_i, a_{i'}\}$, if $C[k, x][i] \leq j$ (meaning that there was a path from $v_{k, x}$ to $v_{i, j}$), we update the lowest-indexed location state for all agents $a_m \in \mathcal{A} \setminus \{a_k\}$ reachable from $v_{k, x}$. If

$C[k, x][m] > C[i', j'][m]$, we update the lowest-index location state of a_m reachable from $v_{k,x}$ to $C[i', j'][m]$ (lines 11-14), because a path $v_{k,x} \rightarrow \dots \rightarrow v_{i,C[k,x][i]} \rightarrow \dots \rightarrow v_{i,j} \rightarrow v_{i',j'} \rightarrow \dots \rightarrow v_{m,C[i',j'][m]}$ is now formed. Similarly, if $C[k, x_0][i] > j$ for some x_0 (no path existed from v_{k,x_0} to $v_{i,j}$), a path from $v_{k,x}$ to $v_{i,j}$ cannot exist either for any $x > x_0$. In this case, we can skip the updates for all $x > x_0$ (lines 15-16).

The worst-case time complexity of updating the connection array (Algorithm 6) is $\mathcal{O}(|\mathcal{A}||\mathcal{V}_{\text{LDG}}|)$, while the time complexity of querying a path in the LDG is $\mathcal{O}(1)$. Although the update complexity can be higher than that of a breadth-first search, the number of updates is often significantly lower than the number of path queries in a single call to **FeasibilityTest** in Algorithm 4, because unsettled edge pairs in case (3) (adding either unsettled edge will not form a cycle in the LDG) do not lead to any update to the LDG (lines 18-23 of Algorithm 4). Therefore, checking cycles with the connection array can significantly reduce the overall computational time of Algorithm 4. Note that Algorithm 6 is one-way, meaning that the update cannot be canceled if an edge is removed. To address this, our implementation backs up the connection array before performing an update if there is a possibility of reverting it later (line 16 of Algorithm 4). Since the required space for the connection array is relatively small ($\mathcal{O}(|\mathcal{A}||\mathcal{V}_{\text{LDG}}|)$), the space and time spent on backup and recovery operations are both negligible.

3.2.3.4 Group Agents in LDG

If there is no predetermined edge between different agents (i.e., in the second and third parts of $\mathcal{E}_{\text{LDG}}$), a very simple method can be used to select the unsettled edges and construct an acyclic complete LDG. First, we assign distinct priorities to all agents (e.g., in the order of their indexes). For each unsettled edge pair, we select the edge that starts from the higher-priority agent and ends at the lower-priority agent. Since no edge is directed from a lower-priority agent to a higher-priority one, the resulting complete LDG must be free of cycles.

To extend this method to deal with predetermined edges, we can reassign priorities to the agents and reuse the same method by utilizing an agent dependency graph (ADG). In the ADG, the nodes represent agents, and the edges are created based

Algorithm 6: Update C when $\langle v_{i,j}, v_{i',j'} \rangle$ is added

```

1 if  $C[i, j][i'] \leq j'$  then
2   return;
3 for  $a_k \in \mathcal{A} \setminus \{a_i\}$  do
4   for  $x \leftarrow j$  downto 1 do
5     if  $C[i, x][k] > C[i', j'][k]$  then
6        $C[i, x][k] \leftarrow C[i', j'][k]$ ;
7     else
8       break;
9 for  $a_k \in \mathcal{A} \setminus \{a_i, a_{i'}\}$  do
10  for  $x \leftarrow 1$  to  $n_k$  do
11    if  $C[k, x][i] \leq j$  then
12      for  $a_m \in \mathcal{A} \setminus \{a_k\}$  do
13        if  $C[k, x][m] > C[i', j'][m]$  then
14           $C[k, x][m] \leftarrow C[i', j'][m]$ ;
15      else
16        break;
```

on the predetermined edges in the LDG. For each predetermined edge from agent a_i to agent a_j in the LDG, we add a corresponding edge $\langle a_i, a_j \rangle$ in the ADG. If there is a path from a_i to a_j in the ADG, the priority of a_i should be assigned higher than that of a_j . However, if there is also a path from a_j to a_i , it creates a conflict in priority assignment, requiring us to assign the same priority to both a_i and a_j .

When two agents share the same priority, they can be grouped together. A strongly connected component in the ADG naturally corresponds to a group of agents, because by definition, each node (agent) in a strongly connected component is reachable from every other node (agent) in the component. We use Tarjan's algorithm [67] to divide the ADG into strongly connected components (agent groups) and then assign a distinct priority to each group of agents. For unsettled edge pairs between agents in different groups, the edges can be selected directly based on group priority. In the feasibility test, we only need to consider unsettled edge pairs between agents within the same group. Furthermore, the unsettled edge pairs in each group can be handled separately, as the selection of edges within one group

does not affect the edge selection in another group, since the selection of cross-group edges based on group priority ensures that no cycle can be formed in the LDG between different groups. Thus, grouping agents in the ADG can reduce the number of unsettled edge pairs to be handled by the feasibility test (Algorithm 4).

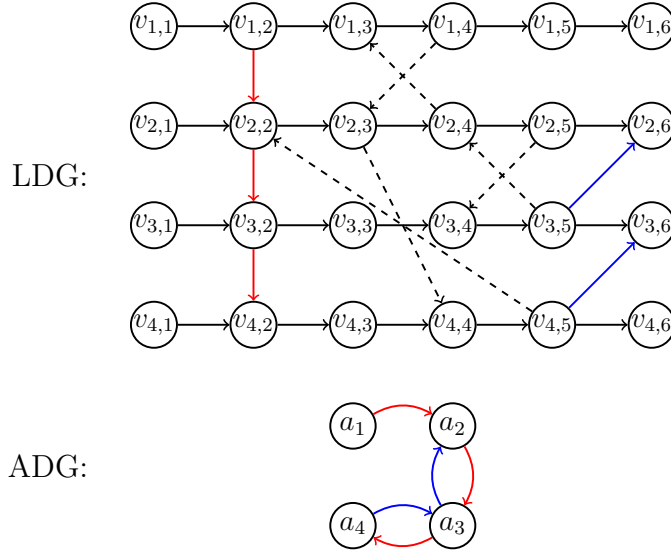


FIGURE 3.4: Grouping the agents in a LDG with its ADG.

Figure 3.4 illustrates an example LDG for four agents and its corresponding ADG. In the LDG, predetermined edges are represented by solid lines and consist of the first three parts of $\mathcal{E}_{\text{LDG}}$. The predetermined edges colored red (e.g., $\langle v_{1,2}, v_{2,2} \rangle$) correspond to the second part of $\mathcal{E}_{\text{LDG}}$, related to the initial locations of agents, while the predetermined edges colored blue (e.g., $\langle v_{3,5}, v_{2,6} \rangle$) represent the third part of $\mathcal{E}_{\text{LDG}}$, related to the goal locations of agents. As seen in the corresponding ADG, $\{a_1\}$ and $\{a_2, a_3, a_4\}$ form two strongly connected components, grouping the agents accordingly. We can assign a higher priority to group $\{a_1\}$ and a lower priority to group $\{a_2, a_3, a_4\}$, since there is a path from the former to the latter in the ADG. This allows us to select unsettled edges between a_1 and other agents directly, since they belong to different groups. For the unsettled edge pair $\langle v_{1,4}, v_{2,3} \rangle$ and $\langle v_{2,4}, v_{1,3} \rangle$, we select the edge $\langle v_{1,4}, v_{2,3} \rangle$ because a_1 has a higher priority than a_2 . The remaining unsettled edge pairs cannot be resolved at this point and should be passed to Algorithm 4 to complete the feasibility test.

3.3 Coordinating Path Execution

Assuming that the initial path plan is feasible (which naturally holds for the output of any MAPF solver), we now present our solution to coordinate the *move* and *wait* actions taken by agents and ensure that all agents can eventually arrive at their goal locations. In this work, we focus on a centralized solution where the agents can sense changes in their location states and send updates to a central controller. Developing distributed solutions is left for future work.

We partition all agents into two sets: unblocked agents U (those taking *move* actions) and blocked agents B (those taking *wait* actions). Recall that a *move* action cannot be canceled before the agent arrives at the adjacent node. Thus, an unblocked agent remains in U until it arrives at the next node on its path. Upon its arrival, we need to decide whether to block the agent from taking a further *move* action. If multiple agents in U arrive at the next nodes on their respective paths at the same timestep, decisions need to be made for all these agents. In addition, we need to decide whether to unblock any agents in B and allow them to take *move* actions. For simplicity, we assume that an agent arriving at the next node on its path is moved from U to B by default. Then, given the current partitions U and B , we focus on designing an algorithm to determine which agents in B to unblock.

If we unblock a set of agents $\Delta U \subseteq B$, the partitions become $U \cup \Delta U$ and $B \setminus \Delta U$. Recall that a blocked agent occupies its current node only, while an unblocked agent occupies both nodes connected by an edge. To guarantee conflict-freeness, we require that the nodes occupied by different agents are all distinct. To ensure deadlock-freeness, we require that if all agents in $U \cup \Delta U$ arrive at their next nodes, the *residual* path plan is feasible. Note that owing to uncertainty, we do not know when these agents will arrive at their next nodes. If the above requirement is satisfied, it also guarantees that the residual path plan is feasible if any subset of the agents in $U \cup \Delta U$ arrive at their next nodes, because we can always wait until the remaining agents in $U \cup \Delta U$ arrive at their next nodes before unblocking any new agent.

For example, in Figure 3.5 (left), if all four agents are unblocked initially and arrive at the next nodes on their paths, none of them can move any further since a

deadlock is formed. As shown in Figure 3.5 (right), there is a cycle in the (partial) LDG of the residual path plan, so it is infeasible. Thus, at most three agents can be unblocked initially.

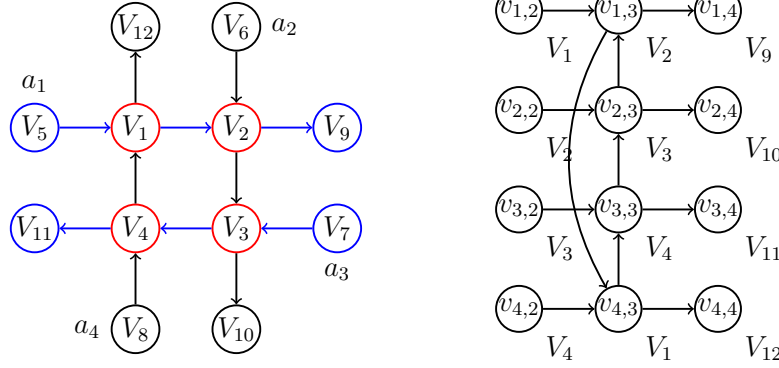


FIGURE 3.5: A path plan (left) and the LDG of the residual path plan if all four agents are unblocked initially (right).

To maximize the path execution effectiveness, the objective of our algorithm is to maximize $|\Delta U|$ (the number of agents to unblock) while meeting the above requirements.

3.3.1 Naive Approach

A naive approach (Algorithm 7) is to iterate through all subsets of B in decreasing order of size and check each subset against the above requirements. Once a subset satisfying the requirements is found, it gives the largest set of agents to unblock. However, the number of all possible subsets of B is $2^{|B|}$, which means that in the worst case, this approach needs to conduct an exponential number of feasibility tests. Even if the largest set of agents has a size i close to $|B|$, we still need to perform at least $\binom{|B|}{i+1} = \frac{|B|!}{(i+1)!(|B|-i-1)!} \approx |B|^{|B|-i-1}$ feasibility tests.

Algorithm 7: Naive approach to unblock agents

```

1 for  $i = |B|$  downto 1 do
2   foreach  $\Delta U \subseteq B$  where  $|\Delta U| = i$  do
3      $\mathcal{P} \leftarrow$  the residual paths of the agents  $\mathcal{A}$  if all agents in  $U \cup \Delta U$  arrive at
       their next nodes;
4     if ( $\mathcal{P}$  is feasible) then
5       return  $\Delta U$ ;

```

3.3.2 Semi-naive Approach

Instead of considering all agents in B , some agents can definitely be unblocked or must be kept blocked. We can first identify these agents (Algorithm 8) and then decide the remaining agents in B with the naive approach.

First, for each agent $a_i \in B$, if the next node v on its path is not occupied and does not appear in the residual path of any other agent, a_i can definitely be unblocked (lines 3-6).

Second, for each agent $a_i \in B$, if the next node v on its path is currently occupied by another agent a_j , a_i cannot be unblocked. This occurs when a_j is taking a *wait* action at node v , or a_j is taking a *move* action from v to an adjacent node, or a_j is taking a *move* action from an adjacent node to v . All identified agents a_i to keep blocked are recorded in the set B_K (lines 7-8).

Next, for each agent $a_i \in B$, if the next node v on its path is its goal location and v also appears in the residual path of another agent a_j , a_i cannot be unblocked. This is because if a_i arrives at v first, it will occupy v infinitely and prevent a_j from passing through v . All identified agents a_i to keep blocked are recorded in the set B_K (lines 9-10).

Finally, we set $B' = B \setminus B_K$ and examine whether these remaining agents can be unblocked with the naive approach by substituting B with B' in Algorithm 7.

The semi-naive approach reduces the size of the blocked agent set B , while still providing the optimal solution to maximize the size of the unblocked agent set. However, it encounters the same issue as the naive approach when the number of agents increases because B' can remain quite large in such cases.

3.3.3 Heuristic Approach

To further optimize the computational efficiency of coordination, we develop a heuristic algorithm (Algorithm 9) to determine whether the agents in B can be unblocked. This algorithm runs much faster than both the naive and semi-naive approaches. Although the heuristic approach does not guarantee an optimal solution,

Algorithm 8: Identify agents that can definitely be unblocked or must be kept blocked

```

1  $B_K \leftarrow \emptyset;$ 
2 foreach agent  $a_i \in B$  do
3    $v \leftarrow$  the next node on  $a_i$ 's path;
4   if  $v$  is not occupied and does not appear in the residual path of any other agent
     then
5      $U \leftarrow U \cup \{a_i\};$ 
6      $B \leftarrow B \setminus \{a_i\};$ 
7   if  $v$  is occupied by another agent then
8      $B_K \leftarrow B_K \cup \{a_i\};$ 
9   if  $v$  is  $a_i$ 's goal location and  $v$  appears in the residual path of another agent
     then
10     $B_K \leftarrow B_K \cup \{a_i\};$ 
11  $B' \leftarrow B \setminus B_K;$ 

```

empirical results show that it performs nearly as well as the optimal approaches (naive and semi-naive).

Before the execution of Algorithm 9, we run Algorithm 8 to identify agents that can definitely be unblocked or must be kept blocked, and set $\Delta U = B \setminus B_K$. Then we set the initial location states of all agents in $U \cup \Delta U$ to the next nodes on their respective paths and run the feasibility test. In our implementation, to improve efficiency, the feasibility test uses the previously found acyclic complete LDG (in the last run of Algorithm 9) as hints for choosing which edge to test first from a pair of unsettled edges (line 13 of Algorithm 4 and line 9 of Algorithm 5). If the residual path plan is feasible, we can safely unblock the agents ΔU (lines 3-5). Otherwise, the feasibility test must return two agents involved in the cycles found in the LDG (line 20 of Algorithm 4). Typically, between these two agents, at least one agent a comes from ΔU , since the residual path plan before unblocking ΔU was feasible. In this case, we remove a from ΔU to break cycles and check the feasibility of the residual path plan again. This process is repeated until the residual path plan becomes feasible (lines 2-7). If both agents returned from the feasibility test are not in ΔU , we simply set ΔU to an empty set (lines 8-9).

In the special case that both U and the resultant ΔU are empty, we need to pick at least one agent to unblock in order to proceed with the path execution. To do so, we examine each agent in $B \setminus B_K$ and check whether unblocking it alone leaves the

residual path plan feasible (lines 11-15). Since the path plan is feasible, at least one agent can be unblocked.

Algorithm 9: Heuristic approach to unblock agents

```

1  $\Delta U \leftarrow B \setminus B_K$ ;
2 while  $\Delta U \neq \emptyset$  do
3    $\mathcal{P} \leftarrow$  the residual paths of the agents  $\mathcal{A}$  if all agents in  $U \cup \Delta U$  arrive at their
   next nodes;
4   if ( $\mathcal{P}$  is feasible) then
5     break;
6   if an agent returned from feasibility test is in  $\Delta U$  then
7     remove the agent from  $\Delta U$ ;
8   else
9      $\Delta U \leftarrow \emptyset$ ;
10 if  $U \cup \Delta U = \emptyset$  then
11   foreach agent  $a_i$  in  $B \setminus B_K$  do
12      $\mathcal{P} \leftarrow$  the residual paths of the agents  $\mathcal{A}$  if all agents in  $U \cup \{a_i\}$  arrive at
     their next nodes;
13     if ( $\mathcal{P}$  is feasible) then
14        $\Delta U \leftarrow \{a_i\}$ ;
15       break;
16 return  $\Delta U$ ;

```

Algorithm 9 ensures that (1) agents taking *move* actions do not collide with each other or with agents taking *wait* actions; and (2) if all or any subset of *move* actions being executed are completed, the residual path plan remains feasible. Thus, running Algorithm 9 repeatedly guarantees that the path execution is conflict-free and deadlock-free.

3.4 Experimental Evaluation

We conduct experiments on four different maps. The first map is room-32-32-4 (Figure 3.6a), which consists of 8 rows and 8 columns of 3×3 rooms separated by walls, with 1-unit-wide corridors (doors) connecting the rooms. This map has more shared locations along agent paths compared to other maps of similar size, as agents must pass through these narrow corridors. The second map is warehouse-10-20-10-2-1 (Figure 3.6d), simulating the layout of a real warehouse with storage racks. The third map is random-32-32-20 (Figure 3.6b), where 20% cells of the grid

map are randomly positioned obstacles, but it is guaranteed to be connected so that any location can be reached from any other location. The above three maps are taken from the Moving AI MAPF benchmark [2]. The fourth map random-32-32-30 (Figure 3.6c, generated by us) is similar to random-32-32-20 but with 30% cells of the grid map being randomly positioned obstacles.

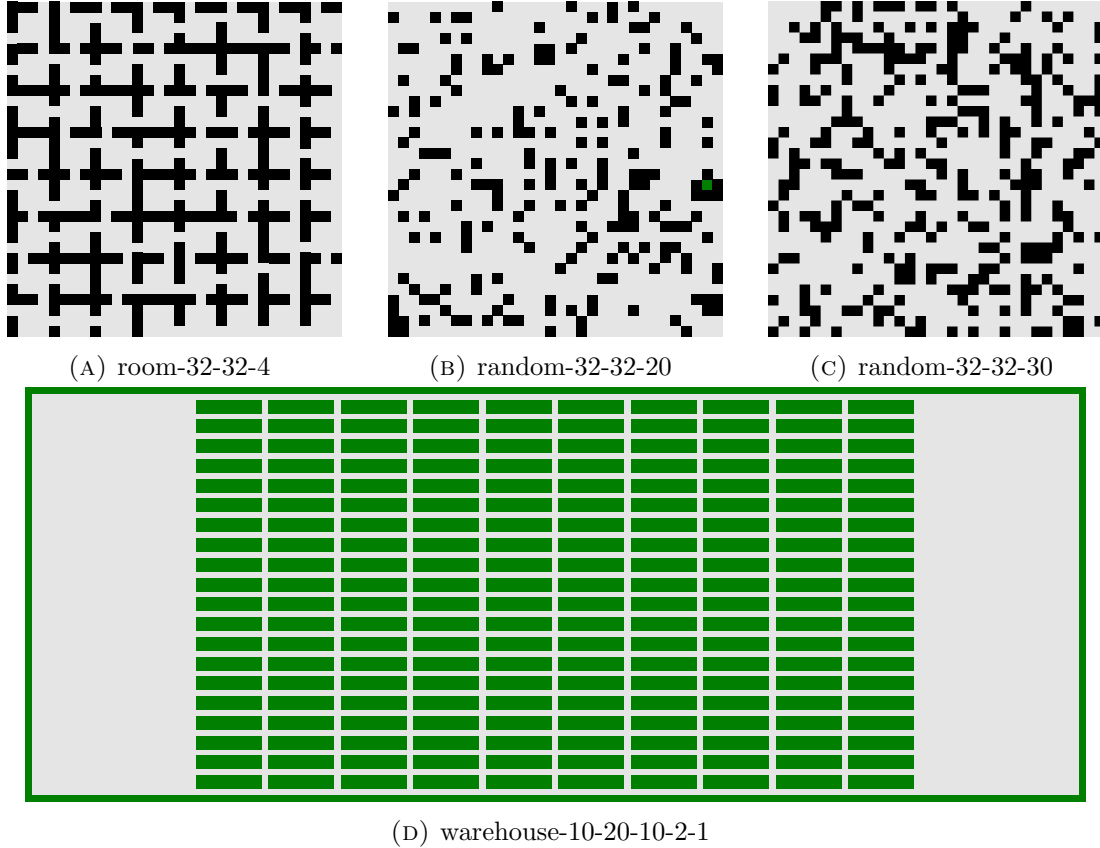


FIGURE 3.6: Maps from the Moving AI MAPF benchmark [2] and generated by us.

We use the even scenario files from the Moving AI MAPF benchmark, which have an even mix of short and long agent paths, to assign the start and goal locations of agents in the first three maps, and we create similar scenario files for the fourth map. Given M agents, we use the first M lines from each scenario file to construct a problem instance. For each instance generated, we run Explicit Estimation CBS (EECBS) [44] with the suboptimality factor set to 1.1 to obtain a near-optimal MAPF solution under the standard assumption that each *move* action takes one timestep to complete. EECBS is a state-of-the-art variant of Conflict-Based Search (CBS) [21] that incorporates many techniques for speeding up CBS.

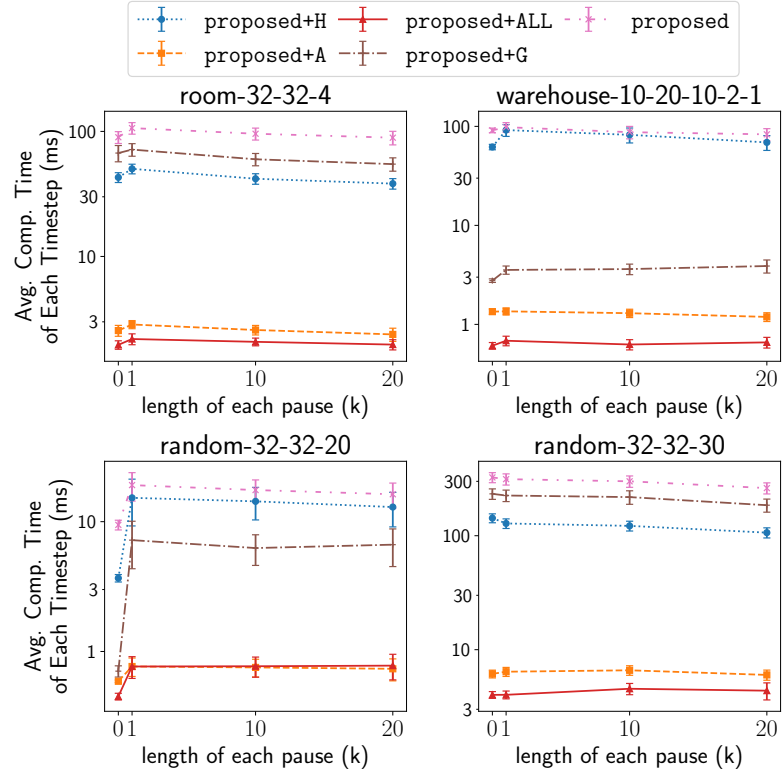


FIGURE 3.7: Comparison of different versions of feasibility test for 40 agents.

We emulate uncertainty by pausing 10% randomly chosen agents every k timesteps. Each pause lasts for k timesteps. If an agent is executing a *wait* action when paused, it will stay at its current node for $k + 1$ timesteps; if an agent is executing a *move* action when paused, it will arrive at the adjacent node after $k + 1$ timesteps. For fair comparison, we pause the same subset of agents at the same timestep for all the algorithms. We measure the sum-of-costs, i.e., the total timesteps taken by all agents to arrive at their goal locations. Given a k setting, we run the path execution for 10 times with different random subsets of agents paused. Overall, there are 250 experiment setups for each map (25 agent placements constructed from scenario files $\times$ 10 uncertainty emulations). We present the average results for all the setups along with the 95% confidence interval. We implement the algorithms in C++ and run the experiments on a machine with Intel Core i9-9820X 3.30GHz CPU and 64GB memory.

3.4.1 Compare Different Versions of Feasibility Test

First, we test different versions of feasibility test in our proposed framework using the heuristic approach for coordinating path execution: (1) **proposed** is the original feasibility test (Algorithm 4) without any optimization. It uses the breadth-first search to detect cycles in the LDG, and does not group the agents; (2) **proposed+H** is the two-level feasibility test which incorporates the heuristic feasibility test (Algorithm 5) to accelerate the original feasibility test; (3) **proposed+A** implements the connection array to optimize cycle detection in the LDG; (4) **proposed+G** groups agents to reduce the number of unsettled edge pairs; (5) **proposed+ALL** combines all optimization methods from (2)-(4).

Figure 3.7 compares the average computational time per timestep among different versions of feasibility test for 40 agents. Since all versions yield the same sum-of-costs, we omit the sum-of-costs comparison in this section. It is evident that **proposed+A** significantly improves the performance, followed by **proposed+H** and **proposed+G**. After applying all optimizations, the average computational time per timestep is reduced by up to 99% (the vertical axis is in log scale) compared with the original feasibility test without any optimization.

3.4.2 Compare Different Approaches in Coordinating Path Execution

Second, we compare different approaches for selecting agents to unblock when coordinating path execution in our proposed framework: the naive, semi-naive and heuristic approaches. In these experiments, we use the **proposed+ALL** feasibility test. Figure 3.8 shows the results for 30 agents. In the experiments, we set a time limit of 6 minutes for each run. The naive approach does not manage to finish within the time limit for a small portion of 250 experiment setups. Thus, we present the average result for only those runs in which the naive approach is completed. The number of agents is limited to 30 because the naive approach runs too slow for more agents.

As seen from the upper row of Figure 3.8, the heuristic approach produces almost the same sum-of-costs as the other two approaches. But the three approaches differ

significantly in their computational times. The lower row of Figure 3.8 compares the average computational time per timestep in path execution. The naive and semi-naive approaches take significantly higher computational time than the heuristic approach (the vertical axis is in log scale). This demonstrates the computational efficiency of the heuristic approach in identifying a good set of agents to unblock.

3.4.3 Compare with Existing Methods

Finally, we compare the heuristic approach for coordinating path execution with five existing methods.

- The first method is a baseline method that follows the precedence dependencies among agents derived from the EECBS solution by applying the MCP execution policy [54].
- The second method is a replanning method that replans the paths of agents whenever any agent not yet reaching its goal location is delayed (by the pause described earlier). We again run EECBS with the suboptimality factor set to 1.1 for replanning. For fair comparison with our method, replanning does not know the length of the pause and it assumes that the pause will finish in the current timestep. Thus, if the pause lasts for multiple timesteps, replanning is carried out repeatedly. Since EECBS cannot always be completed successfully in a predefined timeout (60 seconds) especially as the number of agents increases, when the replan fails, we use the path returned by the most recent successful replan to coordinate the agents with the baseline method for the current timestep. The computation time for unsuccessful replans is excluded from the experimental results.
- The third method is Causal-PIBT+ [68], which is a state-of-the-art online progressive planning method looking one step ahead to avoid conflicts and guarantee reachability (all agents will reach their goal locations but not necessarily be there at the same time). Causal-PIBT+ uses an offline plan as hints to compensate for short-sightedness. We use the aforesaid EECBS solution as hints for Causal-PIBT+.

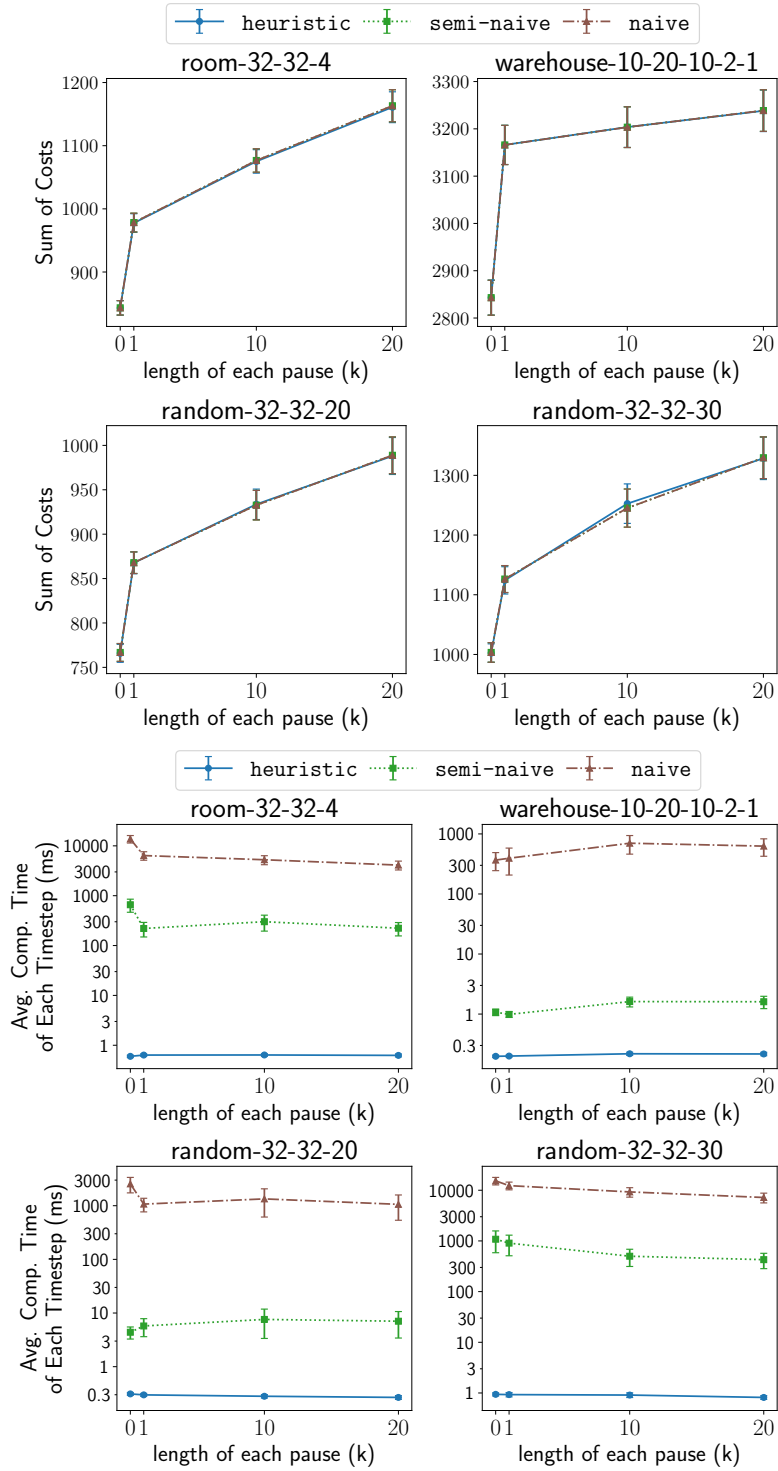


FIGURE 3.8: Comparison of naive, semi-naive and heuristic approaches for 30 agents.

