TARGET ASSIGNMENT AND PATH PLANNING FOR NAVIGATION TASKS WITH TEAMS OF AGENTS

by

Hang Ma

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2020

Acknowledgments

First and foremost, I would like to thank my advisor, Sven Koenig, for his guidance with a profound vision, his patience for leading many inspiring and insightful discussions, and his offering of the greatest freedom for me to pursue my own research ideas, all throughout this five-year journey. I can never learn enough from him about how to keep the enthusiasm and curiosity for both research and life. He has taught me how to be a great advisor of my own students in the future.

I would like to thank T. K. Satish Kumar, with whom I can never spend enough days and nights writing papers together, both on whiteboards and online. Satish has always been a great source of humor who tells only jokes made up by himself. I would like to thank other members in my dissertation committee and proposal guidance committee: Nora Ayanian, Gaurav S. Sukhatme, Satyandra K. Gupta, and Peter Stone, who have provided helpful comments and suggestions on this dissertation. Nora has provided great robotics insights and hardware for this dissertation. Peter has pointed out many interesting research directions on multi-agent/robot systems, and it has always been great fun interacting with his research group members at various conferences.

Much of my research would not have been successful without my colleagues and collaborators. I would like to thank Craig Tovey, through whom I obtained an Erdös number of 2. Our collaboration and discussions built up my confidence in the beginning of this journey and inspired much of my later research. I would like to thank Wolfgang Hönig, without whom my algorithms would have never been able to run on real robots. We have made a great AI+robotics team. I would like to thank Ariel Felner for our fruitful collaboration and discussions, from

which I have learned a lot about heuristic search. I cannot count how many telecons I have had with Daniel Harabor and Peter J. Stuckey, both from the other side of the earth. Our collaboration has always been productive and fun. I have had great fun with all members of the IDM Lab: I would like to thank Jiaoyang Li, who has grown from a mentee to a great colleague of mine and whose undergraduate summer research has become part of this dissertation. I would like to thank Liron Cohen and Tansel Uras for their collaboration and many great discussions. Liron has also been a great source of ideas for entertainment in Los Angeles. My special thanks go to former IDM Lab member William Yeoh, from whom I have learned about how to survive this journey. Many thanks to Hong Xu and Han Zhang, who have made our lab a joyful place. I have also been fortunate to work with and mentor many talented Masters and undergraduate students, including Minghua Liu, Jiangxing Wang, and Jingxing Yang. Many thanks to my great collaborators Guni Sharon, Glenn Wagner, Eli Boyarski, Pavel Surynek, Carlos Hernandez, Roni Stern, Nathan Sturtevant, Thayne Walker, Roman Bartak, and many others. We have formed a great MAPF research community.

I am grateful to Joelle Pineau, my former advisor at McGill University, who has inspired and encouraged me to work in AI. My special thanks go to many USC professors: David Kempe and Shanghua Teng have provided insightful suggestions for several research problems I have had. I have enjoyed interacting with Fei Sha and his group members. I would also like to thank Robert Morris and Corina Pasareanu from NASA Ames for a great blue-sky internship project and Ngai Meng Kou and Cheng Peng from Cainiao, Alibaba for an opportunity to apply my algorithms to warehouse automation.

My deepest gratitude to my parents, Xiaoxuan Ma and Bihua Lin, for their support, education, encouragement, and love that have made me who I am.

Last but not least, I would like to thank my wife, Xiaodong Huang, for her unconditional love, patience, and support. I have also been inspired a lot by just watching our newborn baby figuring out the world. My family has made this journey wonderful.

The research presented in this dissertation was supported by a USC Annenberg fellowship, Cainiao Smart Logistics Network, an internship at NASA Ames via Stinger Ghaffarian Technologies, an internship at ARL West, ONR in form of a MURI project under contract/grant number N00014-09-1-1031, NSF under grant numbers 1319966, 1409987, 1724392, 1817189, and 1837779, and a gift from Amazon.

Contents

List of Tables

List of Figures

List of Important Abbreviations

Abbreviation	Description
Problems:	
MAPD	Multi-Agent Pickup and Delivery: a long-term combined target-assignment and path-planning problem
MAPF	Multi-Agent Path Finding: a one-shot path-planning problem
PERR	Package Exchange Robot Routing: a one-shot path-planning problem
TAPF	Target Assignment and Path Finding: a one-shot combined target-assignment and path-planning problem
Algorithms:	
CBM	Conflict-Based Min-Cost Flow: a TAPF algorithm
CBS	Conflict-Based Search: a MAPF algorithm
CENTRAL	a centralized MAPD algorithm
ILP	Integer Linear Programming or used as the label of an Integer Linear Programming-based algorithm
MAPF-POST	a polynomial-time procedure that post-processes MAPF solutions
SIPP	Safe Interval Path Planning: a one-shot single-agent path-planning algorithm
SIPPwRT	Safe Interval Path Planning with Reservation Table: an improved version of Safe Interval Path Planning that stores paths in a reservation table
TP	Token Passing: a decoupled MAPD algorithm

Abbreviation	Description
TP-SIPPwRT	a version of Token Passing that uses Safe Interval Path Planning with Reservation Table for single-agent path planning and outputs plan-execution schedules
TPTS	Token Passing with Task Swaps: a decoupled MAPD algorithm

Abstract

In many real-world applications of multi-agent systems, teams of agents must assign targets among themselves and plan paths to the targets. The agents must avoid collisions with each other in a congested environment, yet reach their targets as soon as possible. The resulting coordination problems, which model the target-assignment and path-planning operations of the agents, are fundamental for these multi-agent systems but, at the same time, computationally challenging, as there are often many agents and their operating time is long. This dissertation studies two types of multi-agent coordination problems: (1) One-shot coordination problems that assign given targets to and plan paths for a given set of agents to reach their targets on a given graph that models the environment and (2) long-term coordination problems that repeatedly assign incoming tasks to a given set of agents and plan paths for the agents to the targets of their assigned tasks.

This dissertation builds upon the recent successes in tackling a one-shot path-planning problem, called Multi-Agent Path Finding (MAPF), in the artificial intelligence (AI) community as well as theoretical and algorithmic results for other target-assignment and path-planning problems from the theoretical computer science, operations research, AI, and robotics communities. It addresses three central questions when generalizing these results to solving the target-assignment and the path-planning problems jointly: Question 1: How hard is it to jointly assign targets to and plan paths for teams of agents? Question 2: How and how well can one jointly assign targets to and plan paths for teams of agents? Question 3: How do teams of agents execute the computed solutions?

We tackle these three central questions by formalizing novel variants of MAPF that model different target-assignment and path-planning problems, studying their theoretical properties, and developing algorithms for them and thus make the following four contributions. First, we introduce a unified NP-hardness proof structure that can be used to derive complexity results for different target-assignment and path-planning problems. Second, we formalize and study Combined Target Assignment and Path Finding (TAPF), which models the one-shot combined target-assignment and path-planning problem and present complete and optimal algorithms for solving it. Third, we formalize and study Multi-Agent Pickup and Delivery (MAPD), which models the long-term combined target-assignment and path-planning problem and present MAPD algorithms that provide a guarantee on their long-term robustness, namely, allow agents to finish all tasks without deadlocks. Fourth, we demonstrate how our MAPD algorithms can take some of the kinodynamic constraints of real-world agents into account to compute plan-execution schedules, which showcases the benefits of our algorithms for real-world applications of multi-agent systems.

Chapter 1

Introduction

In many real-world applications of multi-agent systems, teams of autonomous agents must assign targets (goal locations) among themselves (target-assignment operations) and plan collision-free paths to their targets (path-planning operations) in order to finish tasks cooperatively. Examples include autonomous aircraft-towing vehicles (Morris et al., 2016), automated warehouse robots (Kou, Peng, Ma, Kumar, & Koenig, 2020; Wurman, D'Andrea, & Mountz, 2008), automated-guided port vehicles (Thurston & Hu, 2002), autonomous intersection management (Dresner & Stone, 2008), forklift robot fleets (Pecora, Andreasson, Mansouri, & Petkov, 2018; Salvado, Krug, Mansouri, & Pecora, 2018), game characters in video games (Ma, Yang, Cohen, Kumar, & Koenig, 2017), object-transportation robots (Mataric, Nilsson, & Simsarin, 1995; Rus, Donald, & Jennings, 1995), patrolling robots (Agmon, Urieli, & Stone, 2011), search-and-rescue robots (Jennings, Whelan, & Evans, 1997), service robots (Ahmadi & Stone, 2006; Khandelwal et al., 2017; Veloso, Biswas, Coltin, & Rosenthal, 2015), swarms of differential-drive robots and quadcopters (Hönig, Kumar, Ma, Ayanian, & Koenig, 2016; Hönig, Preiss, Kumar, Sukhatme, & Ayanian, 2018; Preiss, Hönig, Ayanian, & Sukhatme, 2017), robots in formations (Balch & Arkin, 1998; Li, Sun, et al., 2020; Poduri & Sukhatme, 2004; Smith, Egerstedt, & Howard, 2009; Tanner, Pappas, & Kumar, 2004), and other multi-robot systems (Ma, Hönig, Cohen, et al., 2017). For example, in the near future, autonomous aircraft-towing vehicles will tow aircraft all the way from the runways to their gates (and vice versa), thereby reducing pollution,

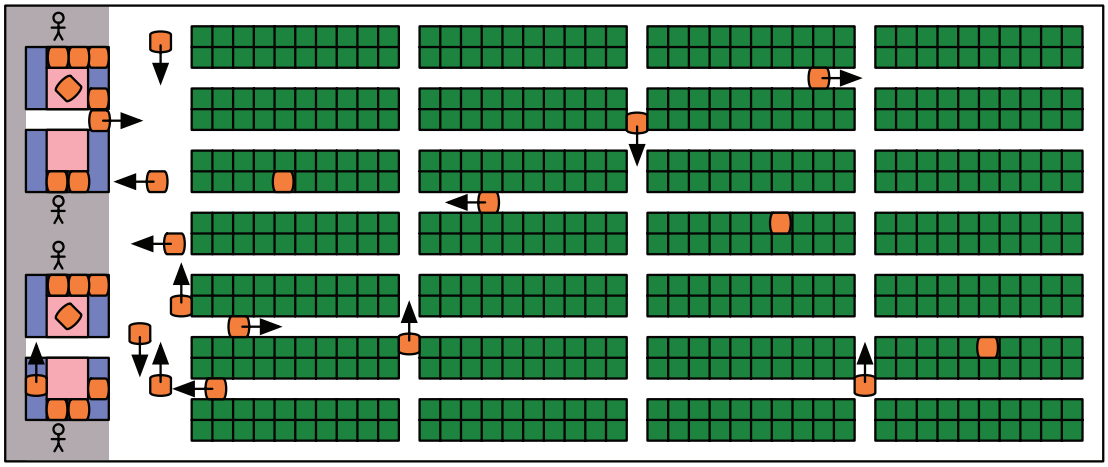


Figure 1.1: The typical layout of part of an Amazon Robotics automated warehouse system, reproduced from Wurman, D'Andrea, and Mountz (2008).

energy consumption, congestion, and human workload (Morris et al., 2016). Today, hundreds of warehouse robots already navigate autonomously in Amazon Robotics automated warehouse systems (formally called Amazon fulfillment centers) to move inventory pods all the way from their storage locations to the inventory stations that need the products they store (and vice versa) (Wurman et al., 2008).

Figure 1.1 shows the typical grid layout of part of an Amazon Robotics automated warehouse system with inventory stations on the left side and storage locations in the storage area to the right of the inventory stations. Each inventory station has an entrance (purple cells) and an exit (pink cells). Each storage location (green cell) can store one inventory pod. Each inventory pod consists of a stack of trays, each of which holds bins with products. Each warehouse robot (orange square) is capable of picking up, carrying, and putting down one inventory pod at a time. The warehouse robots need to move inventory pods all the way from their storage locations to the inventory stations that need the products they store (to ship them to customers) or from inventory stations to their assigned storage locations. After a warehouse robot enters an inventory station, the requested product is removed from its inventory pod by a worker. Once the warehouse robots have delivered all requested products for one shipment to the same inventory station, the worker prepares the shipment to the customer.

The coordination of autonomous operations of teams of agents is a fundamental building block for the above multi-agent systems but often requires a large search space since it involves the following three components: (1) coordination of target-assignment operations, such as assigning inventory pods to warehouse robots that are not carrying inventory pods or assigning empty storage locations to warehouse robots that have picked up inventory pods from the inventory stations in an Amazon Robotics automated warehouse system, (2) coordination of path-planning operations, such as planning paths for warehouse robots to pick up inventory pods assigned to them or to deliver inventory pods to the destinations, while avoiding that they collide with each other, in an Amazon Robotics automated warehouse system, and (3) repeating these target-assignment and path-planning operations if long-term coordination is required, such as repeatedly assigning inventory pods to warehouse robots and plan paths for them to deliver the inventory pods when they finish delivering their current inventory pods in an Amazon Robotics automated warehouse system. Making good decisions for these components requires solving the combined target-assignment and path-planning problem.

Many recent approaches in the artificial intelligence (AI) community have concentrated on studying a one-shot multi-agent path-planning problem called Multi-Agent Path Finding (MAPF) (Ma & Koenig, 2017; Ma, Koenig, et al., 2016). The problem of MAPF is to find collision-free paths for a given set of agents from their given start locations to their given (preassigned) targets in a given environment. While MAPF techniques can potentially be generalized to both the one-shot and the long-term combined target-assignment and path-planning problems for these applications of multi-agent systems, three central questions remain: Question 1: How hard is it to jointly assign targets to and plan paths for teams of agents? Question 2: How and how well can one jointly assign targets to and plan paths for teams of agents? Question 3: How do teams of agents execute the computed solutions? This dissertation addresses these three central questions by evaluating the following hypothesis: **Formalizing and studying new variants of MAPF can result in new theoretical insights into or new algorithms for the one-shot and the long-term combined target-assignment and path-planning problems for teams of agents, which can benefit real-world applications of multi-agent systems.**

To validate this hypothesis, we present four contributions: First, we introduce a unified NP-hardness proof structure that can be used to derive computational complexity results for different target-assignment and path-planning problems, stemming from formalizing and studying a new variant of MAPF, called *Package-Exchange Robot Routing (PERR)*, and its generalizations. Second, we formalize and study a new variant of MAPF, called *Combined Target Assignment and Path Finding (TAPF)*, that models the one-shot combined target-assignment and path-planning problem, and present complete and optimal algorithms for solving it. Third, we formalize and study a new variant of MAPF, called *Multi-Agent Pickup and Delivery (MAPD)*, that models the long-term combined target-assignment and path-planning problem, and present MAPD algorithms that utilize environmental characteristics to guarantee long-term robustness. Fourth, we demonstrate how MAPD algorithms can take kinematic constraints of real-world agents into account to compute plan-execution schedules that allow for the safe execution of the computed solutions, thus showcasing the benefits of these algorithms for real-world applications of multi-agent systems.

1.1 Definitions, Categorizations, and Assumptions

We now define the technical terms, categorize the problems and algorithms, and state the assumptions of this dissertation.

One-shot and Long-Term Coordination The coordination of target-assignment and path-planning operations can be either **one-shot** or **long-term**. In this dissertation, for one-shot coordination, we formalize and study TAPF, where each agent needs to get assigned a target, plan a path to the target without collisions with other agents, and stay at the target. For long-term coordination, we formalize and study MAPD, where each agent needs to get assigned a task that consists of two targets, namely the pickup location and the delivery location, move first to the pickup location and then to the delivery location of the task without collisions with other agents, and continue to attend to new tasks afterward.

Teams Agents in the combined target-assignment and path-planning problem are partitioned into **teams**. Agents in the same team have the same capability and are allowed to exchange their assigned targets or tasks, while agents in different teams are not allowed to do so.

Target-Assignment and Path-Planning Operations The combined target-assignment and path-planning problem involves some or all of the following three components:

- **Component 1:** Determining targets that the agents each needs to visit (**one-shot target-assignment sub-problem**).
- **Component 2:** Planning paths for the agents from their current locations to their targets in a way such that the agents do not collide with each other (**one-shot path-planning sub-problem**).
- **Component 3 (for long-term coordination):** Repeatedly solving the above one-shot target-assignment and path-planning sub-problems as the agents need to execute new tasks after finishing their current ones, which requires a mechanism to coordinate the interplay of Components 1 and 2.

Many recent approaches in the AI community have concentrated on tackling only Component 2 by studying MAPF. The problem of MAPF is to find collision-free paths for a given set of agents from their given start locations to their given targets in a given known environment (modeled as a graph). The quality of a solution is often measured by either the **makespan** (the maximum of the arrival times of all agents at their targets) or the **flowtime** (the sum of the arrival times of all agents at their targets). Many MAPF algorithms can utilize **environmental characteristics** (special properties of the graph that models the given environment) to guarantee that they are complete for some classes of MAPF problem instances (Cáp, Vokrínek, & Kleiner, 2015; Turpin, Mohta, Michael, & Kumar, 2014; Wang & Botea, 2011; Yu, 2017). The one-shot and the long-term target-assignment problems have been studied as various multi-robot task-allocation problems (see the taxonomy by Gerkey and Matarić (2004)) in the robotics community. The one-shot target-assignment problem corresponds to ST-SR-IA (Single-Task Robots, Single-Robot

Tasks, Instantaneous Assignment). The long-term target-assignment problem corresponds to ST-SR-TA (Single-Task Robots, Single-Robot Tasks, Time-Extended Assignment). The one-shot combined target-assignment and path-planning problem TAPF studied in this dissertation addresses Components 1 and 2 jointly. It assumes that all targets to be assigned are known a priori. It is thus **offline**. For TAPF, we focus on **complete** TAPF algorithms, which always return a solution if one exists, and **optimal** TAPF algorithms, which are complete and always return a solution with the smallest makespan. The long-term combined target-assignment and path-planning problem MAPD studied in this dissertation addresses Components 1-3 jointly. It assumes that not all tasks are known a priori and new tasks can appear as time goes by. It is thus **online**. It is NP-hard to solve optimally even if all tasks are known a priori (Brucker, 2010), and there does not exist any online algorithm that computes optimal solutions (Azar, Naor, & Rom, 1995; Kalyanasundaram & Pruhs, 1993). Therefore, for MAPD, we focus on **long-term robustness**, an analogy to completeness for one-shot problems: A long-term robust MAPD algorithm always returns a solution, if one exists, that finishes a finite set of tasks in a finite amount of time. Intuitively, a long-term robust MAPD algorithm avoids deadlocks, where all agents stay idle (because they block each other) and cannot execute the remaining tasks, and livelocks, where the agents constantly move but do not makes progress toward finishing their tasks. We refer to both deadlocks and livelocks as **deadlocks** throughout this dissertation.