- The fourth method is BTPG-optimized [60]. BTPG identifies the precedence dependencies (unsettled edge pairs) that can be switched in any manner without introducing cycles in the dependency graph (LDG). That is, if there are n edge pairs identified, up to 2^n possible dependency graphs (complete LDGs) are required to be acyclic. The advantage is low overhead to switch dependencies during the plan execution as there is no need to check for cycles. But this approach may significantly restrict the set of switchable dependencies. In contrast, all dependencies can possibly be switched in our method (as long as one acyclic complete LDG is identified, even if switching dependencies in other ways create cycles).
- The fifth method is GSES (Graph-based Switchable Edge Search) [59]. SES (Switchable Edge Search) defines a switchable dependency graph that allows for the modification of some precedence relations. It uses an A*-style algorithm to find a new dependency graph based on the switchable graph that minimizes the sum-of-costs of the agents when a delay is detected. Graph-based SES leverages the graph properties of a dependency graph to improve the efficiency of SES.

These five methods are selected as they cover a broad range of approaches in multi-agent path execution, including both classical baselines and recent state-of-the-art methods. They are not the only existing solutions, but they represent the majority of recent research efforts in this area.

Our proposed algorithm guarantees a 100% success rate across all 250 experiment setups in every map. However, other methods (BTPG, Causal-PIBT+, and replanning) may fail due to various issues. For instance, the BTPG code released by the authors encounters segmentation faults in certain setups, and the Causal-PIBT+ code released by the authors does not always find a solution for all agents to arrive at their goal locations. In such cases, we exclude the failed experiment setups from the results and present the average results for the setups successfully completed by all methods compared. We also observe that the GSES code released by the authors fails to prevent all conflicts among agents during path execution, e.g., two agents may occupy the same location at the same timestep. Such conflicts are detected in the simulation of both the initial dependency graph and the replanned dependency graphs. Attempts to execute the replanned dependency graphs in our

own simulator also fail due to the presence of deadlocks in these graphs. This issue is critical because it can result in the sum-of-costs significantly smaller than the theoretical optimal value, as evident from the experimental results.

Figure 3.9 shows the results for 30 agents that compare our proposed method with all optimizations (the heuristic approach for coordinating path execution with the **proposed+ALL** feasibility test) and the five existing methods. The number of agents is limited to 30 because GSES runs too slow for more agents. As seen from the upper row of Figure 3.9, when $k = 0$ (no pause in path execution), the methods except Causal-PIBT+ and GSES have similar sum-of-costs. When $k > 0$, our method substantially reduces the sum-of-costs compared to the baseline method. The improvement generally increases with the length of each pause. The replanning method often produces lower sum-of-costs than our method. This is because replanning can also change the paths for agents to travel, while our method keeps the paths of agents unchanged. It can be seen that the sum-of-costs difference between our method and replanning is much smaller than that between our method and the baseline method. This implies that simply adjusting the order for agents to occupy common nodes can attain most of the benefits by replanning in optimization. The sum-of-costs of BTPG is usually between the baseline method and our method. This is because our method allows more switchable dependencies than BTPG. In addition, BTPG does not perform well in the last map random-32-32-30 where the ratio of obstacles is increased, probably because few dependencies are switchable by using BTPG in this case. The sum-of-costs of GSES is significantly smaller than all other methods even when $k = 0$. Given that GSES only searches for a new dependency graph when a delay occurs, the sum-of-costs of GSES should be the same as to those of the baseline and replanning methods when there is no pause in path execution ($k = 0$), since we feed the same EECBS solution into all methods initially. Unfortunately, after adding a conflict detection block in the GSES code of path execution, a lot of conflicts among agents are detected. This shows that the GSES implementation released by the authors is not correct and fails to avoid all conflicts. Causal-PIBT+ performs worse than the baseline method in many cases, especially when there are more potential conflicts among the agents (in the room map with narrow corridors and the grid map with more obstacles). Causal-PIBT+ is less successful in optimizing the sum-of-costs because it looks only a single step ahead in path planning.

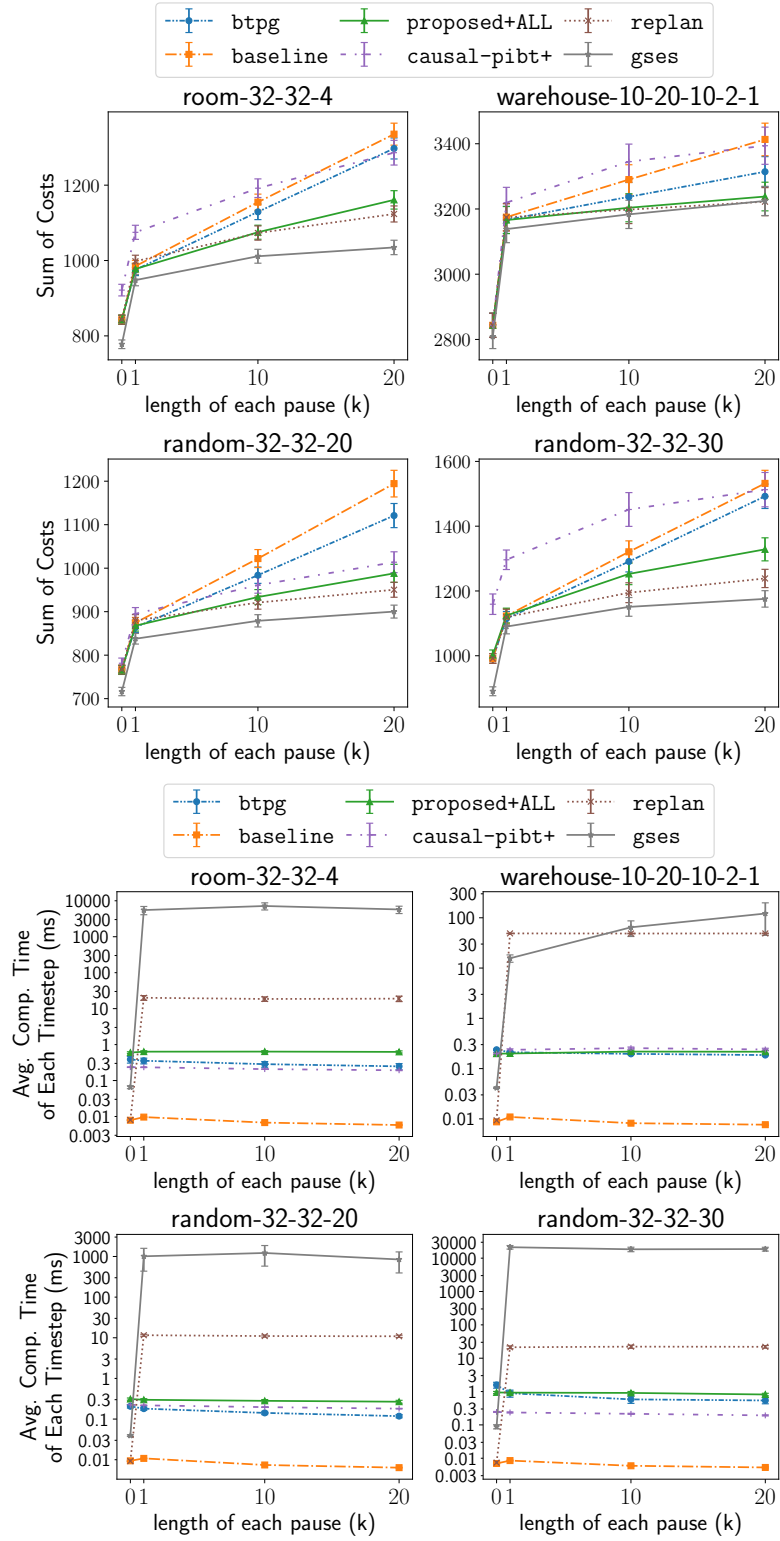


FIGURE 3.9: Comparison of proposed and other methods for 30 agents.

The lower row of Figure 3.9 compares the average computational time per timestep in path execution. Note that only successful replans are considered in deriving the results for the replanning method. The baseline method maintains fairly low computational times because it relies on a pre-computed dependency graph and requires no additional computation in the plan execution. BTPG, Causal-PIBT+, and our proposed method generally exhibit similar computational times when dealing with 40 agents. The replanning method takes much higher computational time than our method. For our method, we observe that running Algorithm 9 on average requires less than 2 feasibility tests and running a feasibility test is much faster than replanning. Although GSES was shown to be faster than replanning with k-Robust CBS in [59], it is much slower than the replanning method with EECBS in our results for most of the maps. We notice that our maps such as the grid maps are more congested (have more obstacles) than those tested in [59], and our delays (pauses) are more frequent than those tested in [59]. This implies that GSES is not competitive in computational efficiency compared to replanning with EECBS in complex situations.

Figure 3.10 shows the results of different methods as a function of the number of agents, where k is set to 20. GSES is not included here because it runs too slow for more than 30 agents. The upper row of Figure 3.10 shows the sum-of-costs normalized by the number of agents. As can be seen, the relative performance among different methods discussed earlier is maintained over different numbers of agents. The lower row of Figure 3.10 compares the average computational time per timestep among various methods. Similar to Figure 3.9, only successful replans are included in the results for the replanning method. The computational times for the baseline and Causal-PIBT+ methods remain relatively constant for different numbers of agents. The computational times for BTPG and our method increase at similar rates as the number of agents rises. Notably, the replanning method exhibits significantly higher computational times than our proposed method for up to 80 agents.

With increasing number of agents, the gap between our proposed method and the replanning method in computational time occasionally appears smaller because each reported value for the replanning method represents a lower bound on the actual computational time, as the durations of unsuccessful replans are not included and can be much longer than those of successful ones. As the number of agents

increases, the frequency of unsuccessful replans also grows, causing the difference between the plotted results and the true computational cost to widen correspondingly.

3.5 Summary

This chapter presented a general framework for multi-agent path execution under discrete-time uncertainty. It introduced a feasibility-based formulation to determine whether all agents can reach their goals without conflicts and proved this problem to be NP-complete. Deterministic and heuristic feasibility test algorithms were developed to coordinate agents online, ensuring both conflict-freeness and deadlock-freeness. Extensive experiments demonstrated the framework's robustness and computational efficiency compared to existing approaches. This study establishes the foundation for handling execution-time uncertainty, motivating the extension to continuous-time scenarios in Chapter 5.

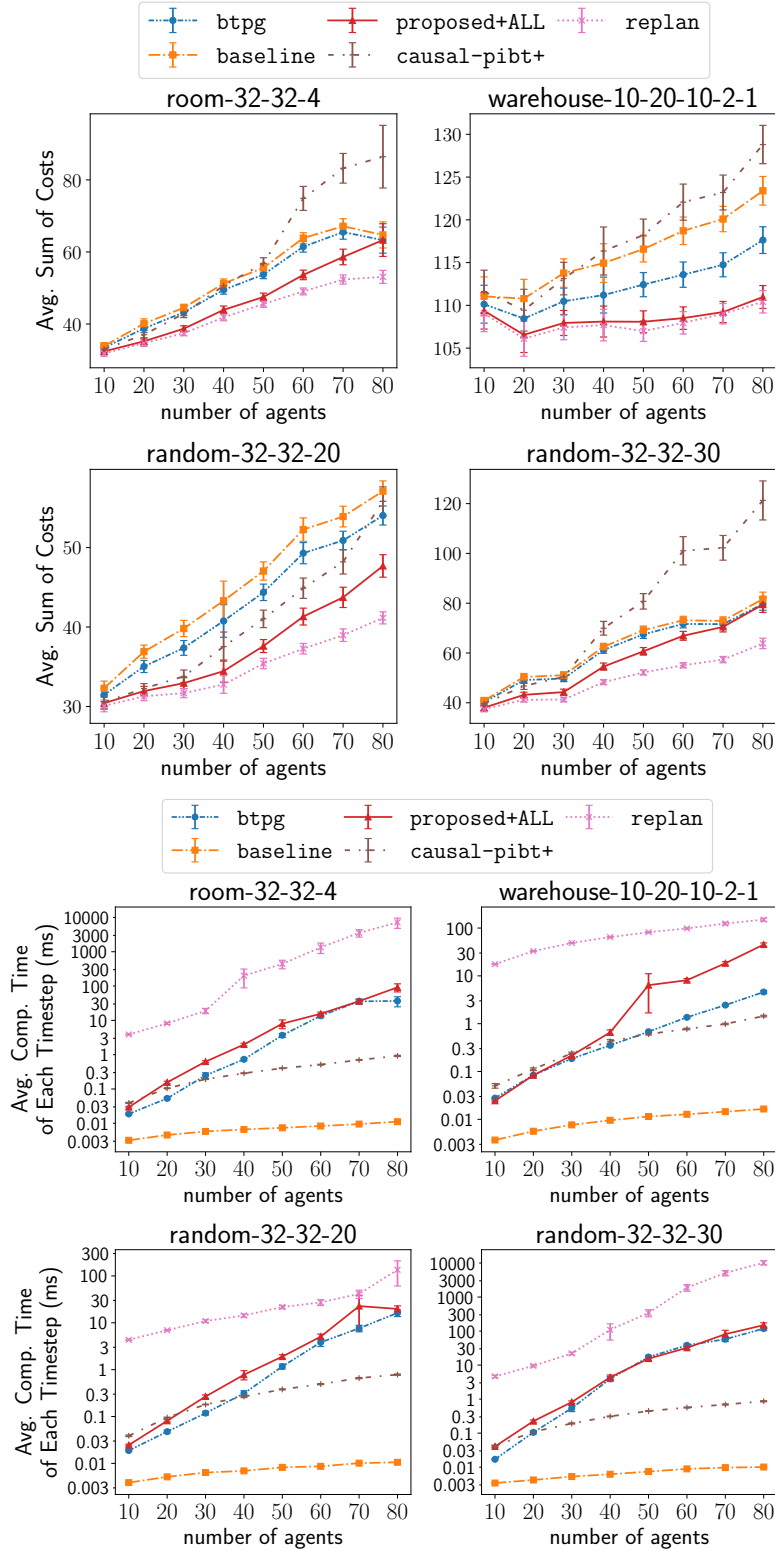


FIGURE 3.10: Comparison of proposed and other methods for different numbers of agents with $k = 20$.

Chapter 4

Continuous-Time Multi-Agent Path Finding

In Section 2.3.1, we stated that the discrete-time multi-agent pathfinding algorithms cannot be applied to many real world applications because of some simplifying assumptions. We also introduced Continuous CBS (CCBS) [48], as an example of an optimal solver for continuous-time MAPF problems. According to the empirical results found by CCBS, we concluded that when the number of agents exceeds a limit (e.g., 40), optimal solvers for continuous-time MAPF problems may not be practical for real-world applications because of the complexity in both path planning and conflict detection.

In this chapter, we propose an efficient hierarchical suboptimal solver MAPF-CLNS for the continuous-time MAPF problem based on Safe Interval Path Planning with Soft Constraints (SIPPS) [10] (for low level search) and Large Neighborhood Search (LNS) [10, 11] (for high level search). The definition of the continuous-time MAPF problem follows the definition in Section 2.3.1. The objective of the problem is to find a *feasible* plan while minimizing the sum-of-costs of all paths in the plan. Our experimental evaluation shows that MAPF-CLNS can handle large-scale continuous-time MAPF problems effectively and efficiently.

4.1 Continuous-Time Single Agent Shortest Path Problem with Dynamic Obstacles

To efficiently solve the Continuous-Time Multi-Agent Path Finding (Continuous-Time MAPF) problem, we need a low-level algorithm that computes a shortest path for a single agent while minimizing collisions with a set of planned paths from other agents. We classify obstacles into two categories: *hard* obstacles, which the agent must strictly avoid (e.g., static walls, permanent barriers, or paths of agents that must not be disrupted), and *soft* obstacles, where collisions are tolerated but penalized (e.g., paths of other agents with some flexibility). We define this as the Continuous-Time Path Finding with Mixed Dynamic Obstacles (CPDMO) problem, where the objective is to find a shortest path without hard obstacle collisions while minimizing the number of soft obstacle collisions. The transformation of planned paths from other agents into hard and soft obstacles varies depending on the high-level algorithm's requirements, which will be explained later.

Similar to the discrete-time MAPF problem (Section 2.2.1), we define three types of obstacles, but each type has a slight difference in terms of the lasting time. A node obstacle (v, t, t') indicates that the node $v \in \mathcal{V}$ is occupied from time instant t to t' . It occurs when another agent is waiting at v in the time period. An edge obstacle (e, t, t') indicates that the edge $e \in \mathcal{E}$ is occupied from time instant t to t' . It occurs when another agent is moving along e in the time period. A goal obstacle (v, t, ∞) indicates that the node $v \in \mathcal{V}$ is occupied from time instant t onward infinitely. It occurs when v is a goal node of an agent, and the agent arrives at its goal node at t . Moreover, all obstacles are bounded with a specific shape parameter. We omit the shape parameter in the above notations as it only takes effect in the geometric calculations of collisions.

Consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with a start node $s \in \mathcal{V}$, a goal node $g \in \mathcal{V}$, and two finite sets of obstacles: $\mathcal{O}^h$ (hard obstacles) and $\mathcal{O}^s$ (soft obstacles). The objective of the CPDMO problem is to find a path $p = \langle \pi_1, \pi_2, \dots, \pi_n \rangle$, which represents a sequence of n actions, that avoids collisions with any hard obstacles while minimizing the number of collisions with soft obstacles. In case of ties, the path with the smallest cost $|p|$ is selected.

4.1.1 SIPPS

Many MAPF algorithms utilize Space-Time A* [15] as their low-level search algorithm, which can also be adapted to solve the CPDMO problem. The adapted version of Space-Time A* conducts an A* search over a state graph where each state is represented by a pair of node v and timestamp t , denoted as (v, t) . The states to search in Space-Time A* are prioritized in an open list based on the number of conflicts (c -value), breaking ties using the heuristic cost of the plan (f -value). These values are also fundamental to our algorithm, which will be introduced in detail later. While Space-Time A* guarantees an optimal solution to the CPDMO problem, it does not do so efficiently. For example, if the minimum number of conflicts (c -value) in the optimal path is one, Space-Time A* must examine and expand all states with $c = 0$ before finding the optimal path. However, since the time dimension is unbounded, an infinite number of zero-collision states may be generated. One potential fix is to restrict the time dimension to the timestamps defined by obstacles and switch to standard A* thereafter; however, even with this modification, the number of generated states remains excessively large, making the search computationally expensive.

Safe Interval Path Planning (SIPP) [16] enhances the Space-Time A* algorithm by extending each state's timestamp t to a safe time interval $[t, t')$, ensuring that no hard obstacles exist within this period at a given node. SIPP prioritizes the states to search based on the earliest arrival time $t_0 \in [t, t')$ and prunes states that arrive later than t_0 within the same safe interval without compromising optimality. Safe Interval Path Planning with Soft Constraints (SIPPS) [10] generalizes SIPP by incorporating soft constraints into the planning process, for solving the Path Finding with Mixed Dynamic Obstacles (PDMO) problem. Empirically, SIPPS runs significantly faster than Space-Time A*, making it a more efficient alternative. We extend the SIPPS algorithm into a continuous-time version to compute the shortest path for an agent without collisions with hard obstacles while minimizing the number of collisions with soft obstacles in CPDMO, achieving a balance between path efficiency and obstacle avoidance.

4.1.1.1 Collisions

In most discrete-time pathfinding problems, detecting collisions between an agent and the three types of obstacles defined above is relatively straightforward. A node collision occurs when an agent collides with a node or goal obstacle. To prevent this, we can construct a Safe Interval Table (SIT) that maps each node to a set of safe intervals, excluding the time intervals occupied by node and goal obstacles. If an agent attempts to occupy a node outside these safe intervals, a collision is detected. An edge collision happens when an agent collides with an edge obstacle. In discrete-time multi-agent pathfinding problems, it occurs only when two agents traverse the same edge in opposite directions at the same timestep. Since the number of edges in the graph is limited (as they are typically straight-line connections between adjacent nodes), we can apply a similar approach to edge obstacles by mapping each edge to its corresponding set of safe intervals. This data structure is usually called a Collision Avoidance Table (CAT), which allows for efficient edge collision detection and resolution.

In continuous-time pathfinding problems such as CPDMO, detecting node and edge collisions becomes more complex because not all collisions can be defined on a single node or edge, making the original versions of SIT and CAT insufficient for detecting all collisions. To address this, we extend the definition of collisions and modify both tables with a more refined approach to detect the collisions to fulfill the requirements of the CPDMO problem.

A collision between an agent executing a *wait* action and an obstacle is detected in the SIT, similar to node collisions in discrete-time problems. It occurs when:

- (1) the shape of a waiting agent (S_a) and the shape of a node/goal obstacle (S_o) overlap geometrically at the same time (since agents and obstacles have volumes, this can happen even when they are positioned at different but nearby nodes);
- (2) the shape of a waiting agent (S_a) and the shape of an edge obstacle (S_o) overlap geometrically at some point along the edge of the obstacle at the same time.

Instead of only updating the safe intervals for one node, we need to update all affected nodes for every obstacle. This ensures that the SIT captures the dynamic interactions between agents and obstacles in continuous space.

A collision between an agent executing a *move* action and an obstacle is detected in the CAT, similar to edge collisions in discrete-time problems. It occurs when:

- (1) the shape of a moving agent (S_a) and the shape of a node/goal obstacle (S_o) overlap geometrically at some point along the edge of the agent at the same time;
- (2) the shape of a moving agent (S_a) and the shape of an edge obstacle (S_o) overlap geometrically at some point along their respective edges at the same time.

Note that two edges (or an edge and a node) do not necessarily share the same nodes, so it is impossible to detect such collisions by node or edge indexes in CPDMO. To address this, we use bounding rectangles as a heuristic to index the collisions, which allows for efficient detection of potential collisions between the moving agent and obstacles.

4.1.1.2 Safe Interval Table

The Safe Interval Table consists of a slot $SIT[v]$ for each node v . Each slot $SIT[v]$ includes a set of time intervals. Each time interval is a continuous period of time with an interval type, $(t, t', type)$, where the time interval is $[t, t')$ and *type* is one of *free*, *soft*, and *hard*. A *free* interval means that the node v is not occupied by any obstacles; a *soft* interval means that the node v is occupied by one or more soft obstacles, but not occupied by any hard obstacles; a *hard* interval means that the node v is occupied by one or more hard obstacles. A safe interval can be either a *free* or *soft* interval, indicating that the node v is available for the agent to occupy without colliding with hard obstacles.

Algorithm 11 shows the procedure of building the Safe Interval Table (SIT) with the obstacles. We first initialize every slot $SIT[v]$ in the table with one time interval of *type* = *free* from the initial time to infinity (lines 1-2), and then divide

it into a minimum disjoint set of different types of time intervals in chronological order based on the obstacles. For each obstacle (v, t, t') or (e, t, t') , we find all nodes that may collide with the obstacles. When an agent executes a *wait* action at a node v , there is a collision if the shape of the agent overlaps geometrically with the obstacle (lines 3-4). Then we find the collision interval $[low, high)$ so that if the agent waits at v in the interval, there is a collision between the agent and the obstacle (line 6). Thus, we add the collision interval $[low, high)$ to the slot $SIT[v]$ with *type* = *hard* or *soft* depending on which category the obstacle belongs to (lines 7-10). Note that there may exist some overlaps among the time intervals in the slot $SIT[v]$, so we should update them by priority (*hard* > *soft* > *free*) to keep them disjoint. If multiple values are assigned to the same time interval, only the value with the highest priority is preserved (line 11).

Algorithm 10: buildSafeIntervalTable($\mathcal{V}, \mathcal{O}^h, \mathcal{O}^s$)

```

1 foreach  $v \in \mathcal{V}$  do
2    $SIT[v] \leftarrow [(0, +\infty, free)]$ ;
3 foreach  $obstacle \in \mathcal{O}^h \cup \mathcal{O}^s$  do
4    $\mathcal{V}^c \leftarrow \{v \in \mathcal{V} \mid \text{a waiting agent at } v \text{ has a collision with } obstacle\}$ ;
5   foreach  $v \in \mathcal{V}^c$  do
6      $[low, high) \leftarrow$  the interval that the agent collides with  $obstacle$  at  $v$ ;
7     if  $obstacle \in \mathcal{O}^h$  then
8       Add  $(low, high, hard)$  to  $SIT[v]$ ;
9     else if  $obstacle \in \mathcal{O}^s$  then
10      Add  $(low, high, soft)$  to  $SIT[v]$ ;
11   Update time intervals in  $SIT[v]$  by priority of values;
```

4.1.1.3 Collision Avoidance Table

We use bounding rectangles as a heuristic to index the obstacles in the Collision Avoidance Table (CAT). For example, in Figure 4.1, two circular agents, a_1 and a_2 , can move along two separate edges, $s_1 \rightarrow g_1$ and $s_2 \rightarrow g_2$, respectively; another circular agent, a_3 , can wait at node v_3 . Suppose we are planning a path for a_1 , i.e., a_1 is executing a *move* action on the edge $s_1 \rightarrow g_1$. Assume $s_2 \rightarrow g_2$ is part of the planned path of a_2 , which forms an edge obstacle; v_3 is part of the planned path of a_3 , which forms a node obstacle. There is a collision between agent a_1 and the edge obstacle (caused by a_2), where none of the corresponding nodes or edges are

identical. To detect the collision, we first check whether the bounding rectangles (in dashed line) of the trajectories of the moving agent and the obstacle overlap. If they do, it indicates a possible collision. If the time intervals of the *move* action and the obstacle overlap as well, we apply geometric algorithms to verify whether a collision actually occurs. More specifically, if both agents start moving at the same time in the example, we can compute the exact collision time between the two agents, ensuring precise detection of the conflict. There is also a potential collision between agent a_1 and the node obstacle (caused by a_3) due to the overlap of their bounding rectangles. However, this conflict can be ruled out by geometric algorithms, as they do not overlap geometrically.

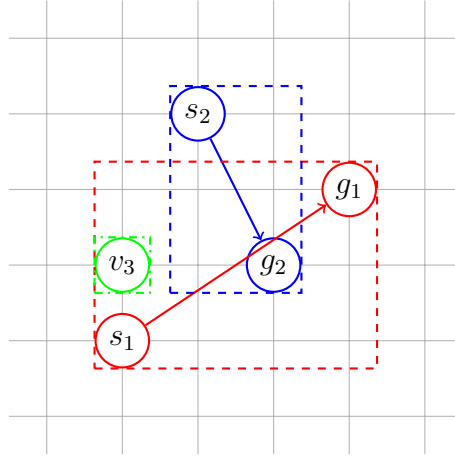


FIGURE 4.1: An example of rectangle collision detection.

In our implementation, we utilize a Quadtree data structure to store the bounding rectangles of obstacles (Algorithm 11). For a node or goal obstacle, we use the bounding rectangle that covers the shape of the obstacle; for an edge obstacle, we use the bounding rectangle that covers the entire trajectory of the obstacle (including its shape) as shown in Figure 4.1. When finding the earliest arrival time of a moving agent without collision with the obstacles (Algorithm 12), we can consider hard obstacles only (*type = hard*) or all obstacles (*type = mixed*) in the algorithm (lines 1-4). By default, if there is no obstacle, the earliest arrival time is the same as the lower bound (*low*) of the provided safe interval (line 5).

We first query the Quadtree *CAT* to retrieve all bounding rectangles that intersect with the bounding rectangle of the moving action π of the agent (lines 6-7), and then filter the possible collisions by checking whether the safe interval $[low, high)$

provided overlaps with the time interval of the obstacle (line 9). For each obstacle, we apply geometric algorithms to find the earliest arrival time t_c at the destination of the moving action ($\pi.e.dest$) within the safe time interval without colliding with the obstacle (line 11). It means that in the interval $[t_c, high)$, π does not collide with the obstacle. If t_c does not exist, π always collides with the obstacle in $[low, high)$, so the overall earliest arrival time low' does not exist as well (lines 12-13). Otherwise, we update the overall earliest arrival time low' with t_c if t_c is larger, since we need to find the earliest arrival time with consideration of all obstacles (lines 14-15). The complexity of identifying each potential collision in $\mathcal{S}$ is $\mathcal{O}(\log N)$, where N is the number of obstacles in $\mathcal{O}$. Empirically, the number of possible collisions remains small since the bounding rectangles of most obstacles are relatively small. Thus, indexing bounding rectangles of obstacles by a Quadtree is more efficient than scanning all the obstacles to check whether a moving agent collides with each obstacle using geometric algorithms.

Algorithm 11: buildCollisionAvoidanceTable($\mathcal{O}^h, \mathcal{O}^s$)

```

1  $CAT \leftarrow$  empty Quadtree of rectangles;
2 foreach  $obstacle = (v/e, t, t') \in \mathcal{O}^h \cup \mathcal{O}^s$  do
3    $S_o \leftarrow$  bounding rectangle of  $obstacle$ ;
4   Add  $\langle S_o, obstacle \rangle$  to  $CAT$ ;
```

4.1.1.4 Search State in SIPPS

A state n in the search tree of SIPPS consists of four elements: a node $n.v \in \mathcal{V}$, a safe interval $[n.low, n.high)$, an index $n.id$ indicating that the safe interval is (a subset of) the id -th safe interval in slot $SIT[n.v]$ of the Safe Interval Table, and a boolean flag $n.is_goal$ indicating whether the state is a goal state. The f -value of state n is the sum of its g -value and h -value, where the g -value is set to $n.low$ (the earliest arrival time) and the h -value is a lower bound on the minimum travel time from the node $n.v$ to g (which can be pre-calculated using an all-pair shortest path algorithm). The c -value of state n represents an underestimated number of collisions with soft obstacles along the partial path from the root state to state n in the search tree. If n' is the parent state of n , we have $n.c = n'.c + c_v + c_e$, where $v = n.v$ and $e = \langle n'.v, n.v \rangle$. Here, c_v is fetched from the Safe Interval Table, and it is 1 if the safe interval of n (i.e., $SIT[n.v][n.id]$) contains soft obstacles and 0 otherwise. c_e is fetched from the Collision Avoidance Table, and it is 1 if

Algorithm 12: findEarliestArrivalTimeWithoutCollision($CAT, \pi, low, high, type$)

```

1 if  $type = hard$  then
2    $\mathcal{O} \leftarrow \mathcal{O}^h$ ;
3 else
4    $\mathcal{O} \leftarrow \mathcal{O}^h \cup \mathcal{O}^s$ ;
5  $low' \leftarrow low$ ;
6  $S_a \leftarrow$  bounding rectangle of  $\pi$ ;
7  $\mathcal{S} = \{\langle S_o, obstacle \rangle \in CAT \mid obstacle \in \mathcal{O}, S_a \cap S_o \neq \emptyset\}$ ;
8 foreach  $\langle S_o, obstacle \rangle \in \mathcal{S}$  do
9   if  $[low, high) \cap [obstacle.t, obstacle.t') = \emptyset$  then
10    continue;
11    $t_c \leftarrow$  earliest arrival time to complete  $\pi$  within  $[low, high)$  without
      colliding with  $obstacle$ ;
12   if  $t_c$  does not exist then
13      $low' \leftarrow$  non-exist;
14   else if  $t_c > low'$  then
15      $low' \leftarrow t_c$ ;
16 return  $low'$ ;

```

executing the action $\pi = (e, t)$ at time $n.low - t$ collides with soft obstacles $\mathcal{O}^s$ and 0 otherwise, where t is the cost $t_a(e)$ of the edge e (the length of e divided by the speed of the agent a).

4.1.1.5 Main SIPPS Algorithm

Algorithm 22 shows the pseudocode of the main SIPPS algorithm. First, we generate the Safe Interval Table as described in the previous section (line 1). We initialize the root state in the search tree with the start node s , the first ($id = 1$) safe interval of the node s , and $is_goal = false$ (line 2). We also create a lower bound T on the cost (travel time) of the path to plan based on the goal node g . If there exist *hard* time intervals in $SIT[g]$, we set T as the maximum end time of the hard time intervals because the plan cannot end before this time to prevent collision with hard obstacles (lines 3-6). Note that $T \neq \infty$ because the hard obstacles on g cannot be a goal obstacle with infinite time (by problem definition, the goal nodes do not overlap with each other). *OPEN* and *CLOSED* are regular open and closed lists, where the root state is added into the open list initially (line 7). They are ordered list implemented by a pairing heap, where the states are

Algorithm 13: Continuous-Time SIPPS($\mathcal{V}, \mathcal{E}, s, g, \mathcal{O}^h, \mathcal{O}^s$)

```

1  $SIT \leftarrow \text{buildSafeIntervalTable}(\mathcal{V}, \mathcal{O}^h, \mathcal{O}^s), CAT \leftarrow$ 
   $\text{buildCollisionAvoidanceTable}(\mathcal{O}^h, \mathcal{O}^s);$ 
2  $root \leftarrow \text{State}(v = s, \text{safe\_interval} = SIT[s][1], id = 1, \text{is\_goal} = \text{false});$ 
3 if  $\exists(t, t', \text{hard}) \in SIT[g]$  then
4    $T \leftarrow \max\{t' \mid (t, t', \text{hard}) \in SIT[g]\};$ 
5 else
6    $T \leftarrow 0;$ 
7  $OPEN \leftarrow \{root\}, CLOSED \leftarrow \emptyset;$ 
8 while  $OPEN \neq \emptyset$  do
9    $n \leftarrow \text{pop the first state from } OPEN;$ 
10  if  $n.\text{is\_goal}$  then
11     $\text{return ExtractPath}(n);$ 
12  if  $n.v = g$  and  $n.\text{low} \geq T$  then
13     $c_{\text{future}} \leftarrow |\{(t, t', \text{soft}) \in SIT[g] \mid t > n.\text{low}\}|;$ 
14    if  $c_{\text{future}} = 0$  then
15       $\text{return ExtractPath}(n);$ 
16     $n' \leftarrow n;$ 
17     $n'.\text{is\_goal} \leftarrow \text{true};$ 
18     $n'.c \leftarrow n'.c + c_{\text{future}};$ 
19     $\text{InsertState}(n', OPEN, CLOSED);$ 
20     $\text{ExpandState}(n', OPEN, CLOSED, SIT, CAT);$ 
21     $CLOSED \leftarrow CLOSED \cup \{n\};$ 
22 return “No Solution”;

```

sorted by their c -values ascendingly (number of conflicts), breaking ties with their f -values (heuristic of cost) ascendingly as well. In each iteration of the search (line 8), we pop the first state n from the open list (line 9) and return the path from the root state to n if it is a goal state (lines 10-11). If the node $n.v$ is the same as the goal node g , but the earliest arrival time $n.\text{low}$ is no smaller than the lower bound T on the cost of the path to plan, it can be a goal state. The agent will wait at g infinitely, but its c -value does not consider collisions with soft obstacles after $n.\text{low}$, so we need to find the number c_{future} of soft time intervals after $n.\text{low}$ on the node g (line 13). If there are no future collisions, the path can be safely returned (lines 14-15). Otherwise, we copy the state n to n' , set $n'.\text{is_goal} = \text{true}$, update the number of collisions of n' , and insert n' back into the open list (lines 16-19). If n' is popped from the open list in a future iteration, we can return its path directly to prevent duplicate calculation of c_{future} at the goal node g . Finally, we expand the search state n (line 20, Algorithm 14) and add the state into the closed list (line

21). When the states in the open list are exhausted, we have searched the whole space but no solution can be found for the problem (line 22).

4.1.1.6 Expanding Search State

To expand a state n (Algorithm 14), we iterate through all adjacent nodes v of $n.v$ and find all reachable safe intervals starting from the safe interval of n . If there is an overlap between a safe interval in $SIT[v]$ and the safe interval of n , delayed by the traveling time $t_a(e)$ on the edge $e = \langle n.v, v \rangle$, the safe interval at v is reachable from n (lines 2-4). Moreover, if there is no hard obstacle before the next safe interval on $SIT[n.v]$, the agent can wait at $n.v$ (lines 5-6). For each safe interval in $\mathcal{I}$, we use $[low, high)$ to represent the start and end times of the time interval, and $type$ to indicate whether it is *soft* or *free* (lines 7-8). If the interval is *free*, the child state is assigned $c_v = 1$; otherwise, $c_v = 0$ (lines 9-12).

The transition to a new safe interval $SIT[v][id]$ can be either a *move* or a *wait* action. In the case of a *move* action, the earliest arrival time low at v is determined within $[low, high)$ to avoid collisions with hard obstacles using the CAT (lines 13-15). If low does not exist, the interval is skipped due to inevitable collisions with hard obstacles (lines 16-17). Next, we compute the earliest arrival time low' at v within the new $[low, high)$ interval with the CAT to account for both soft and hard obstacles (line 18). If low' exists, then arriving within $[low, low')$ leads to at least one soft collision, while arriving within $[low', high)$ ensures no soft collision. If $low' > low$, two child states are generated, where the first one has one more collision with $c_e = 1$ than the second one, which has $c_e = 0$ (lines 21-26). If $low' = low$, only one child state is generated with $c_e = 0$ (lines 27-30). This always occurs when the action is a *wait* action, as low is set to low' beforehand in line 20. If low' does not exist, one child state is generated with $c_e = 1$ because arriving within the whole interval $[low, high)$ leads to at least one soft collision (lines 31-34).

4.1.1.7 Inserting State to Open List

In Algorithm 14, a variety of states are generated and inserted into the open list. If we do not implement an efficient state pruning mechanism, the search space

Algorithm 14: ExpandState($n, OPEN, CLOSED, SIT, CAT$)

```

1  $\mathcal{I} \leftarrow \emptyset$ ;
2 foreach node  $v$  where  $\langle n.v, v \rangle \in \mathcal{E}$  do
3    $e \leftarrow \langle n.v, v \rangle$ ;
4    $\mathcal{I} \leftarrow \mathcal{I} \cup \{(v, id) \mid SIT[v][id] \cap [n.low + t_a(e), n.high + t_a(e)] \neq \emptyset\}$ ;
5 if  $SIT[n.v][n.id + 1]$  is a free or soft interval then
6    $\mathcal{I} \leftarrow \mathcal{I} \cup \{(n.v, n.id + 1)\}$ ;
7 foreach  $v, id \in \mathcal{I}$  do
8    $(low, high, type) \leftarrow SIT[v][id]$ ;
9   if  $type = soft$  then
10     $c_v \leftarrow 1$ ;
11  else
12     $c_v \leftarrow 0$ ;
13  if  $v \neq n.v$  then
14     $\pi = (\langle n.v, v \rangle, n.low, n.low + t_a(\langle n.v, v \rangle))$ ;
15     $low \leftarrow$ 
16      findEarliestArrivalTimeWithoutCollision( $CAT, \pi, low, high, hard$ );
17    if  $low$  does not exist then
18      continue;
19     $low' \leftarrow$ 
20      findEarliestArrivalTimeWithoutCollision( $CAT, \pi, low, high, mixed$ );
21  else
22     $low' \leftarrow low$ ;
23  if  $low'$  exists and  $low' > low$  then
24     $n_1 \leftarrow State(v = v, safe\_interval = [low, low'], id = id, is\_goal =$ 
25       $false)$ ;
26     $n_2 \leftarrow State(v = v, safe\_interval = [low', high], id = id, is\_goal =$ 
27       $false)$ ;
28     $n_1.c = n.c + c_v + 1, n_2.c = n.c + c_v$ ;
29    InsertState( $n_1, OPEN, CLOSED$ );
30    InsertState( $n_2, OPEN, CLOSED$ );
31  else if  $low'$  exists and  $low' = low$  then
32     $n_3 \leftarrow State(v = v, safe\_interval = [low, high], id = id, is\_goal =$ 
33       $false)$ ;
34     $n_3.c = n.c + c_v$ ;
35    InsertState( $n_3, OPEN, CLOSED$ );
36  else
37     $n_4 \leftarrow State(v = v, safe\_interval = [low, high], id = id, is\_goal =$ 
38       $false)$ ;
39     $n_4.c = n.c + c_v + 1$ ;
40    InsertState( $n_4, OPEN, CLOSED$ );

```

will soon explode. This is because each state expansion can potentially generate multiple child states, leading to an exponential growth in the search space as the algorithm progresses. To mitigate this issue, it is essential to prune redundant states, which would otherwise consume unnecessary computational resources. Various techniques can be applied to achieve pruning, such as using dominance relationships between states (e.g., a state that is strictly worse than another state can be discarded) and leveraging state history (in the closed list) to avoid revisiting previously explored configurations. By carefully applying these pruning strategies, we can significantly reduce the size of the search space and improve the efficiency of the algorithm.

We define that two search states n_1 and n_2 have the same identity ($n_1 \sim n_2$) if and only if $n_1.v = n_2.v$, $n_1.id = n_2.id$, and $n_1.is_goal = n_2.is_goal$. If two search states in $OPEN \cup CLOSED$ have the same identity, one of them may be pruned from the lists. We further define a dominance relationship $n_1 \succsim n_2$ if and only if $n_1 \sim n_2$, $n_1.low \leq n_2.low$, and $c(n_1) \leq c(n_2)$. If n_1 dominates n_2 ($n_1 \succsim n_2$), the earliest arrival time and number of conflicts of n_1 are both smaller than those of n_2 , which means that the safe intervals expanded by n_1 always include those intervals expanded by n_2 . Thus, we can safely prune n_2 to avoid duplicate search efforts on smaller time intervals. Furthermore, if there is no dominance relationship between n_1 and n_2 where $n_1.low \leq n_2.low$, then we know that $c(n_1) > c(n_2)$. If there is an overlap between the time intervals of n_1 and n_2 , we can shrink the interval of the search state n_1 by splitting it into two search states n'_1 and n''_1 , such that the time interval of n'_1 is $[n_1.low, n_2.low]$ and the time interval of n''_1 is $[n_2.low, n_1.high]$, with $c(n_1) = c(n'_1) = c(n''_1)$. Since $n_2 \succsim n''_1$ ($n_2.low = n''_1.low$, $c(n_2) < c(n''_1)$), we can prune n''_1 and replace n_1 with n'_1 by updating $n_1.high$ to $n_2.low$.

To insert a state n (Algorithm 11), we first identify all states that share the same identity as n (including n itself), which may be candidates for pruning from the open and closed lists, and iterate over them (lines 1-2). If there exists a state $n' \succsim n$, then there is no need to insert n into the open list, as n' dominates n (lines 3-4). If there exists a state $n \succsim n'$, we prune n' from both the open and closed lists (lines 5-6). Otherwise, if there is an overlap between the time intervals of n and n' , we shrink the interval of either n or n' to avoid redundant search, by adjusting their respective time intervals (lines 7-11).

Algorithm 15: InsertState($n, OPEN, CLOSED$)

```

1  $\mathcal{N} \leftarrow \{n' \in OPEN \cup CLOSED \mid n \sim n'\};$ 
2 foreach  $n' \in \mathcal{N}$  do
3   if  $n'.low \leq n.low$  and  $c(n') \leq c(n)$  then
4     return;
5   else if  $n.low \leq n'.low$  and  $c(n) \leq c(n')$  then
6     Remove  $n'$  from  $OPEN$  and  $CLOSED$ ;
7   else if  $n.low < n'.high$  and  $n'.low < n.high$  then
8     if  $n.low < n'.low$  then
9        $n.high \leftarrow n'.low$ ;
10    else
11       $n'.high \leftarrow n.low$ ;

```

4.1.2 Example Applications of SIPPS

4.1.2.1 Conflicted-Based Search (CBS)

Discrete-time multi-agent pathfinding (MAPF) problems have been extensively studied, with conflict-based search (CBS) being a well-known optimal algorithm for solving classical MAPF instances [21]. CBS employs a two-level approach: a low-level search, such as Space-Time A* or SIPP, finds individual paths while treating all obstacles as hard constraints, and a high-level search resolves conflicts between these paths. CBS prioritizes search states based on the sum of path costs over all agents, ensuring an optimal solution.

In a continuous-time MAPF problem, conflicts can occur between agents traversing different edges, as well as between an agent waiting at a node and another agent moving along an edge. Continuous-time conflict based search (CCBS) [48] solves the problem optimally by considering all types of conflicts (node-node, edge-edge and node-edge conflict) and generating corresponding constraints in the search states. The conflicts are detected and resolved with a geometry-aware collision detection mechanism, and the constraints are defined by actions (*move* or *wait*) and time intervals.

In the original CCBS algorithm, SIPP is used as the low-level algorithm where all constraints are converted into hard obstacles. This can be inefficient because the paths that have not been added into the constraints may cause conflicts in

the replanning, leading to potential duplicate computations. We can convert these paths into soft obstacles and use the SIPPS algorithm (Algorithm 22) for low-level search to speed up CCBS.

4.1.2.2 Prioritized Planning (PP)

Prioritized Planning with SIPPS (PP-SIPPS) can solve the continuous-time MAPF problem that tolerates the existence of conflicts. The problem can be solved by converting the paths of all agents into soft obstacles. Different from the original Prioritized Planning algorithm which uses A* or SIPP as the low-level search [38], PP-SIPPS ensures completeness by not treating any obstacles as hard obstacles. The solution of PP-SIPPS can be provided to another continuous-time MAPF solver for further optimization. The details of PP-SIPPS can be found in the next section.

4.2 Continuous-Time MAPF via Large Neighborhood Search

Due to the fact that MAPF is NP-hard to solve optimally [13], CBS can only find optimal solutions for small instances in limited time, especially for continuous-time instance because the search space is much larger when the number of edges in the graph increases. Some bounded suboptimal variants of CBS, such as BCBS [35] and EECBS [44], scale to larger instances with bounded loss in optimality, but they are still exponential-time algorithms. There also exist some linear-time MAPF algorithms that can scale to much larger instances. For example, prioritized planning (PP) uses low-level search (such as A* or SIPP) to plan a path for each agent from the highest priority agent to the lowest priority one and avoid collision with the planned paths of higher priority agents. However, these algorithms do not always find a valid solution, and the quality of the found solutions is far from optimal.

MAPF via Large Neighborhood Search (MAPF-LNS) combines the above techniques and uses LNS as a heuristic for finding good solutions [11]. It starts with an initial solution found by a fast but not optimal MAPF solver such as PP, deletes a neighborhood (the paths of a certain set of agents) from the solution, and optimizes

the paths in the neighborhood by replanning while fixing the remaining part of the solution. In this section, we extend MAPF-LNS to find a suboptimal solution of the continuous-time MAPF problem efficiently.

4.2.1 Find Solution in a Neighborhood

In large neighborhood search, a fast MAPF solver is required to find a solution to the reduced MAPF problem within a neighborhood $\mathcal{A}_n$. Theoretically, any non-optimal algorithm can be used for this purpose. In our implementation, we develop Prioritized Planning with SIPPS (PP-SIPPS) and convert the paths of all agents into soft obstacles to solve the reduced problem (Algorithm 16). The initial plan before the first iteration of the large neighborhood search can be generated by setting $\mathcal{A}_n = \mathcal{A}$, ensuring that the neighborhood initially includes all agents.

In Algorithm 16, we begin by assigning each agent in the neighborhood $\mathcal{A}_n$ a distinct and random priority through shuffling (line 1). The set $\mathcal{O}^s$ is then initialized by converting the paths of agents outside the neighborhood ($\mathcal{A} \setminus \mathcal{A}_n$) into soft obstacles (line 2). We proceed by planning paths for agents in order of priority, from highest to lowest (line 3). If SIPPS fails to find a solution for an agent's path, we immediately return "no solution" (lines 4-5). Otherwise, the computed path is converted into soft obstacles and added to $\mathcal{O}^s$ (lines 7-8). Once paths for all agents are successfully planned, they are returned as the final solution (line 9).

Algorithm 16: PP-SIPPS($\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{A}_n$)

```

1 Randomly shuffle the agents in  $\mathcal{A}_n$ ;
2  $\mathcal{O}^s \leftarrow$  convert the paths of  $\mathcal{A} \setminus \mathcal{A}_n$  into soft obstacles;
3 foreach agent  $a_i \in \mathcal{A}_n$  do
4    $p_i \leftarrow$  SIPPS( $\mathcal{V}, \mathcal{E}, a_i.s, a_i.g, \emptyset, \mathcal{O}^s$ );
5   if  $p_i$  does not exist then
6     return No solution;
7    $\mathcal{O}_i \leftarrow$  convert  $p_i$  into soft obstacles;
8    $\mathcal{O}^s \leftarrow \mathcal{O}^s \cup \mathcal{O}_i$ ;
9 return  $\mathcal{P}$ ;
```

In the original Prioritized Planning (PP) algorithm, A* or SIPP is used as the low-level search algorithm to compute paths for individual agents. Both A* and SIPP treat all obstacles as hard obstacles, aiming to minimize path costs while avoiding

collisions. However, due to its incompleteness, PP may fail to find a solution in some instances, as the hard obstacles formed by previously planned paths can make future paths unsolvable even when a valid solution exists but is not consistent with the assigned priorities. In contrast, PP-SIPPS ensures completeness by not treating any obstacles as hard obstacles. If SIPPS (without hard obstacles) fails to find a solution, it indicates that no valid path exists from the start node to the goal node, regardless of the paths of other agents. This implies that no feasible solution can be found within the entire neighborhood. Otherwise, a solution with a minimal number of collisions under the assigned priorities is always obtainable in the neighborhood.

4.2.2 Collision-Based Neighborhoods

The neighborhoods can be generated by selecting a subset of agents from $\mathcal{A}$ whose paths collide with each other. We define a collision graph $\mathcal{G}_c = (\mathcal{V}_c, \mathcal{E}_c)$, where $\mathcal{V}_c = \mathcal{A}$ correspond to the agents and $\mathcal{E}_c = \{\langle a_i, a_j \rangle \mid p_i \text{ collides with } p_j\}$ correspond to the collisions between the paths of the agents. In Algorithm 17, we generate a collision-based neighborhood of size N , where N is a user-defined parameter.

We begin by randomly selecting an agent $a_v \in \mathcal{V}_c$ that is a non-isolated node in the collision graph $\mathcal{G}_c$ and identifying the largest connected component $\mathcal{G}'_c = (\mathcal{V}'_c, \mathcal{E}'_c)$ using a breadth-first search (BFS), ensuring that $\mathcal{G}'_c \subseteq \mathcal{G}_c$ and $a_v \in \mathcal{V}'_c$ (lines 1-2). a_v can always be found when there exist collisions between the paths of the agents. If the size of $\mathcal{V}'_c$ is no larger than N , we add all agents from $\mathcal{V}'_c$ into the neighborhood $\mathcal{A}_n$, though the number of agents may still be insufficient (lines 3-4). To reach the required size, we perform a random walk, incrementally adding agents until $\mathcal{A}_n$ contains exactly N agents. During each random walk (lines 12-29), we randomly select an agent a_k from $\mathcal{A}_n$, choose a random node from its path to start the random walk, and add agents that collide with this random walk path (lines 5-7). If $\mathcal{V}'_c$ contains more agents than required, we randomly select up to $N - |\mathcal{A}_n|$ agents from $\mathcal{V}'_c$, ensuring that they remain connected within $\mathcal{G}'_c$. This constraint guarantees that the neighborhood size does not exceed the predefined limit N (lines 8-9).

In the random walk function (lines 12-29), suppose $\pi_{k,j}$ represents a random action on path p_k for $j \in [1, n_k]$. If $\pi_{k,j}$ is a *wait* action, the waiting node x is selected as the

starting node; otherwise, for a *move* action, the source node x is used (lines 13-15). At each step, we determine potential neighbor nodes N_x of x that might provide a lower-cost path to the goal node $a_k.g$. The minimal time required for agent a_k to travel from x to y is precomputed as $t_{a_k}(x, y)$. If $t + t_{a_k}(x, v) + t_{a_k}(v, a_k.g) < |p_k|$, then agent a_k may be able to reach its goal sooner than by strictly following p_k , making v a potential neighbor (lines 17-20). If no potential neighbors exist, the random walk terminates (lines 21-22). Otherwise, a node y is randomly selected from N_x , and an action π is generated with edge $x \rightarrow y$ starting at time t (lines 23-24). If this action collides with the path of another agent a_i that is not already in $\mathcal{A}_n$, we add a_i to the neighborhood while ensuring that at most $N - |\mathcal{A}_n|$ agents are added, maintaining the neighborhood size constraint (lines 25-27). Finally, before proceeding to the next step of the random walk, we update (x, t) to $(y, t + t_{a_k}(x, y))$ (line 28), ensuring the step is properly executed.

4.2.3 Failure-Based Neighborhoods

The neighborhoods can be generated by selecting agents that caused failures when attempting to find collision-free paths in previous LNS iterations. Although our low-level algorithm (SIPPS) never fails since it always finds a path for an agent whenever its start node s and goal node g are connected in the graph $\mathcal{G}$, we define failure based on whether the resulting path contains any collisions. The failure-based neighborhood selection focuses on an agent a_i in two common failure scenarios in PP-SIPPS: (1) collisions occur between a_i and the paths of other agents at the start node s_i and (2) collisions occur between the path p_i from s_i to g_i and the goal nodes of other agents.

In Algorithm 18, we begin by initializing the probability of an agent a_v being randomly selected into the neighborhood as $\deg(a_v)/(2|\mathcal{E}_c|)$ (we know $\sum_{a_j \in \mathcal{V}_c} \deg(a_j) = 2|\mathcal{E}_c|$ from the property of the graph $\mathcal{G}_c$, where $\deg(a_j)$ is the degree of agent a_j in the graph $\mathcal{G}_c$, representing the number of paths that collide with a_j 's path), ensuring that agents whose paths have more collisions with others are more likely to be selected, while agents with zero collisions are never chosen (lines 1-4). Next, we identify two sets of agents that may contribute to failures: $\mathcal{A}_s$ and $\mathcal{A}_g$. The set $\mathcal{A}_s$ consists of agents whose paths collide with the start node s_v of agent a_v (line 5), while $\mathcal{A}_g$ includes agents whose goal nodes collide with a_v 's path p_v (line 6).

Algorithm 17: GenerateCollisionBasedNeighborhood($\mathcal{V}_c, \mathcal{E}_c, N$)

```

1  $a_v \leftarrow$  randomly select an agent from  $\mathcal{V}_c$  that is a non-isolated node in  $\mathcal{G}_c$ ;
2  $\mathcal{G}'_c = (\mathcal{V}'_c, \mathcal{E}'_c) \leftarrow$  largest connected component where  $\mathcal{G}'_c \subseteq \mathcal{G}_c$  and  $a_v \in \mathcal{V}'_c$ ;
3 if  $|\mathcal{V}'_c| \leq N$  then
4    $\mathcal{A}_n \leftarrow \mathcal{V}'_c$ ;
5   while  $|\mathcal{A}_n| < N$  do
6      $a_k \leftarrow$  randomly select an agent from  $\mathcal{A}_n$ ;
7      $\mathcal{A}_n \leftarrow \text{RandomWalk}(\mathcal{A}_n, a_k)$ ;
8 else
9    $\mathcal{A}_n \leftarrow$  randomly select at most  $N - |\mathcal{A}_n|$  agents from  $\mathcal{V}'_c$  so that these
    agents are connected in  $\mathcal{G}'_c$ ;
10 return  $\mathcal{A}_n$ ;
11
12 Function RandomWalk( $\mathcal{A}_n, a_k$ ):
13    $(x, t, t') \leftarrow \pi_{k,j}$  where  $j$  is a random integer in  $[1, n_k]$ ;
14   if  $x \in \mathcal{E}$  then
15      $x \leftarrow$  the source node of  $x$ ;
16   while  $|\mathcal{A}_n| < N$  do
17      $N_x \leftarrow \emptyset$ ;
18     foreach node  $v$  so that  $\langle x, v \rangle \in \mathcal{E}$  do
19       if  $t + t_{a_k}(x, v) + t_{a_k}(v, a_k.g) < |p_k|$  then
20          $N_x \leftarrow N_x \cup \{v\}$ ;
21     if  $N_x = \emptyset$  then
22       break;
23      $y \leftarrow$  randomly select a node in  $N_x$ ;
24      $\pi \leftarrow (\langle x, y \rangle, t, t + t_{a_k}(x, y))$ ;
25      $\mathcal{A}'_n \leftarrow \{a_i \in \mathcal{A} \setminus \mathcal{A}_n \mid p_i \text{ collides with } \pi\}$ ;
26      $\mathcal{A}'_n \leftarrow$  randomly select at most  $N - |\mathcal{A}_n|$  agents from  $\mathcal{A}'_n$ ;
27      $\mathcal{A}_n \leftarrow \mathcal{A}_n \cup \mathcal{A}'_n$ ;
28      $(x, t) \leftarrow (y, t + t_{a_k}(x, y))$ ;
29 return  $\mathcal{A}_n$ ;

```

If there are not enough agents in $\mathcal{A}_s \cup \mathcal{A}_g$ to form a neighborhood of size N (lines 7-8), we iteratively add more agents until $|\mathcal{A}_n| = N$ by selecting agents whose goal nodes collide with the paths of agents already included in $\mathcal{A}_s$ (lines 10-17). In each iteration, we randomly select an agent a_j from the neighborhood $\mathcal{A}_n$ and find $\mathcal{A}_g$ where the goal nodes of agents in $\mathcal{A}_g$ collide with p_j and these colliding agents have not been added into the neighborhood $\mathcal{A}_n$ yet (line 11). To avoid repeatedly finding an empty $\mathcal{A}_g$ when selecting the agent a_j , we maintain a visited set $\mathcal{A}_v$ to record the agents that have an empty corresponding $\mathcal{A}_g$ (line 9). The visited set contains a_v initially because all agents whose goal nodes collide with a_v have been added into $\mathcal{A}_n$ in line 8. Thus, in line 11, we should exclude the agents in $\mathcal{A}_v$ when randomly selecting the agent a_j . After a_j is selected, if $\mathcal{A}_g$ is not empty, we randomly select an agent from $\mathcal{A}_g$ and add it to the neighborhood $\mathcal{A}_n$ (lines 13-15). We update the visited set (add a_j to $\mathcal{A}_v$) when $\mathcal{A}_g$ contains zero or one agent. Since no more agents can be added via a_j , further unnecessary checks are prevented (lines 16-17). After this process, the number of agents in the neighborhood $\mathcal{A}_n$ can still be smaller than N . In this case, we will proceed the LNS iteration with a smaller neighborhood.

If there are too many agents in $\mathcal{A}_s \cup \mathcal{A}_g$, we select a subset to form the neighborhood. We prioritize agents in $\mathcal{A}_g$ over those in $\mathcal{A}_s$, as empirical results suggest that the scenario involving $\mathcal{A}_g$ occurs more frequently in PP [10]. To avoid duplication, we first remove agents that exist in both sets from $\mathcal{A}_s$ (line 19). If $\mathcal{A}_s$ is empty, we randomly select $N - 1$ agents from $\mathcal{A}_g$ (lines 20-22). Otherwise, we randomly select at most $N - 2$ agents from $\mathcal{A}_g$ to ensure that at least one agent from $\mathcal{A}_s$ remains (line 24), as removing all agents from $\mathcal{A}_s$ tends to degrade performance. Next, we sort the agents in $\mathcal{A}_s$ in ascending order based on the collision time of their paths with s_v and select the first $N - 1 - |\mathcal{A}_g|$ agents (line 25). This process guarantees that the neighborhood $\mathcal{A}_n$ always contains exactly N agents.