Kinematic Constraints For the definitions of target-assignment and path-planning problems, we assume a given undirected graph whose vertices model the locations in the environment, point **agents** that each occupies a vertex at a discrete time step, uniform edge lengths (unweighted edges), and synchronized agent movements. For the execution of a given solution of these target-assignment and path-planning problems, we consider essential kinodynamic constraints (Kavraki & LaValle, 2016), namely first-order dynamic constraints (temporal constraints, such as velocity limits) and kinematic constraints (geometric constraints, such as agent sizes and placement of obstacles in the environment), of **real-world agents**, for example, mobile robots. We do not distinguish first-order dynamic constraints and kinematic constraints and refer to them

as **kinematic constraints** throughout this dissertation. We ignore other kinodynamic constraints and assume that they are handled by the controllers of real-world agents.

1.2 Central Questions

State-of-the-art MAPF algorithms can compute collision-free paths for hundreds of agents in minutes (Ma et al., 2018a). However, while MAPF algorithms are potentially applicable to solving both the one-shot and the long-term combined target-assignment and path-planning problems, three central questions remain:

- **Question 1:** *How hard is it to jointly assign targets to and plan paths for teams of agents?*

  There has not been any comprehensive study yet on the complexity of solving either the one-shot or the long-term combined target-assignment and path-planning problem. Some results are known for related problems: Finding an optimal solution to the one-shot multi-agent path-planning problem MAPF is NP-hard (Goldreich, 2011; Ratner & Warmuth, 1986; Surynek, 2010; Yu & LaValle, 2013c). For the special case of one team of agents, the one-shot combined target-assignment and path-planning problem can be solved optimally in polynomial time (Yu & LaValle, 2013a). However, none of these existing results applies to either the one-shot or the long-term combined target-assignment and path-planning problem in general.

- **Question 2:** *How and how well can one jointly assign targets to and plan paths for teams of agents?*

  Solving the combined target-assignment and path-planning problem requires solving the target-assignment and the path-planning sub-problems. MAPF algorithms solve the one-shot path-planning sub-problem (Felner et al., 2017; Ma & Koenig, 2017; Ma, Koenig, et al., 2016; Stern et al., 2019). But they assume that the assignment of targets to agents is fixed and do not consider different target assignments. Target-assignment algorithms solve either the one-shot (Bertsekas, 1992; Garfinkel, 1971; Gross, 1959; Kuhn, 1955; Shapley

& Shubik, 1971) or the long-term (Azar et al., 1995; Kalyanasundaram & Pruhs, 1993; Khuller, Mitchell, & Vazirani, 1994) target-assignment sub-problem. But they do not consider actual paths of agents when assigning targets, and the resulting target assignment is not optimal for the combined target-assignment and path-planning problem. Algorithms that solve the target-assignment and the path-planning sub-problems jointly can potentially determine solutions of higher quality than a trivial combination of individual solutions for all sub-problems (Srivastava et al., 2014; Turpin, Mohta, et al., 2014). However, the following issues need to be addressed to generalize MAPF algorithms to solving the target-assignment and the path-planning sub-problems jointly:

- How and how well do the good theoretical properties of MAPF algorithms carry over to algorithms that solve the combined target-assignment and path-planning problem? For example, an optimal MAPF solution may not be optimal for the combined problem.
- How and how well can such algorithms exploit the combinatorial structure of the combined target-assignment and path-planning problem? For example, it is not known how much better/worse or faster/slower such algorithms can solve the target-assignment and the path-planning sub-problems jointly than trivially combining individual solutions for the sub-problems.

Specifically, for the one-shot combined problem, it remains unclear how one can generalize target-assignment and MAPF algorithms there. Moreover, while, for the special case of one team of agents, the one-shot combined problem can be solved with a max-flow algorithm (Yu & LaValle, 2013a), it remains unclear how to generalize this algorithm to the general case of multiple teams of agents. For the long-term combined problem, it remains unclear whether and how the completeness of a MAPF algorithm can be generalized to the long-term robustness of an algorithm for the long-term combined problem. Specifically, while some MAPF algorithms can utilize environmental characteristics to guarantee completeness for some classes of MAPF problem instances (Cáp et al., 2015;

Turpin, Mohta, et al., 2014; Wang & Botea, 2011; Yu, 2017), it remains unclear whether and how algorithms for the long-term combined problem can guarantee long-term robustness, for example, by also utilizing environmental characteristics. To summarize, while existing target-assignment algorithms and path-planning algorithms provide some promising directions, none of them is directly applicable to solving the one-shot or the long-term combined target-assignment and path-planning problem.

- **Question 3:** *How do teams of agents execute the computed solutions?*

  There has not been any study of how to take kinematic constraints of teams of real-world agents into account to let them safely execute solutions for the combined target-assignment and path-planning problem. Existing MAPF algorithms can use the polynomial-time procedure MAPF-POST, developed in our recent research (Hönig, Kumar, Cohen, et al., 2016) (not covered as a contribution in this dissertation), in a post-processing step to transform a MAPF solution into a plan-execution schedule that takes kinematic constraints of real-world agents into account, which allows the agents to execute the MAPF solution safely. It remains unclear how the agents can execute the computed solutions to the combined target-assignment and path-planning problem and whether MAPF-POST can also be used to transform the computed solutions into plan-execution schedules for the combined target-assignment and path-planning problem.

1.3 Hypothesis and Contributions

The hypothesis of this dissertation is the following:

> *Formalizing and studying new variants of Multi-Agent Path Finding (MAPF) can result in new theoretical insights into or new algorithms for the one-shot and the long-term combined target-assignment and path-planning problems for teams of agents, which can benefit real-world applications of multi-agent systems.*

This dissertation makes four contributions to validate the above hypothesis, which leverages insights and tools from (1) operations research and theoretical computer science to characterize the computational complexity and capture the combinatorial structure of the new variants of MAPF that model the one-shot and the long-term combined target-assignment and path-planning problems, (2) AI to develop new algorithms that solve these new variants of MAPF by exploiting their combinatorial structure, and (3) robotics to take kinematic constraints of real-world agents into account and guarantee long-term robustness for the long-term combined target-assignment and path-planning problem by utilizing environmental characteristics.

- **Contribution 1:** To validate the hypothesis that formalizing and studying new variants of MAPF can result in new theoretical insights into the one-shot and the long-term combined target-assignment and path-planning problems, we introduce a unified NP-hardness proof structure that can easily be used to derive computational complexity results for these and many other target-assignment and path-planning problems. This NP-hardness proof structure stems from formalizing and studying a new variant of MAPF, called PERR.

- **Contribution 2:** To validate the hypothesis that formalizing and studying new variants of MAPF can result in new algorithms for the one-shot combined target-assignment and path-planning problem, which can benefit real-world applications of multi-agent systems, we formalize and study a new variant of MAPF, called TAPF, that models both the target-assignment and the path-planning sub-problems in a one-shot combined problem formulation. We present complete and optimal TAPF algorithms. These TAPF algorithms can use MAPF-POST to transform their solutions into plan-execution schedules that allow for the safe execution of their solutions on real-world agents.

- **Contribution 3:** To validate the hypothesis that formalizing and studying new variants of MAPF can result in new algorithms for the long-term combined target-assignment and path-planning problem, we formalize and study a new variant of MAPF, called MAPD, that models both the target-assignment and the path-planning sub-problems in a long-term

combined problem formulation. We present MAPD algorithms that utilize environmental characteristics to guarantee long-term robustness.

- **Contribution 4:** To validate the hypothesis that MAPD algorithms can benefit real-world applications of multi-agent systems, we present two methods to take kinematic constraints of real-world agents into account. First, we present an adapted MAPD algorithm that takes kinematic constraints of real-world agents into account during planning and still guarantees its long-term robustness. Second, as a baseline method, our MAPD algorithms can use MAPF-POST to transform their solutions into plan-execution schedules that take kinematic constraints of real-world agents into account. We conduct a case study of MAPD with kinematic constraints in a simulated automated warehouse system, where we demonstrate the benefits of our methods using both an agent simulator and a standard robot simulator.

Table 1.1 provides a summary of the problem formulations and contributions of this dissertation. In the following, we describe the contributions in detail.

1.3.1 Contribution 1: Theoretical Analysis of Target Assignment and Path Planning

Many theoretical results are known for the one-shot path-planning problem MAPF. The solvability of a MAPF problem instance can be determined and a MAPF solution, if there exists one, can be found in polynomial time (Kornhauser, Miller, & Spirakis, 1984; Röger & Helmert, 2012). However, it is NP-hard to solve MAPF optimally (Goldreich, 2011; Ratner & Warmuth, 1986; Surynek, 2010; Yu & LaValle, 2013c). Many theoretical results are also known for both the one-shot and the long-term target-assignment problems. On one hand, special cases of the one-shot target-assignment problem, including the classic assignment problem (Bertsekas, 1992; Kuhn, 1955; Shapley & Shubik, 1971) and the bottleneck assignment problem (Fulkerson, Glicksberg, & Gross, 1953; Garfinkel, 1971; Gross, 1959), can be solved optimally in polynomial time. On the other hand, the long-term target-assignment problem is NP-hard to solve optimally

Table 1.1: Summary of problem formulations and contributions.

	MAPF	PERR	TAPF	MAPD
novelty of formulation	not novel	novel [**Contribution 1**]	novel [**Contribution 2**]	novel [**Contribution 3**]
coordination type	one-shot	one-shot	one-shot	long-term
operation type	path-planning	path-planning	target-assignment and path-planning	target-assignment and path-planning
known complexity results	NP-hard	-	polynomial-time solvable for one team of agents, NP-hard for single-agent teams	-
new complexity results [**Contribution 1**]	fixed-parameter inapproximable (for makespan minimization)	fixed-parameter inapproximable (for makespan minimization), NP-hard (for flowtime minimization)	fixed-parameter inapproximable for more than one team of agents	fixed-parameter inapproximable (for makespan minimization), NP-hard (for service time minimization)
example algorithms	ILP, CBS	ILP, Adapted CBS	ILP, CBM [**Contribution 2**]	TP, TPTS, CENTRAL [**Contribution 3**], TP-SIPPwRT [**Contribution 4**]
properties of algorithms	complete, optimal	complete, optimal	complete, optimal [**Contribution 2**]	long-term robust for well-formed problem instances [**Contributions 3&4**]
how to include kinematic constraints?	post-process via MAPF-POST	-	post-process via MAPF-POST	post-process via MAPF-POST, include directly via TP-SIPPwRT [**Contribution 4**]

in general (Brucker, 2010). It is also known that the special case of the one-shot combined target-assignment and path-planning problem for one team of agents can be solved optimally in polynomial time (Yu & LaValle, 2013a). However, none of these existing results are applicable to the general case of either the one-shot or the long-term combined target-assignment and path-planning problem for multiple teams of agents.

Therefore, we study the relationships between different target-assignment and path-planning problems and introduce a unified NP-hardness proof structure for these problems, which stems from formalizing and studying a new variant of MAPF, called PERR. In PERR, each agent starts in a given start location and carries a package that needs to be delivered to a given goal location (preassigned target). Packages can be reassigned to agents in a proactive way—two agents in adjacent locations can exchange their packages, thereby exchanging their targets as well. We prove that PERR is NP-hard to approximate within any constant factor less than $4/3$ for makespan minimization and to solve optimally for flowtime minimization by using our unified NP-hardness proof structure to establish a reduction from an NP-complete version of the Boolean satisfiability problem called $\leq$3,=3-SAT (Tovey, 1984) to PERR. Studying PERR also lays the theoretical foundation for studying other target-assignment and path-planning problems, including MAPF, TAPF, and MAPD. For example, we demonstrate how our unified NP-hardness proof structure can be directly applied to proving the NP-hardness of optimally solving MAPF and MAPD for different objectives with a reduction from the same Boolean satisfiability problem $\leq$3,=3-SAT. We demonstrate how to derive similar NP-hardness results for TAPF by using our unified NP-hardness proof structure to establish a reduction from a newly constructed NP-complete version of the Boolean satisfiability problem called $2/\overline{2}/3$-SAT to TAPF. Notably, we provide the first fixed-parameter inapproximability results for both MAPF and TAPF and the first NP-hardness results for MAPD, thereby improving the state of the art.

1.3.2 Contribution 2: One-Shot Combined Target Assignment and Path Planning

We consider the one-shot combined target-assignment and path-planning problem, where teams of agents must assign targets among themselves and plan collision-free paths to their assigned

targets. We formalize and study TAPF, that models this one-shot combined target-assignment and path-planning problem for multiple teams of agents. In TAPF, each team is given the same number of targets as there are agents in the team. The problem of TAPF is to assign the targets of each team to agents in the same team and plan collision-free paths for the agents to their targets so that each agent reaches exactly one target and each target is reached by an agent. Existing research has considered only two extremes of this one-shot combined target-assignment and path-planning problem. On one hand, the special case of TAPF with one team of agents can be solved optimally in polynomial time (Yu & LaValle, 2013a) using a max-flow algorithm. On the other hand, MAPF algorithms (Felner et al., 2017; Ma & Koenig, 2017; Ma, Koenig, et al., 2016; Stern et al., 2019) assume that each agent forms a single-agent team. It remains unclear how and how well one can solve the general case of TAPF with multiple teams of agents.

Therefore, we present a complete and optimal TAPF algorithm called Conflict-Based Min-Cost Flow (CBM). CBM breaks TAPF down to the NP-hard sub-problem of coordinating different teams of agents and the polynomial-time solvable sub-problems of coordinating the agents in every team. It then tackles these sub-problems by using a combination of a combinatorial search algorithm called Conflict-Based Search (Sharon, Stern, Felner, & Sturtevant, 2015) for the NP-hard sub-problem and a min-cost max-flow algorithm (Goldberg & Tarjan, 1987) for the polynomial-time solvable sub-problems. We also present an Integer Linear Programming (ILP) based TAPF algorithm that solves an ILP encoding resulting from a reduction of TAPF to the integer multi-commodity flow problem. Experimentally, we demonstrate that CBM can compute solutions faster than the ILP-based algorithm, indicating better exploitation of the combinatorial structure of TAPF. We also demonstrate that CBM can compute solutions for more than 400 agents in minutes of runtime, showcasing its potential for large-scale multi-agent systems. Our TAPF algorithms can use the existing polynomial-time procedure MAPF-POST in a post-processing step to transform their solutions into plan-execution schedules that take kinematic constraints of teams of real-world agents into account and thus allow for the safe execution of their solutions on these agents. These results showcase the benefits of our TAPF algorithms

for the one-shot coordination of target-assignment and path-planning operations for real-world applications of multi-agent systems.

1.3.3 Contribution 3: Long-Term Target Assignment and Path Planning

We consider the long-term combined target-assignment and path-planning problem, where teams of agents must assign tasks among themselves, plan collision-free paths to the targets of their assigned tasks, and repeatedly attend to new tasks after finishing their current tasks. We formalize and study MAPD, that models this long-term combined target-assignment and path-planning problem. In MAPD, agents have to attend to a stream of tasks that each consists of two targets, the pickup location and the delivery location. The system changes dynamically as each task is added to system at an unknown time. An agent that is currently not executing any task can be assigned an unexecuted task. To execute the task, the agent has to first move from its current location to the pickup location of the task and then from there to the delivery location of the task, while avoiding collisions with other agents. Existing research has considered the one-shot target-assignment (Bertsekas, 1992; Garfinkel, 1971; Gross, 1959; Kuhn, 1955; Shapley & Shubik, 1971), the one-shot path-planning (Felner et al., 2017; Ma & Koenig, 2017; Ma, Koenig, et al., 2016; Stern et al., 2019), and the long-term target-assignment (Azar et al., 1995; Kalyanasundaram & Pruhs, 1993; Khuller et al., 1994) problems. However, it remains unclear how and how well one can generalize the existing results to the long-term combined target-assignment and path-planning problem MAPD and how one can guarantee long-term robustness for MAPD.

Therefore, we demonstrate how MAPD algorithms can utilize environmental characteristics to guarantee long-term robustness for *well-formed* MAPD problem instances, a class of MAPD problem instances that are realistic for many real-world applications of multi-agent systems. We design MAPD algorithms that allow agents to rest (that is, stay for a long period) only in locations where they cannot block other agents, thereby avoiding deadlocks. This is inspired by warehouse robots that are only allowed to charge batteries or pick up and drop off inventory pods in locations where they cannot block other robots. Specifically, we present two decoupled MAPD algorithms, TP and TPTS, that assign a task to and plan a path for one agent at a time and

one centralized MAPD algorithm, CENTRAL, that assigns tasks to and plans paths for multiple agents at a time. These MAPD algorithms break MAPD down to a sequence of one-shot target-assignment and one-shot path-planning sub-problems in chronological order and then apply one-shot target-assignment and one-shot path-planning algorithms to solve them. We prove that these MAPD algorithms can solve the one-shot target-assignment and one-shot path-planning sub-problems without backtracking in time, while avoiding deadlocks, for all well-formed MAPD problem instances. They are thus long-term robust for well-formed MAPD problem instances. We compare them experimentally with up to 500 agents and 1,000 tasks, thus showcasing their potential for the long-term coordination of target-assignment and path-planning operations for large-scale multi-agent systems.