4.2.4 Random Neighborhoods

The neighborhoods can also be generated randomly, a simple yet effective approach for many problems. In Algorithm 19, we first assign a weight w_i to each path p_i based on the number of paths that collide with p_i ($\deg(a_i)$), plus one (lines 1-2). The additional one ensures that even collision-free paths have a chance of being

Algorithm 18: GenerateFailureNeighborhood($\mathcal{V}_c, \mathcal{E}_c, N$)

```

1 foreach  $a_i \in \mathcal{V}_c$  do
2    $P_i \leftarrow \deg(a_i)/(2|\mathcal{E}_c|)$ ;
3  $a_v \leftarrow$  randomly select an agent from  $\mathcal{A}$  with probabilities  $P_i$ 's;
4  $\mathcal{A}_n \leftarrow \{a_v\}$ ;
5  $\mathcal{A}_s \leftarrow \{a_i \in \mathcal{V}_c \setminus \mathcal{A}_n \mid p_i \in \mathcal{P} \text{ collides with } s_v\}$ ;
6  $\mathcal{A}_g \leftarrow \{a_i \in \mathcal{V}_c \setminus \mathcal{A}_n \mid p_v \text{ collides with } g_i\}$ ;
7 if  $|\mathcal{A}_s \cup \mathcal{A}_g| < N - 1$  then
8    $\mathcal{A}_n \leftarrow \mathcal{A}_n \cup \mathcal{A}_s \cup \mathcal{A}_g$ ;
9    $\mathcal{A}_v \leftarrow \{a_v\}$ ;
10  while  $|\mathcal{A}_n| < N$  and  $\mathcal{A}_n \neq \mathcal{A}_v$  do
11     $a_j \leftarrow$  randomly select an agent from  $\mathcal{A}_n \setminus \mathcal{A}_v$ ;
12     $\mathcal{A}_g \leftarrow \{a_i \in \mathcal{A} \setminus \mathcal{A}_n \mid p_j \text{ collides with } g_i\}$ ;
13    if  $\mathcal{A}_g \neq \emptyset$  then
14       $a_k \leftarrow$  randomly select an agent from  $\mathcal{A}_g$ ;
15       $\mathcal{A}_n \leftarrow \mathcal{A}_n \cup \{a_k\}$ ;
16    if  $|\mathcal{A}_g| \leq 1$  then
17       $\mathcal{A}_v \leftarrow \mathcal{A}_v \cup \{a_j\}$ ;
18 else
19    $\mathcal{A}_s \leftarrow \mathcal{A}_s \setminus \mathcal{A}_g$ ;
20   if  $\mathcal{A}_s = \emptyset$  then
21      $\mathcal{A}_g \leftarrow$  randomly select  $N - 1$  agents from  $\mathcal{A}_g$ ;
22      $\mathcal{A}_n \leftarrow \mathcal{A}_n \cup \mathcal{A}_g$ ;
23   else
24      $\mathcal{A}_g \leftarrow$  randomly select at most  $N - 2$  agents from  $\mathcal{A}_g$ ;
25      $\mathcal{A}_s \leftarrow$  select the first  $N - 1 - |\mathcal{A}_g|$  agents from  $\mathcal{A}_s$  in ascending order
        based on the collision time of their paths and  $s_v$ ;
26      $\mathcal{A}_n \leftarrow \mathcal{A}_n \cup \mathcal{A}_s \cup \mathcal{A}_g$ ;
27 return  $\mathcal{A}_n$ ;

```

selected, but with a lower probability. We then compute the selection probability P_i for each path p_i by dividing w_i by the sum of all weights (lines 3-4). Using these probabilities, we randomly sample a neighborhood from $\mathcal{A}$. During this sampling process, we ensure that each agent is selected at most once, guaranteeing that the neighborhood $\mathcal{A}_n$ consists of distinct agents.

Algorithm 19: GenerateRandomNeighborhood($\mathcal{V}_c, \mathcal{E}_c, N$)

```

1 foreach  $a_i \in \mathcal{V}_c$  do
2    $w_i \leftarrow \deg(a_i) + 1$ ;
3 foreach  $a_i \in \mathcal{V}_c$  do
4    $P_i \leftarrow \frac{w_i}{\sum_{a_j \in \mathcal{V}_c} w_j}$ ;
5  $\mathcal{A}_n \leftarrow$  randomly sample from  $\mathcal{A}$  with probabilities  $P_i$ 's;
6 return  $\mathcal{A}_n$ ;

```

4.2.5 Adaptive LNS (ALNS)

Different neighborhood selection methods may be effective in some instances but not in others, or in different stages of solving the same instance. Relying on a single selection method for all instances or throughout the entire solving process may therefore be suboptimal. Adaptive LNS (ALNS) leverages multiple neighborhood selection methods within a single instance, dynamically choosing the most promising one based on its past performance. This is achieved by tracking the number of reduced collisions (*gain*) for each method in previous LNS iterations and updating the weight of the corresponding method based on the *gain* in each iteration.

Algorithm 20 shows our complete LNS algorithm with Adaptive LNS. First, we select all agents in $\mathcal{A}$ as a neighborhood and find the initial path with PP-SIPPS with some collisions between the agents on their paths (lines 1-2). Similar to the random neighborhood selection method, we maintain a weight w_m (initially set to 1) for each method m and update it iteratively (lines 3-5). In each LNS iteration, a method m is selected with probability $w_m / \sum_j w_j$ to generate the next neighborhood and replan the paths with PP-SIPPS again (lines 6-9). If the number of collisions does not decrease compared to the previous solution, we set *gain* to 0; otherwise, it is set to the number of collisions reduced, and the best plan $\mathcal{P}$ is updated as well (lines 10-15). The weight w_m is then updated as $lr \cdot \text{gain} + (1 - lr) \cdot w_m$, where $lr \in [0, 1]$ is a user-defined learning rate that controls how quickly w_m adapts during LNS (lines 16). Typically, a small learning rate (e.g., 0.1) is used to prevent the algorithm from overreacting to a single iteration with a large *gain*, which could cause it to become stuck in a local minimum. The number of iterations can be set as a fixed user-defined value or determined by the maximum number of iterations that can be completed within a given time limit. If collisions still exist after all

Algorithm 20: ALNS

```

1  $\mathcal{P} \leftarrow \text{PP-SIPPS}(\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{A});$ 
2  $C \leftarrow$  number of collisions in  $\mathcal{P}$  in CAT;
3  $methods \leftarrow \{\text{collision-based, failure-based, random}\};$ 
4 foreach  $method\ m$  in  $methods$  do
5    $w_m \leftarrow 1;$ 
6 foreach  $LNS$  iteration do
7    $m \leftarrow$  randomly choose a method from  $methods$  with weights  $w$ ;
8    $\mathcal{A}_n \leftarrow$  generate a neighborhood with  $m$ ;
9    $\mathcal{P}' \leftarrow \text{PP-SIPPS}(\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{A}_n);$ 
10   $C' \leftarrow$  number of collisions in  $\mathcal{P}'$ ;
11  if  $C \geq C'$  then
12     $gain \leftarrow 0;$ 
13  else
14     $gain \leftarrow C - C';$ 
15     $\mathcal{P} \leftarrow \mathcal{P}', C \leftarrow C';$ 
16   $w_m \leftarrow lr \cdot gain + (1 - lr) \cdot w_m;$ 
17 if  $C > 0$  then
18   return “No collision-free solution”,  $\mathcal{P};$ 
19 return  $\mathcal{P};$ 

```

LNS iterations, a collision-free solution cannot be found. However, we can still return $\mathcal{P}$ as the best-effort solution obtained, containing the minimal number of collisions encountered during the search (lines 17-19).

4.3 Experimental Evaluation

We conduct experiments on five distinct maps as shown in Figure 4.2 from the Moving AI MAPF benchmark [2]. The first map, empty-32-32 (Figure 4.2a), contains no obstacle and serves as a baseline for testing algorithmic optimality. The second map, room-32-32-4 (Figure 4.2b), consists of 8 rows and 8 columns of 3×3 rooms separated by walls, with 1-unit-wide corridors connecting them. This map introduces more shared locations along agent paths, as they must navigate through narrow passageways. The third map, random-32-32-20 (Figure 4.2c), has 20% of its grid cells randomly occupied by obstacles while ensuring full connectivity, allowing any location to be reached. The fourth map, warehouse-10-20-10-2-1 (Figure 4.2d),

simulates a real warehouse layout with storage racks. The fifth map, den520d (Figure 4.2e), originates from the game Dragon Age: Origins (DAO) and is significantly larger (257×256), making it ideal for evaluating algorithmic efficiency in extreme cases.

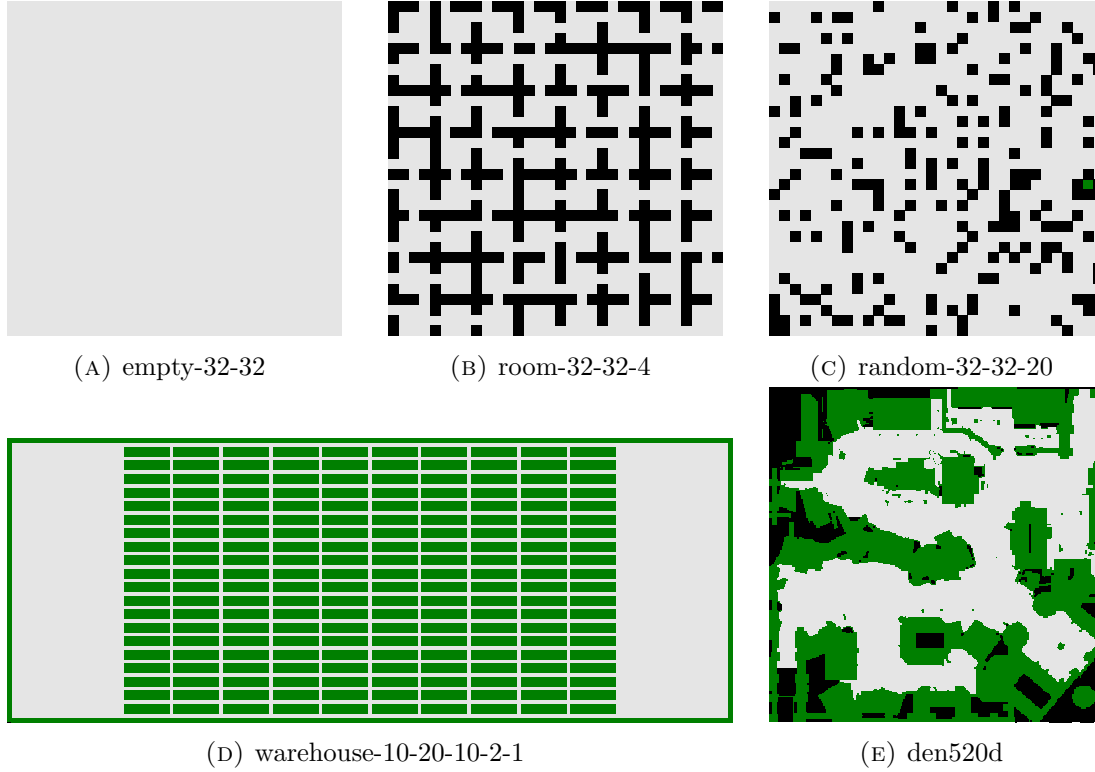


FIGURE 4.2: Maps from the Moving AI MAPF benchmark [2].

We use the even scenario files from the same benchmark where the shortest paths of the agents have an even mix of short and long in length. Each map comes with 25 scenario files, from which we extract the first M rows to generate start and goal locations for different numbers of agents M . To enable continuous-time *move* actions, we conduct experiments on 2^k -connected variants of each map, where k is a parameter. A 2^k -connected grid means that each node connects to 2^k neighboring nodes, allowing agents to move accordingly if no collisions occur. When $k = 2$ (Figure 4.3a), agents can move in four directions and the distance of each move is 1, making the instance solvable by discrete MAPF solvers. When $k = 3$ (Figure 4.3b), agents can move in eight directions, enabling more efficient navigation by allowing diagonal movements. This provides a shorter transition via diagonal edges of length $\sqrt{2}$ compared to taking two consecutive orthogonal steps with a total length of 2. When $k = 4$ (Figure 4.3c), eight additional edges are introduced, further increasing

movement flexibility. We use $k = 2, 3, 4$ to generate three different instances per setup, where a larger k theoretically results in shorter travel distances and lower sum-of-costs.

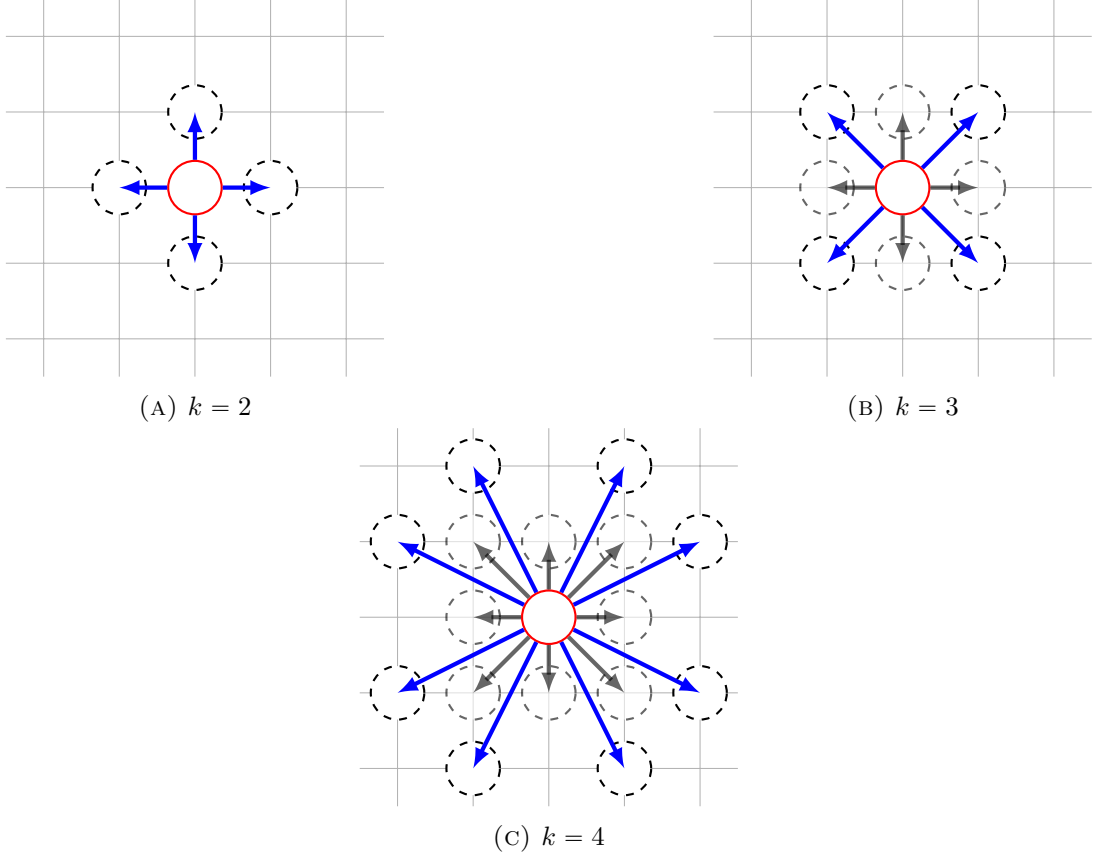


FIGURE 4.3: Illustration of 2^k -connected grids.

In our experiments, we employ the Continuous-time Large Neighborhood Search (CLNS) framework with a neighborhood size of $N = 8$ and Adaptive LNS with a learning rate of $lr = 0.1$, running for a maximum of 1000 LNS iterations to plan agent paths for each instance. We compare CLNS with the state-of-the-art Continuous-time CBS (CCBS) algorithm in terms of both the sum of costs and computation time. All agents are modeled as disk-shaped entities with a radius of $\sqrt{2}/4$, moving at a speed of 1 per time unit. This radius, chosen by CCBS, ensures comparability with discrete MAPF solvers such as CBS when $k = 2$, as it represents the largest radius that allows agents to safely execute moves where they follow each other. However, we observe that CCBS has a low success rate on certain maps within a reasonable time limit (e.g., 60 seconds) when the number of agents exceeds 20, making fair comparisons across all instances infeasible.

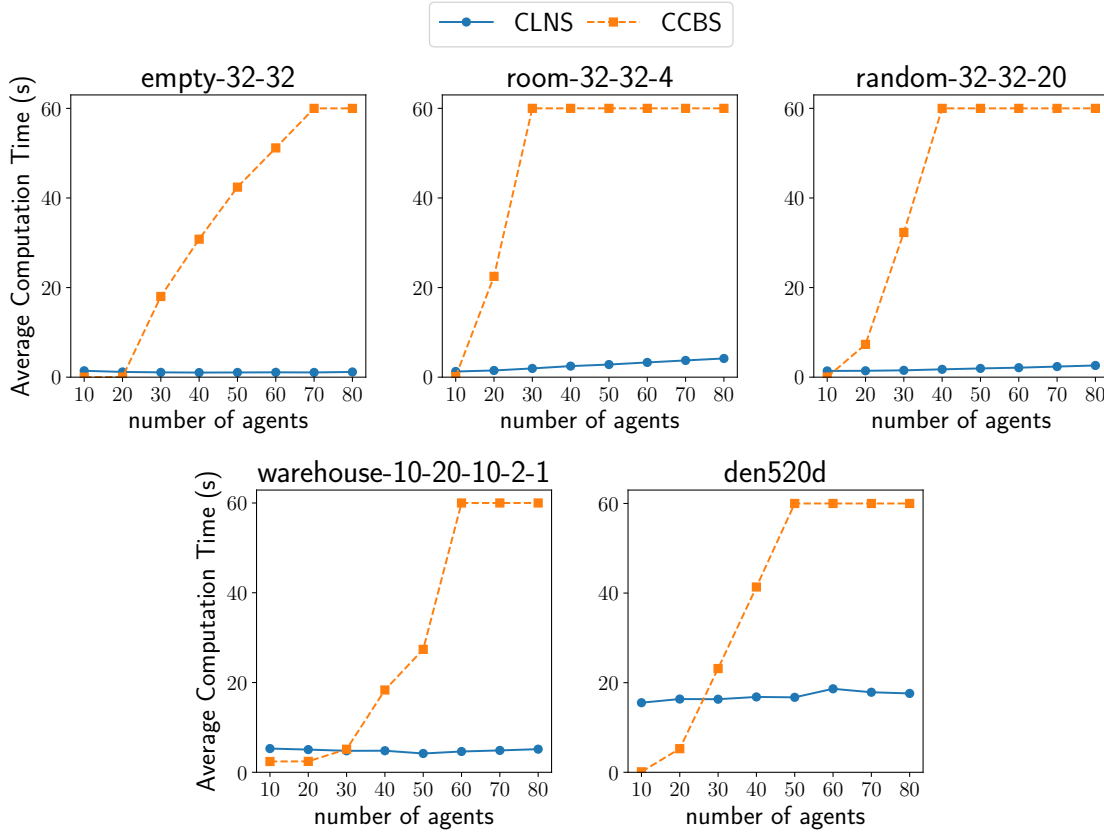
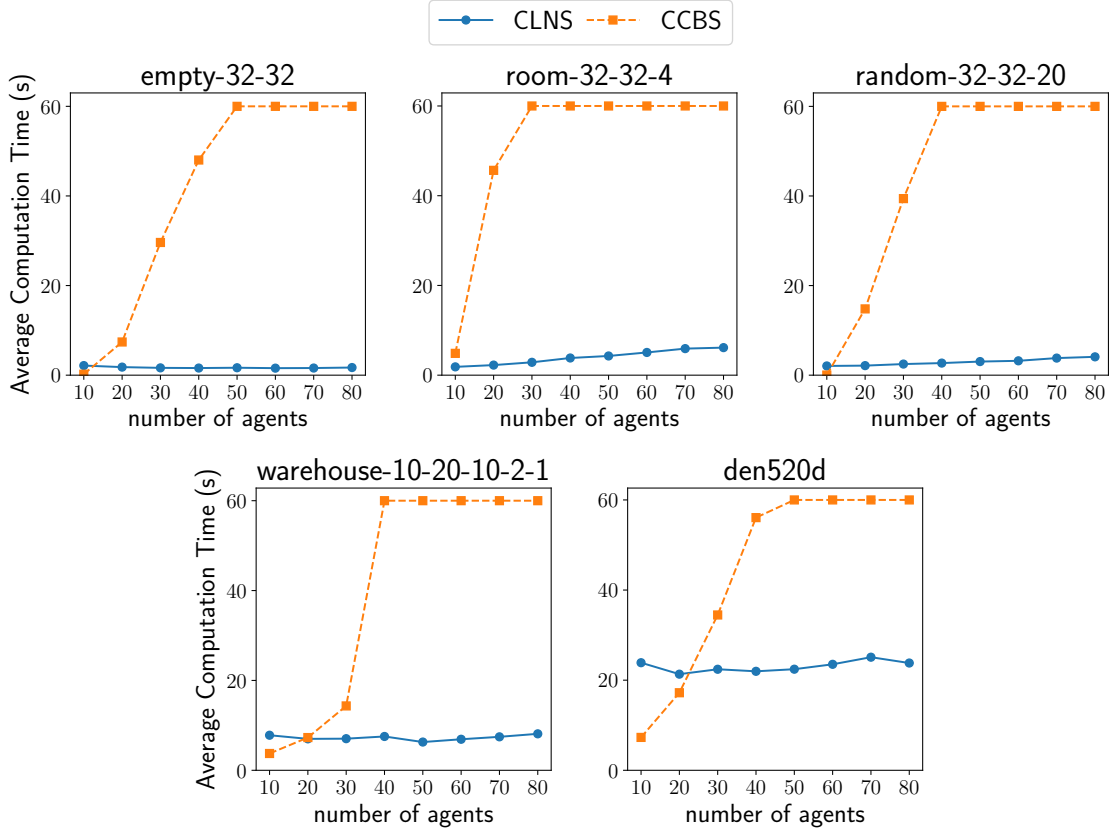


FIGURE 4.4: Results for 2^k -connected maps when $k = 2$.

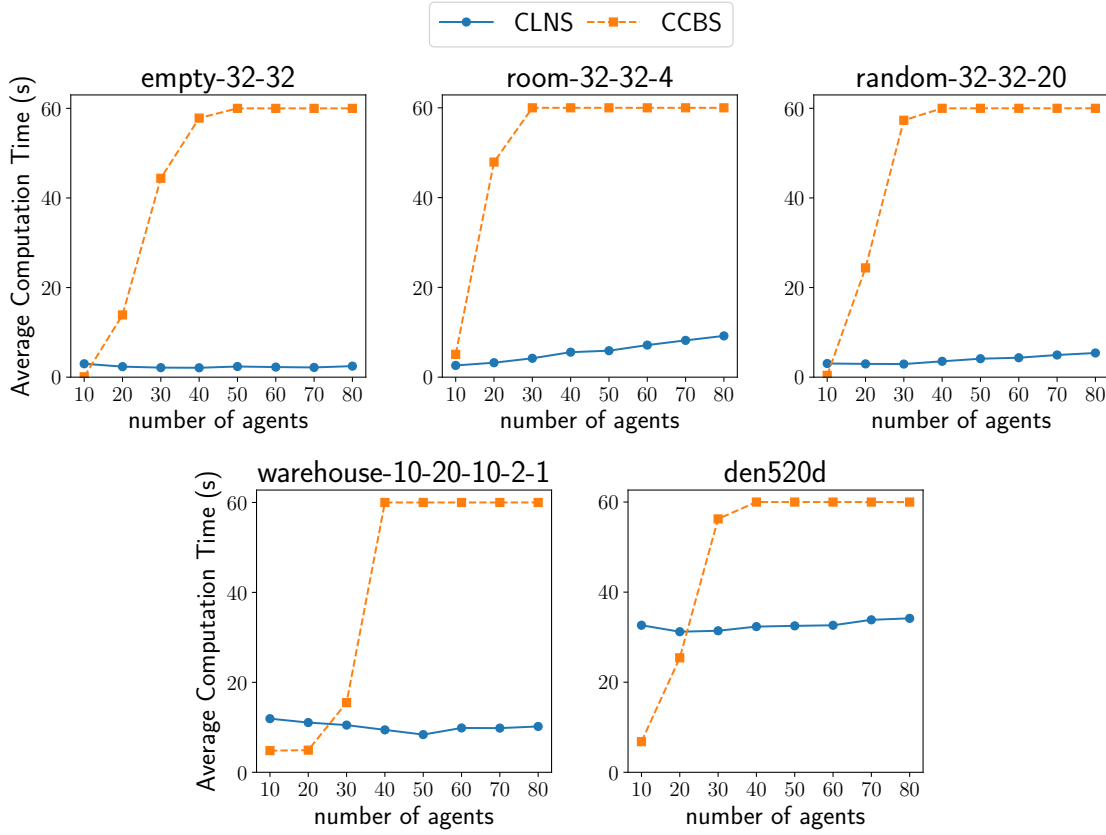
We omit the plots that compare the sum of costs for solutions generated by CLNS and CCBS because they remain nearly identical. The difference is smaller than 1% between two algorithms for the same instance whenever CCBS successfully finds a solution within 60 seconds. Given that CCBS is an optimal continuous-time MAPF solver, this indicates that the solutions produced by CLNS are of high quality.

In Figures 4.4, 4.5, and 4.6, we compare the average computation times of CLNS and CCBS. If CCBS fails to find a solution within 60 seconds for a given instance, we set its computation time to 60 seconds in the plot, meaning that its actual computation time could be even larger. For CCBS data points with a computation time of 60 seconds, this indicates that CCBS fails to solve any instance for the given M and k within the time limit. When the number of agents is small ($M = 10, 20, 30$), CCBS sometimes outperforms CLNS. This is because CLNS runs 1000 iterations per instance regardless of the number of agents, leading to unnecessary overhead when the number of agents is too low to benefit from further iterations. It is possible to terminate the iterations early. For example, after each iteration, if no conflicts remain and the sum-of-costs shows little change across

FIGURE 4.5: Results for 2^k -connected maps when $k = 3$.

several consecutive iterations, the process can be stopped. However, the optimal threshold for determining when to terminate may vary across different maps and agent densities. This is a potential direction for future work.

However, when there are more than 30 agents, CLNS consistently requires significantly less computation time than CCBS. More interestingly, while the computation time of CCBS increases exponentially as the number of agents grows, the computation time of CLNS does not increase rapidly. For the room-32-32-4, random-32-32-20, and den520d maps, the computation time of CLNS increases linearly with a relatively small slope, whereas for the empty-32-32 and warehouse-10-20-10-2-1 maps, the computation time of CLNS remains nearly constant. This may be due to two factors: (1) in empty-32-32, PP can already find an initial solution with few collisions, reducing the computation in LNS iterations; (2) in warehouse-10-20-10-2-1, the open spaces on the left and right sides are similar to the empty map, and corridor conflicts in the middle can be efficiently handled.

FIGURE 4.6: Results for 2^k -connected maps when $k = 4$.

4.4 Summary

This chapter proposed a hierarchical and efficient solver for continuous-time Multi-Agent Path Finding (MAPF). The method integrates Safe Interval Path Planning with Soft Constraints (SIPPS) at the low level and Large Neighborhood Search (LNS) at the high level to improve solution quality and runtime scalability. Several neighborhood selection strategies were explored to enhance performance across different map structures. Experimental evaluations confirmed that the proposed solver consistently outperforms existing continuous-time approaches. The developed solver not only advances continuous-time MAPF research but also provides high-quality plans that serve as inputs for robust execution under uncertainty, which is investigated in Chapter 5.

Chapter 5

Robust Continuous-Time Multi-Agent Path Execution

In this chapter, we develop a robust continuous-time path execution framework to deal with unexpected delays arising from various reasons. Robust path executions are crucial but also challenging because agents may suffer from unforeseen failures and anomalies like breakdowns and dynamic obstacles in the environment. These unexpected conditions are generally difficult to model or predict beforehand. If they are not handled carefully, the robustness and effectiveness of path executions would be significantly affected.

The framework takes the planned paths of agents as input and coordinates the path execution in an online fashion based on the evolving states of agents. We first define all possible conflicts between the agents, taking into account non-uniform action durations, volumes of individual agents and distinct trajectories of paths. We then construct a state dependency graph to model the conflicts and formulate a feasibility problem to infer whether all the agents can successfully reach their goal locations following their respective paths. We develop execution policies to guarantee that the path execution is conflict-free and deadlock-free. To optimize the effectiveness of path execution, our execution policies enable as many agents as possible to move towards their goal locations concurrently. Experimental results demonstrate the correctness, effectiveness and efficiency of our algorithms.

5.1 Problem Definition

We mostly follow the problem definition of the continuous multi-agent pathfinding problem in Section 2.3.1. Consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of M agents $\mathcal{A} = \{a_1, \dots, a_M\}$, each of which has a given shape and size, and can move with a specified speed. Each agent a_i has a start location $s_i \in \mathcal{V}$, a goal location $g_i \in \mathcal{V}$, and a predefined path p_i from s_i to g_i expressed by a sequence of nodes $s_i = v_{i,1} \rightarrow v_{i,2} \rightarrow \dots \rightarrow v_{i,n_i} = g_i$, where each node $v_{i,j} \in \mathcal{V}$, each edge $\langle v_{i,j}, v_{i,j+1} \rangle \in \mathcal{E}$ and n_i is the number of nodes on the path. A path plan $\mathcal{P} = \{p_1, \dots, p_M\}$ is a collection of paths for all agents. To execute a path plan $\mathcal{P} = \{p_1, \dots, p_M\}$, each agent a_i moves from its start location s_i to its goal location g_i through the path p_i . Note that our representation of a path plan is more general than the output of a MAPF solver (the path plan defined in Section 2.3.1) in that we do not associate timestamps with locations or actions. Thus, our analysis and execution policies can also be applied to any agent paths specified by applications such as security patrolling.

At any time, an agent can be executing either a *wait* action or a *move* action. When the agent executes a *wait* action, it stays at its current node. When the agent executes a *move* action, it travels from the current node to an adjacent node along an edge on its path. The time taken to arrive at the adjacent node is not deterministic due to unexpected conditions, which results in the *uncertainty* that we aim to deal with. Once starting to execute a *move* action, the agent has to continue executing the *move* action until it reaches the adjacent node. When the agent arrives at the adjacent node, it may then execute either a *wait* action to stay at the node or a *move* action to start traveling along the next edge on its path.

We consider three types of conflicts between agents:

- A *node-node conflict* occurs when two agents are both executing *wait* actions and their shapes overlap geometrically. Note that since agents have volumes, a node-node conflict can occur even when agents are staying at different (but nearby) nodes (see Figure 5.1(a)).
- An *edge-edge conflict* occurs when two agents are both executing *move* actions and there exist locations along the respective trajectories of their edges

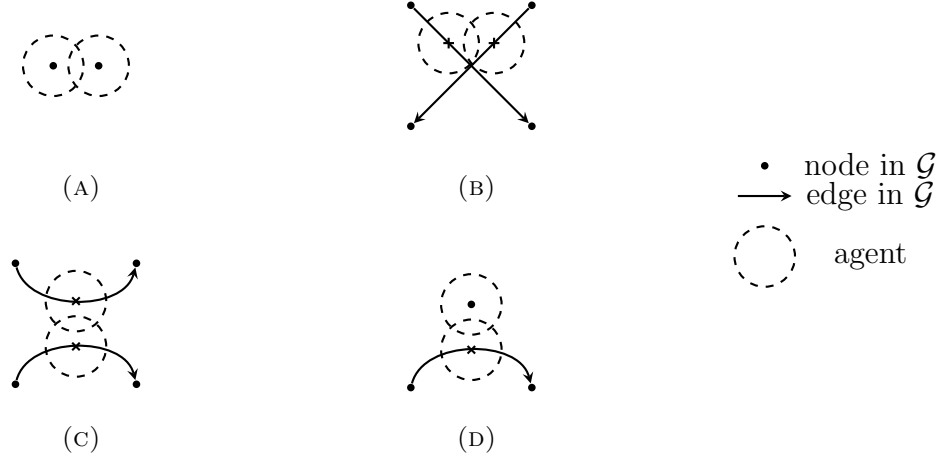


FIGURE 5.1: Some example conflicts.

where the shapes of the two agents overlap geometrically. In general, an edge-edge conflict occurs when two edges intersect geometrically (see Figure 5.1(b)). Due to agent volumes, an edge-edge conflict can also occur even if two edges do not intersect geometrically (see Figure 5.1(c)).

- A *node-edge conflict* occurs when one agent is executing a *wait* action and the other agent is executing a *move* action, and there is geometric overlap between the shape of the former agent at its node and the shape of the latter agent somewhere along the trajectory of its edge. Again, owing to agent volumes, a node-edge conflict can occur even if the node and edge do not intersect geometrically (see Figure 5.1(d)).

We assume that no conflict exists initially (resp. finally) when all agents stay at their start (resp. goal) locations. Given a path plan $\mathcal{P}$, we would like to coordinate the *move* and *wait* actions taken by agents to ensure that there is no conflict of any type throughout the execution and all agents eventually arrive at their goal locations.

5.1.1 Feasibility and State Dependency Graph

A path plan $\mathcal{P}$ is said *feasible* if it can be executed without conflicts and deadlocks and all agents can arrive at their goal locations. Otherwise, the path plan is said

infeasible. To infer the feasibility of a path plan, we need to resolve all potential conflicts between agents by deciding the order for them to occupy the node(s) and/or edge(s) involved in the conflicts. We design a state dependency graph for this purpose.

On the path $v_{i,1} \rightarrow v_{i,2} \rightarrow \dots \rightarrow v_{i,n_i}$ of an agent a_i , we refer to each node $v_{i,j}$ as a *node state*, and each edge $e_{i,j} = \langle v_{i,j}, v_{i,j+1} \rangle$ as an *edge state*. To execute the path, the agent a_i has to achieve the node and edge states alternately and sequentially in the order of $v_{i,1}, e_{i,1}, v_{i,2}, e_{i,2}, \dots, e_{i,n_i-1}, v_{i,n_i}$. The state dependency graph (SDG) is a directed graph $\mathcal{G}_{\text{SDG}} = (\mathcal{V}_{\text{SDG}}, \mathcal{E}_{\text{SDG}})$, where $\mathcal{V}_{\text{SDG}}$ is the set of all possible node and edge states of the agents, and $\mathcal{E}_{\text{SDG}}$ specifies the order for the agents to achieve their states. $\mathcal{E}_{\text{SDG}}$ is composed of three parts: $\mathcal{E}_{\text{SDG}_1}$, $\mathcal{E}_{\text{SDG}_2}$ and $\mathcal{E}_{\text{SDG}_3}$. The first part $\mathcal{E}_{\text{SDG}_1}$ is based on the path of each individual agent and includes the edges $\langle v_{i,j}, e_{i,j} \rangle$'s and $\langle e_{i,j}, v_{i,j+1} \rangle$'s as illustrated by solid lines in Figure 5.2, meaning that agent a_i cannot achieve the edge state $e_{i,j}$ before the node state $v_{i,j}$, or achieve the node state $v_{i,j+1}$ before the edge state $e_{i,j}$. The edges in $\mathcal{E}_{\text{SDG}_1}$ are called *predetermined edges* because these orders are fixed in any path execution.

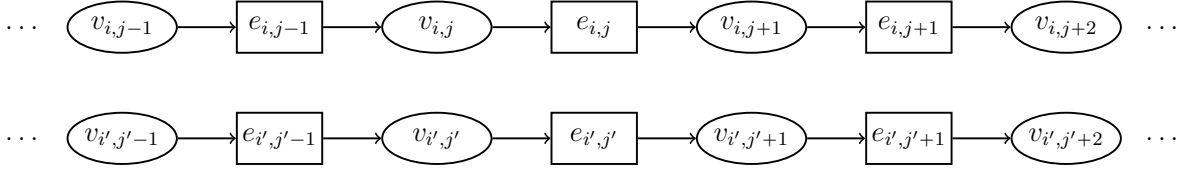


FIGURE 5.2: An illustration of $\mathcal{E}_{\text{SDG}_1}$ in SDG.

The other two parts $\mathcal{E}_{\text{SDG}_2}$ and $\mathcal{E}_{\text{SDG}_3}$ are to resolve the order of two states from different agents that are involved in potential conflicts. Two possible dependency edges can be formed for each conflict according to the type of the conflict.

Node-node conflict. Suppose the shapes of two agents a_i and $a_{i'}$ overlap geometrically when a_i stays at node $v_{i,j}$ and $a_{i'}$ stays at node $v_{i',j'}$. Due to uncertainty, when an agent is executing a *move* action on an edge, it can be located anywhere along the edge, including the two nodes incident to the edge. Thus, to avoid conflicts, a_i (resp. $a_{i'}$) must arrive at the next node after $v_{i,j}$ (resp. $v_{i',j'}$) on its path before $a_{i'}$ (resp. a_i) starts to move on the edge whose destination is at node $v_{i,j}$ (resp. $v_{i',j'}$). Hence, as illustrated by dashed lines in Figure 5.3, a dependency edge $\langle v_{i,j+1}, e_{i',j'-1} \rangle$ can be formed if a_i arrives at node $v_{i,j+1}$ before $a_{i'}$ starts to move on edge $e_{i',j'-1}$, or a dependency edge $\langle v_{i',j'+1}, e_{i,j-1} \rangle$ can be formed if $a_{i'}$ arrives

at node $v_{i',j'+1}$ before a_i starts to move on edge $e_{i,j-1}$. Thus, the conflict can be denoted as $\{\langle v_{i,j+1}, e_{i',j'-1} \rangle, \langle v_{i',j'+1}, e_{i,j-1} \rangle\}$.

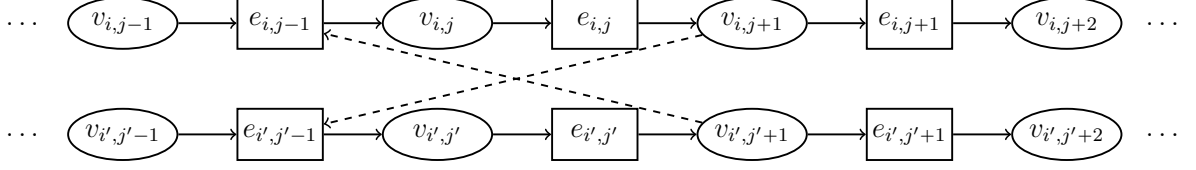


FIGURE 5.3: Dependency edges for a node-node conflict.

Edge-edge conflict. Suppose geometric overlap exists between the trajectories of two agents a_i and $a_{i'}$ when a_i moves on edge $e_{i,j}$ and $a_{i'}$ moves on edge $e_{i',j'}$. To avoid conflicts, a_i and $a_{i'}$ cannot traverse edges $e_{i,j}$ and $e_{i',j'}$ at the same time. Hence, as illustrated by dashed lines in Figure 5.4, a dependency edge $\langle v_{i,j+1}, e_{i',j'} \rangle$ can be formed if a_i arrives at node $v_{i,j+1}$ before $a_{i'}$ starts to move on edge $e_{i',j'}$, or a dependency edge $\langle v_{i',j'+1}, e_{i,j} \rangle$ can be formed if $a_{i'}$ arrives at node $v_{i',j'+1}$ before a_i starts to move on edge $e_{i,j}$. Thus, the conflict can be denoted as $\{\langle v_{i,j+1}, e_{i',j'} \rangle, \langle v_{i',j'+1}, e_{i,j} \rangle\}$.

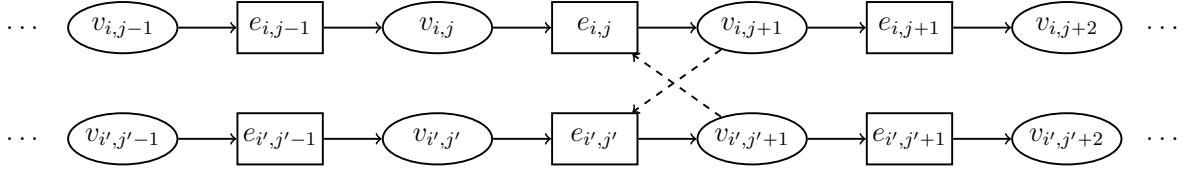


FIGURE 5.4: Dependency edges for an edge-edge conflict.

Node-edge conflict. Suppose geometric overlap exists between the trajectory of agent a_i when a_i moves on edge $e_{i,j}$ and another agent $a_{i'}$ when $a_{i'}$ stays at node $v_{i',j'}$. To avoid conflicts, a_i can only traverse edge $e_{i,j}$ before $a_{i'}$ starts to move on the edge whose destination is at node $v_{i',j'}$ or finishes moving on the edge sourced from node $v_{i',j'}$. Hence, as illustrated by dashed lines in Figure 5.5, a dependency edge $\langle v_{i,j+1}, e_{i',j'-1} \rangle$ can be formed if a_i arrives at node $v_{i,j+1}$ before $a_{i'}$ starts to move on edge $e_{i',j'-1}$, or a dependency edge $\langle v_{i',j'+1}, e_{i,j} \rangle$ can be formed if $a_{i'}$ arrives at node $v_{i',j'+1}$ before a_i starts to move on edge $e_{i,j}$. Thus, the conflict can be denoted as $\{\langle v_{i,j+1}, e_{i',j'-1} \rangle, \langle v_{i',j'+1}, e_{i,j} \rangle\}$.

We can observe that all dependency edges point from a node state to an edge state in any type of conflicts between two agents defined above. Note that one or both of the two dependency edges for a conflict can be *invalid* when the source or target

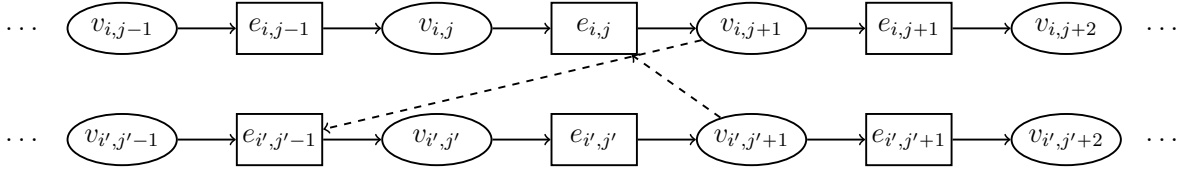


FIGURE 5.5: Dependency edges for a node-edge conflict.

of the edge is not defined in $\mathcal{V}_{\text{SDG}}$. By the above definitions, an invalid dependency edge can only occur if the node state involved in the conflict is the start location or goal location of an agent. For example, in a node-node conflict (Figure 5.3), if $v_{i,j}$ is the start location of a_i (i.e., $j = 1$), then the edge state $e_{i,j-1}$ does not exist, so the dependency edge $\langle v_{i',j'+1}, e_{i,j-1} \rangle$ is invalid. If $v_{i,j}$ is the goal location of a_i (i.e., $j = n_i$), then the edge state $e_{i,j}$ and the node state $v_{i,j+1}$ do not exist, so the dependency edge $\langle v_{i,j+1}, e_{i',j'-1} \rangle$ is invalid. The two dependency edges are both invalid only when $v_{i,j}$ and $v_{i',j'}$ are both start locations or both goal locations. Since no conflict exists initially (resp. finally) when all agents stay at their start (resp. goal) locations, such a situation would not occur in the initial SDG of the path plan.¹

If only one dependency edge for a conflict is invalid, it means that the other edge is a *predetermined edge* because the execution order to resolve the conflict is unique. In this case, the *valid* edge is added to $\mathcal{E}_{\text{SDG}_2}$, the conflict is then called a *presettled conflict* and added into the set $\mathcal{C}_{\text{presettled}}$. For example, in a node-node conflict (Figure 5.3), if $v_{i,j}$ is a start location while $v_{i',j'}$ is not, then a_i must move away from $v_{i,j}$ before $a_{i'}$ can arrive at $v_{i',j'}$ in the plan execution. If both dependency edges for a conflict are valid, we refer to these two dependency edges as a pair of *unsettled edges*. The conflict is then called an *unsettled conflict* and added into the set $\mathcal{C}_{\text{unsettled}}$. To ensure conflict-free path execution, one edge in the pair of *unsettled edges* must be chosen and added to $\mathcal{E}_{\text{SDG}_3}$. We refer to the SDG with predetermined edges $\mathcal{E}_{\text{SDG}_1} \cup \mathcal{E}_{\text{SDG}_2}$ only as a *partial* SDG, and refer to the SDG after choosing and including one edge from each pair of unsettled edges as a *complete* SDG.

If there is no directed cycle in the complete SDG, we can derive a topological ordering for the agents to achieve their node/edge states and eventually arrive at their goal locations. If there is a directed cycle in the complete SDG, a deadlock

¹However, as the plan execution progresses, such a situation may happen, meaning that a conflict occurs since the execution does not comply with either dependency edge. Thus, this should be avoided for conflict-free plan execution, as discussed later.

will be formed and not all the agents will be able to arrive at their goal locations. Given a path plan $\mathcal{P}$, if unsettled edges can be chosen to convert the partial SDG into an acyclic complete SDG, then $\mathcal{P}$ is feasible. Otherwise, $\mathcal{P}$ is infeasible.

5.1.2 Comparison with Existing Work

Our SDG is designed to address real-world challenges where agents with varying characteristics operate in unpredictable environments. It differs from existing work in the following aspects. First, our SDG includes both node and edge states. [54] includes only node states in the temporal plan graph, which is not able to model conflicts involving edges in scenarios where agents of varying shapes and sizes move in arbitrary directions. [8] models only actions in the action dependency graph, which is also inadequate because with uncertain delays, agents may need to stop at any node to avoid conflicts and deadlocks even if the initial path plan is to perform move actions along adjacent edges consecutively. Second, our SDG includes unsettled edges which offer the flexibility to dynamically adjust the order for agents to occupy nodes and edges during the execution when facing unexpected delays. In contrast, existing graph models [8, 54] have fixed edges only and the execution must follow such order.

5.2 Feasibility Test

Before presenting the feasibility test algorithm for the continuous-time setting, it is important to clarify the key differences from the discrete-time case. In continuous time, agents can move asynchronously, and conflicts may occur at arbitrary time points rather than at discrete steps, which significantly increases the complexity of detecting and resolving collisions. Moreover, the safe intervals used for coordination are no longer uniformly aligned across agents, making the feasibility evaluation and synchronization strategies developed in Chapter 3 inapplicable without major modifications. These challenges necessitate a new formulation and algorithmic design tailored for continuous motion, which are introduced in the following sections.

5.2.1 Feasibility Test Algorithm

We design a feasibility test algorithm (Algorithm 21) as a building block for our path execution solution. Suppose that the partial SDG is acyclic. The algorithm repeatedly scans through the unsettled conflicts $\mathcal{C}_{\text{unsettled}}$ one by one and selects an unsettled edge for each conflict to produce complete SDGs. Conceptually, we conduct a brute-force search over all the complete SDGs to derive the feasibility of the path plan. The search stops once an acyclic complete SDG is found. The search space is pruned once a cycle is formed.

Specifically, for each unsettled conflict, there are three possible cases for the associated pair of unsettled edges: (1) adding either unsettled edge will form a cycle in the SDG; (2) adding one unsettled edge will form a cycle in the SDG, but adding the other unsettled edge will not; (3) adding either unsettled edge will not form a cycle in the SDG. In case (1), the algorithm returns that there does not exist an acyclic complete SDG based on the current construction (lines 5-6). In case (2), the algorithm adds the edge not forming a cycle to the SDG (lines 7-10). In case (3), we tentatively skip the pair of unsettled edges. If all remaining pairs of unsettled edges fall in case (3), we then pick an arbitrary pair of unsettled edges $\langle \varepsilon, \varepsilon' \rangle$ (lines 12-13). First, we add ε to the SDG and recursively call the algorithm to see if the remaining unsettled edges can be selected to produce an acyclic complete SDG (lines 14-16). If so, the algorithm returns a positive result with the acyclic complete SDG constructed. Otherwise, we remove ε , add ε' to the SDG and make another recursive call (lines 17-18). Only when both calls return negative results, it concludes that no acyclic complete SDG exists.

Besides the optimizations developed in the discrete-time version of the feasibility test (see Chapter 3), we introduce two additional techniques that are specifically effective in the continuous-time version: removing redundant conflicts and constructing conflict groups. By eliminating conflicts that are implied by others and grouping related conflicts together, the feasibility test can scale better and perform more efficiently in large continuous-time MAPF instances.

Algorithm 21: FeasibilityTest($\mathcal{C}_{\text{unsettled}}$)

```

1 while  $\mathcal{C}_{\text{unsettled}} \neq \emptyset$  do
2    $\text{selected} \leftarrow \text{false}$ ;
3   foreach unsettled conflict  $c$  in  $\mathcal{C}_{\text{unsettled}}$  do
4      $\langle \varepsilon, \varepsilon' \rangle \leftarrow$  a pair of unsettled edges for conflict  $c$ ;
5     if  $\varepsilon$  and  $\varepsilon'$  both form a cycle in SDG then
6       return infeasible (together with conflict  $c$ );
7     else if one of  $\varepsilon$  and  $\varepsilon'$  forms a cycle in SDG then
8       Add the edge not forming a cycle to SDG;
9       Remove conflict  $c$  from  $\mathcal{C}_{\text{unsettled}}$ ;
10       $\text{selected} \leftarrow \text{true}$ ;
11   if  $\text{selected} = \text{false}$  then
12     Remove a conflict  $c$  from  $\mathcal{C}_{\text{unsettled}}$ ;
13      $\langle \varepsilon, \varepsilon' \rangle \leftarrow$  a pair of unsettled edges for conflict  $c$ ;
14     Add  $\varepsilon$  to SDG;
15     if FeasibilityTest( $\mathcal{C}_{\text{unsettled}}$ ) = feasible then
16       return feasible (with the acyclic complete SDG found);
17     Remove  $\varepsilon$  from SDG and add  $\varepsilon'$  to SDG;
18     return FeasibilityTest( $\mathcal{C}_{\text{unsettled}}$ );
19 return feasible (with the acyclic complete SDG found);

```

5.2.2 Remove Redundant Conflicts

We observe that some conflicts between two agents may be induced from other conflicts between the same agents. One main reason is that owing to uncertainty, when an agent is executing a *move* action on an edge, it can be located anywhere along the edge, including the two nodes incident to the edge. As a result, if a node state conflicts with another node/edge state, the edge states preceding and succeeding the former node state must also conflict with the latter node/edge state. In this case, the conflicts can be *redundant*. A redundant conflict means that if we remove it from the unsettled conflicts $\mathcal{C}_{\text{unsettled}}$, no constraint is lost in the SDG and the feasibility test results are always the same before and after removing the conflict. Thus, we can remove redundant conflicts in SDG construction to improve the efficiency of the feasibility test. We use a similar technique of dominance pruning in A* search algorithms to detect and prune redundant conflicts.

Recall that a *conflict* between two agents a_i and $a_{i'}$ can be represented by two *dependency edges* generated from the conflict, regardless of the type of the conflict (unsettled or presettled). For example, if the generated dependency edges

are $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$ and $\langle v_{i',j_{i'}}, e_{i,k_i} \rangle$, we have $c = \{\langle v_{i,j_i}, e_{i',k_{i'}} \rangle, \langle v_{i',j_{i'}}, e_{i,k_i} \rangle\}$, where $v_{i,j_i}, v_{i',j_{i'}}$ are node states and $e_{i,k_i}, e_{i',k_{i'}}$ are edge states of the agents a_i and $a_{i'}$. We then have Definition 5.1 on the dominance relationship of the conflicts between the same two agents.

Definition 5.1. For any two conflicts c and $\hat{c}$ between the same two agents a_i and $a_{i'}$, where $c = \{\langle v_{i,j_i}, e_{i',k_{i'}} \rangle, \langle v_{i',j_{i'}}, e_{i,k_i} \rangle\}$ and $\hat{c} = \{\langle v_{i,\hat{j}_i}, e_{i',\hat{k}_{i'}} \rangle, \langle v_{i',\hat{j}_{i'}}, e_{i,\hat{k}_i} \rangle\}$, we define a dominance relationship $\hat{c} \succsim c$ ($\hat{c}$ dominates c) if and only if (1) $j_i \leq \hat{j}_i$; (2) $k_{i'} \geq \hat{k}_{i'}$; (3) $j_{i'} \leq \hat{j}_{i'}$; and (4) $k_i \geq \hat{k}_i$.

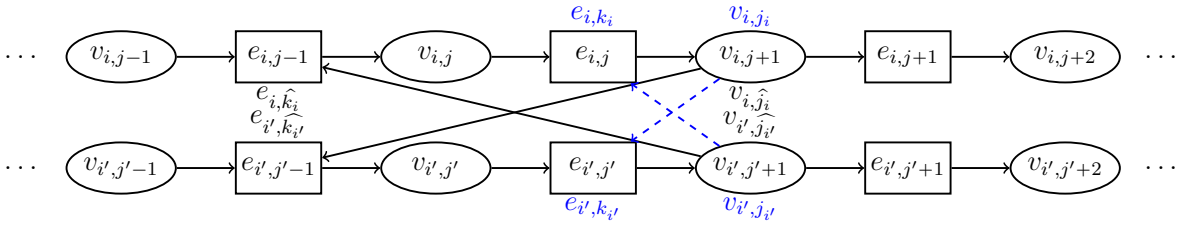


FIGURE 5.6: Dominance relationship of two conflicts between the same two agents.

For example, in Figure 5.6, suppose there is an edge-edge conflict c between $e_{i,j}$ and $e_{i',j'}$, where $c = \{\langle v_{i,j+1}, e_{i',j'} \rangle, \langle v_{i',j'+1}, e_{i,j} \rangle\}$ (represented by the dashed lines), and a node-node conflict $\hat{c}$ between $v_{i,j}$ and $v_{i',j'}$, where $\hat{c} = \{\langle v_{i,j+1}, e_{i',j'-1} \rangle, \langle v_{i',j'+1}, e_{i,j-1} \rangle\}$ (represented by the solid lines). We can conclude that $\hat{c} \succsim c$ because the four conditions hold.

Lemma 5.1. $\succsim$ is a partial ordering for all conflicts between the same two agents a_i and $a_{i'}$.

Proof. For any three conflicts $c, \hat{c}$ and $\tilde{c}$ between agents a_i and $a_{i'}$, we need to prove the following properties: (1) $c \succsim c$; (2) if $c \succsim \hat{c}$ and $\hat{c} \succsim c$, then $c = \hat{c}$; (3) if $\tilde{c} \succsim \hat{c}$ and $\hat{c} \succsim c$, then $\tilde{c} \succsim c$. The first two properties are set naturally. For the last property, let $\tilde{c} = \{\langle v_{i,\tilde{j}_i}, e_{i',\tilde{k}_{i'}} \rangle, \langle v_{i',\tilde{j}_{i'}}, e_{i,\tilde{k}_i} \rangle\}$. It follows from $\tilde{c} \succsim \hat{c}$ and $\hat{c} \succsim c$ that: (1) $j_i \leq \hat{j}_i \leq \tilde{j}_i$; (2) $k_{i'} \geq \hat{k}_{i'} \geq \tilde{k}_{i'}$; (3) $j_{i'} \leq \hat{j}_{i'} \leq \tilde{j}_{i'}$; and (4) $k_i \geq \hat{k}_i \geq \tilde{k}_i$, which means that $\tilde{c} \succsim c$. $\square$

Definition 5.2. The redundant conflicts are defined as $\mathcal{C}_{\text{redundant}} = \{c \in \mathcal{C}_{\text{unsettled}} \mid \exists \hat{c} \in (\mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}}) \setminus \{c\}, \hat{c} \succsim c\}$.

Lemma 5.2. For each redundant conflict $c \in \mathcal{C}_{\text{redundant}}$, the results of feasibility tests with $\mathcal{C}_{\text{unsettled}}$ and $\mathcal{C}_{\text{unsettled}} \setminus \{c\}$ are the same.

Proof. Let $c = \{\langle v_{i,j_i}, e_{i',k_{i'}} \rangle, \langle v_{i',j_{i'}}, e_{i,k_i} \rangle\}$ be a redundant conflict between two agents a_i and $a_{i'}$.

First, if the path plan $\mathcal{P}$ is feasible before removing the conflict c from $\mathcal{C}_{\text{unsettled}}$, an acyclic complete SDG $\mathcal{G}_{\text{SDG}} = (\mathcal{V}_{\text{SDG}}, \mathcal{E}_{\text{SDG}})$ can be found, in which one of the dependency edges $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$ and $\langle v_{i',j_{i'}}, e_{i,k_i} \rangle$ is selected. After removing the conflict c , we can remove the selected edge ε and the new complete SDG $\mathcal{G}'_{\text{SDG}} = (\mathcal{V}_{\text{SDG}}, \mathcal{E}_{\text{SDG}} \setminus \{\varepsilon\})$ must still be acyclic, which means that the result of the new feasibility test is also positive.

Second, if the path plan $\mathcal{P}$ is infeasible before removing the conflict c , we prove that it is also infeasible after removing the conflict by contradiction. Assume on the contrary that the result of the feasibility test without the conflict c is positive, i.e., an acyclic complete SDG $\mathcal{G}_{\text{SDG}} = (\mathcal{V}_{\text{SDG}}, \mathcal{E}_{\text{SDG}})$ can be formed. This means that adding either dependency edge $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$ and $\langle v_{i',j_{i'}}, e_{i,k_i} \rangle$ forms a cycle in $\mathcal{G}_{\text{SDG}}$. Let $\hat{c} = \{\langle v_{i,\hat{j}_i}, e_{i',\hat{k}_{i'}} \rangle, \langle v_{i',\hat{j}_{i'}}, e_{i,\hat{k}_i} \rangle\}$ be the conflict satisfying $\hat{c} \succeq c$. Without loss of generality, suppose the dependency edge $\langle v_{i,\hat{j}_i}, e_{i',\hat{k}_{i'}} \rangle$ for the conflict $\hat{c}$ is selected in $\mathcal{G}_{\text{SDG}}$. Since $j_i \leq \hat{j}_i$ and $k_{i'} \geq \hat{k}_{i'}$, a path $v_{i,j_i} \rightarrow \dots \rightarrow v_{i,\hat{j}_i} \rightarrow e_{i',\hat{k}_{i'}} \rightarrow \dots \rightarrow e_{i',k_{i'}}$ exists in $\mathcal{G}_{\text{SDG}}$. Moreover, since adding the dependency edge $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$ forms a cycle in $\mathcal{G}_{\text{SDG}}$, there must exist a path from $e_{i',k_{i'}}$ to v_{i,j_i} in $\mathcal{G}_{\text{SDG}}$. Then, the aforesaid two paths form a cycle in $\mathcal{G}_{\text{SDG}}$ even without the edge $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$. This leads to a contradiction with the assumption that $\mathcal{G}_{\text{SDG}}$ is acyclic and the result of the feasibility test without the conflict c is positive. $\square$

Recall the example in Figure 5.6. If the dependency edge $\langle v_{i,j+1}, e_{i',j'-1} \rangle$ is selected from the conflict $\hat{c}$, then the dependency edge $\langle v_{i,j+1}, e_{i',j'} \rangle$ must also be selected from the conflict c . Otherwise, the selection would require both dependency edges $\langle v_{i',j'+1}, e_{i,j-1} \rangle$ and $\langle v_{i',j'+1}, e_{i,j} \rangle$ to be selected to prevent a cycle in the SDG. In this specific example with only two conflicts, removing either c or $\hat{c}$ preserves the result of the feasibility test. However, in scenarios with more conflicts between two agents, following the dominance relationship $\succeq$ that forms a partial order among all conflicts between the same two agents is essential to ensure the correctness of redundant conflict removal.