1.3.4 Contribution 4: Long-Term Target Assignment and Path Planning with Kinematic Constraints

We consider generating plan-execution schedules for MAPD that take kinematic constraints of real-world agents into account. Realistic simulations of multi-agent systems with kinematic constraints provide important test beds for demonstrating the practicability of multi-agent coordination algorithms for their real-world applications. However, existing target-assignment and path-planning problems do not model kinematic constraints. For example, they assume discrete agent movements with uniform velocity and ignore the sizes and the velocities of real-world agents. The resulting solutions may be unsafe for real-world agents to execute. MAPF and TAPF algorithms can use the existing polynomial-time procedure MAPF-POST in a post-processing step to take kinematic constraints into account, thus transforming their solutions into plan-execution schedules that can be safely executed by real-world agents. However, it remains unclear whether and, if so, how MAPD algorithms can take kinematic constraints into account to compute such plan-execution schedules.

Therefore, we conduct a case study of MAPD with kinematic constraints of real-world agents in a simulated automated warehouse system. We demonstrate how MAPD algorithms can be made to produce kinematically feasible plan-execution schedules, showcasing their generality

for real-world applications of multi-agent systems. In particular, we present two methods: First, we adapt one of our MAPD algorithms, TP, using a novel one-shot single-agent path-planning algorithm SIPPwRT that computes paths with continuous agent movements with given velocities. The resulting algorithm TP-SIPPwRT directly takes kinematic constraints of real-world agents into account during planning, computes plan-execution schedules for the agents, provides guaranteed user-specified safety distances between the agents, and remains long-term robust for well-formed MAPD problem instances. Second, as a baseline method, MAPD algorithms can use the existing polynomial-time procedure MAPF-POST to transform their solutions into plan-execution schedules in a post-processing step. We compare the two methods and demonstrate their benefits for real-world applications of multi-agent systems using both an agent simulator and a standard robot simulator. For example, we demonstrate that TP-SIPPwRT can compute solutions for 250 agents and 2,000 tasks in seconds of runtime. These results showcase the benefits of our MAPD algorithms for the long-term coordination of target-assignment and path-planning operations for real-world applications of multi-agent systems.

1.4 Dissertation Outline

The outline of this dissertation is as follows: In Chapter 2, we provide formal definitions of the problems studied in this dissertation. In Chapter 3, we provide an in-depth overview of different target-assignment and path-planning problems. In Chapter 4, we introduce a unified NP-hardness proof structure that lays the theoretical foundations for problems covered in this dissertation. In Chapter 5, we study TAPF, that models the one-shot coordination of target-assignment and path-planning operations. In Chapter 6, we study MAPD, that models the long-term coordination of target-assignment and path-planning operations. In Chapter 7, we study MAPD with kinematic constraints of real-world agents. In Chapter 8, we conclude the dissertation by summarizing our contributions and providing an outline of possible future work.

Chapter 2

Problem Definitions

In this chapter, we formalize different target-assignment and path-planning problems on a graph whose vertices model the locations in the environment and whose edges model the connections between locations. We formally define MAPF in Section 2.1 and Anonymous MAPF in Section 2.2. We then formally define three novel target-assignment and path-planning problems, TAPF in Section 2.3, PERR in Section 2.4, and MAPD in Section 2.5, that we study in the later chapters of this dissertation. We formalize these problems as variants of MAPF, discuss their relationships to MAPF and between themselves, and give examples of their problem instances. Finally, we conclude the chapter in Section 2.6.

2.1 Problem Definition of MAPF

We now formalize the problem of Multi-Agent Path Finding (MAPF). A MAPF problem instance consists of:

- A given finite connected undirected graph $G = (V, E)$, whose vertices V correspond to locations and whose edges E correspond to connections between locations that the agents can move along.

- A given set of M agents $\{a_i | i \in [M]\}$.[1] Each agent a_i has a *start vertex* $s_i \in V$ and a *goal vertex* $g_i \in V$ (that represents the preassigned target). All start vertices are pairwise different. All goal vertices are also pairwise different.

At each discrete time step, each agent a_i either moves to an adjacent vertex or waits at the same vertex. Let $\pi_i(t)$ denote the vertex occupied by agent a_i at time step $t = 0, \ldots, \infty$.

Definition 2.1. A *path* $\pi_i = \langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i), \pi_i(T_i + 1), \ldots \rangle$ for agent a_i satisfies the following conditions:

1. The agent starts at its start vertex, that is, $\pi_i(0) = s_i$.

2. The agent ends at its goal vertex at the *arrival time* T_i, which is the minimal time step T_i such that, for all time steps $t = T_i, \ldots, \infty$, $\pi_i(t) = g_i$.

3. The agent always either moves to an adjacent vertex or does not move between two consecutive time steps, that is, for all time steps $t = 0, \ldots, \infty$, $(\pi_i(t), \pi_i(t + 1)) \in E$ or $\pi_i(t + 1) = \pi_i(t)$.

In this dissertation, path π_i is written in its short form $\langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i) \rangle$ (agent a_i continues to occupy the last vertex of its path after the arrival time).

Definition 2.2. A *vertex collision* is a tuple $\langle a_i, a_j, v, t \rangle$, where agents a_i and a_j occupy the same vertex $v = \pi_i(t) = \pi_j(t)$ at time step t. An *edge collision* is a tuple $\langle a_i, a_j, u, v, t \rangle$, where agents a_i and a_j traverse the same edge (u, v), where $u = \pi_i(t) = \pi_j(t+1)$ and $v = \pi_j(t) = \pi_i(t+1)$, in opposite directions between time steps t and $t + 1$.

Definition 2.3. A MAPF *plan* consists of a path π_i assigned to each agent a_i. A MAPF *solution* is a MAPF plan whose paths are collision-free.

Definition 2.4. The *makespan* $\max_{i \in [M]} T_i$ of a MAPF plan is the maximum of the arrival times of all agents at their goal vertices.

[1] We let $[M]$ denote the positive integer set $\{1, \ldots, M\}$.

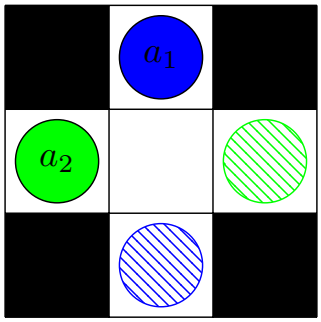


Figure 2.1: Example of a MAPF problem instance.

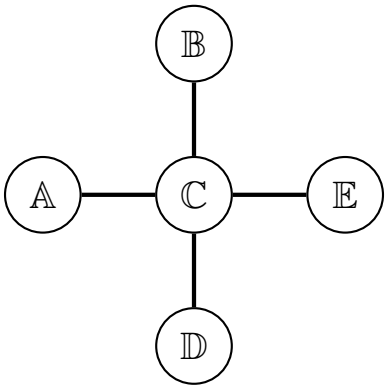


Figure 2.2: Graph representation of the MAPF problem instance shown in Figure 2.1.

The problem of MAPF is to find a solution with the smallest makespan. Some existing research on MAPF also aims to minimize the flowtime (Felner et al., 2017).

Definition 2.5. The *flowtime* $\sum_{i \in [M]} T_i$ of a MAPF plan is the sum of the arrival times of all agents at their goal vertices.

MAPF is NP-hard to solve optimally for both makespan minimization (Surynek, 2010) and flowtime minimization (Yu & LaValle, 2013c). The optimal makespan and optimal flowtime of any MAPF problem instance are both bounded by $O(|V|^3)$ (Yu & Rus, 2015).

2.1.1 MAPF Example

Figure 2.1 shows an example of a MAPF problem instance on a 2D 4-neighbor grid. White cells are traversable, and black cells are blocked. Colored circles are agents in their given start cells.

A hatched circle is placed in the given goal cell of each agent in the same color as the agent. Figure 2.2 shows the corresponding graph representation of the 2D 4-neighbor grid. Agent a_1 with $s_1 = \mathbb{B}$ and $g_1 = \mathbb{D}$ and agent a_2 with $s_2 = \mathbb{A}$ and $g_2 = \mathbb{E}$ are given. An optimal solution is $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle\}$ with makespan 3.

2.2 Problem Definition of Anonymous MAPF

We now formalize the problem of Anonymous MAPF. An Anonymous MAPF problem instance consists of:

- A given finite connected undirected graph $G = (V, E)$, whose vertices V correspond to locations and whose edges E correspond to connections between locations that the agents can move along.
- A given set of M agents $\{a_i | i \in [M]\}$ and M *targets* $g_1, \ldots, g_M \in V$. Each agent a_i has a *start vertex* $s_i \in V$. All start vertices are pairwise different. All targets are also pairwise different.

Definition 2.6. An *assignment* of targets to agents is a one-to-one mapping σ, determined by a permutation of $[M]$, that maps each agent a_i to a target $g_{i'} = \sigma(a_i)$.

An agent can be assigned any one of the targets as long as every target is assigned to and reached by an agent eventually. Therefore, the agents are interchangeable and thus "anonymous".

At each discrete time step, each agent a_i either moves to an adjacent vertex or waits at the same vertex. Let $\pi_i(t)$ denote the vertex occupied by agent a_i at time step $t = 0, \ldots, \infty$.

Definition 2.7. A *path* $\pi_i = \langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i), \pi_i(T_i + 1), \ldots \rangle$ for agent a_i satisfies the following conditions:

1. The agent starts at its start vertex, that is, $\pi_i(0) = s_i$.

2. The agent ends at its assigned target at the *arrival time* T_i, which is the minimal time step T_i such that, for all time steps $t = T_i, \ldots, \infty$, $\pi_i(t) = g_{i'} = \sigma(a_i)$.

3. The agent always either moves to an adjacent vertex or does not move between two consecutive time steps, that is, for all time steps $t = 0, \ldots, \infty$, $(\pi_i(t), \pi_i(t+1)) \in E$ or $\pi_i(t+1) = \pi_i(t)$.

In this dissertation, path π_i is written in its short form $\langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i) \rangle$ (agent a_i continues to occupy the last vertex of its path after the arrival time).

Definition 2.8. A *vertex collision* is a tuple $\langle a_i, a_j, v, t \rangle$, where agents a_i and a_j occupy the same vertex $v = \pi_i(t) = \pi_j(t)$ at the same time step t. An *edge collision* is a tuple $\langle a_i, a_j, u, v, t \rangle$, where agents a_i and a_j traverse the same edge (u, v), where $u = \pi_i(t) = \pi_j(t+1)$ and $v = \pi_j(t) = \pi_i(t+1)$, in opposite directions between time steps t and $t+1$.

Definition 2.9. An Anonymous MAPF *plan* consists of an assignment σ of targets to agents and a path π_i assigned to each agent a_i. An Anonymous MAPF *solution* is an Anonymous MAPF plan whose paths are collision-free.

The assignment σ of an Anonymous MAPF plan is often not written explicitly since it can be inferred from the paths of the plan.

Definition 2.10. The *makespan* $\max_{i \in [M]} T_i$ of an Anonymous MAPF plan is the maximum of the arrival times of all agents at their assigned targets.

The problem of Anonymous MAPF is to find a solution with the smallest makespan. It can be solved optimally in polynomial time (Yu & LaValle, 2013a). All Anonymous MAPF problem instances are solvable, and there always exists a solution with makespan no larger than $M + |V| - 1$ for any Anonymous MAPF problem instance (Yu & LaValle, 2013a). The optimal makespan of any Anonymous MAPF problem instance is thus bounded from above by $M + |V| - 1$.

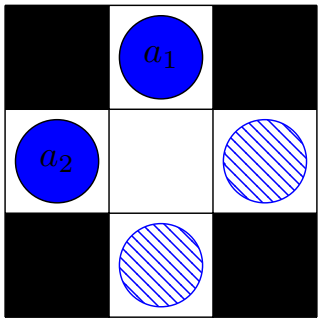


Figure 2.3: Example of an Anonymous MAPF problem instance.

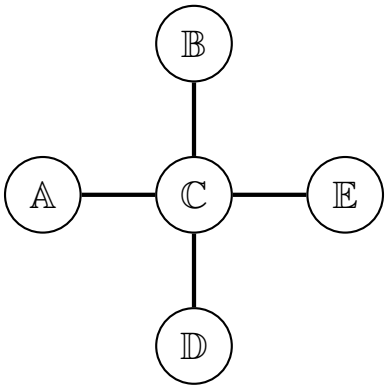


Figure 2.4: Graph representation of the Anonymous MAPF problem instance shown in Figure 2.3.

2.2.1 Anonymous MAPF Example

Figure 2.3 shows an example of an Anonymous MAPF problem instance on a 2D 4-neighbor grid. White cells are traversable, and black cells are blocked. Blue circles are agents in their given start cells. A blue hatched circle is placed in each target cell. Figure 2.4 shows the corresponding graph representation of the 2D 4-neighbor grid. Agent a_1 with $s_1 = \mathbb{B}$ and agent a_2 with $s_2 = \mathbb{A}$ are given. Two targets $g_1 = \mathbb{D}$ and $g_2 = \mathbb{E}$ are given. An optimal solution is $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{E} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{D} \rangle\}$ (where $\sigma(a_1) = g_2$ and $\sigma(a_2) = g_1$) with makespan 3.

2.3 Problem Definition of TAPF

We now formalize the problem of Target Assignment and Path Finding (TAPF). A TAPF problem instance consists of:

1. A given finite connected undirected graph $G = (V, E)$, whose vertices V correspond to locations and whose edges E correspond to connections between locations that the agents can move along.

2. A given set of M agents that are partitioned into K disjoint *teams* $team_1, \ldots, team_K$. Each team $team_k$ consists of M_k agents $a_1^k, \ldots, a_{M_k}^k$ and is given M_k *targets* $g_1^k, \ldots, g_{M_k}^k$, where $M = \sum_{k=1}^{K} M_k$. Each agent a_j^k has a unique *start vertex* s_j^k. All start vertices are pairwise different. All targets are also pairwise different.

Definition 2.11. An *assignment* of targets of team $team_k$ to agents in the same team is a one-to-one mapping σ^k, determined by a permutation of $[M_k]$, that maps each agent a_j^k in team $team_k$ to a target $g_{j'}^k = \sigma^k(a_j^k)$ of the same team.

An agent in a team can be assigned any one of the targets of the same team as long as every target of the team is assigned to and reached by an agent in the team eventually. An agent in a team cannot be assigned a target of a different team.

At each discrete time step, each agent a_j^k either moves to an adjacent vertex or waits at the same vertex. Let $\pi_j^k(t)$ denote the vertex occupied by agent a_j^k at time step $t = 0, \ldots, \infty$.

Definition 2.12. A *path* $\pi_j^k = \langle \pi_j^k(0), \pi_j^k(1), \ldots, \pi_j^k(T_j^k), \pi_j^k(T_j^k + 1), \ldots \rangle$ for agent a_j^k satisfies the following conditions:

1. The agent starts at its start vertex, that is, $\pi_j^k(0) = s_j^k$.

2. The agent ends at its assigned target at the *arrival time* T_j^k, which is the minimal time T_j^k such that, for all time steps $t = T_j^k, \ldots, \infty$, $\pi_j^k(t) = g_{j'}^k = \sigma^k(a_j^k)$.

3. The agent always either moves to an adjacent vertex or does not move between two consecutive time steps, that is, for all time steps $t = 0, \ldots, \infty$, $(\pi_j^k(t), \pi_j^k(t+1)) \in E$ or $\pi_j^k(t+1) = \pi_j^k(t)$.

In this dissertation, path π_j^k is written in its short form $\langle \pi_j^k(0), \pi_j^k(1), \ldots, \pi_j^k(T_j^k) \rangle$ (agent a_j^k continues to occupy the last vertex of its path after the arrival time).

Definition 2.13. A *vertex collision* between an agent a_j^k in team $team_k$ and a different agent $a_{j'}^{k'}$ in team $team_{k'}$ is a tuple $\langle team_k, team_{k'}, v, t \rangle$, where agents a_j^k and $a_{j'}^{k'}$ occupy the same vertex $v = \pi_j^k(t) = \pi_{j'}^{k'}(t)$ at the same time step t. An *edge collision* between an agent a_j^k in team $team_k$ and a different agent $a_{j'}^{k'}$ in team $team_{k'}$ is a tuple $\langle team_k, team_{k'}, u, v, t \rangle$, where agents a_j^k and $a_{j'}^{k'}$ traverse the same edge (u, v), where $u = \pi_j^k(t) = \pi_{j'}^{k'}(t+1)$ and $v = \pi_{j'}^{k'}(t) = \pi_j^k(t+1)$, in opposite directions between time steps t and $t+1$. A collision can occur between two agents in the same team (in this case, $k = k'$) and between two agents in different teams (in this case, $k \neq k'$).

Definition 2.14. A TAPF *plan* consists of an assignment σ^k of targets to agents for each team $team_k$ and a path π_j^k assigned to each agent a_j^k in each team $team_k$. A TAPF *solution* is a TAPF plan whose paths are collision-free.

The assignments σ^k, for all $k \in K$, of a TAPF plan are often not written explicitly since they can be inferred from the paths of the plan.

Definition 2.15. Given paths for all agents in team $team_k$, the *team cost* $\max_{j \in [M_k]} T_j^k$ of team $team_k$ is the maximum of the arrival times of all agents in the team at their targets.

Definition 2.16. The *makespan* $\max_{k \in [K], j \in [M_k]} T_j^k$ of a TAPF plan is the maximum of the arrival times of all agents at their assigned targets.

The problem of TAPF is to find a solution with the smallest makespan.

2.3.1 TAPF as a Generalization of MAPF

TAPF generalizes both Anonymous MAPF and (Non-Anonymous) MAPF:

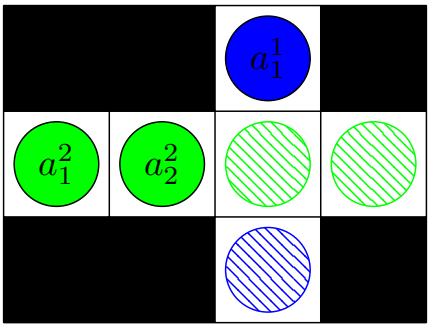


Figure 2.5: Example of a TAPF problem instance.

- **Anonymous MAPF** [$K = 1$] (also called Permutation-Invariant (Kloder & Hutchinson, 2006) or Unlabeled MAPF (Solovey & Halperin, 2016)) results from TAPF if only one team exists that consists of all M agents. It is called "anonymous" because a target can be assigned to any agent, and the agents are thus interchangeable.

- **(Non-Anonymous) MAPF** [$K = M$] (often just called MAPF) results from TAPF if every team consists of exactly one agent and the number of teams is thus equal to the number of agents. It is called "non-anonymous" because a target can be assigned to only one specific agent (meaning that the assignments of targets to agents are pre-determined), and the agents are thus non-interchangeable.

If we compare the problem definition of TAPF to that of MAPF, then we observe that a MAPF problem instance can be obtained from a TAPF problem instance by fixing the assignment of targets to agents and setting the goal vertices of the agents corresponding to their assigned targets. Therefore, TAPF is a generalization of MAPF that allows any assignment of targets of a team to the agents in the same team. Any solution to a TAPF problem instance is thus also a solution to its corresponding MAPF problem instance for a suitable assignment of targets to agents. Since the makespan of any optimal MAPF solution is bounded from above by $O(|V|^3)$ (Yu & Rus, 2015), the makespan of any optimal TAPF solution is also bounded from above by $O(|V|^3)$.

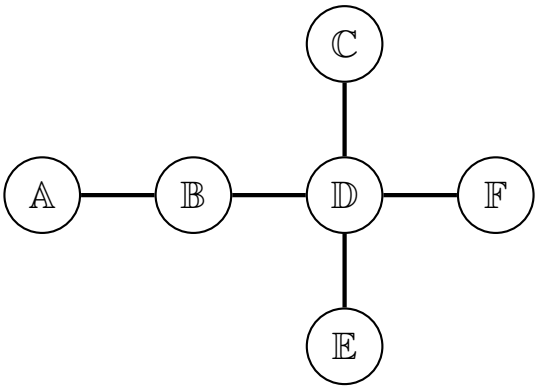


Figure 2.6: Graph representation of the TAPF problem instance shown in Figure 2.5.

2.3.2 TAPF Example

Figure 2.5 shows an example of a TAPF problem instance on a 2D 4-neighbor grid. White cells are traversable, and black cells are blocked. Colored circles are agents in their given start cells. Each color represents a team. A hatched circle is placed in a target cell in the color of its team. Figure 2.6 shows the corresponding graph representation of the 2D 4-neighbor grid. Two teams $team_1$ (blue) and $team_2$ (green) are given. Team $team_1$ is given target $g_1^1 = \mathbb{E}$ and consists of agent a_1^1 with start vertex $\mathbb{C}$. Team $team_2$ is given targets $g_1^2 = \mathbb{D}$ and $g_2^2 = \mathbb{F}$ and consists of agents a_1^2 and a_2^2 with start vertices $\mathbb{A}$ and $\mathbb{B}$, respectively. Both teams have paths of team cost 2 that are collision-free among agents in the same team: $\{\pi_1^1 = \langle \mathbb{C}, \mathbb{D}, \mathbb{E} \rangle\}$ and $\{\pi_1^2 = \langle \mathbb{A}, \mathbb{B}, \mathbb{D} \rangle, \pi_2^2 = \langle \mathbb{B}, \mathbb{D}, \mathbb{F} \rangle\}$, respectively. However, there is a vertex collision $\langle team_1, team_2, \mathbb{D}, 1 \rangle$ between agent a_1^1 in team $team_1$ and agent a_2^2 in team $team_2$ for these paths. An optimal solution is $\{\pi_1^1 = \langle \mathbb{C}, \mathbb{D}, \mathbb{E} \rangle, \pi_1^2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{B}, \mathbb{D} \rangle, \pi_2^2 = \langle \mathbb{B}, \mathbb{B}, \mathbb{D}, \mathbb{F} \rangle\}$ (where $\sigma^1(a_1^1) = g_1^1$, $\sigma^2(a_1^2) = g_1^2$, and $\sigma^2(a_2^2) = g_2^2$) with makespan 3.

2.4 Problem Definition of PERR

We now formalize the problem of Package-Exchange Robot Routing (PERR). A PERR problem instance consists of:

1. A given finite connected undirected graph $G = (V, E)$, whose vertices V correspond to locations and whose edges E correspond to connections between locations that the agents can move along.

2. A given set of M packages $\{p_i | i \in [M]\}$ and a given set of M agents. Each package p_i has a *start vertex* s_i and a *goal vertex* g_i (that represents the preassigned target) and is carried by an agent initially. All start vertices are pairwise different. All goal vertices are also pairwise different.

At each discrete time step, each package p_i, carried by an agent, either moves to an adjacent vertex or waits at its current vertex. Let $\pi_i(t)$ denote the vertex occupied by the agent carrying package p_i at time step $t = 0, \ldots, \infty$.

Definition 2.17. A *path* $\pi_i = \langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i), \pi_i(T_i + 1), \ldots \rangle$ for package p_i satisfies the following conditions:

1. The package starts at its start vertex, that is, $\pi_i(0) = s_i$.

2. The package ends at its goal vertex at the *arrival time* T_i, which is the minimal time step T_i such that, for all time steps $t = T_i, \ldots, \infty$, $\pi_i(t) = g_i$.

3. The package (carried by an agent) always either moves to an adjacent vertex or does not move, that is, for all time steps $t = 0, \ldots, \infty$, $(\pi_i(t), \pi_i(t+1)) \in E$ or $\pi_i(t+1) = \pi_i(t)$.

In this dissertation, path π_i is written in its short form $\langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i) \rangle$ (package p_i continues to be at the last vertex of its path after the arrival time).

Definition 2.18. A *vertex collision* is a tuple $\langle p_i, p_j, v, t \rangle$, where the agent carrying package p_i and the agent carrying package p_j occupy the same vertex $v = \pi_i(t) = \pi_j(t)$ at the same time step t.

Definition 2.19. A PERR *plan* consists of a path π_i assigned to each package p_i. A PERR *solution* is a PERR plan whose paths are collision-free.

Definition 2.20. The *makespan* $\max_{i \in [M]} T_i$ of a PERR plan is the maximum of the arrival times of all packages at their goal vertices.

The problem of PERR is to find a solution with the smallest makespan.

Our unified NP-hardness proof structure presented in Chapter 4 can be used to derive computational complexity results for both makespan minimization and flowtime minimization.

Definition 2.21. The *flowtime* $\sum_{i \in [M]} T_i$ of a PERR plan is the sum of the arrival times of all packages at their goal vertices.

2.4.1 PERR as a Relaxation of MAPF

In PERR, two packages p_i and p_j can be exchanged in a single time step by the agents carrying them when they are at adjacent vertices. If this happens, then p_i and p_j traverse the same edge in opposite directions, resulting in $\pi_i(t) = \pi_j(t+1)$ and $\pi_j(t) = \pi_i(t+1)$.

Definition 2.22. For packages p_i with $\pi_i(t) = u$ and p_j with $\pi_j(t) = v$ where $(u, v) \in E$, an *exchange operation* moves package p_i from vertex u to vertex v and package p_j from vertex v to vertex u between time steps t and $t+1$.

If we compare the problem definition of PERR to that of MAPF, PERR is almost identical to MAPF except that MAPF does not allow exchange operations and treats them as edge collisions (Definition 2.2). If we view the packages that are moved by agents (from their given start vertices to their given goal vertices) as packages that move by themselves (just like the agents in MAPF that move from their given start vertices to their given goal vertices), then PERR is a relaxation of MAPF that permits exchange operations (and thus omits the edge collisions of MAPF). Therefore, there is a correspondence between a MAPF problem instance on a graph $G = (V, E)$ that has agents $\{a_i | i \in [M]\}$, with start vertex $s_i \in V$ and goal vertex $g_i \in V$ for each agent a_i, and a PERR problem instance on the same graph $G = (V, E)$ that has packages $\{p_i | i \in [M]\}$, with start vertex $s_i \in V$ and goal vertex $g_i \in V$ for each package p_i.

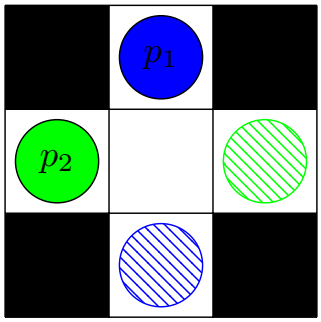


Figure 2.7: Example of a PERR problem instance.

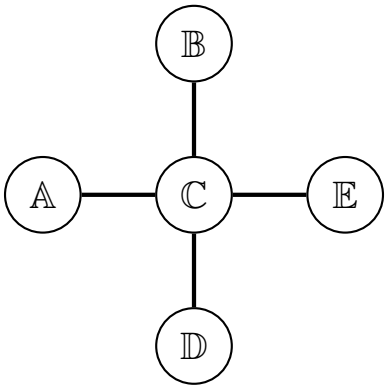


Figure 2.8: Graph representation of the PERR problem instance shown in Figure 2.7.

2.4.2 PERR Example

Figure 2.7 shows an example of a PERR problem instance on a 2D 4-neighbor grid. It corresponds to the MAPF problem instance shown in Figure 2.1. White cells are traversable, and black cells are blocked. Colored circles are packages, each carried by an agent, in their given start cells. A hatched circle is placed in the given goal cell of each package in the same color as the package. Figure 2.8 shows the corresponding graph representation of the 2D 4-neighbor grid. Package p_1 with $s_1 = \mathbb{B}$ and $g_1 = \mathbb{D}$ and package p_2 with $s_2 = \mathbb{A}$ and $g_1 = \mathbb{E}$ are given. An optimal solution is $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle\}$ with makespan 3.

2.4.3 K-PERR

We also formalize a generalization of PERR, namely K-Type Package-Exchange Robot Routing (K-PERR), where the packages and goal vertices are partitioned into K types. If there are M_k packages of type k ($k \in [K]$), then there must also be M_k goal vertices of type k. Packages of the same type are interchangeable: Each package of type k must be delivered to a different goal vertex of type k. Each goal vertex of type k must receive a package of type k. The problem of K-PERR is to minimize the makepsan, namely the maximum of the arrival times of all packages at their goal vertices. PERR is thus a special case of K-PERR with $K = M$ by definition.

We recall that, in TAPF, agents are partitioned into teams and agents in the same team are interchangeable. K-PERR is thus a relaxation of TAPF with K teams that permits exchange operations (and thus omits the edge collisions of TAPF) in the same sense that PERR is a relaxation of MAPF. Without loss of generality, we assume that packages and goal vertices in K-PERR with the same type are given contiguous indices, that is, we can group them into K types such that package p_i and goal vertex g_i are of (a unique) type k where $i \in [1 - M_k + \sum_1^k M_k, \sum_1^k M_k]$. Therefore, there is a correspondence between a TAPF problem instance on a graph $G = (V, E)$ that has K teams of agents $\{a_j^k | k \in [K], j \in [M_k]\}$, with start vertex $s_j^k \in V$ for each agent a_j^k, and targets $g_1^k, \ldots g_{M_k}^k \in V$ for each $team_k$, and a K-PERR problem instance on the same graph $G = (V, E)$ that has K types of packages $\{p_i | i \in [M]\}$, with start vertex $s_i = s_j^k \in V$ and goal vertex $g_i = g_j^k \in V$ of type k for each package p_i of type k, where $i = j - M_k + \sum_1^k M_k$.

2.4.4 1-PERR

1-PERR is a special case of K-PERR where only $K = 1$ type of packages exists. We know from above that 1-PERR is a relaxation of Anonymous MAPF (TAPF with one team) that permits exchange operations (and thus omits the edge collisions of Anonymous MAPF). There is a correspondence between an Anonymous MAPF problem instance on a graph $G = (V, E)$ that has agents $\{a_i | i \in [M]\}$, with start vertex $s_i \in V$ and goal vertex $g_i \in V$ for each agent a_i,

and a 1-PERR problem instance on the same graph $G = (V, E)$ that has packages $\{p_i | i \in [M]\}$, with start vertex $s_i \in V$ and goal vertex $g_i \in V$ for each package p_i.

2.5 Problem Definition of MAPD

We now formalize the problem of Multi-Agent Pickup and Delivery (MAPD). A MAPD problem instance consists of:

1. A given finite connected undirected graph $G = (V, E)$, whose vertices V correspond to locations and whose edges E correspond to connections between locations that the agents can move along.

2. A given set of M agents $\{a_i | i \in [M]\}$. Each agent has an *initial vertex*. All initial vertices are pairwise different.

3. A task set $\mathcal{T}$ that contains the set of unexecuted tasks in the system. The task set changes dynamically as, at each time step, new tasks can be added to the system. Each task $\tau_j \in \mathcal{T}$ is characterized by a *pickup vertex* $s_j \in V$ and a *delivery vertex* $g_j \in V$ and is added to the system at an unknown (finite) time step. A task is known and available for execution only from the time step on when it has been added to the system.

At each discrete time step, each agent a_i either moves to an adjacent vertex or waits at the same vertex. Let $\pi_i(t) \in V$ denote the vertex occupied by agent a_i at time step t. When the system starts (at time step 0), agent a_i starts at its given initial vertex $\pi_i(0)$.

For ease of exposition, we refer to both (1) an entire sequence of vertices traversed by an agent in a MAPD solution (when describing the long-term problem) and (2) a sequence of vertices for an agent to execute a task (when describing one-shot sub-problems) as a *path* for MAPD. We thus do not require a path to start at a predefined vertex and end at a predefined vertex.

Definition 2.23. A *path* $\pi_i = \langle \pi_i(0), \pi_i(1), \ldots, \pi_i(T_i), \pi_i(T_i + 1), \ldots \rangle$ for agent a_i satisfies the following condition: The agent always either moves to an adjacent vertex or does not move, that is, for all time steps $t = 0, \ldots, \infty$, $(\pi_i(t), \pi_i(t + 1)) \in E$ or $\pi_i(t + 1) = \pi_i(t)$.

Agents need to avoid collisions with each other:

Definition 2.24. A *vertex collision* is a tuple $\langle a_i, a_{i'}, v, t \rangle$ where agent a_i and agent $a_{i'}$ occupy the same vertex $v = \pi_i(t) = \pi_{i'}(t)$ at the same time step t. An *edge collision* is a tuple $\langle a_i, a_{i'}, u, v, t \rangle$ where agent a_i and agent $a_{i'}$ traverse the same edge (u, v), where $u = \pi_i(t) = \pi_{i'}(t+1)$ and $v = \pi_{i'}(t) = \pi_i(t+1)$, in opposite directions between time steps t and $t+1$.

We use the following definitions to model single-agent tasks that each can be assigned to one agent at a time and single-task agents that each can execute one task at a time (Gerkey & Matarić, 2004).

Definition 2.25. An agent is called *free* if and only if it is currently not executing any task. Otherwise, it is called *occupied*.

Definition 2.26. A task can be assigned to one agent at a time. A free agent can be assigned any task $\tau_j \in \mathcal{T}$. In order to execute task τ_j, it then has to move first from its current vertex to the pickup vertex s_j of the task and then from there to the delivery vertex g_j of the task. When the agent reaches the pickup vertex, it *starts* to execute the task and removes the task from $\mathcal{T}$. When it reaches the delivery vertex, it *finishes* executing the task, which implies that it becomes free again and is no longer assigned the task. Any free agent can be assigned any task in the task set. An agent can be assigned a different task in the task set while it is still moving to the pickup vertex of its currently assigned task but has to finish executing the task after it has reached the pickup vertex of the task before it can be assigned another task.

Definition 2.27. The *service time* is the average number of time steps needed to finish executing each task after it was added to the system.

Definition 2.28. The *makespan* is the earliest time step when all tasks are finished.

The problem of MAPD is to find collision-free paths for the agents to finish executing all tasks. The effectiveness of a MAPD algorithm is evaluated by the service time or makespan.

Definition 2.29. A MAPD algorithm *solves* a MAPD problem instance if and only if the resulting service time of all tasks is bounded (or, equivalently, the resulting makespan is bounded). A MAPD algorithm is *long-term robust* if and only if it solves all MAPD problem instances with finitely many tasks.

2.5.1 MAPD as a Long-Term Generalization of One-Shot Problems

In MAPD, the one-shot problem at any time step can be viewed as a variant of Anonymous MAPF, MAPF, and TAPF. The free agents are anonymous since they can be assigned any task in the task in the task set. Therefore, the one-shot problem for any given set of free agents, namely to assign (pickup vertices of) tasks in the task set to and find collision-free paths for the agents in the set to the pickup vertices of their assigned tasks, is similar to Anonymous MAPF. The difference is that the number of free agents in the set is not necessarily equal to the number of (pickup vertices of) tasks available for assignment. The occupied agents are non-anonymous since they cannot change their current targets (delivery vertices of their assigned tasks). Therefore, the one-shot problem for any given set of occupied agents, namely to find collision-free paths for the agents in the set to the delivery vertices of the tasks they are currently executing, is similar to MAPF. Similarly, the one-shot problem for any given set of free and occupied agents is similar to TAPF where all free agents in the set form one team and each occupied agent in the set forms a single-agent team.