Lemma 5.3. *If $\tilde{c} \in \mathcal{C}_{\text{redundant}}$, then $\mathcal{C}_{\text{redundant}} \setminus \{\tilde{c}\} = \{c \in \mathcal{C}_{\text{unsettled}} \setminus \{\tilde{c}\} \mid \exists \hat{c} \in \mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}} \setminus \{c, \tilde{c}\}, \hat{c} \succeq c\}$.*

Proof. From Definition 5.2, we know that for each $c \in \mathcal{C}_{\text{redundant}} \setminus \{\tilde{c}\}$, there exists $\hat{c} \in \mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}} \setminus \{c\}$ such that $\hat{c} \succsim c$. If $\hat{c} \neq \tilde{c}$, the lemma is set naturally. Otherwise, since $\tilde{c} \in \mathcal{C}_{\text{redundant}}$ and $\tilde{c} = \hat{c}$, there exists $\bar{c} \in \mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}} \setminus \{\tilde{c}\}$ such that $\bar{c} \succsim \hat{c}$. It follows that $\bar{c} \succsim \hat{c} = \tilde{c} \succsim c$. Since $\tilde{c} \neq \bar{c}$, we have $\hat{c} \neq \bar{c}$. Therefore, we can conclude that $\bar{c} \in \mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}} \setminus \{c, \hat{c}\}$ and $\bar{c} \succsim c$. $\square$

Theorem 5.4. *The results of feasibility tests with $\mathcal{C}_{\text{unsettled}}$ and $\mathcal{C}_{\text{unsettled}} \setminus \mathcal{C}_{\text{redundant}}$ are the same.*

Proof. According to Lemma 5.3, after removing a conflict $c \in \mathcal{C}_{\text{redundant}}$ from $\mathcal{C}_{\text{unsettled}}$, the other conflicts in $\mathcal{C}_{\text{redundant}}$ are still redundant. Thus, by iteratively applying Lemma 5.2, the results of feasibility tests are always the same. $\square$

Therefore, we can remove all redundant conflicts $\mathcal{C}_{\text{redundant}}$ during the construction of the SDG. To identify these redundant conflicts, we develop two approaches: one is based on straightforward pairwise iterations over conflicts using the dominance relationship, and the other leverages the physical meanings of the conflicts to detect redundant conflicts more efficiently. These approaches help reduce the size of the SDG and improve the overall efficiency of the feasibility test.

The first intuitive approach to find $\mathcal{C}_{\text{redundant}}$ between any two agents is to iterate over all pairs of conflicts and mark certain conflicts as redundant based on the dominance relationship $\succsim$. This approach allows us to identify all the redundant conflicts, as $\succsim$ forms a partial ordering over all conflicts between the same two agents (Lemma 5.1).

Formally, for each pair of agents a_i and $a_{i'}$ in the set $\{\langle a_i, a_{i'} \rangle \mid a_i, a_{i'} \in \mathcal{A}, a_i \neq a_{i'}\}$, let $\mathcal{C}_{i,i'}$ denote the set of all (unsettled and presettled) conflicts between a_i and $a_{i'}$. In Algorithm 22, we apply a two-level iteration over all pairs of conflicts c and c' in $\mathcal{C}_{i,i'}$, and we add conflict c to the set $\mathcal{C}_{\text{redundant}}$ if $c' \succsim c$. This procedure directly follows the definition of $\mathcal{C}_{\text{redundant}}$ as given in Definition 5.2.

The time complexity of the above approach on a pair of agents a_i and $a_{i'}$ is $\mathcal{O}(|\mathcal{C}_{i,i'}|^2)$, where $|\mathcal{C}_{i,i'}|$ is the number of conflicts between a_i and $a_{i'}$. Since there are $M(M-1)/2$ unique pairs of agents in a problem with M agents, the overall complexity can become a bottleneck as the number of agents increases, making the approach less efficient in large-scale scenarios.

Algorithm 22: Detect Redundant Conflicts between Agents a_i and $a_{i'}$

```

1  $\mathcal{C}_{\text{redundant}} \leftarrow \emptyset;$ 
2 foreach  $c \in \mathcal{C}_{i,i'}$  do
3   foreach  $c' \in \mathcal{C}_{i,i'} \setminus \{c\}$  do
4     if  $c' \succsim c$  then
5        $\mathcal{C}_{\text{redundant}} \leftarrow \mathcal{C}_{\text{redundant}} \cup \{c\};$ 

```

The second approach is based on the observation that some conflicts can be induced from others through their physical meanings.

First, some edge-edge conflicts can be induced from a node-node conflict. Suppose that there is a node-node conflict between two agents a_i and $a_{i'}$ at the nodes $v_{i,j}$ and $v_{i',j'}$ respectively. Figure 5.7 illustrates the unsettled edges for the node-node conflict by solid lines. Then, the edge states incident to $v_{i,j}$ and $v_{i',j'}$ would also conflict with each other by definition. Specifically, we can induce that the following edge-edge conflicts must exist (if the corresponding edges are valid) and they are redundant by Definition 5.2: (1) conflict between $e_{i,j-1}$ and $e_{i',j'-1}$ (dotted lines in upper part of Figure 5.7); (2) conflict between $e_{i,j}$ and $e_{i',j'}$ (dashed lines in upper part of Figure 5.7); (3) conflict between $e_{i,j-1}$ and $e_{i',j'}$ (dotted lines in lower part of Figure 5.7); (4) conflict between $e_{i,j}$ and $e_{i',j'-1}$ (dashed lines in lower part of Figure 5.7).

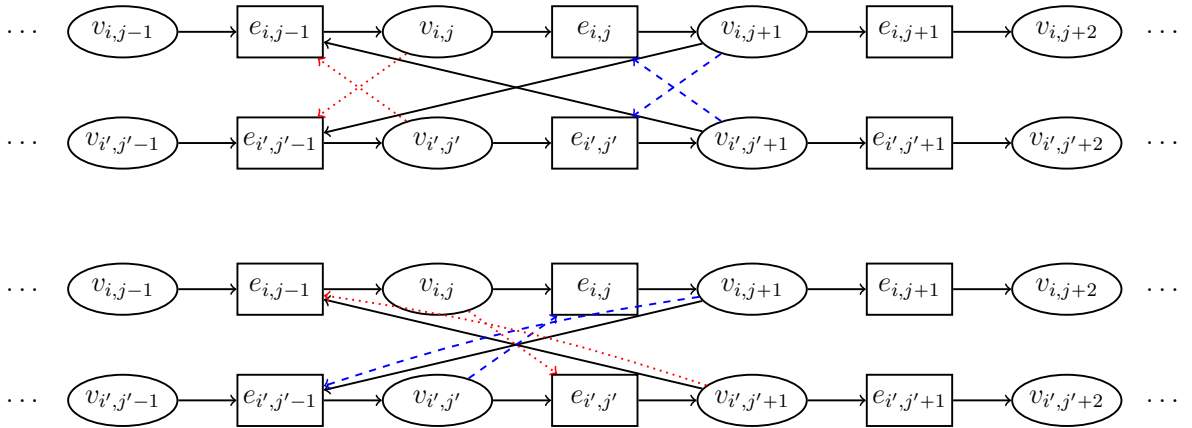


FIGURE 5.7: Edge-edge conflicts induced from a node-node conflict.

Second, some node-edge conflicts can be induced from a node-node conflict. With the same node-node conflict in Figure 5.7, the edge states incident to $v_{i,j}$ (resp.

$v_{i',j'}$) would also conflict with the node state $v_{i',j'}$ (resp. $v_{i,j}$) by definition. Specifically, we can induce that the following node-edge conflicts must exist (if the corresponding edges are valid) and they are redundant by Definition 5.2: (1) conflict between $e_{i',j'-1}$ and $v_{i,j}$ (dotted lines in Figure 5.8); (2) conflict between $e_{i,j}$ and $v_{i',j'}$ (dashed lines in Figure 5.8); (3) conflict between $e_{i,j-1}$ and $v_{i',j'}$; (4) conflict between $e_{i',j'}$ and $v_{i,j}$ (the last two conflicts are symmetric to those shown in Figure 5.8).

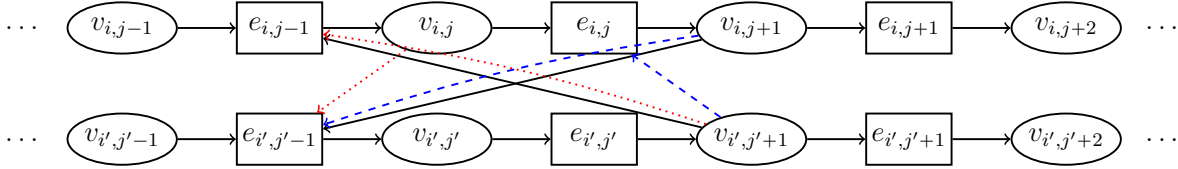


FIGURE 5.8: Node-edge conflicts induced from a node-node conflict.

Third, some edge-edge conflicts can be induced from a node-edge conflict. Suppose that there is a node-edge conflict between two agents a_i and $a_{i'}$ at the edge $e_{i,j}$ and node $v_{i',j'}$ respectively. Figure 5.9 illustrates the unsettled edges for the node-edge conflict by solid lines. Then, the edge states incident to $v_{i',j'}$ would also conflict with the edge state $e_{i,j}$ by definition. Specifically, we can induce that the following edge-edge conflicts must exist (if the corresponding edges are valid) and they are redundant by Definition 5.2: (1) conflict between $e_{i,j}$ and $e_{i',j'-1}$ (dotted lines in Figure 5.9); (2) conflict between $e_{i,j}$ and $e_{i',j'}$ (dashed lines in Figure 5.9).

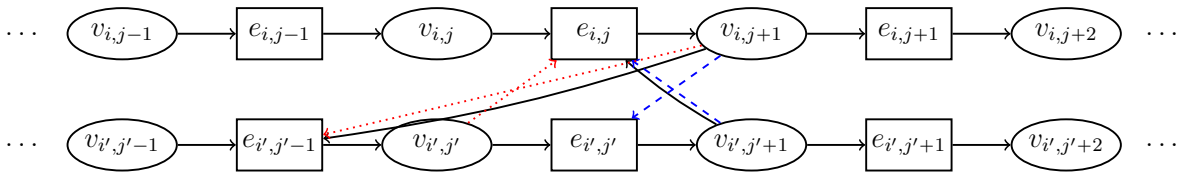


FIGURE 5.9: Edge-edge conflicts induced from a node-edge conflict.

Given an arbitrary node-node conflict or node-edge conflict, all three types of redundant conflicts described above can be detected in amortized constant time by indexing and storing the conflicts with their corresponding dependency edges in a hash table. As a result, the overall time complexity of the method is $\mathcal{O}(|\mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}}|)$, since each conflict $c \in \mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}}$ requires only constant time to check for redundant conflicts. This significantly improves the efficiency of redundant conflict detection compared to the pairwise iteration-based approach.

Note that, unlike the pairwise iteration-based approach, this is a heuristic method and does not guarantee exhaustive detection of all the redundant conflicts. However, it offers several notable advantages. First, the method is grounded in the physical interpretation of conflicts, making it easier to understand and implement. Second, it is capable of identifying a near-maximum set of redundant conflicts, yielding a comparable empirical speedup in the feasibility test. Third, due to its linear time complexity, the method scales effectively to large instances that involve a huge number of agents and conflicts.

5.2.3 Construct Conflict Groups

We also observe that certain conflicts between two agents may naturally be grouped together. For example, if two conflicts share the same dependency edge, then to prevent a cycle in the SDG, either that shared dependency edge or the two other edges involved in the individual conflicts must be selected and added to the SDG. This grouping is enabled by the *mutually exclusive* property of dependency edges, which ensures that only one edge in a conflict can be selected without forming a cycle in the SDG.

In this section, the term “SDG” specifically refers to the initial SDG that includes only the first set of dependency edges, denoted as $\mathcal{E}_{\text{SDG}_1}$, since conflict groups are constructed before path execution (or before any feasibility test). We exclude the second set of dependency edges ($\mathcal{E}_{\text{SDG}_2}$), which are formed by presettled conflicts where only one dependency edge is valid for each conflict. This exclusion allows us to group unsettled and presettled conflicts together in the construction process. Specifically, if an unsettled conflict belongs to the same conflict group as a presettled conflict, it can be transformed into a presettled conflict, since only one of its dependency edges does not form a cycle in the SDG and can be selected. This transformation further eliminates the unsettled conflicts in the feasibility test, which helps improve its efficiency.

Definition 5.3. Two dependency edges ε and ε' are *mutually exclusive* if and only if adding both dependency edges into the SDG always forms a cycle in the SDG.

Lemma 5.5. *Two dependency edges in a conflict are mutually exclusive.*

Proof. This can be easily proved by contradiction. Suppose $c = \{\langle v_{i,j_i}, e_{i',k_{i'}} \rangle, \langle v_{i',j_{i'}}, e_{i,k_i} \rangle\}$ is a conflict between two agents a_i and $a_{i'}$, if both dependency edges are selected, there is a cycle $v_{i,j_i} \rightarrow e_{i',k_{i'}} \rightarrow \dots \rightarrow v_{i',j_{i'}} \rightarrow e_{i,k_i} \rightarrow \dots \rightarrow v_{i,j_i}$ since $k_{i'} < j_{i'}$ and $k_i < j_i$ (according to the definition of conflicts). The cycle can also be seen in Figures 5.3, 5.4 and 5.5 if both dependency edges (dashed lines) are selected. $\square$

The mutually exclusive property between two dependency edges can be extended to two sets of dependency edges. This extension allows reasoning about conflicts at the group level rather than the individual level.

Definition 5.4. Two sets of dependency edges $\mathcal{E}$ and $\mathcal{E}'$ are *mutually exclusive* if and only if the following three conditions hold: (1) only one of $\mathcal{E}$ and $\mathcal{E}'$ contains **invalid** dependency edges; (2) after adding all **valid** dependency edges in $\mathcal{E}$ to the SDG, adding any **valid** dependency edge $\varepsilon' \in \mathcal{E}'$ results in a cycle in the SDG, and (3) after adding all **valid** dependency edges in $\mathcal{E}'$ to the SDG, adding any **valid** dependency edge $\varepsilon \in \mathcal{E}$ results in a cycle in the SDG.

Definition 5.5. Suppose there is a set of conflicts $\mathcal{C}_G = \{\langle \varepsilon_{i_1}, \varepsilon_{i'_1} \rangle, \dots, \langle \varepsilon_{i_N}, \varepsilon_{i'_N} \rangle\}$, where N is the number of conflicts in $\mathcal{C}_G$. $\mathcal{C}_G$ is a *conflict group* if and only if there exists a selection list $\{\langle \theta_j, \theta'_j \rangle = \langle i_j, i'_j \rangle \text{ or } \langle i'_j, i_j \rangle \mid j \in [1, N], j \in \mathbb{Z}\}$ such that the two sets of dependency edges $\{\varepsilon_{\theta_1}, \dots, \varepsilon_{\theta_N}\}$ and $\{\varepsilon_{\theta'_1}, \dots, \varepsilon_{\theta'_N}\}$ are mutually exclusive.

We develop two approaches to construct and merge conflict groups for each pair of agents: one is based on straightforward pairwise iterations, and the other is a heuristic approach based on some observation of the SDG to merge conflict groups more efficiently. The correctness of the merging of conflict groups are guaranteed by the following lemmas:

Lemma 5.6. $\mathcal{C}_G$ is a conflict group if there is only one conflict in $\mathcal{C}_G$.

Lemma 5.7. Two conflict groups $\mathcal{C}_G$ (with corresponding mutually exclusive sets of dependency edges $\mathcal{E}$ and $\mathcal{E}'$) and $\widehat{\mathcal{C}}_G$ (with corresponding mutually exclusive sets of dependency edges $\widehat{\mathcal{E}}$ and $\widehat{\mathcal{E}}'$) can be merged into one conflict group if and only if one of the following two conditions holds: (1) $\mathcal{E} \cup \widehat{\mathcal{E}}$ and $\mathcal{E}' \cup \widehat{\mathcal{E}}'$ are mutually exclusive; or (2) $\mathcal{E} \cup \widehat{\mathcal{E}}'$ and $\mathcal{E}' \cup \widehat{\mathcal{E}}$ are mutually exclusive.

When a conflict is resolved in the feasibility test and the dependency edge is selected and added into the SDG, all other conflicts in the same conflict group can be resolved directly, with all edges in its corresponding set of dependency edges selected and added into the SDG. This optimization reduces the number of conflicts to be resolved to the number of conflict groups to be resolved, which helps improve the efficiency of the feasibility test.

Similar to the definition in removing redundant conflicts, for each pair of agents a_i and $a_{i'}$ in the set $\{\langle a_i, a_{i'} \rangle \mid a_i, a_{i'} \in \mathcal{A}, a_i \neq a_{i'}\}$, let $\mathcal{C}_{i,i'}$ denote the set of all (unsettled and presettled) conflicts between a_i and $a_{i'}$. The first approach can merge the conflicts $\mathcal{C}_{i,i'}$ into a minimum number of conflict groups based on the above lemmas.

Algorithm 23 illustrates the first approach. Initially, each individual conflict is treated as a separate conflict group, as justified by Lemma 5.6, and the collection of all conflict groups is denoted by Γ (line 1). The algorithm proceeds by repeatedly attempting to merge conflict groups in Γ until no further merges (i.e., updates to Γ) are possible. In each merge attempt, the algorithm iterates through the conflict groups in Γ , selecting a random conflict group $\mathcal{C}_G \in \Gamma$ that has not yet been processed in the current attempt, and removing it from Γ (lines 5-7). It then compares $\mathcal{C}_G$ with all remaining conflict groups in Γ to check if any of them can be merged with $\mathcal{C}_G$ based on Lemma 5.7 (lines 8-9). If a merge is possible, the matched conflict group is removed from Γ to prevent redundant checks during the current attempt, and a flag is set to indicate that an update has occurred (lines 11-12). The merged result is stored in a temporary set Γ' , which is reassigned to Γ at the end of the current merge attempt (line 14). This iterative merging process continues until no further updates to Γ are detected, ensuring that all mergeable conflict groups are eventually unified.

However, in the worst case, the approach needs to perform $\mathcal{O}(|\mathcal{C}_{i,i'}|^3)$ cycle checks (line 9), where $|\mathcal{C}_{i,i'}|$ is the number of conflicts between two agents a_i and $a_{i'}$. It is even worse than the time complexity of Algorithm 22, making the approach not applicable to large-scale scenarios.

Note that we do not place any restriction on the number of agents involved in the definitions of mutually exclusive sets of dependency edges (Definition 5.4) and

Algorithm 23: Construct Conflict Groups (Iterative)

```

1  $\Gamma \leftarrow \{\{c\} \mid c \in \mathcal{C}_{i,i'}\};$ 
2 do
3    $\Gamma' \leftarrow \emptyset;$ 
4    $updated \leftarrow false;$ 
5   while  $\Gamma \neq \emptyset$  do
6      $\mathcal{C}_G \leftarrow$  randomly select a conflict group in  $\Gamma;$ 
7      $\Gamma \leftarrow \Gamma \setminus \{\mathcal{C}_G\};$ 
8     foreach  $\widehat{\mathcal{C}}_G \in \Gamma$  do
9       if  $\mathcal{C}_G$  and  $\widehat{\mathcal{C}}_G$  can be merged into a conflict group then
10          $\mathcal{C}_G \leftarrow \mathcal{C}_G \cup \widehat{\mathcal{C}}_G;$ 
11          $\Gamma \leftarrow \Gamma \setminus \{\widehat{\mathcal{C}}_G\};$ 
12          $updated \leftarrow true;$ 
13      $\Gamma' \leftarrow \Gamma' \cup \{\mathcal{C}_G\};$ 
14    $\Gamma \leftarrow \Gamma';$ 
15 while  $updated = true;$ 

```

conflict groups (Definition 5.5), meaning these concepts can naturally extend to an arbitrary number of agents. Consequently, Algorithm 23 can also be applied to construct conflict groups among more than two agents by expanding $\mathcal{C}_{i,i'}$ to a larger set of conflicts involving all agents under consideration.

Despite the possibility, in this work, we intentionally avoid merging conflict groups across different agent pairs using Algorithm 23 for several practical reasons: (1) the number of cycle checks required increases dramatically with the number of conflicts, significantly impacting efficiency; (2) empirically, conflict groups involving multiple agent pairs are rare compared to those between the same two agents; and (3) we already construct agent groups as a separate optimization in the discrete-time version of the algorithm (Chapter 3), and since the optimization has been adapted to the continuous-time version, the effectiveness of additionally constructing conflict groups across different agent pairs is reduced.

The second approach is based on the observation that some conflicts that are close to each other can be merged into a conflict group. While some of these conflicts can also be explained or induced from their physical meanings, our analysis in this approach is primarily conducted on the structure of the SDG itself. In the analysis, for simplicity, we generally omit the states that precede the relevant states of a given

conflict type due to symmetry. By focusing on the demonstrated states, we can effectively reduce redundancy in the analysis while preserving correctness.

First, some node-node conflicts can be merged with a node-node conflict. Suppose that there is a node-node conflict c between two agents a_i and $a_{i'}$ at the nodes $v_{i,j}$ and $v_{i',j'}$ respectively. Figure 5.10 illustrates the unsettled edges for the node-node conflict by solid lines. There may also exist node-node conflicts among $v_{i,j}$, $v_{i',j'}$ and some nearby node states ($v_{i,j+1}$ and $v_{i',j'+1}$). Specifically, if the following node-node conflicts exist, they can be grouped with the conflict c by Lemma 5.7: (1) conflict between $v_{i,j+1}$ and $v_{i',j'+1}$ (dashed lines in Figure 5.10); (2) conflict between $v_{i,j}$ and $v_{i',j'+1}$ (dotted lines in Figure 5.10); (3) conflict between $v_{i',j'}$ and $v_{i,j+1}$ (symmetric to (2), omitted in Figure 5.10).

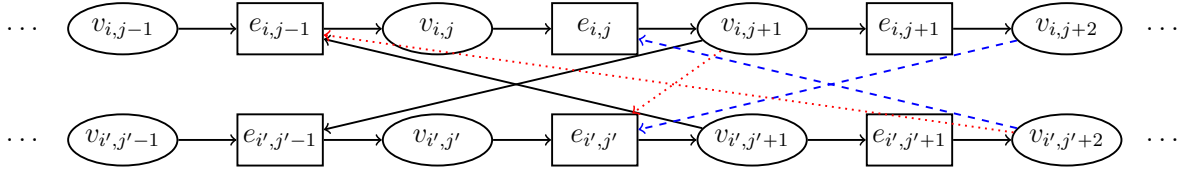


FIGURE 5.10: Node-node conflicts that can be merged with a node-node conflict.

Second, some node-edge conflicts can be merged with a node-node conflict. Suppose that there is a node-node conflict c between two agents a_i and $a_{i'}$ at the nodes $v_{i,j}$ and $v_{i',j'}$ respectively. Figure 5.11 illustrates the unsettled edges for the node-node conflict by solid lines. There may also exist node-edge conflicts among $v_{i,j}$, $v_{i',j'}$ and some nearby node states ($v_{i,j+1}$ and $v_{i',j'+1}$) and edge states ($e_{i,j}$ and $e_{i',j'}$). Specifically, if the following node-edge conflicts exist, they can be grouped with the conflict c by Lemma 5.7: (1) conflict between $e_{i,j}$ and $v_{i',j'+1}$ (dotted lines in Figure 5.11); (2) conflict between $e_{i',j'}$ and $v_{i,j+1}$ (symmetric to (1), omitted in Figure 5.11); (3) conflict between $e_{i,j}$ and $v_{i',j'}$ (dashed lines in Figure 5.11); (4) conflict between $e_{i',j'}$ and $v_{i,j}$ (symmetric to (3), omitted in Figure 5.11). Note that the conflicts (3) and (4) are also redundant conflicts induced from a node-node conflict (see Figure 5.8), which can be ignored when merging conflicts.

Third, some node-node conflicts can be merged with a node-edge conflict. Suppose that there is a node-node conflict c between two agents a_i and $a_{i'}$ at the nodes $e_{i,j}$ and $v_{i',j'}$ respectively. Figure 5.12 illustrates the unsettled edges for the node-edge conflict by solid lines. There may also exist node-node conflicts among $v_{i,j}$, $v_{i',j'}$

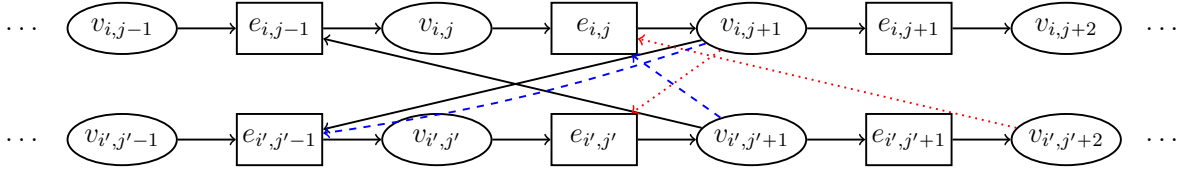


FIGURE 5.11: Node-edge conflicts that can be merged with a node-node conflict.

and some nearby node states ($v_{i,j+1}$ and $v_{i',j'+1}$). Specifically, if the following node-edge conflicts exist, they can be grouped with the conflict c by Lemma 5.7: (1) conflict between $v_{i,j+1}$ and $v_{i',j'+1}$ (dashed lines in Figure 5.12); (2) conflict between $v_{i,j}$ and $v_{i',j'+1}$ (dotted lines in Figure 5.12); (3) conflict between $v_{i',j'}$ and $v_{i,j+1}$ (dash-dotted lines in Figure 5.12).

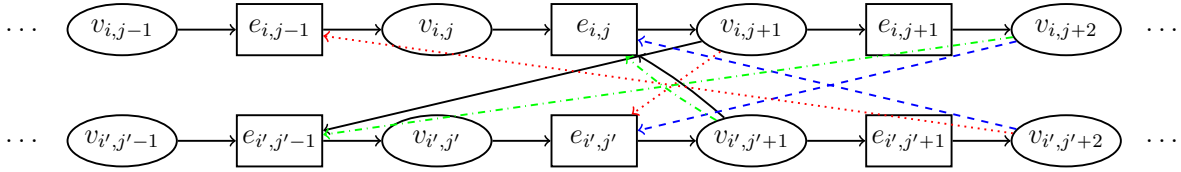


FIGURE 5.12: Node-node conflicts that can be merged with a node-edge conflict.

Last, some node-edge conflicts can be merged with a node-edge conflict. Suppose that there is a node-node conflict c between two agents a_i and $a_{i'}$ at the nodes $e_{i,j}$ and $v_{i',j'}$ respectively. Figure 5.13 illustrates the unsettled edges for the node-edge conflict by solid lines. There may also exist node-edge conflicts among some nearby node states ($v_{i,j+1}$ and $v_{i',j'+1}$) and edge states ($e_{i,j}$ and $e_{i',j'}$). Specifically, if the following node-edge conflicts exist, they can be grouped with the conflict c by Lemma 5.7: (1) conflict between $e_{i,j}$ and $v_{i',j'+1}$ (dotted lines in Figure 5.13); (2) conflict between $e_{i',j'}$ and $v_{i,j+1}$ (dashed lines in Figure 5.13).

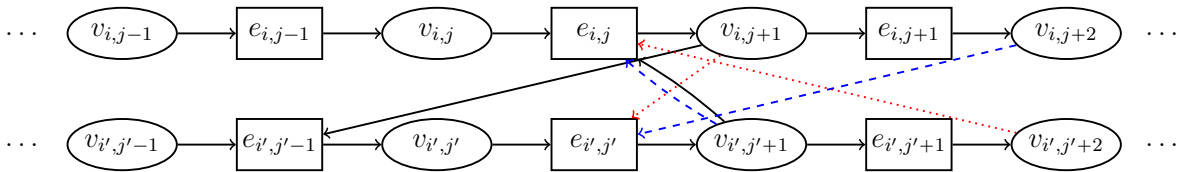


FIGURE 5.13: Node-edge conflicts that can be merged with a node-edge conflict.

With three types of conflicts, there are nine possible combinations when merging conflicts based on their types. However, only four of these combinations are demonstrated above. This simplification is due to two main reasons: (1) two edge-edge conflicts cannot be merged in our analysis framework; and (2) any edge-edge

conflict that can be merged with a node-node or node-edge conflict is considered a redundant conflict (see Figure 5.7 and Figure 5.9). If redundant conflicts are already removed prior to the feasibility test, such edge-edge conflicts can be ignored during the merging process. Otherwise, it is still possible to merge these edge-edge conflicts, but doing so at this stage is less efficient than eliminating them earlier during redundant conflict detection. As such, we omit them from the analysis above.

In Algorithm 24, we begin by initializing all conflicts $\mathcal{C}_{i,i'}$ between each pair of agents a_i and $a_{i'}$ as individual conflict groups (line 1). For each conflict $c \in \mathcal{C}_{i,i'}$, we identify the set of conflicts that can be merged with c using the method described earlier (lines 2–3). As in the second approach for removing redundant conflicts, these mergeable conflicts can be retrieved in amortized constant time using the same hash table indexed by conflicts and their corresponding dependency edges. For each mergeable conflict c' , we check whether its corresponding conflict group $\mathcal{C}'_G$ has already been merged with the conflict group $\mathcal{C}_G$ containing c (lines 5–7). If not, we proceed to merge $\mathcal{C}'_G$ into $\mathcal{C}_G$, and the merge process always follows the conditions specified in Lemma 5.7 (lines 8–9). After executing the algorithm, Γ contains the merged conflict groups which can be further used in the feasibility test.