2.5.2 MAPD Example

Figure 2.9 shows an example of a MAPD problem instance on a 2D 4-neighbor grid. Colored circles are agents. Dashed circles represent pickup and delivery vertices. Figure 2.2 shows the corresponding graph representation of the 2D 4-neighbor grid. Two free agents a_1 (in blue) with $\pi_1(0) = \mathbb{A}$ and a_2 (in green) with $\pi_2(0) = \mathbb{E}$ are given. There is only one task τ_1 with $s_1 = \mathbb{A}$ and $g_1 = \mathbb{E}$, which is added to the system at time step 0. A solution is to assign task τ_1 and path $\langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle$ to agent a_1 and path $\langle \mathbb{E}, \mathbb{C}, \mathbb{B} \rangle$ to agent a_2 with service time 3 and makespan

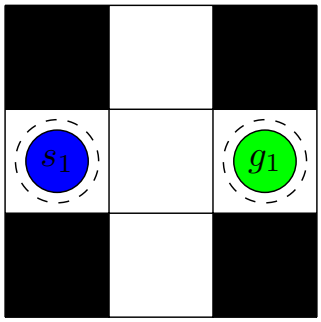


Figure 2.9: Example of a MAPD problem instance.

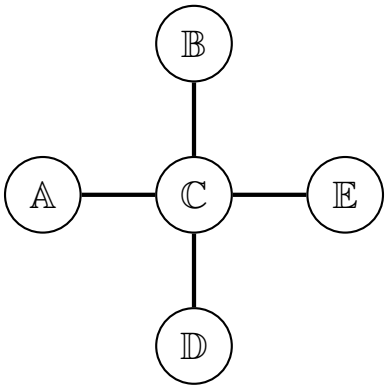


Figure 2.10: Graph representation of the MAPD problem instance shown in Figure 2.9

3. Specifically, agent a_1 becomes an occupied agent and starts to execute task τ_1 at time step 0 and finishes executing the task at time step 3.

2.6 Summary

In this chapter, we gave the formal definitions of MAPF, Anonymous MAPF, TAPF, PERR, and MAPD. Figure 2.11 summarizes the relationships between these problems. Table 2.1 summarizes the symbols used in the definitions.

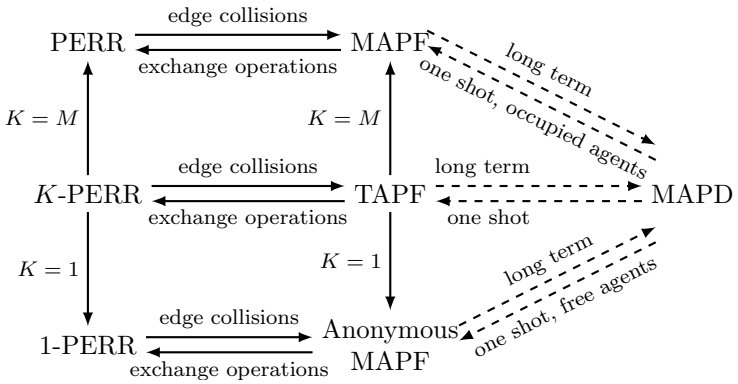


Figure 2.11: Relationships between different problems.

Table 2.1: Summary of symbols. The last column specifies if the symbols are used for specific problems.

symbol	description	problem
$G = (V, E)$	graph that models the environment	
$u, v \in V$	used for vertices	
$\mathbb{A}, \mathbb{B}, \ldots, \mathbb{F}$	specific vertices in examples	
a	used for agents	
p	used for packages	(K-)PERR
M	number of agents or packages	
K	number of teams of agents or types of packages	TAPF, K-PERR
τ	used for tasks	MAPD
$\mathcal{T}$	task set	MAPD
$s \in V$	used for start vertices	all except MAPD
	used for pickup vertices of tasks	MAPD
$g \in V$	used for goal vertices (preassigned targets)	MAPF, (K-)PERR
	used for targets to be assigned to agents	Anonymous MAPF, TAPF
	used for delivery vertices of tasks	MAPD
π	used for paths	
t	discrete time step	
T	used for arrival times	

Chapter 3

Target Assignment and Path Planning for Teams of Agents

In this chapter, we provide a review of literature on different target-assignment and path-planning problems. We first provide an overview of the one-shot path-planning problem MAPF as an essential element of coordinating target-assignment and path-planning operations of teams of agents in Section 3.1. We provide detailed descriptions of three MAPF algorithms in Section 3.2. We describe how MAPF solutions can be safely executed by real-world agents in Section 3.3. We then survey existing research on other one-shot and long-term target-assignment and path-planning problems in Section 3.4. Finally, we conclude the chapter in Section 3.5.

3.1 One-Shot Path-Planning Problem: MAPF

Many recent research efforts have concentrated on tackling the one-shot path-planning problem for multiple agents by studying the standard problem formulation MAPF (Ma & Koenig, 2017; Ma, Koenig, et al., 2016). MAPF has been well-studied by researchers from the AI, robotics, theoretical computer science, and operations research communities under different names, including Cooperative Path Finding (Silver, 2005), Pebble Motion on Graphs (Kornhauser et al., 1984), and Multi-Robot Path Planning (Yu & LaValle, 2016).

In the following, we first give a survey of theoretical results for MAPF in Section 3.1.1. We then provide an overview of MAPF algorithms in Section 3.1.2. Finally, we survey other one-shot path-planning problems that are modeled as extensions of MAPF in Section 3.1.3.

3.1.1 Theoretical Results for MAPF

Now, we provide an overview of existing theoretical results for MAPF. We categorize them into results for pebble motion on graphs with a time-irrelevant objective function (total number of edge traversals) and results for MAPF with time-relevant objective functions (makespan and flowtime).

3.1.1.1 15-Puzzle and Pebble Motion on Graphs with Time-Irrelevant Objectives

In the theoretical computer science and operations research communities, MAPF originates from the study of the 15-puzzle (Johnson & Story, 1879), that can be viewed as a special case of MAPF on a 4×4 2D 4-neighbor grid with 15 agents. The solvability of a 15-puzzle problem instance depends on the parity of the permutation. Pebble motion on graphs (Kornhauser et al., 1984) is a generalization of the 15-puzzle and can also be viewed as a special case of MAPF with at most $M = |V| - 1$ agents. In the problem of pebble motion on graphs, an agent can move from its current vertex only to an adjacent vertex that is not currently occupied by another agent. The solution quality is measured by the total number of edge traversals that move all agents from their start vertices to their goal vertices. There exists a complete $O(|V|^3)$-time algorithm that finds a solution of $O(|V|^3)$ edge traversals to any problem instance of pebble motion on graphs or decides that the problem instance is unsolvable (Kornhauser et al., 1984; Röger & Helmert, 2012). There even exists a linear-time algorithm that decides whether a problem instance of pebble motion on graphs is solvable (Goraly & Hassin, 2010). However, it is NP-hard to find a solution with the minimum total number of edge traversals to a problem instance of pebble motion on graphs (Goldreich, 2011; Ratner & Warmuth, 1986).

3.1.1.2 MAPF with Time-Relevant Objectives

In the AI community, MAPF attempts to model robots in real-world applications rather than pebbles on game boards and thus often uses time-relevant objective functions, such as the makespan or the flowtime, that assign costs to wait actions (staying at the same vertex for one time step) as in Definitions 2.1 and 2.4 in additional to the time-irrelevant objective function—the total number of edge traversals. The makespan and flowtime of any MAPF problem instance are bounded from above by $O(|V|^3)$, based on the result that there exists a complete $O(|V|^3)$-time algorithm that finds a solution of $O(|V|^3)$ edge traversal to any MAPF problem instance or decides that the MAPF problem instance is unsolvable (Yu & Rus, 2015). However, it is NP-hard to find a solution with the minimum makespan (Surynek, 2010) or the minimum flowtime (Yu & LaValle, 2013c) to a MAPF problem instance, even if the given graph is a planar graph (Yu, 2016) or a 2D 4-neighbor grid (Banfi, Basilico, & Amigoni, 2017). Any two of the objective functions, the total number of edge traversals, makespan, and flowtime, cannot be simultaneously minimized for MAPF (Yu & LaValle, 2013c).

3.1.2 MAPF Algorithms

Now, we provide an overview of existing MAPF algorithms. We categorize them into reduction-based, rule-based, and search-based algorithms, based on their methodologies. We highlight their properties in terms of completeness (complete for all MAPF problem instances, complete for MAPF problem instances on graphs with special properties, or incomplete) and optimality (optimal, bounded-suboptimal, or suboptimal with respect to different objectives). A survey and limited experimental evaluation of some of these algorithms can be found in Felner et al. (2017).

3.1.2.1 Reduction-Based Algorithms

Reduction-based algorithms reduce MAPF to other well-studied combinatorial problems, such as Boolean Satisfiability (Surynek, 2012), Integer Linear Programming (ILP) (Yu & LaValle,

2013b), and Answer Set Programming (Erdem, Kisa, Oztok, & Schueller, 2013). These algorithms often construct an explicit representation of the state space of a MAPF problem instance up to some value of the makespan (planning horizon) by using variables and posting constraints for the variables. They solve a decision problem for each value of the makespan and increase the value if no solution exists for the current one. They are complete for solving MAPF and naturally designed for minimizing the makespan. They can be modified to solve MAPF with other objectives optimally (Surynek, Felner, Stern, & Boyarski, 2016; Yu & LaValle, 2013b), bounded-suboptimally (within a user-provided suboptimality factor) (Surynek, Felner, Stern, & Boyarski, 2017), and suboptimally (Surynek, 2015). We describe an ILP-based MAPF algorithm in Section 3.2.3.

3.1.2.2 Rule-Based Algorithms

Rule-based algorithms solve MAPF using a set of primitive operations that specify the actions of the agents in different situations. They often guarantee completeness for a restricted class of MAPF problem instances. Rule-based algorithms are often very efficient by simply following the predefined primitive operations but provide no guarantee on the solution quality. Push and Swap (Luna & Bekris, 2011) and its extension (Sajid, Luna, & Bekris, 2012) provide no completeness guarantee. One of their descendants, Push and Rotate (de Wilde, ter Mors, & Witteveen, 2013), is complete for MAPF problem instances on graphs with at most $M = |V| - 2$ agents. TASS (Khorshid, Holte, & Sturtevant, 2011) is complete for MAPF problem instances on "solvable" trees based on prior work on solving multi-robot motion planning on trees (Masehian & Nejad, 2009). BIBOX (Surynek, 2009) is complete for MAPF problem instances on bi-connected graphs with at most $M = |V| - 2$ agents. Split and Group (Yu, 2017) is complete for MAPF problem instances on grid-like "well-connected" graphs, runs in polynomial time, and provides a constant-factor approximation guarantee for minimizing the makespan on such graphs.

Some algorithms combine both primitive operations and search. FAR (Wang & Botea, 2008) and MAPP (Wang & Botea, 2011) explore different ways of combining paths of individual agents. MAPP is complete for MAPF problem instances on "slidable" graphs (Wang & Botea,

2011). This research has resulted in Wang's dissertation (Wang, 2012). There is also a MAPF algorithm that uses a combination of A* searches on a graph abstraction, primitive operations, and reductions to constraint satisfaction problems (Ryan, 2008; Ryan, 2010).

3.1.2.3 Search-Based Algorithms

The main computational challenge of optimally solving MAPF with a search algorithm is that the number of possible states of a MAPF problem instance is exponential in the number of agents: Each (joint) state is the product of the vertices of all agents, and each state-transition operator is the product of the actions (moving to one of the adjacent vertices or waiting) of all agents. Search-based MAPF algorithms tackle this challenge by using different strategies to reduce the size of the exponential state space.

Decoupled Algorithms The search-based algorithms in the first category decouple MAPF completely into one-shot single-agent path-finding problems. These algorithms are often efficient but provide no optimality or even completeness guarantee. Many of them are based on Prioritized Planning (Bennewitz, Burgard, & Thrun, 2002; Erdmann & Lozano-Pérez, 1987) that plans a path for each agent at a time in a fixed order according to their predefined priorities. Cooperative A* and Hierarchical Cooperative A* (HCA*) (Silver, 2005) are prioritized planning algorithms that use a space-time A* search (see Section 3.2.1.1) to plan a path for each agent, one at a time, and avoid collisions of this agent with the agents whose paths they have planned earlier. Cooperative A* uses pre-computed heuristic values for the space-time A* searches. HCA* was initially designed for video games with a limited amount of memory and thus needs to compute the heuristic values for the space-time A* searches at runtime. We describe Cooperative A* in Section 3.2.1. One of its extensions, Windowed-HCA* (Silver, 2005), only considers the paths of other agents within a limited planning horizon (time window). Cooperative Partial-Refinement A* (Sturtevant & Buro, 2006) further improves upon HCA* by abstracting the state space of an agent and thus reducing the runtime needed to calculate the heuristic values. Conflict-Oriented

Windowed-HCA* (Bnaya & Felner, 2014) first ignores collisions and then dynamically places time windows around collisions to adjust the paths it has planned.

A*-Based Algorithms The search-based algorithms in the second category use an A* search to plan with joint states but try to reduce the size of the state space they need to explore. They are complete for all MAPF problem instances and can be used for either makespan minimization or flowtime minimization. Independence Detection (Standley, 2010) partitions the agents into independent groups where agents in different groups do not collide with each other and thus reduces the original MAPF problem into several MAPF sub-problems, one for each group. The size of its state space is thus exponential only in the cardinality of the largest group. Operator Decomposition (Standley, 2010) decomposes a state-transition operator into M (single-agent) actions and applies one action at a time, thus affording the A* search pruning opportunities for any two actions that lead to a collision. Enhanced Partial Expansion A* (Goldenberg et al., 2014) uses an "operator selection function" to select state-transition operators to avoid generating search nodes with f-values larger than the optimal solution cost. M* (Wagner & Choset, 2015) dynamically changes the branching factor and searches with joint states of a set of agents locally only if it has found a collision among them. M* is part of Wagner's dissertation (Wagner, 2015).

Hierarchical Algorithms The search-based algorithms in the third category decouple MAPF into one-shot single-agent path-planning problems on the low level and dynamically couple the resulting single-agent paths using a best-first tree search on the high level. They are complete for all MAPF problem instances. Increasing Cost Tree Search (Sharon, Stern, Goldenberg, & Felner, 2013) minimizes the flowtime. On the high level, it systematically considers each combination of the arrival times of all agents and branches on possible ways of increasing the flowtime by one if no collision-free paths exist for a combination of arrival times. On the low level, it determines whether collision-free paths exist for a combination of arrival times. Conflict-Based Search (CBS) (Sharon et al., 2015) minimizes either the makespan or the flowtime. We describe CBS in Section 3.2.2. CBS first finds individually optimal paths for all agents (ignoring collisions). On

the high level, it then systematically resolves each collision of the computed paths by imposing constraints on individual agents that forbid them from occupying a vertex or traversing an edge at a given time step. On the low level, it uses a space-time A* search to find a path for an agent that obeys its constraints. The high-level search of CBS branches on which collision to resolve. Increasing Cost Tree Search and CBS are part of Sharon's dissertation (Sharon, 2015).

3.1.3 MAPF Extensions and Related Problems

We now survey other one-shot path-planning problems that are modeled as extensions of MAPF.

3.1.3.1 MAPF with Deadlines

In the problem of MAPF with Deadlines (Ma et al., 2018a; 2018b), a deadline (time step) is given. Its objective is to maximize the number of successful agents, defined as the agents that can reach their given goal vertices from their given start vertices within the given deadline without colliding with each other. Its applications include robots that need to evacuate before a disaster and robots that need to finish tasks before a deadline.

3.1.3.2 MAPF with Delay Probabilities and Related Problems

MAPF with Delay Probabilities (MAPF-DP) (Ma, Kumar, & Koenig, 2017) generalizes MAPF to the case where the uncertainty of agent motion has to be considered during planning to ensure a collision-free execution of the plan. In MAPF-DP, the uncertainty of each agent is characterized by a given delay probability with which the agent stays in its current vertex whenever it intends to traverse an outgoing edge of its current vertex. The problem of MAPF-DP is to find a plan that consists of a path for each agent and a plan-execution policy that controls with GO or STOP commands how each agent proceeds along its path such that no collisions occur during plan execution. It is also studied as MAPF with Uncertainty (Wagner & Choset, 2017), where the paths are planned in the belief space of the agents and the execution of the resulting plan is not guaranteed to be collision-free. There is also research on a similar extension of MAPF that

enforces a certain number of time steps for which a vertex must be unoccupied after it has been occupied by an agent, which reduces the possibility of collisions during plan execution (Atzmon et al., 2018).

Other related problems use the Markov Decision Process (MDP) or Partially Observable Markov Decision Process (POMDP) framework to plan paths under uncertainty for multiple agents. These problems include POMDP planning for robot navigation (Kurniawati, Hsu, & Lee, 2008; Ma & Pineau, 2015), transition-independent decentralized MDPs (Becker, Zilberstein, Lesser, & Goldman, 2004; Goldman & Zilberstein, 2004), multi-agent MDPs (Boutilier, 1996), decentralized sparse-interaction MDPs (Melo & Veloso, 2011), transition-independent multi-agent MDPs (Scharpff, Roijers, Oliehoek, Spaan, & de Weerdt, 2016), and a framework for approximating multi-agent MDPs (Liu & Michael, 2016) for multi-agent path planning with motion dynamics.

3.1.3.3 MAPF for Large Agents

In the problem of MAPF for Large Agents (Li, Surynek, et al., 2019), an agent can occupy more than one vertex at one time step according to its given shape and volume. Two agents collide if both of them occupy some vertex at the same time step.

3.2 MAPF Algorithm Examples

Now, we describe one incomplete (and thus suboptimal) and two optimal MAPF algorithms introduced in Section 3.1.2, which inspire the design of some of the target-assignment and path-planning algorithms covered in the later chapters of this dissertation. The first one is Cooperative A* (Silver, 2005), a decoupled search-based algorithm based on Prioritized Planning (Bennewitz et al., 2002; Erdmann & Lozano-Pérez, 1987). The second one is Conflict-Based Search (CBS) (Sharon et al., 2015), a hierarchical search-based algorithm. The third one is an ILP-based MAPF algorithm (Yu & LaValle, 2013b), which reduces MAPF to integer multi-commodity flow problems that are then solved with an ILP formulation.

3.2.1 Cooperative A*

Prioritized Planning (Bennewitz et al., 2002; Erdmann & Lozano-Pérez, 1987) is a decoupled scheme for MAPF. It uses a predefined total order on the agents and reduces MAPF to a one-shot single-agent path-planning problem for each agent: Prioritized Planning plans for the highest priority agent first and compute its optimal path. Prioritized Planning then plans for lower and lower priority agents and computes, for each agent, its individually optimal path that avoids collisions with all higher priority agents, which are *dynamic obstacles* that follow their (already planned) paths. Decoupled MAPF algorithms are complete for all *well-formed* MAPF problem instances (Cáp et al., 2015).

Definition 3.1. A MAPF problem instance is *well-formed* if and only if:

1. The start and goal vertices of all agents, called *endpoints*, are different from each other except that the start and goal vertices of the same agent can be the same.

2. For any two endpoints, there exists a path between them that traverses no other endpoints.

We now describe a version of Cooperative A* that implements Prioritized Planning by using a *space-time A* search* (Silver, 2005) to solve the one-shot single-agent path-planning problem for each agent individually.

3.2.1.1 Space-Time A* Search

Space-time A* is a one-shot single-agent path-planning algorithm that is used in not only Cooperative A* but also other search-based MAPF algorithms. A space-time A* search is an A* search whose states are pairs of vertices and time steps. A directed edge exists from state $\langle u, t \rangle$ to state $\langle v, t+1 \rangle$ if and only if $u = v \in V$ or $(u, v) \in E$. A space-time A* search finds a *time-minimal path* (that is, with the minimum arrival time at its goal vertex) for some agent a_i that obeys a set of *constraints* to avoid collisions with other agents.

Algorithm 3.1: Cooperative A*

```
   Input: MAPF problem instance
1  Plan ← ∅;
2  for i ← 1 . . . M do
3      if Space-time A* search for a_i returns no path then
4          return "No Solution";
5      Add the returned path to Plan;
6  return Plan;
```

Definition 3.2. A *constraint* for agent a_i is either a *vertex constraint* $\langle a_i, v, t \rangle$, that prohibits agent a_i from occupying vertex v at timestep t, or an *edge constraint* $\langle a_i, u, v, t \rangle$, that prohibits agent a_i from moving from vertex u to vertex v between timesteps t and $t + 1$.

Therefore, state $\langle v, t \rangle$ is removed from the state space of agent a_i if and only if there is a vertex constraint $\langle a_i, v, t \rangle$. Similarly, the edge from state $\langle u, t \rangle$ to state $\langle v, t + 1 \rangle$ is removed from the state space of agent a_i if and only if there is an edge constraint $\langle a_i, u, v, t \rangle$.

3.2.1.2 Pseudocode

Algorithm 3.1 shows the pseudocode of Cooperative A*. Given a MAPF problem instance with graph $G = (V, E)$ and the set of agents $\{a_i | i \in [M]\}$, it plans paths for the agents in increasing order of their indices [Line 2]. It then uses a space-time A* search to find a path for each agent a_i that obeys the constraints imposed by the already planned paths π_j of all higher priority agents a_j [Lines 3-5]. Specifically, each such path π_j imposes the set of constraints $\{\langle a_i, \pi_j(0), 0 \rangle, \langle a_i, \pi_j(1), \pi_j(0), 0 \rangle, \langle a_i, \pi_j(1), 1 \rangle, \langle a_i, \pi_j(2), \pi_j(1), 1 \rangle, \ldots\}$ for agent a_i. Therefore, the space-time A* search finds a time-minimal path π_i for agent a_i that has no collisions with the paths π_j of all higher priority agents a_j. All planned paths are thus collision-free. Cooperative A* reports that no solution exists if the space-time A* search of any agent returns no path [Line 4]. Otherwise, it returns a MAPF plan that contains collision-free paths for all agents [Line 6].

More details on Cooperative A*, including an analysis of its properties, can be found in Silver (2005).

3.2.1.3 Example

We describe how Cooperative A* solves the MAPF problem instance shown in Figure 2.1. Cooperative A* first plans for agent a_1 and finds the time-minimal path $\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle$. It then plans for agent a_2 and finds the time-minimal path $\pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle$ that obeys all constraints imposed by path π_1.

3.2.2 Conflict-Based Search

Conflict-Based Search (CBS) (Sharon et al., 2015) is a two-level search-based MAPF algorithm. On the low level, CBS uses a space-time A* search (see Section 3.2.1.1) to compute a time-minimal path for each agent. On the high level, CBS performs a best-first tree search to resolve collisions in the computed paths. We now describe a version of CBS that is optimal for makespan minimization.

3.2.2.1 High-Level Search of CBS

CBS performs a best-first search on the high level to resolve collisions among the agents and build a constraint tree. Each node N contains a set of constraints (Definition 3.2) $N.constraints$, a plan $N.plan$ that obeys these constraints, and a cost $N.cost$ equal to the makespan of its plan. The list OPEN stores all generated but unexpanded nodes. Algorithm 3.2 shows the high-level search of CBS. CBS starts with the root node, that has an empty set of constraints [Line 1]. It performs a low-level space-time A* search to find an individually time-minimal path for each agent (without any constraints) independently. It terminates unsuccessfully if the low-level space-time A* search for any agent returns no path. Otherwise, the plan of the root node contains paths for all agents [Line 2-6]. The cost of the root node is the makespan of its plan [Line 7]. CBS inserts the root node into OPEN [Line 8]. If OPEN is empty, then CBS terminates unsuccessfully [Lines 9 and 24]. Otherwise, it expands a node N in OPEN with the smallest cost [Line 10] and removes the node from OPEN [Line 11]. Ties are broken in favor of the node whose plan has the smallest number of pairs of colliding agents, which speeds up the high-level search

Algorithm 3.2: High-Level Search of CBS

Input: MAPF problem instance

1 *Root.constraints* $\leftarrow \emptyset$;
2 *Root.plan* $\leftarrow \emptyset$;
3 **foreach** $i \in [M]$ **do**
4 **if** *LowLevel*(a_i, *Root*) returns no path **then**
5 **return** "No Solution";
6 Add the returned paths to *Root.plan*;
7 *Root.cost* $\leftarrow$ *Makespan*(*Root.plan*);
8 OPEN $\leftarrow \{Root\}$;
9 **while** OPEN $\neq \emptyset$ **do**
10 $N \leftarrow \arg\min_{N' \in \text{OPEN}} N'.cost$;
11 OPEN $\leftarrow$ OPEN $\setminus \{N\}$;
12 **if** $N.plan$ has no collision **then**
13 **return** $N.plan$;
14 *collision* $\leftarrow$ a vertex or edge collision $\langle a_i, a_j, \dots \rangle$ in $N.plan$;
15 **foreach** a_i involved in *collision* **do**
16 $N' \leftarrow$ new node;
17 $N'.plan \leftarrow N.plan$;
18 $N'.constraints \leftarrow N.constraints$;
19 $N'.constraints \leftarrow N'.constraints \cup \{\langle a_i, \dots \rangle\}$;
20 **if** *LowLevel*(a_i, N') returns a path **then**
21 Update $N'.plan$ with returned path;
22 $N'.cost \leftarrow$ *Makespan*($N'.plan$);
23 OPEN $\leftarrow$ OPEN $\cup \{N'\}$;
24 **return** "No Solution";

of CBS experimentally (Sharon et al., 2015). If the plan of node N has no collisions, then it is a goal node and CBS terminates successfully with this plan [Lines 12-13]. Otherwise, CBS finds a collision that it needs to resolve [Line 14]. CBS then generates two child nodes N_1 and N_2 of N [Lines 15-16]. Each child node inherits the plan and all constraints from node N [Lines 17-18]. If the collision to resolve is a vertex collision $\langle a_i, a_j, v, t \rangle$, then CBS adds the vertex constraint $\langle a_i, v, t \rangle$ to the constraints of N_1 and the vertex constraint $\langle a_j, v, t \rangle$ to the constraints of N_2 [Line 19]. If the collision to resolve is an edge collision $\langle a_i, a_j, u, v, t \rangle$, then CBS adds the edge constraint $\langle a_i, u, v, t \rangle$ to the constraints of N_1 and the edge constraint $\langle a_j, v, u, t \rangle$ to the constraints of N_2 [Line 19]. For node N_1 (respectively N_2), CBS performs a low-level

space-time A* search to find a time-minimal path for agent a_i (respectively a_j) that obeys all constraints in N_1.*constraints* (respectively N_2.*constraints*) relevant to agent a_i (respectively a_j). If the low-level space-time A* search successfully returns such a path, CBS replaces the old path of agent a_i (respectively a_j) in N_1.*plan* (respectively N_2.*plan*) with the returned one [Lines 20-21], updates the cost of N_1 (respectively N_2) accordingly [Line 22], and inserts N_1 (respectively N_2) into OPEN [Line 23]. Otherwise, it discards the node.

3.2.2.2 Low-Level Search of CBS

Similar to the space-time A* search for Cooperative A* (see Section 3.2.1.1), *LowLevel*(a_i, N) performs a space-time A* search to find a time-minimal path for agent a_i that obeys all constraints of node N. In addition, it breaks ties among all time-minimal paths in favor of one that has the fewest collisions with the paths of other agents in N.*plan*, which speeds up the high-level search of CBS experimentally (Sharon et al., 2015).

CBS uses an upper bound $\mathcal{U}$ on the makespan, such as the one given in Yu and Rus (2015), to guarantee completeness and optimality. The makespan of a plan is bounded from above by $\mathcal{U}$ if and only if the arrival times of all agents are bounded from above by $\mathcal{U}$. Therefore, the low-level space-time A* search of CBS for agent a_i also uses $\mathcal{U}$ as the upper bound on the arrival time at its goal vertex. Therefore, it terminates unsuccessfully and returns no path when it tries to expand a state (a pair of a vertex and a time step) whose time step is larger than $\mathcal{U}$. More details on CBS, including an analysis of its properties, can be found in Sharon et al. (2015).

3.2.2.3 Flowtime Objective

CBS can also be used for other objectives (Sharon et al., 2015). In particular, it returns a solution with the minimum flowtime if the flowtime of the plan of a node is assigned to the cost of the node on Lines 7 and 22.

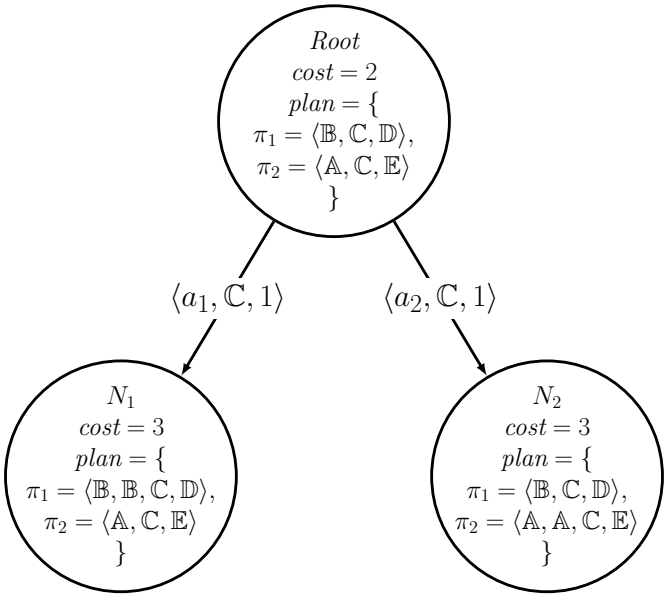


Figure 3.1: The constraint tree for the MAPF problem instance shown in Figure 2.1.

3.2.2.4 Example

Figure 3.1 shows the constraint tree constructed for the MAPF problem instance shown in Figure 2.1. CBS starts with the root node, that has an empty set of constraints. Its plan is a set of individually time-minimal paths $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle\}$. When CBS expands the root node and resolves the collision $\langle a_1, a_2, \mathbb{C}, 1 \rangle$, it generates two child nodes N_1 and N_2 with additional constraints $\langle a_1, \mathbb{C}, 1 \rangle$ and $\langle a_2, \mathbb{C}, 1 \rangle$, respectively. For N_1 (respectively N_2), *LowLevel*(a_1, N_1) (respectively *LowLevel*(a_2, N_2)) returns path $\pi_1 = \langle \mathbb{B}, \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle$ for a_1 (respectively $\pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle$ for a_2) that obeys the constraints of N_1 (respectively N_2). When CBS tries to expand either N_1 or N_2 and detects no collisions in the plan of the node, it returns the plan as solution.

3.2.2.5 CBS Variants

Many improvements to CBS have been proposed: Meta-Agent CBS (Sharon et al., 2015) dynamically groups multiple agents into a meta-agent on the high level and uses an A* search to plan paths for these agents with their joint states on the low level. ICBS (Boyarski et al., 2015)

always first resolves collisions that result in child nodes whose costs are larger than that of the current node, thus affording the high-level search of CBS pruning opportunities. CBSH (Felner et al., 2018) and its improvement (Li, Boyarski, Felner, Ma, & Koenig, 2019) use an admissible heuristic to improve the high-level best-first search of CBS. Recent search develops a CBS variant (Li, Harabor, Stuckey, Felner, et al., 2019) that expands each node in a way such that any solution is admitted by the subtree under only one but not both of its child nodes, thus reducing duplicate search effort of the high-level search of CBS. Other recent research develops a CBS variant (Li, Gange, et al., 2020; Li, Harabor, Stuckey, Ma, & Koenig, 2019) that adds multiple constraints to a child node at a time. Other CBS variants use different searches on the high level: ECBS (Barer, Sharon, Stern, & Felner, 2014) and its improvement (Cohen et al., 2016) perform a bounded-suboptimal search on the constraint tree. Some recent research (Cohen et al., 2018) develops an anytime version of the bounded-suboptimal search on the constraint tree. Other recent research (Ma, Harabor, Stuckey, Li, & Koenig, 2019) develops a greedy depth-first search on the constraint tree.

3.2.3 ILP-Based MAPF Algorithm

The ILP-based MAPF algorithm (Yu & LaValle, 2013b) first reduces MAPF to the integer multi-commodity flow problem on a time-expanded flow network (an idea that originated in the operations research literature (Aronson, 1989)) and then uses this reduction to solve MAPF optimally for makespan minimization.

3.2.3.1 Reducing MAPF to Multi-Commodity Flow

We now describe the reduction used by the ILP-based MAPF algorithm.

Given a MAPF problem instance on graph $G = (V, E)$ and a limit T on the number of time steps, we construct a T-step *time-expanded flow network* $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V} = \bigcup_{v \in V}(\{v_0^{out}\} \cup \bigcup_{t=1}^{T}\{v_t^{in}, v_t^{out}\})$ and directed edges $\mathcal{E}$ with unit capacity. Each vertex $v \in V$ is translated to a vertex $v_t^{out} \in \mathcal{V}$ for all $t = 0 \ldots T$ (which represents vertex v at the end of time step t) and a vertex $v_t^{in} \in \mathcal{V}$ for all $t = 1 \ldots T$ (which represents vertex v in the beginning of time

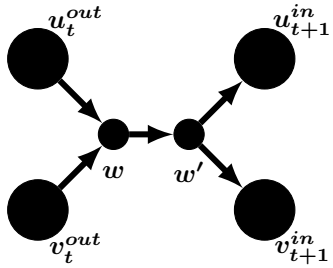


Figure 3.2: Example of the construction of the gadgets in $\mathcal{N}$ for edge $(u, v) \in E$ and time step t.

step t). For each agent a_i, we set a supply of one at (start) vertex $(s_i)_0^{out}$ and a demand of one at (goal) vertex $(g_i)_T^{out}$, both for commodity type i (corresponding to agent a_i). Each vertex $v \in V$ is also translated to an edge $(v_t^{out}, v_{t+1}^{in}) \in \mathcal{E}$ for all time steps $t = 0 \dots T-1$ (which represents an agent waiting at vertex v between time steps t and $t+1$). Each vertex $v \in V$ is translated to an edge $(v_t^{in}, v_t^{out}) \in \mathcal{E}$ for all time steps $t = 1 \dots T$ (which prevents vertex collisions of the form $\langle *, *, v, t \rangle$ among all agents since only one agent can occupy vertex v between time steps t and $t+1$). Each edge $(u, v) \in E$ is translated to a *gadget* of vertices in $\mathcal{V}$ and edges in $\mathcal{E}$ for all time steps $t = 0 \dots T-1$, which consists of two auxiliary vertices $w, w' \in \mathcal{V}$ that are unique to the gadget (but have no superscripts/subscripts here for ease of readability) and the edges $(u_t^{out}, w), (v_t^{out}, w), (w, w'), (w', u_{t+1}^{in}), (w, v_{t+1}^{in}) \in \mathcal{E}$. This gadget prevents edge collisions of the forms $\langle *, *, u, v, t \rangle$ and $\langle *, *, v, u, t \rangle$ among all agents since only one agent can move along the edge (u, v) in any direction between time steps t and $t+1$. Figure 3.2 shows an example of the construction of the gadgets. By construction, there is a correspondence between all feasible integer multi-commodity flows on the T-step time-expanded flow network of a number of units equal the number of agents and all solutions of the MAPF problem instance with makespans of at most T (Yu & LaValle, 2013b).