Algorithm 24: Construct Conflict Groups (Heuristic)

```

1  $\Gamma \leftarrow \{\{c\} \mid c \in \mathcal{C}_{i,i'}\};$ 
2 foreach  $c \in \mathcal{C}_{i,i'}$  do
3    $\mathcal{C}' \leftarrow$  find the conflicts that can be merged with  $c$ ;
4   foreach  $c' \in \mathcal{C}'$  do
5      $\mathcal{C}_G \leftarrow$  the conflict group in  $\Gamma$  in so that  $c \in \mathcal{C}_G$ ;
6      $\mathcal{C}'_G \leftarrow$  the conflict group in  $\Gamma$  in so that  $c' \in \mathcal{C}'_G$ ;
7     if  $\mathcal{C}_G \neq \mathcal{C}'_G$  then
8        $\Gamma \leftarrow \Gamma \setminus \{\mathcal{C}_G, \mathcal{C}'_G\};$ 
9        $\Gamma \leftarrow \Gamma \cup \{\mathcal{C}_G \cup \mathcal{C}'_G\};$ 

```

The time complexity of Algorithm 24 is $\mathcal{O}(|\mathcal{C}_{i,i'}| \log |\mathcal{C}_{i,i'}|)$, where $|\mathcal{C}_{i,i'}|$ is the number of conflicts between two agents a_1 and a_2 . Although the number of mergeable conflicts $\mathcal{C}'$ for each conflict is constant in each iteration due to efficient hashing, the set union operation performed during merging (line 9) requires at least $\log |\mathcal{C}_{i,i'}|$ time in the worst case. This can be achieved by using the union-find data structure

to maintain the conflict groups in Γ . The high efficiency of Algorithm 24 makes it suitable for generating conflict groups even in large-scale MAPF instances with many agents and conflicts.

5.3 Path Execution Solution

5.3.1 Offline Path Execution

Assuming that the initial path plan is feasible (which naturally holds for the output of a continuous-time MAPF solver), we first present our offline solution to ensure conflict-free and deadlock-free path execution. However, unexpected delays during the path execution cannot be effectively addressed in the offline solution because they are completely unknown prior to the path execution.

Before the path execution, we run the feasibility test (Algorithm 21) and obtain an acyclic complete SDG $\mathcal{G}_{\text{SDG}} = (\mathcal{V}_{\text{SDG}}, \mathcal{E}_{\text{SDG}})$ to fix the order for the agents to achieve relevant states in all potential conflicts. For each conflict $c \in \mathcal{C}_{\text{unsettled}}$, we can find a dependency edge $\varepsilon \in \mathcal{E}_{\text{SDG}}$ which is one of the two unsettled edges for c and implies the moving order of the two agents to avoid this conflict. For each conflict $c \in \mathcal{C}_{\text{presettled}}$, only one dependency edge is valid, so the order of the two agents is determined as well. Let $\mathcal{C}_i \subset \mathcal{C}_{\text{presettled}} \cup \mathcal{C}_{\text{unsettled}}$ be the set of all potential conflicts involving the agent a_i , and $\mathcal{E}_i = \{\varepsilon \in \mathcal{E}_{\text{SDG}} \mid \exists c \in \mathcal{C}_i, \varepsilon \in c\}$ be the set of dependency edges in $\mathcal{G}_{\text{SDG}}$ for the conflicts in $\mathcal{C}_i$.

The path execution based on the offline solution can be decentralized (Algorithm 25) because each agent can decide whether to take *move* and *wait* actions by itself, with the help of some communications with the other agents. When an agent a_i achieves a node state $v_{i,j}$, a_i first sends a message including its current state $v_{i,j}$ to all other agents $a_{i'}$ s where there exists an $e_{i',j'}$ such that $\langle v_{i,j}, e_{i',j'} \rangle \in \mathcal{E}_i$, i.e., a_i should achieve the node state $v_{i,j}$ first to avoid a potential conflict with $a_{i'}$ according to the complete SDG (lines 3-4). Then, a_i examines whether it has received all messages containing $v_{i',j'}$ s such that $\langle v_{i',j'}, e_{i,j} \rangle \in \mathcal{E}_i$. i.e., $a_{i'}$ should achieve the node state $v_{i',j'}$ before a_i achieves the edge state $e_{i,j}$ (which is the edge state of a_i immediately following $v_{i,j}$) to avoid a potential conflict according to the complete SDG. If any one of these messages is not received, a_i takes a *wait* action

to remain in the node state $v_{i,j}$ until all messages are received (lines 6-8). After all messages are received, a_i then takes a *move* action and achieves the edge state $e_{i,j}$ (line 9). When the agent a_i is in an edge state, since the *move* action cannot be interrupted by the problem definition, the algorithm does nothing until the agent a_i arrives at the adjacent node on its path and achieves the next node state (line 10).

Algorithm 25: Offline path execution (of agent a_i)

```

1 achieve node state  $v_{i,1}$ ;
2 for  $j \leftarrow 1$  to  $n_i - 1$  do
3   foreach  $\varepsilon \in \mathcal{E}_i$  where  $\varepsilon = \langle v_{i,j}, e_{i',j'} \rangle$  do
4     send message  $\langle i, j \rangle$  to agent  $a_{i'}$ ;
5   foreach  $\varepsilon \in \mathcal{E}_i$  where  $\varepsilon = \langle v_{i',j'}, e_{i,j} \rangle$  do
6     if message  $\langle i', j' \rangle$  is not received then
7       do wait action;
8       yield until message  $\langle i', j' \rangle$  is received;
9   start move action and achieve edge state  $e_{i,j}$ ;
10  yield until achieving node state  $v_{i,j+1}$ ;
```

In the feasibility test, if each unsettled conflict has both its dependency edges eligible for selection (line 11 of Algorithm 21), we randomly select an unsettled edge from a conflict c to proceed (line 14 of Algorithm 21). The order of selecting the unsettled edge here does not affect the correctness of the feasibility test because if the first edge ε selected cannot lead to an acyclic complete SDG, a second attempt with the other edge ε' will be made (line 18 of Algorithm 21). However, such SDG construction may not optimize the sum-of-costs or makespan in offline path execution, because the first acyclic complete SDG found can be far from optimal. To overcome this drawback, we may use as hints the timestamp associated with each edge in the path of every agent a_i , which is provided in the path plan produced by a continuous-time MAPF solver. Specifically, in a conflict $c = \{\langle v_{i,j_i}, e_{i',k_{i'}} \rangle, \langle v_{i',j_{i'}}, e_{i,k_i} \rangle\}$ between two agents a_i and $a_{i'}$, we compare the timestamps t_{i,k_i} and $t_{i',k_{i'}}$ of e_{i,k_i} and $e_{i',k_{i'}}$. Here t_{i,k_i} (resp. $t_{i',k_{i'}}$) refers to the time when a_i (resp. $a_{i'}$) starts executing the *move* action on edge e_{i,k_i} (resp. $e_{i',k_{i'}}$). If $t_{i,k_i} < t_{i',k_{i'}}$, we first select the edge $\langle v_{i',j_{i'}}, e_{i,k_i} \rangle$ because a_i is supposed to achieve the edge state e_{i,k_i} first to avoid the conflict if there is no unexpected delay; otherwise, we first select the edge $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$.

5.3.2 Online Path Execution

We now present our online solution to coordinate the *move* and *wait* actions taken by agents on the fly and ensure conflict-free and deadlock-free path execution. The online solution considers unexpected delays arising in the path execution and optimizes the execution effectiveness dynamically. For ease of exposition, we focus on a centralized solution where the agents sense changes in their states and send updates to a central controller. Developing a decentralized solution is an interesting direction for future work.

We partition all agents into two sets: unblocked agents U (those executing *move* actions, i.e., in the edge states) and blocked agents B (those executing *wait* actions, i.e., in the node states). Recall that a *move* action cannot be canceled before the agent arrives at the adjacent node. Thus, an unblocked agent remains in U until it arrives at the next node on its path. Upon its arrival, we need to decide whether to block the agent from taking a further *move* action. In addition, we need to decide whether to unblock any agents in B and allow them to take *move* actions. We assume that an agent arriving at the next node on its path is moved from U to B by default. Given the current partitions U and B , we focus on designing an algorithm to determine which agents in B to unblock.

If we unblock a set of agents $\Delta U \subseteq B$, the partitions become $U \cup \Delta U$ and $B \setminus \Delta U$. If all agents in ΔU are in their next edge states, some dependency edges may become invalid. For example, suppose two agents $a_i, a_{i'} \in \Delta U$, and there is an unsettled conflict $c = \{\langle v_{i,j_i}, e_{i',k_{i'}} \rangle, \langle v_{i',j_{i'}}, e_{i,k_i} \rangle\}$ between a_i and $a_{i'}$. Note that by the definition of dependency edges, we must have $k_i < j_i$ and $k_{i'} < j_{i'}$. Let the current state of a_i be v_{i,k_i} , and the current state of $a_{i'}$ be $v_{i',k_{i'}}$. If a_i transits to its next edge state is e_{i,k_i} , the dependency edge $\langle v_{i',j_{i'}}, e_{i,k_i} \rangle$ becomes invalid because the agent $a_{i'}$ has not achieved the node state $v_{i',j_{i'}}$ yet. Hence, the dependency edge $\langle v_{i,j_i}, e_{i',k_{i'}} \rangle$ must be selected and c becomes a presettled conflict. If $a_{i'}$ also transits to its next edge state $e_{i',k_{i'}}$ at the same time, both dependency edges become invalid and a conflict must occur because the execution does not comply with either dependency edge. In this case, we say that c becomes a *violated* conflict. Similarly, suppose there is a presettled conflict c between a_i and $a_{i'}$ with only one valid dependency edge $\langle v_{i',j_{i'}}, e_{i,k_i} \rangle$. If a_i transits to its next edge state is e_{i,k_i} , the dependency edge becomes invalid and c also becomes a violated conflict.

To ensure conflict-freeness and deadlock-freeness, we require that (1) no presettled conflict has its only valid dependency edge become invalid; (2) no unsettled conflict has both its dependency edges become invalid; and (3) if the valid dependency edges of new presettled conflicts (coming from originally unsettled conflicts) are added to the partial SDG, the feasibility test with the *remaining* unsettled conflicts returns a positive result. If these requirements are satisfied, it guarantees that the *leftover* path plan is feasible if all or any subset of the agents in $U \cup \Delta U$ arrive at their next nodes. This is because the dependency edges for the unsettled conflicts never become invalid if the agents transit from edge states to their next node states. Hence, the unsettled conflicts are not changed after all or some agents in $U \cup \Delta U$ arrive at their next nodes. Thus, the feasibility test result must remain positive. The function **UnblockTest** (lines 1-9 of Algorithm 26) checks if a set of agents ΔU can be safely unblocked according to the above requirements.

To optimize the path execution effectiveness, the objective of our algorithm is to maximize $|\Delta U|$ (the number of agents to unblock). We develop a heuristic algorithm (Algorithm 26). First, for each agent $a_i \in B$, we check if a_i can be unblocked while keeping all other agents in B blocked. If not, a_i must be blocked (lines 11-13). Then, we initialize ΔU by excluding from B all agents that must be blocked (line 14) and check whether all the agents in ΔU can be unblocked together (line 15). If not, the **UnblockTest** function must return a violated conflict or a conflict causing cycles in the feasibility test. Typically, between the two agents involved in the conflict returned, at least one agent a comes from ΔU , since the path plan before unblocking ΔU was feasible. In this case, we remove a from ΔU (lines 16-17) and check whether all the remaining agents in ΔU can be unblocked together. If both agents involved in the conflict returned are not in ΔU , we simply set ΔU to an empty set (lines 18-19). This process is repeated until all the agents in ΔU can be unblocked together or ΔU becomes empty (lines 15-19). In the case that both U and the resultant ΔU are empty, we pick an agent randomly from those that can be unblocked alone (lines 20-21). Since the path plan before unblocking is feasible, such an agent must exist.

Algorithm 26 ensures that (1) agents executing *move* actions do not collide with each other or with agents executing *wait* actions; and (2) if all or any subset of *move* actions being executed are completed, the leftover path plan remains feasible. Thus,

running Algorithm 26 repeatedly guarantees that the path execution is conflict-free and deadlock-free.

Algorithm 26: Unblock agents in online path execution

```

1 Function UnblockTest( $\Delta U$ ):
2   Tentatively set all agents in  $\Delta U$  to their next edge states;
3    $\mathcal{C}_{\text{violated}} \leftarrow \{c \in \mathcal{C}_{\text{unsettled}} \mid \text{both dependency edges for } c \text{ become}$ 
    $\text{invalid}\} \cup \{c \in \mathcal{C}_{\text{presettled}} \mid \text{the valid dependency edge for } c \text{ becomes invalid}\};$ 
4   if  $\mathcal{C}_{\text{violated}} \neq \emptyset$  then
5     return a conflict in  $\mathcal{C}_{\text{violated}}$ ;
6    $\mathcal{C}_{\text{fixed}} \leftarrow \{c \in \mathcal{C}_{\text{unsettled}} \mid \text{one dependency edge for } c \text{ becomes invalid}\};$ 
7   if  $\text{FeasibilityTest}(\mathcal{C}_{\text{unsettled}} \setminus \mathcal{C}_{\text{fixed}}) = \text{infeasible}$  then
8     return the conflict returned from the feasibility test;
9   return success;
10  $B_K \leftarrow \emptyset;$ 
11 foreach agent  $a_i \in B$  do
12   if UnblockTest( $\{a_i\}$ )  $\neq \text{success}$  then
13      $B_K \leftarrow B_K \cup \{a_i\};$ 
14  $\Delta U \leftarrow B \setminus B_K;$ 
15 while  $\Delta U \neq \emptyset$  and UnblockTest( $\Delta U$ )  $\neq \text{success}$  do
16   if the conflict returned involves an agent in  $\Delta U$  then
17      $\Delta U \leftarrow \Delta U \setminus \{a_i\};$ 
18   else
19      $\Delta U \leftarrow \emptyset;$ 
20 if  $U \cup \Delta U = \emptyset$  then
21    $\Delta U \leftarrow$  a randomly chosen agent in  $B \setminus B_K;$ 
22 return  $\Delta U;$ 

```

5.4 Experimental Evaluation

We conduct experiments using a setup largely consistent with that described in Section 4.3. Specifically, we use four distinct maps (the first four maps in Section 4.3, which are empty-32-32, room-32-32-4, random-32-32-20 and warehouse-10-20-10-2-1) from the Moving AI MAPF benchmark [2], and the length of each grid cell side is set to 1 unit. We experiment with the 2^n -connected variants of each map, where $n = 2, 3, 4$, and use the 25 different random placements of the start and goal locations of the agents from the scenario files for each map. All agents are modeled as disks with a radius of $\sqrt{2}/4$ unit [48], and their movement speed is fixed at 1

unit per time unit. While these parameters are chosen for ease of experimental evaluation, our algorithm is general and can be applied to agents with arbitrary shapes and speeds.

For each instance, we run Continuous-time Large Neighborhood Search (CLNS) as described in Chapter 4 to obtain a suboptimal solution to the continuous-time MAPF problem. We do not employ an optimal solver such as Continuous-time CBS [48, 49], a state-of-the-art variant of Conflict-Based Search [21] designed for continuous-time MAPF. This decision is based on the limited scalability of CCBS, which typically handles only around 20 to 30 agents on the benchmark maps and fails to produce solutions for many instances in reasonable time. In contrast, CLNS scales to hundreds of agents while still achieving solutions with sum-of-costs close to optimal, making it more suitable for large-scale experimental evaluation.

To the best of our knowledge, there is no existing mechanism for robust plan execution in the continuous time setting. An intuitive method is to replan the paths of agents with continuous-time MAPF solvers such as CCBS and CLNS whenever any agent is unexpectedly delayed. However, the success rate of CCBS is quite low on the maps within a graceful time limit (e.g., 10 seconds), and CLNS also requires around 10 seconds to find a near-optimal solution for some large instances. Since hundreds of replans are required in a single path execution (by the pauses described below), the computation time of the replanning method will be huge, so we do not include the replanning method for comparison.

Because of the limitations of the replanning method, we compare our proposed online path execution algorithm with the offline algorithm as a baseline. Note that the solution produced by continuous-time MAPF solvers includes a sequence of *move* actions with action start times for each agent. However, we cannot derive the order for the agents to occupy the nodes or edges involved in potential conflicts directly from the MAPF solution because of the uncertainty we want to handle. For example, if there is an edge-edge conflict between two agents, it is possible that they can be traversing the edges concurrently with certain offsets in starting their *move* actions to avoid a collision in the MAPF solution. This is because these continuous-time MAPF solvers assume that agents move at constant speeds without interruption. But it is unsafe in our problem since the durations of *move* actions are not deterministic and agents can be located anywhere along the edges

in the action durations. Our offline algorithm derives the order for the agents to occupy the nodes or edges involved in potential conflicts with the feasibility test (Section 5.3.1), which may result in a slightly different execution from the MAPF solution even if no unexpected delay is introduced.

We emulate uncertainty by pausing 10% randomly chosen agents every k time units, and each pause lasts for k time units as well. This is the same as the experimental setup we used in Chapter 3. Specifically, if an agent is executing a *wait* action at a node when paused, it remains at the node for additional k time units; if an agent is executing a *move* action on an edge when paused, it is delayed for k time units in arriving at the adjacent node. Note that there can be consecutive pauses on the same agent due to random choices of agents to pause, in which case the delays will be aggregated. For fair comparison, we pause the same agents at the same time instant for both offline and online algorithms. We measure the sum-of-costs, which refers to the total time taken by all agents to reach their goal locations. An algorithm with a lower sum-of-costs is more effective in path execution. Given a k setting, we run the path execution for 10 times with different random agents paused. We present the average result for all runs together with the 95% confidence interval (e.g., 25 agent placements $\times$ 10 uncertainty emulations = 250 runs for each map). We implement the algorithms in C++ and run the experiments on a machine with Intel Core i9-9820X 3.30GHz CPU and 64GB memory.

5.4.1 Offline vs. Online Algorithm

First, we compare the online algorithm with all optimizations (**online-all-opt**), including removing redundant conflicts, constructing conflict groups and all optimizations in the discrete-time version, with the offline algorithm on different maps. Both algorithms can complete path execution successfully in all runs. On each map, we perform experiments for different numbers of agents on 4-connected (2^2), 8-connected (2^3) and 16-connected (2^4) variants of each map. A 2^n -connected grid map means that each node in the map is connected to 2^n different nodes nearby.

In Figures 5.14, 5.15 and 5.16, we present the results for scenarios with 40 agents across the three connectivity variants of each map. When $k = 0$ (i.e., no pause during path execution), the online and offline algorithms exhibit similar sum-of-costs. Nonetheless, a consistent yet modest advantage is observed for the online

algorithm in most configurations. This is likely due to the fact that the initial plan generated by CLNS is suboptimal, and the online execution algorithm is able to refine the plan dynamically during execution.

For $k > 0$, the online algorithm significantly outperforms the offline counterpart in terms of sum-of-costs. The performance gap widens as k increases, which reflects the growing uncertainty introduced by larger delays. The longer the pause time, the more the online algorithm benefits from its ability to adapt and coordinate in real time. However, the performance difference between the two approaches diminishes on maps with denser connectivity. In particular, the 2^4 -connected variants (Figure 5.16) show the smallest difference in sum-of-costs between the online and offline algorithms. This is likely because the higher connectivity provides more alternative paths and resolution options, reducing the impact of delays and making the initial path plan more robust.

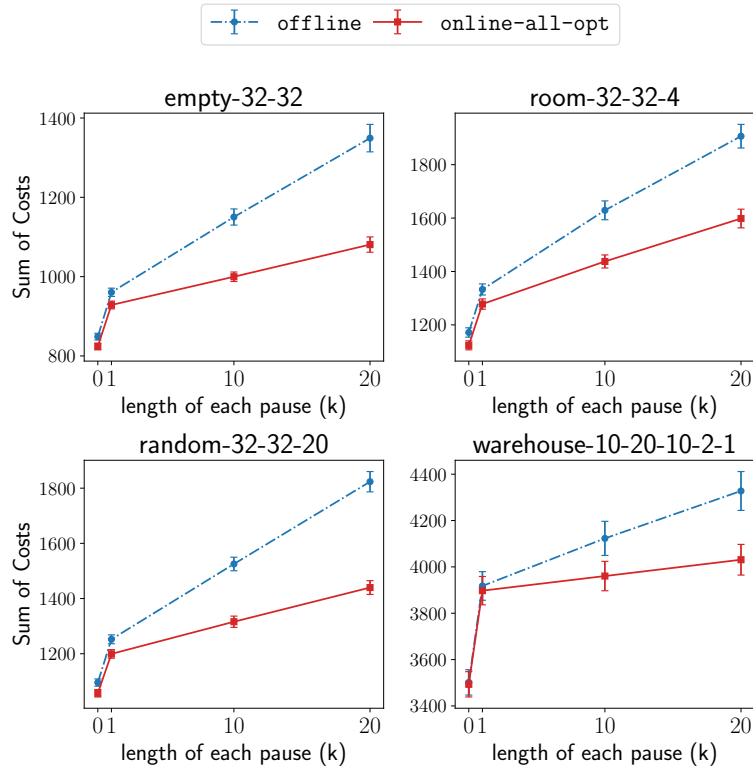


FIGURE 5.14: Comparison of offline and online algorithms for 40 agents on 2^2 -connected maps.

Figure 5.17, 5.18 and 5.19 show the sum-of-costs results of offline and online algorithms as a function of the number of agents, where k is set to 20. As can

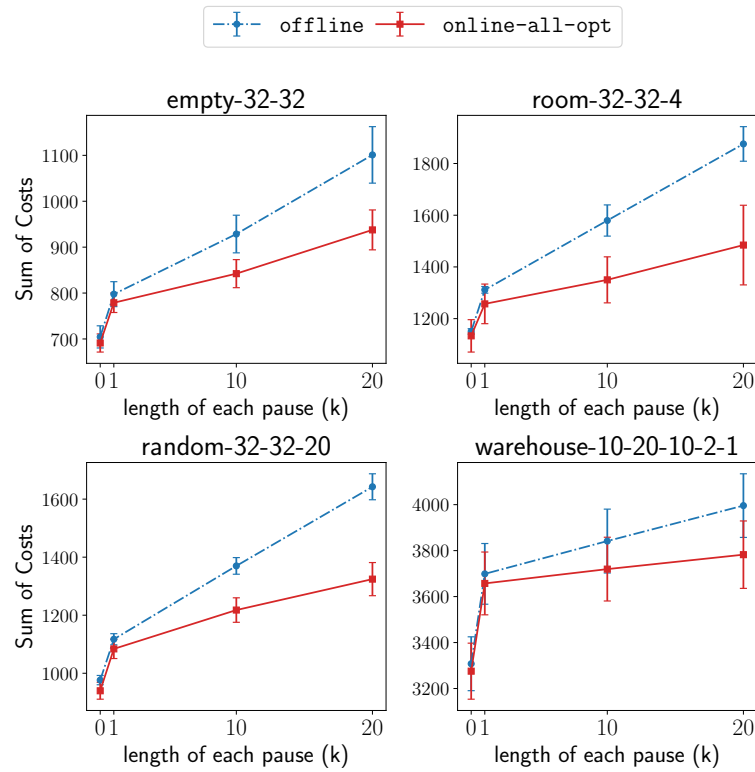


FIGURE 5.15: Comparison of offline and online algorithms for 40 agents on 2^3 -connected maps.

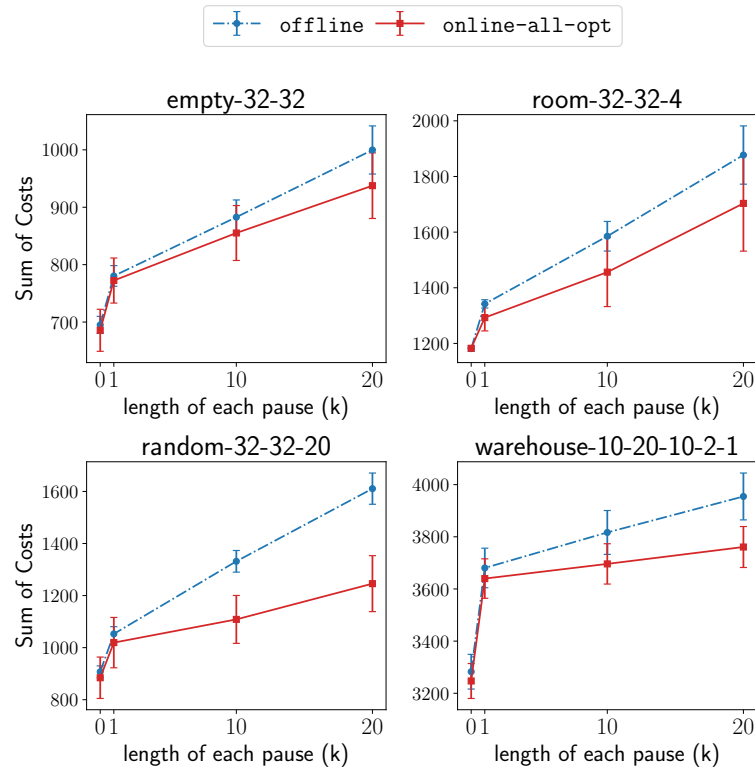


FIGURE 5.16: Comparison of offline and online algorithms for 40 agents on 2^4 -connected maps.

be seen, the relative performance between the two methods is maintained over different numbers of agents.

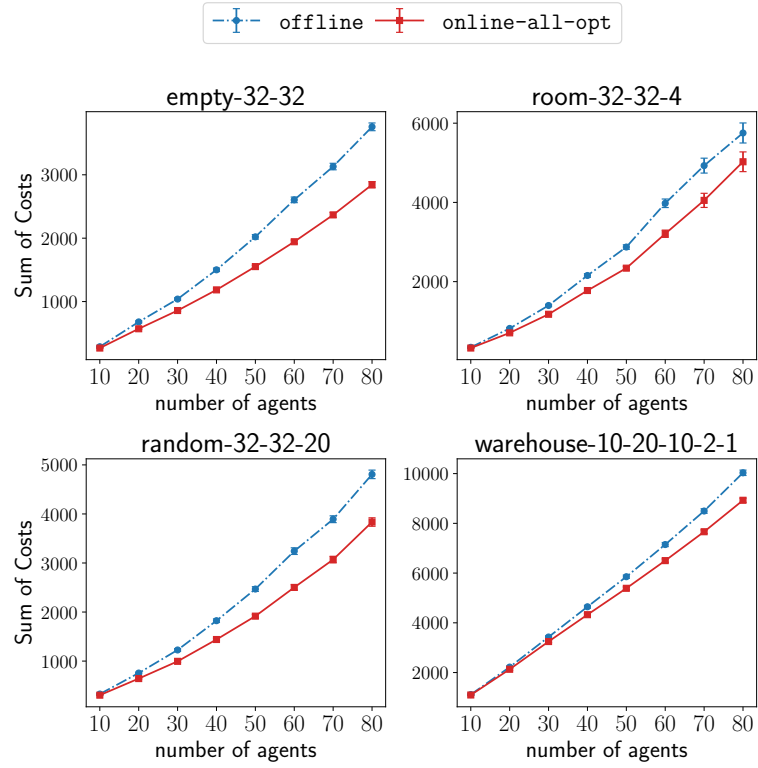


FIGURE 5.17: Comparison of offline and online algorithms for different number of agents with $k = 20$ on 2^2 -connected maps.

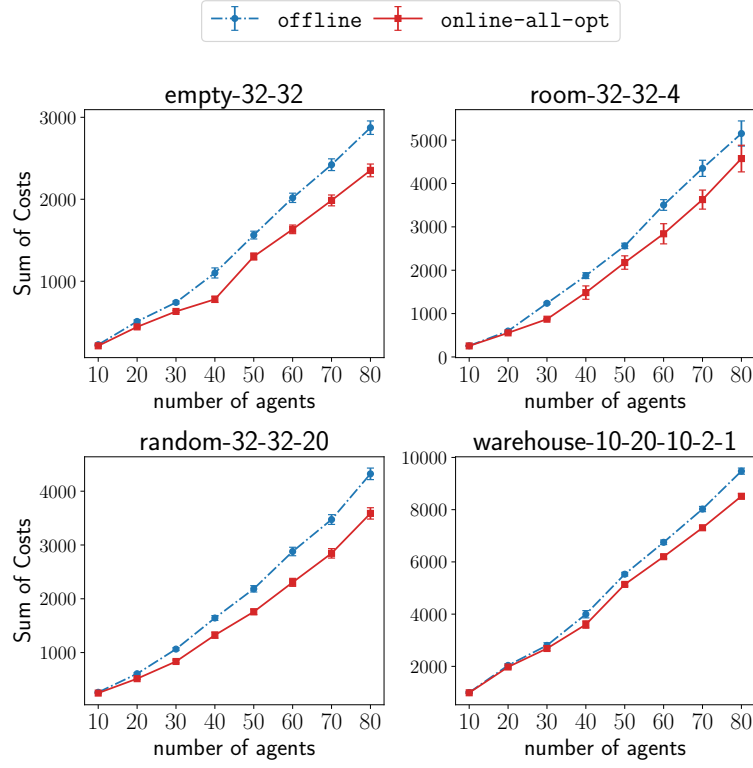


FIGURE 5.18: Comparison of offline and online algorithms for different number of agents with $k = 20$ on 2^3 -connected maps.

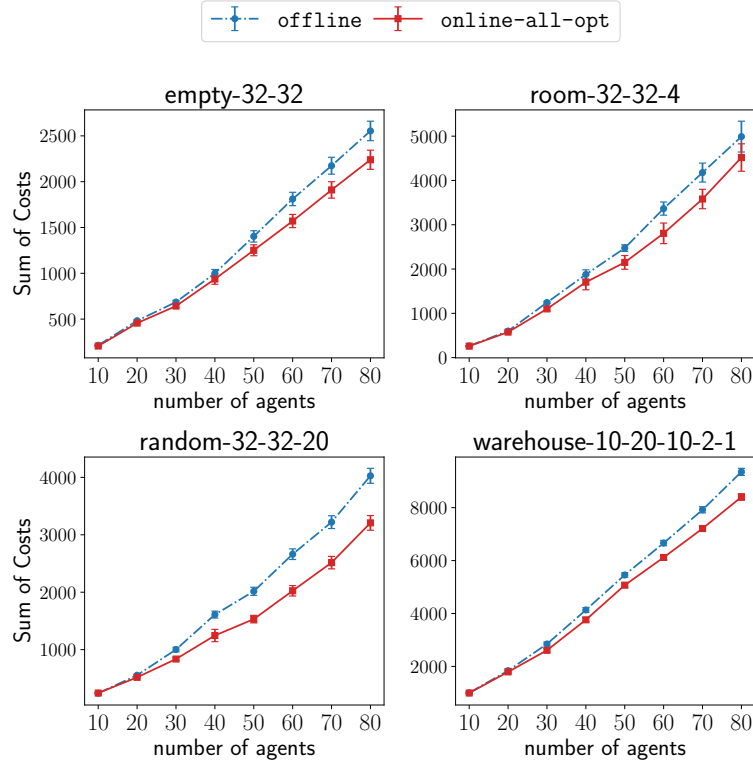


FIGURE 5.19: Comparison of offline and online algorithms for different number of agents with $k = 20$ on 2^4 -connected maps.