3.2.3.2 Example

Figure 3.3 shows a feasible integer multi-commodity flow in the 3-step time-expanded flow network reduced from the MAPF problem instance shown in Figure 2.1. Solid colored circles

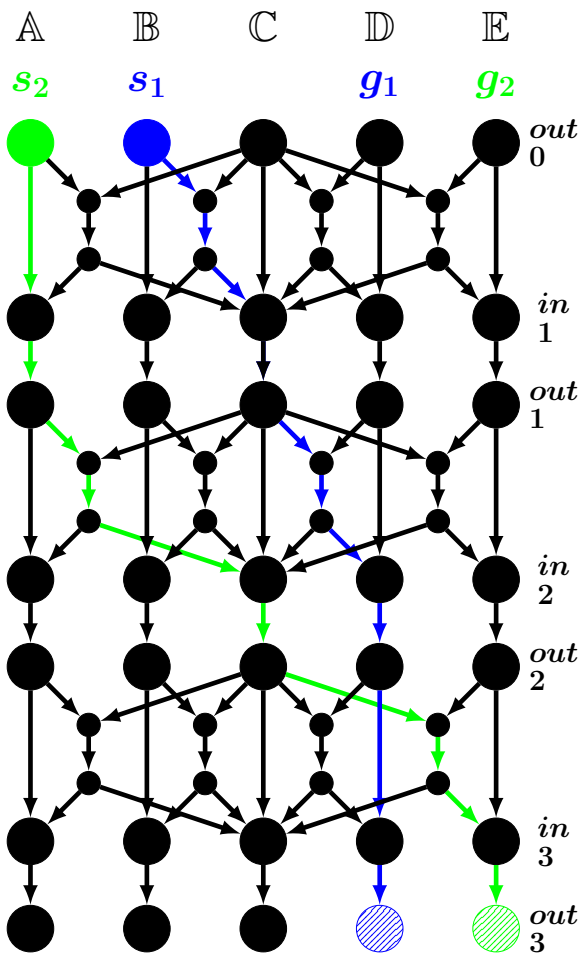


Figure 3.3: A feasible integer multi-commodity flow for the MAPF problem instance shown in Figure 2.1.

are (start) vertices with a supply. Hatched colored circles are (goal) vertices with a demand. The blue edges represent a unit flow of commodity type 1, corresponding to a path for agent a_1. The green edges represent a unit flow of commodity type 2, corresponding to a path for agent a_2. The feasible integer multi-commodity flow corresponds to the optimal solution $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle\}$.

3.2.3.3 Solving MAPF via ILP

The ILP-based MAPF algorithm then employs the reduction to solve MAPF optimally for makespan minimization via a standard ILP formulation. Let $\delta^+(v)$ (respectively $\delta^-(v)$) be the set of incoming (respectively outgoing) edges of v. Let $x_i[e]$ be the Boolean variable representing the amount of flow of commodity type i on edge e. An integer multi-commodity flow problem on the T-step time-expanded flow network can be written as an ILP using the following standard formulation.

$$0 \leq \sum_{i \in [M]} x_i[e] \leq 1 \qquad \forall e \in \mathcal{E}.$$

$$\sum_{e \in \delta^+(v)} x_i[e] - \sum_{e \in \delta^-(v)} x_i[e] = 0 \qquad \forall i \in [M], \; \forall v \in \mathcal{V} \text{ such that } v \neq (s_i)_0^{out} \text{ and } v \neq (g_i)_T^{out}.$$

$$\sum_{e \in \delta^-((s_i)_0^{out})} x_i[e] = \sum_{e \in \delta^+((g_i)_T^{out})} x_i[e] = 1 \qquad \forall i \in [M].$$

$$x_i[e] \in \{0, 1\} \qquad \forall i \in [M], \forall e \in \mathcal{E}.$$

An optimal solution can thus be found by starting with a lower bound on T and iteratively checking for increasing values of T whether a feasible integer multi-commodity flow of M units exists in the corresponding T-step time-expanded flow network (which is an NP-hard problem), until an upper bound on T is reached (such as the one provided in Yu and Rus (2015)). One can use the maximum over $i \in [M]$ of the length of a shortest path from s_i to g_i in G as the lower bound. Each T-step time-expanded flow network is encoded as an ILP in the above way, which is then solved with an ILP solver. A feasible integer multi-commodity flow for the smallest value of T corresponds to a MAPF solution with the smallest makespan. More details on the ILP-based MAPF algorithm, including an analysis of its properties, can be found in Yu and LaValle (2013b).

3.3 One-Shot Path Planning with Kinematic Constraints

In our recent research (Hönig, Kumar, Cohen, et al., 2016; Hönig, Kumar, Ma, et al., 2016), we have helped to develop a post-processing procedure (not covered as a contribution of this dissertation) called MAPF-POST. MAPF-POST is part of Hönig's dissertation (Hönig, 2019). MAPF-POST makes use of a Simple Temporal Network (STN) to postprocess a MAPF solution in polynomial time to create a plan-execution schedule that takes kinematic constraints of real-world agents into account. Specifically, this plan-execution schedule works for different types of real-world agents, takes their maximum and minimum velocities into account, and provides a guaranteed safety distance between them.

MAPF-POST assumes that (1) each agent traverses each edge with constant velocity and stays at each location (vertex along the path of the agent in the given MAPF solution) only for an instant before it reaches its target and (2) the velocity of each agent can change instantaneously at each location. These assumptions can be approximated by controllers of real-world agents.

Given a MAPF solution on graph $G = (V, E)$, MAPF-POST constructs an STN, which is a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each vertex $\mathfrak{v} \in \mathcal{V}$ represents an event corresponding to an agent a_i entering a location $\pi_i(t) \in V$ given by its path in the MAPF solution. Each directed edge $(\mathfrak{u}, \mathfrak{v}) \in \mathcal{E}$, labeled by an STN bound $[LB(\mathfrak{u}, \mathfrak{v}), UB(\mathfrak{u}, \mathfrak{v})]$, represents a temporal constraint between events $\mathfrak{u}$ and $\mathfrak{v}$ indicating that event $\mathfrak{u}$ must be scheduled between $LB(\mathfrak{u}, \mathfrak{v})$ and $UB(\mathfrak{u}, \mathfrak{v})$ time units before event $\mathfrak{v}$. The STN imposes two types of temporal constraints between events as dictated by the MAPF solution. Type 1: For each agent a_i, it enters locations in the order given by its path in the MAPF solution, namely a_i enters $\pi_i(t)$ before it enters $\pi_i(t')$ for every consecutive locations ($\pi_i(t) \neq \pi_i(t')$) with $t < t'$. The STN bound of such a Type 1 constraint is obtained by using the length of the edge connecting the locations and velocity limits of the agent. Type 2: For each pair of agents a_i and a_j and each location $v = \pi_i(t) = \pi_j(t') \in G$ that they both enter, they enter the location in the order given by their paths in the MAPF solution, namely a_i enters v before a_j enters v assuming $t < t'$ without loss of generality. The STN bound of such a Type 2 constraint is $[0, \infty)$. In effect, the MAPF solution discretizes time

and specifies a total order among the events. The STN, however, does not discretize time and specifies only a partial order among the events, which provides it with the flexibility to take kinematic constraints of real-world agents into account. If a non-zero safety distance δ is to be enforced between the agents, the STN also constructs additional events representing agents entering auxiliary locations, that are additional locations along edges (and between vertices) of graph G, and impose extra temporal constraints for these events.

Definition 3.3 (Hönig (2019))**.** A *plan-execution schedule* assigns a real-valued execution time to each event, corresponding to an entry time for each location. Real-world agents that execute a plan-execution schedule enter all locations at these entry times.

MAPF-POST either solves a linear program or uses the Bellman-Ford algorithm (Bellman, 1958; Ford Jr & Fulkerson, 2015) to compute a plan-execution schedule that assigns (continuous) execution times $t(\mathfrak{v})$ to all events $\mathfrak{v}$ that satisfy all the temporal constraints. For graph G with unit-cost edges and a user-specified parameter $\delta \in (0, 1)$, the resulting plan-execution schedule guarantees that the distance between any two agents is at least δ on graph G and that the Euclidean distance between (the centers of) any two circular real-world agents is at least $\delta/\sqrt{2}$ if graph G is a 2D 4-neighbor grid (Hönig, 2019).

More details on MAPF-POST, including an analysis of its properties, can be found in Hönig (2019).

3.4 Other One-Shot and Long-Term Target-Assignment and Path-Planning Problems

We now survey existing research on other one-shot and long-term target-assignment and path-planning problems.

3.4.1 One-Shot Target-Assignment Problem

In the one-shot target-assignment problem, M targets need to be assigned to a team of M agents. The cost of assigning each target to each agent is given, often capturing the time it takes the agent to move from its current location to the target. An assignment is a one-to-one mapping from the targets to the agents. Many results are known for the classic assignment problem (Kuhn, 1955). The problem is to find an assignment that minimizes the flowtime, namely the total cost. This problem is also called the weighted bipartite matching problem. It can be solved in polynomial time using the Hungarian algorithm (Kuhn, 1955), via an economic model (Shapley & Shubik, 1971), or using a distributed auction-based algorithm (Bertsekas, 1992). There is also research on the linear bottleneck assignment problem (Burkard, Dell'Amico, & Martello, 2009; Fulkerson et al., 1953; Gross, 1959). The problem aims to find an assignment that minimizes the makespan, namely the maximum cost. It can be solved optimally in polynomial time by using a flow algorithm or the Hungarian algorithm as a subroutine (Derigs & Zimmermann, 1978; Garfinkel, 1971).

3.4.2 One-Shot Combined Target-Assignment and Path-Planning Problem for One Team of Agents: Anonymous MAPF

The special case of the one-shot combined target-assignment and path-planning problem for one team of agents has been studied as Anonymous MAPF, also known as Permutation-Invariant MAPF or Unlabeled MAPF (see Section 2.2). We recall that, in Anonymous MAPF, M targets and a team of M *anonymous* agents are given. The problem is to assign the targets to the agents and find collision-free paths for each agent from its given start vertex to its assigned target such that each target is assigned to a unique agent and is occupied by the agent in the end. The objective is to minimize the makespan. Anonymous MAPF can be solved optimally in polynomial time by using a reduction to the max-flow problem on the time-expanded network (Yu & LaValle, 2013a), similar to the reduction from MAPF to the integer multi-commodity flow problem as described in Section 3.2.3. The reduction is as follows. Given an Anonymous

MAPF problem instance on undirected graph $G = (V, E)$ and a limit T on the number of time steps, we construct a T-step time-expanded flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ in the same way as described in Section 3.2.3.1 with the following change: There is only a single commodity type. There are a supply of one unit of this commodity type at vertex $(s_i)_0^{out}$ and a demand of one unit of this commodity type at vertex $(g_i)_T^{out}$ for all $i \in [M]$. The integer multi-commodity flow problem thus becomes a regular feasible circulation problem, which is easily converted to a maximum flow problem. Since all supply and demand values are one, any polynomial or pseudopolynomial time algorithm for maximum flow determines the feasibility of Anonymous MAPF for any particular T in polynomial time. Since the optimal makespan of any Anonymous MAPF problem instance is bounded from above by $M + |V| - 1$ (Yu & LaValle, 2013a), binary search on T thus solves Anonymous MAPF optimally for makespan minimization in polynomial time.

3.4.3 Long-Term Target-Assignment Problem

In the long-term target-assignment problem, targets are constantly added to the system at unknown times and need to be reached by M agents. Each agent might need to get assigned a new target after it reaches its current target. The problem has been well studied as different versions of Online Multi-Robot Task Allocation (Gerkey & Matarić, 2004; Korsah, Stentz, & Dias, 2013) or Cooperative Multi-Agent Tracking (Parker, 1999) in the robotics community and Online Weighted Matching (Kalyanasundaram & Pruhs, 1993) in the theoretical computer science community. It is often solved by utility-based greedy algorithms (Werger & Matarić, 2000) or a reduction to a sequence of one-shot target-assignment problems (Gerkey & Matarić, 2002; Khuller et al., 1994).

3.5 Summary

In this chapter, we provided an overview of MAPF, which is used as an essential element for one-shot and long-term target-assignment and path-planning problems in general. We provided

detailed descriptions of three MAPF algorithms that inspire the design of some of the target-assignment and path-planning algorithms in the later chapters. We described how MAPF solutions can be transformed into plan-execution schedules in a post-processing step by using MAPF-POST, which is also used for TAPF in Chapter 5 and MAPD in Chapter 7. We also surveyed existing research on other one-shot and long-term target-assignment and path-planning problems.

Chapter 4

Theoretical Analysis of Target Assignment and Path Planning

In this chapter, we present the first major contribution of this dissertation. Specifically, we introduce a unified NP-hardness proof structure that can be used to derive computational complexity results for different MAPF variants that model the one-shot and the long-term coordination of autonomous target-assignment and path-planning operations of teams of agents. For example, this unified NP-hardness proof structure can be used to derive new fixed-parameter inapproximability results for the one-shot and the long-term combined target-assignment and path-planning problems TAPF and MAPD. This unified NP-hardness proof structure stems from formalizing and studying a new variant of MAPF, called *Package-Exchange Robot Routing (PERR)*, and its generalization K-PERR. Therefore, these results validate the hypothesis that formalizing and studying new variants of MAPF can result in new theoretical insights into the one-shot and the long-term combined target-assignment and path-planning problems for teams of agents. We follow our formalization of different problems in Chapter 2.

This chapter is based on Ma, H., Tovey, C., Sharon, G., Kumar, T. K. S., & Koenig, S. (2016). Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI Conference on Artificial Intelligence* (pp. 3166–3173). The reductions from $\leq$3,=3-SAT to PERR and from $2/\overline{2}/3$-SAT to 2-PERR are contributions of Craig Tovey.

The remainder of this chapter is structured as follows. In Section 4.1, we reiterate the motivation behind introducing our unified NP-hardness proof structure by studying PERR. In Section 4.2, we introduce our unified NP-hardness proof structure and derive fixed-parameter inapproximability results for PERR. In Section 4.3, we demonstrate how our unified NP-hardness proof structure can also provide fixed-parameter inapproximability results for other one-shot and long-term target-assignment and path-planning problems. The above results lay the theoretical foundation of this dissertation. We then present other results from our study of PERR: In Section 4.4, we show that, unlike MAPF, PERR is always solvable. In Section 4.5, we describe how MAPF algorithms can be adapted to solving PERR by viewing it as a relaxation of MAPF and also provide theoretical results for the special case of 1-PERR. Finally, we summarize the contributions of this chapter in Section 4.6.

4.1 Introduction

In this chapter, we study the computational complexity of optimally solving one-shot and long-term target-assignment and path-planning problems. From the survey in Section 3.1.1, we learned that many theoretical results are known for the one-shot path-planning problem MAPF. The solvability of a MAPF problem instance can be determined and a solution to the MAPF problem instance, if one exists, can be found in polynomial time but with a makespan of $O(|V|^3)$ (Kornhauser et al., 1984; Röger & Helmert, 2012; Yu & Rus, 2015). However, it is NP-hard to solve MAPF optimally (Goldreich, 2011; Ratner & Warmuth, 1986; Surynek, 2010; Yu & LaValle, 2013c). Many theoretical results are also known for the one-shot and the long-term target-assignment problems. Special cases of the one-shot target-assignment problem for one team of agents, including the classic assignment problem (Bertsekas, 1992; Kuhn, 1955; Shapley & Shubik, 1971) and the bottleneck assignment problem (Fulkerson et al., 1953; Garfinkel, 1971; Gross, 1959), can be solved optimally in polynomial time. The special case of the one-shot combined target-assignment and path-planning problem for one team of agents, namely Anonymous MAPF (see Section 2.2), can be solved optimally in polynomial time (Yu & LaValle,

2013a). But the long-term target-assignment problem for multiple agents is NP-hard in general (Brucker, 2010). However, none of the existing results or NP-hardness proof structures can be directly applied to the general case of either the one-shot or the long-term version of the combined target-assignment and path-planning problem for multiple teams of agents.

Therefore, we introduce a unified NP-hardness proof structure, stemming from formalizing and studying a new variant of MAPF, called PERR, and its generalization K-PERR (see Section 2.4). In PERR, each package is preassigned to an agent at a given start vertex and needs to be delivered to a given goal vertex (preassigned target) in a given graph. Each agent carries one package. Packages can be reassigned to agents in a proactive way—two agents in adjacent vertices can *exchange* their packages, and thus exchanging their goal vertices. We use our unified NP-hardness proof structure to establish a reduction from an NP-complete version of the Boolean satisfiability problem, called $\leq$3,=3-SAT (Tovey, 1984), to PERR and prove that PERR is NP-hard to approximate within any constant factor less than $4/3$.