5.4.2 Removing Redundant Conflicts and Constructing Conflict Groups

Second, we compare the computational efficiency of different versions of feasibility test in the online path execution algorithm with and without the optimizations. Note that we apply all discrete-time optimizations, including heuristic feasibility test, connection array cycle detection and grouping agents in all versions of the feasibility tests. The five different labels in the figures of this section mean the following: (1) **online** is the feasibility test without any further optimizations; (2) **online-remove-redundant-graph** uses the straightforward pairwise iteration method (Algorithm 22) to remove redundant conflicts; (3) **online-remove-redundant-physical** uses the physical meaning induction method to remove redundant conflicts (described towards the end of Section 5.2.2); (4) **online-group** uses the heuristic method (Algorithm 24) to construct groups; (5) **online-all-opt** used all optimizations, which combines **online-remove-redundant-physical** and **online-group** together.

In Figures 5.20, 5.21 and 5.22, results are presented for instances with 40 agents, while the computation time of the online algorithm without the proposed optimizations becomes prohibitively long when there are more agents. Each figure reports the average computation time of running Algorithm 26 per simulated time unit until all 40 agents successfully reach their goal locations and the path execution completes.

The results show that removing redundant conflicts using either of the proposed methods significantly reduces computation time, achieving up to five times of speedup. Additionally, constructing conflict groups also leads to notable performance gains in three of the four maps. The exception is the **empty-32-32** map, where the benefit is limited, likely due to the sparse number of conflict groups formed in open environments. When both optimizations (redundant conflict removal and conflict group construction) are applied together, the marginal benefit is reduced. This is because many of the conflicts that could be grouped have already been eliminated during the redundancy removal stage. Furthermore, the average computation time decreases as the parameter k increases, since a larger delay reduces the number of agents considered in each coordination step as moving agents do not need to be considered when paused.

Finally, we verify that the sum-of-costs remains nearly identical across all five versions of the feasibility test. The figures that compare the sum-of-costs are omitted because the differences are indistinguishable. Minor variations may occur due to slight differences in execution order, but the results validate the correctness of the proposed optimizations.

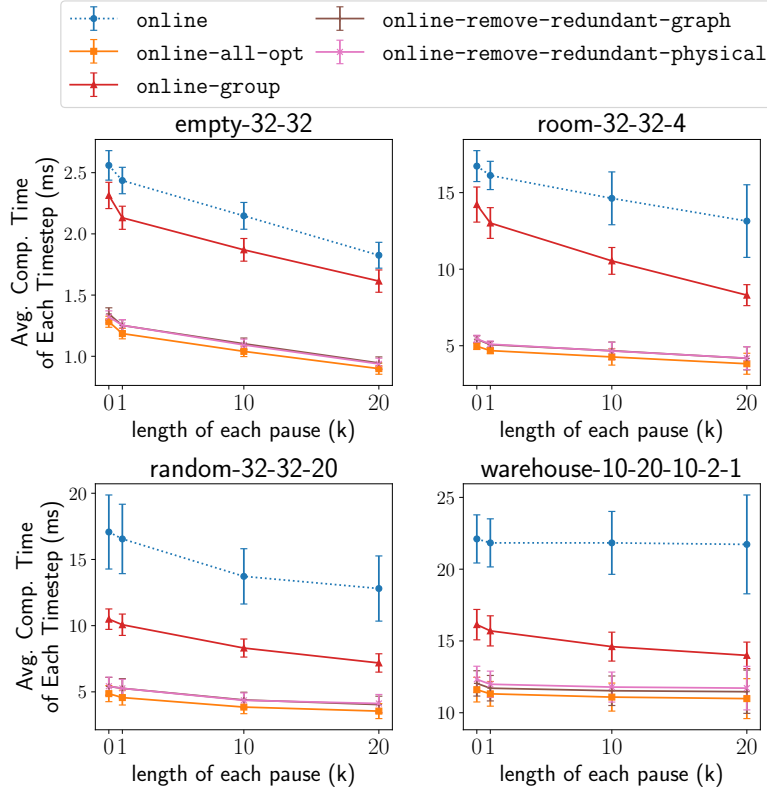


FIGURE 5.20: Comparison of online algorithm and its optimized versions for 40 agents on 2^2 -connected maps.

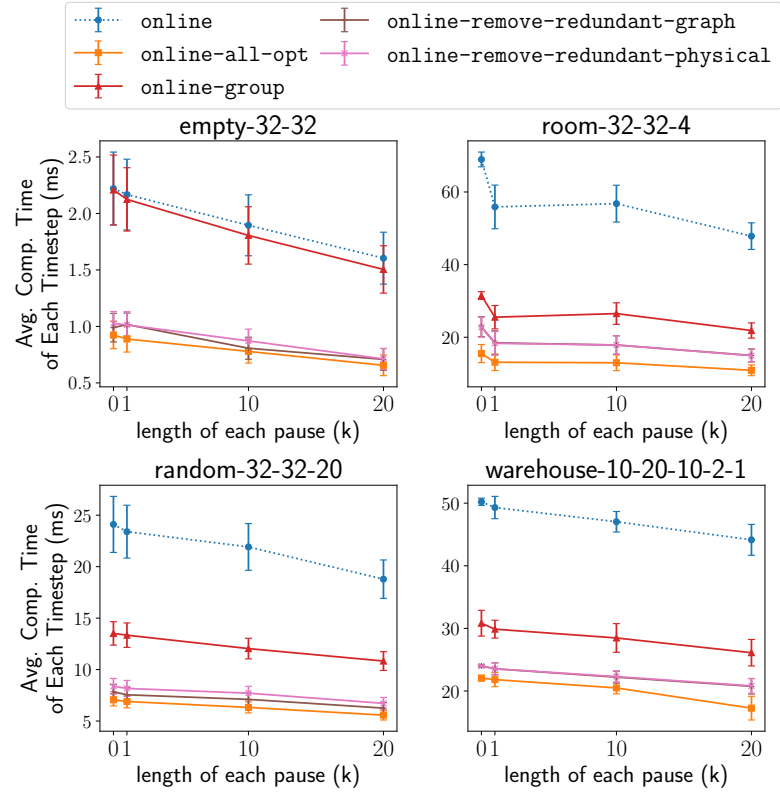


FIGURE 5.21: Comparison of online algorithm and its optimized versions for 40 agents on 2^3 -connected maps.

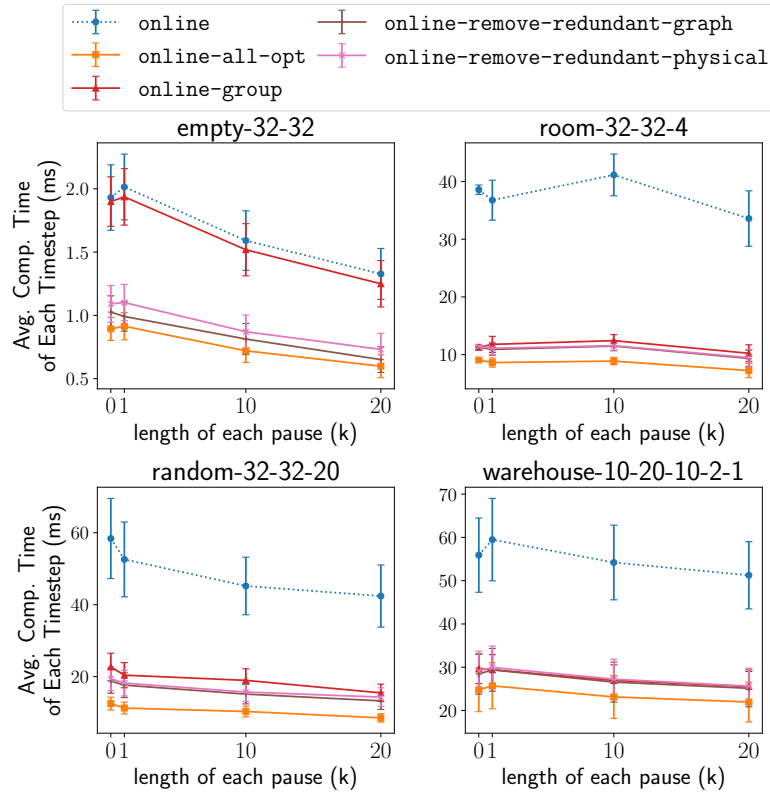


FIGURE 5.22: Comparison of online algorithm and its optimized versions for 40 agents on 2^4 -connected maps.

5.5 Summary

This chapter extended the discrete-time execution framework in Chapter 3 to continuous-time settings, addressing the additional complexities of asynchronous agent motion and temporal uncertainty. The initial paths for execution were generated using the hierarchical solver developed in Chapter 4, providing a strong foundation for robust coordination. New feasibility tests and execution policies were designed to handle continuous-time conflicts, supported by targeted optimizations to ensure efficiency and scalability. Experimental results showed significant improvements in robustness and reliability over prior methods. This chapter completes the transition from discrete to continuous-time execution frameworks, offering a unified and practical solution for real-world multi-agent coordination.

Chapter 6

Conclusion & Future Work

6.1 Conclusion

This thesis investigated the planning and execution of multi-agent systems, focusing on developing robust and efficient coordination strategies under uncertainty. The study addressed both discrete-time and continuous-time settings, extending the classical Multi-Agent Path Finding (MAPF) framework to more realistic situations where agents may face delays, asynchronous motion, or unexpected disturbances during execution. Through systematic problem formulation, algorithm design, and empirical evaluation, this work provided a unified view of how planning and execution can be closely integrated to achieve conflict-free and deadlock-free coordination in uncertain environments. The results contribute to both the theoretical understanding and the practical implementation of multi-agent coordination, showing how principled algorithmic design can effectively bridge the gap between abstract models and real-world robotic systems.

The work began by formalizing the Multi-Agent Path Execution with Uncertainty problem in discrete-time settings, where agents may experience delays or other unexpected disturbances during plan execution (Chapter 3). A feasibility problem was proposed to determine whether all agents can still reach their goals without conflicts despite these uncertainties. This problem was shown to be NP-complete, motivating the design of both deterministic and heuristic feasibility test algorithms that

coordinate agents online. The framework ensures conflict-freeness and deadlock-freeness while maintaining high concurrency, and experimental results confirmed its robustness and efficiency compared with existing approaches.

Building upon these results, the thesis introduced a hierarchical solver for Continuous-Time Multi-Agent Path Finding, combining Safe Interval Path Planning with Soft Constraints (SIPPS) at the low level and Large Neighborhood Search (LNS) at the high level (Chapter 4). This solver balances computational efficiency with path quality and demonstrates scalability across large and diverse benchmark maps. The approach generalizes well to various hierarchical MAPF frameworks, bridging the gap between discrete and continuous planning.

The research then advanced to Robust Continuous-Time Multi-Agent Path Execution, which extended the discrete-time execution framework to handle asynchronous and continuous agent motion (Chapter 5). The initial paths were generated using the hierarchical solver from Chapter 4, serving as inputs for the robust execution framework. New feasibility tests and execution policies were developed to resolve continuous-time conflicts while accounting for timing deviations and dynamic interactions. Targeted optimizations further enhanced efficiency and scalability, and empirical evaluations demonstrated superior robustness and reliability compared with prior methods.

From a theoretical standpoint, this thesis formalized feasibility testing for execution under uncertainty and introduced generalized conflict types for continuous-time coordination. The frameworks presented here enrich the understanding of how timing deviations and asynchronous interactions affect multi-agent behavior, offering tools that can serve as building blocks for future uncertainty-aware systems. From a practical standpoint, the methods provide computationally efficient solutions for applications such as automated warehouses, multi-robot logistics, and autonomous vehicle coordination, where agents must operate safely despite timing variations and imperfect synchronization.

Reflecting on this research journey, the work evolved from tackling discrete-time coordination to developing robust continuous-time execution models, revealing the delicate balance between optimality, robustness, and computational efficiency. The insights gained highlight the importance of combining theoretical soundness with

practical design, paving the way toward more adaptive, uncertainty-aware multi-agent coordination frameworks that can eventually support large-scale, long-term autonomous systems.

6.2 Future Work

While the proposed methods are effective, there are still limitations and open challenges. The frameworks assume accurate knowledge of agent dynamics and static environments, which may not always hold in real-world deployments. The current algorithms also rely on deterministic feasibility checks, leaving room for improvement in handling unpredictable or time-varying conditions. These limitations naturally suggest several directions for future research.

In the future, this work can be extended in several ways to make the proposed frameworks more flexible and applicable to complex environments. One natural extension is to support lifelong multi-agent path execution under uncertainty, where agents continuously take on new tasks instead of operating within a single, fixed plan. It would also be valuable to integrate uncertainty handling directly into the planning phase, so that the generated paths are naturally more tolerant to execution-time variations. In addition, improving the optimization of suboptimal plans in large-scale systems remains a meaningful challenge, as better plan refinement could further enhance both efficiency and adaptability in practical deployments.

6.2.1 Lifelong Multi-Agent Path Execution with Uncertainty

Recall that in Section 2.1.4, we defined two simplified behaviors for agents at goal locations: *staying at goal*, where agents remain idle at their goal positions indefinitely, and *disappearing at goal*, where agents are removed from the system after completing their planned paths. While these assumptions are convenient for classical MAPF, they are simplifications of agent behavior in real-world applications.

To address more realistic scenarios, the Lifelong Multi-Agent Path Finding (Lifelong MAPF) problem generalizes classical MAPF by allowing agents to receive new goals and continue navigating after completing their current path plans. In this sense, classical MAPF represents only the “one-shot” variant of a broader, ongoing coordination problem.

Although Lifelong MAPF has been actively studied in recent years [69, 70], there has been limited research on the robust execution of lifelong path plans under uncertainty. The execution frameworks proposed in Chapters 3 and 5 can be extended to support lifelong MAPF by continuously updating the Location Dependency Graph (LDG) or State Dependency Graph (SDG) as new paths are assigned to agents.

The primary challenge in this extension lies in maintaining the feasibility of the overall execution plan as the dependency graph evolves. Specifically, it is necessary to preserve cycle-freeness when selecting unsettled dependency edges, even as new edges are introduced. Achieving this may require redesigning either or both components of the system, including the path planning algorithm and the path execution framework, to support dynamic, conflict-free updates.

Furthermore, the execution framework can be adapted to the Multi-Agent Pickup and Delivery (MAPD) problem, which is closely related to Lifelong MAPF. In MAPD, agents are tasked with visiting one or more pairs of goal locations, each consisting of a pickup location and a corresponding delivery location [71]. Similar execution challenges arise in MAPD due to the continuous assignment of tasks and the need for conflict-free coordination during extended operations.

6.2.2 Multi-Agent Path Finding with Uncertainty

The execution frameworks proposed in Chapters 3 and 5 primarily address the execution stage of multi-agent problems. The initial path plans can be generated by any existing MAPF solver, provided that the solver accounts for the same constraints (the types of conflicts) by the execution frameworks. This design provides flexibility and generality in choosing MAPF solvers. However, it may also compromise the effectiveness of execution in practice.

This limitation arises because most MAPF solvers do not account for execution-time uncertainty during the planning stage. As a result, path plans may contain time windows with densely packed agent movements over shared nodes or edges. Though such time windows do not introduce conflicts if agents never delay, they may lead to potential bottlenecks in robust path execution with uncertainty, which hinder progress and reduce the overall efficiency of the system in real-world deployments.

Some prior work, such as MAPF with Time Uncertainty (MAPF-TU) [50] and k -robust MAPF [51], attempts to incorporate robustness into the planning stage. However, these methods are not specifically designed to complement the execution frameworks developed in this thesis because they are all targeted at a certain type of uncertainty and cannot be applied to a generalized setup.

In the end, this observation suggests that future improvements should consider uncertainty with a co-design of both the path planning and execution stages. Integration between the two stages would enable coordinated reasoning about uncertainty, resulting in higher-quality, robust, and scalable solutions for Multi-Agent Path Planning and Execution with Uncertainty.

6.2.3 Optimization of Suboptimal Multi-Agent Path Plans

As observed in Chapters 4 and 5, both the path planning and path execution stages tend to have lower sum-of-costs on the same map as the connectivity increases (i.e., when moving from 2^2 -connected to 2^4 -connected grid maps). However, this improvement in solution quality comes at the cost of increased computation time. Planning algorithms such as CCBS [48] become significantly slower with higher map connectivity, as the number of edges increases, leading to more computational work in both the low-level single-agent planning and high-level conflict resolution. This reveals a fundamental trade-off between the efficiency of the resulting path plan and the computational efficiency of the MAPF solver.

The feasibility test in the continuous-time path execution framework introduced in Chapter 5 offers a novel opportunity to optimize path plans after initial planning. Specifically, it enables transformations of the map or path plan without

full re-planning. For example, the map can be locally transformed from a lower-connectivity version to a higher-connectivity one to explore potentially shorter transitions. Additionally, even without changing the underlying map, the optimizer can examine opportunities to replace two or more consecutive edges with a single shorter edge that was not used in the original plan.

When such an optimization is proposed, the corresponding states in the State Dependency Graph (SDG) can be merged accordingly. The updated SDG is then passed to the feasibility test to verify whether the new transition is valid under uncertainty. If the feasibility test passes, the optimization is accepted. Otherwise, the transformation can be rejected, or further local adjustments can be made to the original plan to resolve any conflicts revealed by the test.

This optimization strategy is considerably more efficient than fully replanning with CBS-based methods. It leverages the strengths of the feasibility test which has been shown to be efficient through extensive optimizations, and provides a lightweight mechanism to improve path quality while preserving robustness and scalability in execution.

List of Author's Awards, Patents, and Publications

Awards

- **Best Paper Awards Finalist**, “Playground: A Safe Building Operating System.” In *2024 ACM/IEEE 15th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 111-122. IEEE, 2024..

Conference Proceedings

- **Liu, Yihao**, Xueyan Tang, Wentong Cai, and Jingning Li. “Multi-Agent Path Execution with Uncertainty.” In *Proceedings of the International Symposium on Combinatorial Search*, vol. 17, pp. 64-72. 2024.
- Fu, Xiaohan, **Yihao Liu**, Jason Koh, Dezhi Hong, Rajesh Gupta, and Gabe Fierro. “Playground: A safe building operating system.” In *2024 ACM/IEEE 15th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 111-122. IEEE, 2024.
- Ramanathan, Saravanan, **Yihao Liu**, Xueyan Tang, Wentong Cai, and Jingning Li. “Minimising Task Tardiness for Multi-Agent Pickup and Delivery.” In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pp. 2349-2351. 2023.
- Wu, Xiaohu, **Yihao Liu**, Xueyan Tang, Wentong Cai, Funing Bai, Gilbert Khonstantine, and Guopeng Zhao. “Multi-agent pickup and delivery with task deadlines.” In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pp. 360-367. 2021.

Bibliography

- [1] Chen Wang. *Kernel learning for visual perception*. PhD thesis, 2019. [ix](#)
- [2] 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. In *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS)*, pages 151–158, 2019. [xix](#), [10](#), [20](#), [49](#), [50](#), [74](#), [107](#), [108](#), [138](#)
- [3] Peter R. Wurman, Raffaello D’Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1): 9–9, 2008. [1](#)
- [4] R. Morris, C. S. Pasareanu, K. Luckow, W. Malik, H. Ma, T. K. S. Kumar, and S. Koenig. Planning, scheduling and monitoring for airport surface operations. In *Proceedings of the AAAI Workshop on Planning for Hybrid Systems*, 2016. [1](#)
- [5] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. CoBots: robust symbiotic autonomous mobile service robots. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4423–4429, 2015. [1](#)
- [6] Didi Chuxing. Didi Chuxing Corporate Citizenship Report, 2017. http://img-ys011.didistatic.com/static/didiglobal/do1_p53rQtxhA6BjW6uWpF6t, 4 2018. [2](#)
- [7] Ariel Felner, Roni Stern, Solomon Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS)*, pages 29–37, 2017. [2](#), [3](#), [44](#), [49](#)
- [8] Wolfgang Hönl, Scott Keisel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. Persistent and robust execution of mapf schedules in warehouses. *IEEE Robotics and Automation Letters*, 4(2):1125–1131, 2019. [2](#), [46](#), [49](#), [119](#)

- [9] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Iterative refinement for real-time multi-robot path planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9690–9697, 2021. doi: 10.1109/IROS48987.2021.9636355. [2](#), [49](#)
- [10] 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. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10256–10265, 2022. [3](#), [85](#), [87](#), [104](#)
- [11] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Anytime multi-agent path finding via large neighborhood search. In *International Joint Conference on Artificial Intelligence 2021*, pages 4127–4135. Association for the Advancement of Artificial Intelligence (AAAI), 2021. [3](#), [39](#), [85](#), [99](#)
- [12] Hang Ma, Daniel Harabor, Peter J Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7643–7650, 2019. [11](#), [23](#), [31](#)
- [13] Jingjin Yu and Steven M LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013. [12](#), [99](#)
- [14] Ariel Felner, Roni Stern, Solomon Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Tenth Annual Symposium on Combinatorial Search*, 2017. [12](#), [21](#)
- [15] Steve Rabin. *AI Game Programming Wisdom 3 (Game Development Series)*. Charles River Media, Inc., 2006. [14](#), [87](#)
- [16] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE international conference on robotics and automation*, pages 5628–5635. IEEE, 2011. [16](#), [87](#)
- [17] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernández, Ariel Felner, TK Satish Kumar, and Sven Koenig. Anytime focal search with applications. In *IJCAI*, pages 1434–1441, 2018. [17](#)
- [18] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 2010. [22](#)
- [19] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert C Holte, and Jonathan Schaeffer. Enhanced partial expansion a. *Journal of Artificial Intelligence Research*, 50:141–187, 2014. [23](#)

- [20] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013. [23](#)
- [21] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66, 2015. [23](#), [24](#), [74](#), [98](#), [139](#)
- [22] Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K Brayton. Algorithms for discrete function manipulation. In *1990 IEEE international conference on computer-aided design*, pages 92–93. IEEE Computer Society, 1990. [24](#)
- [23] Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. Don’t split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 47–51, 2015. [28](#), [39](#)
- [24] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Eyal Shimony. Icbs: The improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 6, pages 223–225, 2015. [28](#), [39](#), [43](#)
- [25] Jiaoyang Li, Daniel Harabor, Peter J Stuckey, Ariel Felner, Hang Ma, and Sven Koenig. Disjoint splitting for multi-agent path finding with conflict-based search. In *Proceedings of the international conference on automated planning and scheduling*, volume 29, pages 279–283, 2019. [28](#), [43](#)
- [26] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, TK Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, pages 83–87, 2018. [28](#), [43](#)
- [27] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *IJCAI*, volume 2019, pages 442–449, 2019. [28](#), [39](#), [43](#)
- [28] Malcolm Ross Kinsella Ryan. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, 31:497–542, 2008. [28](#), [34](#)
- [29] Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter J Stuckey. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research*, 144:105809, 2022. [28](#), [29](#)
- [30] Graeme Gange, Daniel Harabor, and Peter J Stuckey. Lazy cbs: implicit conflict-based search using lazy clause generation. In *Proceedings of the international conference on automated planning and scheduling*, volume 29, pages 155–162, 2019. [28](#)

- [31] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient sat approach to multi-agent path finding under the sum of costs objective. In *ECAI 2016*, pages 810–818. IOS Press, 2016. [28](#)
- [32] Esra Erdem, Doga Kisa, Umut Oztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, pages 290–296, 2013. [28](#)
- [33] Jingjin Yu and Steven M LaValle. Planning optimal paths for multiple robots on graphs. In *2013 IEEE International Conference on Robotics and Automation*, pages 3612–3617. IEEE, 2013. [28](#)
- [34] Pavel Surynek. Compilation-based solvers for multi-agent path finding: a survey, discussion, and future opportunities. *arXiv preprint arXiv:2104.11809*, 2021. [28](#)
- [35] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the international symposium on combinatorial Search*, volume 5, pages 19–27, 2014. [30](#), [99](#)
- [36] Liron Cohen, Tansel Uras, and Sven Koenig. Feasibility study: Using highways for bounded-suboptimal multi-agent path finding. In *Eighth Annual Symposium on Combinatorial Search*, 2015.
- [37] Liron Cohen, Tansel Uras, TK Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. Improved solvers for bounded-suboptimal multi-agent path finding. In *IJCAI*, pages 3067–3074, 2016. [30](#)
- [38] Jur P Van Den Berg and Mark H Overmars. Prioritized motion planning for multiple robots. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 430–435. IEEE, 2005. [30](#), [99](#)
- [39] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence*, 310:103752, 2022. [32](#)
- [40] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. winpibt: Extended prioritized algorithm for iterative multi-agent path finding. *arXiv preprint arXiv:1905.10149*, 2019. [32](#)
- [41] David Silver. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, volume 1, pages 117–122, 2005. [33](#), [34](#)
- [42] Zahy Bnaya and Ariel Felner. Conflict-oriented windowed hierarchical cooperative a*. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3743–3748. IEEE, 2014. [34](#)

- [43] Nathan R Sturtevant and Michael Buro. Improving collaborative pathfinding using map abstraction. In *AIIDE*, pages 80–85. Marina del Rey, 2006. [34](#)
- [44] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 12353–12362, 2021. [37](#), [74](#), [99](#)
- [45] Jordan Thayer and Wheeler Ruml. Using distance estimates in heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19, pages 382–385, 2009. [37](#)
- [46] Jordan Tyler Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI*, volume 2011, pages 674–679, 2011. [37](#)
- [47] Jordan Thayer, Austin Dionne, and Wheeler Ruml. Learning inadmissible heuristics during search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 21, pages 250–257, 2011. [38](#)
- [48] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. *Artificial Intelligence*, 305:103662, 2022. [42](#), [85](#), [98](#), [138](#), [139](#), [155](#)
- [49] Anton Andreychuk, Konstantin Yakovlev, Eli Boyarski, and Roni Stern. Improving continuous-time conflict based search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11220–11227, 2021. [43](#), [139](#)
- [50] Tomer Shahar, Shashank Shekhar, Dor Atzmon, Abdallah Saffidine, Brendan Juba, and Roni Stern. Safe multi-agent pathfinding with time uncertainty. *Journal of Artificial Intelligence Research*, 70:923–954, 2021. [45](#), [155](#)
- [51] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research*, 67:549–579, 2020. [45](#), [155](#)
- [52] Glenn Wagner and Howie Choset. Path planning for multiple agents under uncertainty. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 577–585, 2017. [46](#)
- [53] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R Sturtevant, and Sven Koenig. Probabilistic robust multi-agent path finding. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 29–37, 2020. [46](#)
- [54] Hang Ma, TK Satish Kumar, and Sven Koenig. Multi-agent path finding with delay probabilities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017. [46](#), [77](#), [119](#)

- [55] Wolfgang Hönig, TK Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26, pages 477–485, 2016. [46](#)
- [56] Alexander Berndt, Niels Van Duijkeren, Luigi Palmieri, and Tamas Keviczky. A feedback scheme to reorder a multi-agent execution schedule by persistently optimizing a switchable action dependency graph. *arXiv preprint arXiv:2010.05254*, 2020. [46](#), [53](#)
- [57] Adem Coskun, Jason O’Kane, and Marco Valtorta. Deadlock-free online plan repair in multi-robot coordination with disturbances. In *Proceedings of the International FLAIRS Conference*, volume 34, 2021. [53](#)
- [58] Adittyo Paul, Ying Feng, and Jiaoyang Li. A fast rescheduling algorithm for real-time multi-robot coordination [extended abstract]. In *Proceedings of the 16th International Symposium on Combinatorial Search (SoCS)*, pages 175–176, 2023.
- [59] Ying Feng, Adittyo Paul, Zhe Chen, and Jiaoyang Li. A real-time rescheduling algorithm for multi-robot plan execution. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 201–209, 2024. <https://github.com/YinggggFeng/Switchable-Edge-Search>. [46](#), [79](#), [82](#)
- [60] Yifan Su, Rishi Veerapaneni, and Jiaoyang Li. Bidirectional temporal plan graph: Enabling switchable passing orders for more efficient multi-agent path finding plan execution. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, pages 17559–17566, 2024. <https://github.com/YifanSu1301/BTPG>. [47](#), [79](#)
- [61] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 837–845, 2017. [49](#)
- [62] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1898–1900, 2020. [49](#)
- [63] Alessandro Mascis and Dario Pacciarelli. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3): 498–517, 2002. [54](#)
- [64] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103, Boston, MA, 1972. Springer US. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9. [55](#)

- [65] Esther M Arkin, Aritra Banik, Paz Carmi, Gui Citovsky, Matthew J Katz, Joseph SB Mitchell, and Marina Simakov. Selecting and covering colored points. *Discrete Applied Mathematics*, 250:75–86, 2018. 55
- [66] Claudio Arbib, Gluseppe F. Italiano, and Alessandro Panconesi. Predicting deadlock in store-and-forward networks. *Networks*, 20(7):861–881, 1990. 60
- [67] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. 67
- [68] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Time-independent planning for multiple moving agents. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, pages 11299–11307, 2021. <https://github.com/Kei18/time-independent-planning>. 77
- [69] Qian Wan, Chonglin Gu, Sankui Sun, Mengxia Chen, Hejiao Huang, and Xiaohua Jia. Lifelong multi-agent path finding in a dynamic environment. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 875–882. IEEE, 2018. 154
- [70] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11272–11281, 2021. 154
- [71] Qinghong Xu, Jiaoyang Li, Sven Koenig, and Hang Ma. Multi-goal multi-agent pickup and delivery. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9964–9971. IEEE, 2022. 154