Studying PERR is inspired by practical applications, including ride-sharing (or taxis) with passenger transfers (Coltin & Veloso, 2014) and office robots with package exchanges (Veloso et al., 2015). But, more importantly, it lays the theoretical foundation for studying general multi-agent coordination problems, including MAPF and the one-shot and the long-term combined target-assignment and path-planning problems TAPF (see Section 2.3) and MAPD (see Section 2.5). For example, we demonstrate how our unified NP-hardness proof structure, with a similar reduction from $\leq$3,=3-SAT, can be directly applied to proving the NP-hardness of optimally solving MAPF and MAPD. We also demonstrate how to derive similar NP-hardness results for K-PERR and generalize them to TAPF by using our unified NP-hardness proof structure to establish a reduction from a newly constructed NP-complete version of the Boolean satisfiability problem, called $2/\overline{2}/3$-SAT. Notably, we present the first fixed-parameter inapproximability results for both MAPF and TAPF.

We also present other results from our study of PERR. We prove that all PERR and K-PERR problem instances are solvable. We demonstrate how optimal MAPF algorithms, such as CBS (see Section 3.2.2) and the ILP-based MAPF algorithm (see Section 3.2.3), can be adapted

to solving PERR optimally by viewing PERR as a relaxation of MAPF that permits exchange operations of packages.

4.2 Unified NP-Hardness Proof Structure and Intractability of PERR

We now describe our unified NP-hardness proof structure that can be used to establish reductions NP-complete versions of the Boolean satisfiability problem to target-assignment and path-planning problems. Our unified NP-hardness proof structure requires an NP-complete version of the Boolean satisfiability problem whose Boolean variables each appear in at most a constant number of its disjunctive clauses. Moreover, the total number of times that each variable can appear uncomplemented or complemented affects the optimal makespan of the target-assignment and path-planning problem instance resulting from the reduction and thus the inapproximability ratio for makespan minimization. A reduction from the general version of the Boolean satisfiability problem or the 3-satisfiability problem, where the number of times that each variable can appear is not bounded, thus does not yield constant-factor inapproximability results with our unified NP-hardness proof structure.

As an example, we now prove that PERR is NP-hard to approximate within any constant factor less than $4/3$ for makespan minimization by reducing an NP-complete version of the Boolean satisfiability problem, called $\leq$3,=3-SAT (Tovey, 1984), to PERR. We then derive a corollary for a version of PERR that minimizes the flowtime: PERR is NP-hard to solve optimally for flowtime minimization.

A $\leq$3,=3-SAT problem instance consists of $\mathfrak{N}$ Boolean variables $X_1, \ldots, X_{\mathfrak{N}}$ and $\mathfrak{M}$ disjunctive clauses $C_1, \ldots, C_{\mathfrak{M}}$. Each variable appears in exactly three clauses, uncomplemented at least once and complemented at least once. Each clause contains at most three literals. Its decision question is as follows: Is there an assignment of *True* or *False* to the variables that satisfies the problem instance? $\leq$3,=3-SAT is known to be NP-complete (Tovey, 1984).

Theorem 4.1. For any $\epsilon > 0$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to PERR for makespan minimization.

Proof. We construct a PERR problem instance that has a solution with makespan three if and only if a given $\leq$3,=3-SAT problem instance is satisfiable.

For each variable X_i in the $\leq$3,=3-SAT problem instance, we construct two "literal" packages, p_{iT} and p_{iF}, with start vertices s_{iT} and s_{iF} and goal vertices t_{iT} and t_{iF}, respectively. For each literal package, we construct two paths to get to its goal vertex in three time steps: a "shared" path, namely $\langle s_{iT}, u_{iT}, v_i, t_{iT} \rangle$ for p_{iT} and $\langle s_{iF}, u_{iF}, v_i, t_{iF} \rangle$ for p_{iF}, and a "private" path, namely $\langle s_{iT}, w_{iT}, x_{iT}, t_{iT} \rangle$ for p_{iT} and $\langle s_{iF}, w_{iF}, x_{iF}, t_{iF} \rangle$ for p_{iF}. The shared paths for p_{iT} and p_{iF} intersect at vertex v_i. Only one of the two paths can thus be used if a makespan of three is to be achieved. Sending literal package p_{iT} (or p_{iF}) along the shared path corresponds to assigning *True* (or *False*) to X_i in the $\leq$3,=3-SAT problem instance.

For each clause C_j in the $\leq$3,=3-SAT problem instance, we construct a "clause" package p_j with start vertex c_j and goal vertex d_j. It has multiple (but at most three) "clause" paths to get to its goal vertex in three time steps, which have a one-to-one correspondence to the literals in C_j. Every literal X_i (or $\overline{X_i}$) can appear in at most two clauses. If C_j is the first clause that it appears in, then the clause path is $\langle c_j, w_{iT}, b_j, d_j \rangle$ (or $\langle c_j, w_{iF}, b_j, d_j \rangle$). If C_j is the second clause that it appears in, a vertex α_j is introduced and the clause path is instead $\langle c_j, \alpha_j, x_{iT}, d_j \rangle$ (or $\langle c_j, \alpha_j, x_{iF}, d_j \rangle$). The clause path of each C_j with respect to any literal in that clause and the private path of the literal intersect. Only one of the two paths can thus be used if a makespan of three is to be achieved.

Suppose that a satisfying assignment to the $\leq$3,=3-SAT problem instance exists. Then, a solution with makespan three is obtained by sending literal packages of true literals along their shared paths, the other literal packages along their private paths, and clause packages along the clause paths corresponding to one of the true literals in those clauses.

Conversely, suppose that a solution with makespan three exists. Then, each clause package traverses the clause path corresponding to one of the literals in that clause, and the corresponding

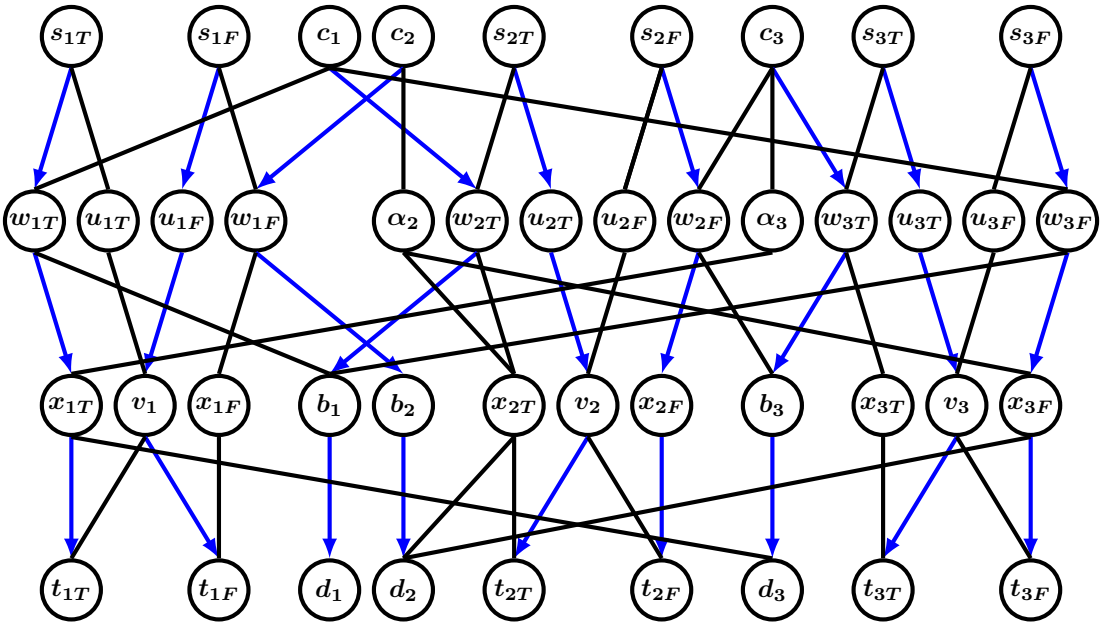


Figure 4.1: Example of the reduction from a $\leq$3,=3-SAT problem instance to a PERR problem instance.

literal package traverses its shared path. Since the packages of a literal and its complement cannot both use their shared path if a makespan of three is to be achieved, we can assign *True* to every literal whose package uses its shared path without assigning *True* to both the uncomplemented and complemented literals. If the packages of both literals use their private paths, we can assign *True* to any one of the literals and *False* to the other one. A solution to the PERR problem instance with makespan three thus yields a satisfying assignment to the $\leq$3,=3-SAT problem instance.

To summarize, the PERR problem instance has a solution with makespan three if and only if the $\leq$3,=3-SAT problem instance is satisfiable. Also, the PERR problem instance cannot have a solution with makespan smaller than three and always has a solution with makespan four, even if the $\leq$3,=3-SAT problem instance is unsatisfiable. For any $\epsilon > 0$, any approximation algorithm for PERR with ratio $4/3 - \epsilon$ thus computes a solution with makespan three whenever the $\leq$3,=3-SAT problem instance is satisfiable and therefore solves $\leq$3,=3-SAT. □

Figure 4.1 shows a PERR problem instance reduced from the $\leq$3,=3-SAT problem instance $(X_1 \vee X_2 \vee \overline{X}_3) \wedge (\overline{X}_1 \vee X_2 \vee \overline{X}_3) \wedge (X_1 \vee \overline{X}_2 \vee X_3)$. Clause C_1 is the first clause that literal X_1 appears in. The corresponding clause path is $\langle c_1, w_{1T}, b_1, d_1 \rangle$. Since clause C_2 is the second clause that X_2 appears in, vertex α_2 is introduced. The corresponding clause path is $\langle c_2, \alpha_2, x_{2T}, d_2 \rangle$. The blue (directed) edges represent one optimal solution to the PERR problem instance of makespan three, which corresponds to the satisfying assignment $(X_1, X_2, X_3) =$ (*False*,*True*,*True*).

The PERR problem instance in the proof of Theorem 4.1 has the property that the length of every path from a start vertex to the corresponding goal vertex is at least three. Therefore, if the makespan is three, then every package is delivered in exactly three time steps and the flowtime is $3M$ $(= 3(2\mathfrak{N} + \mathfrak{M}))$. Moreover, if the makespan exceeds three, then the flowtime exceeds $3M$, yielding the following corollary:

Corollary 4.2. It is NP-hard to find an optimal solution to PERR for flowtime minimization.

4.3 Other Target-Assignment and Path-Planning Problems

We now demonstrate how the unified NP-hardness proof structure introduced in Section 4.2 can be used to derive computational complexity results for other target-assignment and path-planning problems. A key insight is that the unified NP-hardness proof structure does not require exchange operations since any exchange operation would move one package further away from its goal vertex and thus not result in a smaller makespan.

4.3.1 Complexity Results for MAPF

Our unified NP-hardness proof structure applies to MAPF (see Section 2.1). Specifically, we first construct a PERR problem instance for a given $\leq$3,=3-SAT problem instance as described in the proof of Theorem 4.1. We then obtain a MAPF problem instance that corresponds to the resulting PERR problem instance as described in Section 2.4.1. Clearly, the MAPF problem instance has a solution with makespan three if and only if the $\leq$3,=3-SAT problem instance is

satisfiable. Also, the MAPF problem instance cannot have a solution with makespan smaller than three and always has a solution with makespan four, even if the $\leq$3,=3-SAT problem instance is unsatisfiable. If the makespan is three, then the flowtime is $3M$. Moreover, if the makespan exceeds three, then the flowtime exceeds $3M$. This yields the following theorem[1] and corollary:

Theorem 4.3. For any $\epsilon > 0$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to MAPF for makespan minimization.

Corollary 4.4. It is NP-hard to find an optimal solution to MAPF for flowtime minimization.

4.3.2 Complexity Results for MAPD

Our unified NP-hardness proof structure applies to MAPD (see Section 2.5). Specifically, we also reduce from $\leq$3,=3-SAT and use a similar construction to that for PERR and MAPF.

Theorem 4.5. For any $\epsilon > 0$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to MAPD for makespan minimization.

Proof. We use a similar reduction to the one in the proof of Theorem 4.1, but from $\leq$3,=3-SAT to MAPD. Specifically, we construct a MAPD problem instance that has a solution with makespan three if and only if a given $\leq$3,=3-SAT problem instance is satisfiable.
We point out the differences in the construction: For each variable X_i in the $\leq$3,=3-SAT problem instance, we construct two "literal" agents, a_{iT} and a_{iF}, with initial vertices s_{iT} and s_{iF}, respectively, and two tasks, τ_{iT} and τ_{iF}, with pickup vertices s_{iT} and s_{iF} and delivery vertices t_{iT} and t_{iF}, respectively, that are added to the system at time step 0. For each clause C_j in the $\leq$3,=3-SAT problem instance, we construct a "clause" agent a_j with initial vertex c_j and a task τ_j with pickup vertex c_j and delivery vertex d_j that is added to the system at time step 0. Therefore, an optimal solution must assign every task to the agent whose initial vertex is the pickup vertex of the task at time step 0 and let the agent execute the task.

[1] Theorem 4.3 improves the state-of-the-art NP-hardness result of MAPF for makespan minimization (Surynek, 2010), since it shows not only the NP-hardness of optimally solving MAPF but also the NP-hardness of constant-factor approximation. The previous NP-hardness proof structures for MAPF by Surynek (2010) and Yu and LaValle (2013c) do not transfer to PERR and other MAPF variants.

To summarize, using the same arguments as in the proof of Theorem 4.1, the MAPD problem instance has a solution with makespan three if and only if the $\leq$3,=3-SAT problem instance is satisfiable. Also, the MAPD problem instance cannot have a solution with makespan smaller than three and always has a solution with makespan four, even if the $\leq$3,=3-SAT problem instance is unsatisfiable. For any $\epsilon > 0$, any approximation algorithm for MAPD with ratio $4/3-\epsilon$ thus computes a solution with makespan three whenever the $\leq$3,=3-SAT problem instance is satisfiable and therefore solves $\leq$3,=3-SAT. □

Similar to the PERR problem instance constructed in the proof of Theorem 4.1, the MAPD problem instance in the proof of Theorem 4.5 has the property that, if the makespan is three, then every task is finished in exactly three time steps and the service time is three. Moreover, if the makespan exceeds three, then the service time exceeds three, yielding the following corollary:

Corollary 4.6. It is NP-hard to find an optimal solution to MAPD for service time minimization.

These results hold even for an offline version of MAPD where all tasks are known a priori.

4.3.3 Complexity Results for K-PERR

Our unified NP-hardness proof structure also applies to K-PERR (see Section 2.4.3), where packages are partitioned into types.

The construction in the proof of Theorem 4.1 almost applies in case all literal packages are of the same type and all clause packages are of the same type—but not quite since, if clause C_k is the first clause that literal X_i appears in and clause C_j is the second such clause, then clause package p_k could travel from its start vertex c_k along path $\langle c_k, w_{iT}, x_{iT}, d_j \rangle$ of length three to goal vertex d_j. For example, in Figure 4.1, clause package p_1 could travel from its start vertex c_1 along path $\langle c_1, w_{1T}, x_{1T}, d_3 \rangle$ of length three to goal vertex d_3. Therefore, a solution to the PERR problem instance with makespan three does not necessarily yield a satisfying assignment for the $\leq$3,=3-SAT problem instance.

Our unified NP-hardness proof structure thus requires another NP-complete version of the Boolean satisfiability problem for K-PERR. This version of the Boolean satisfiability problem forces any clause package p_j to reach goal vertex d_j if a makespan of three is to be achieved.

Specifically, we now prove that even 2-PERR is NP-hard to approximate within any constant factor less than $4/3$ for makespan minimization, by using our unified NP-hardness proof structure to establish a reduction from an NP-complete version of the Boolean satisfiability problem, called $2/\overline{2}/3$-SAT, to 2-PERR. We show a corollary for the flowtime objective: 2-PERR is NP-hard to solve optimally for flowtime minimization. We then generalize the results to K-PERR for other values of K.

A $2/\overline{2}/3$-SAT problem instance consists of $\mathfrak{N}$ Boolean variables $X_1, \ldots, X_{\mathfrak{N}}$ and $\mathfrak{M}$ disjunctive clauses $C_1, \ldots, C_{\mathfrak{M}}$. Each variable appears complemented in one clause of size two, appears uncomplemented in one clause of size two and appears a third time in a clause of size three. Its decision question is as follows: Is there an assignment of *True* or *False* to the variables that satisfies the problem instance?

We first prove the following Lemma:

Lemma 4.7. $2/\overline{2}/3$-SAT is NP-complete.

Proof. $2/\overline{2}/3$-SAT is clearly in NP. The 3-satisfiability problem is NP-complete and can be reduced to $2/\overline{2}/3$-SAT as follows, similar to Tovey (1984): Given a standard 3-satisfiability problem instance with variables Y_i and exactly three literals per clause, we delete all clauses that contain a variable that does not appear in any other clause. Then, we consider each remaining variable Y_i in turn. Let $\mathcal{K}_i > 1$ be the number of literals that it occurs in. We replace the κ-th occurrence of variable Y_i by a new variable $X_{i,\kappa}$, that is, replace literal Y_i (or $\overline{Y}_i$) with literal $X_{i,\kappa}$ (or $\overline{X}_{i,\kappa}$). Then, we append the following clauses of two literals each to the constructed problem instance: $\left(\bigwedge_{k=1}^{\mathcal{K}_i-1}(X_{i,\kappa} \vee \overline{X}_{i,\kappa+1})\right) \wedge (X_{i,\mathcal{K}_i} \vee \overline{X}_{i,1})$. The clause $X_{i,\kappa} \vee \overline{X}_{i,\kappa+1}$ implies that $X_{i,\kappa+1}$ must be false if $X_{i,\kappa}$ is false and that $X_{i,\kappa}$ must be true if $X_{i,\kappa+1}$ is true. The cyclic structure of the clauses therefore forces all $X_{i,1}, \ldots, X_{i,\mathcal{K}_i}$ to have the same truth value. Therefore, the constructed problem instance is satisfiable if and only if the original 3-satisfiability

problem instance is satisfiable. Each $X_{i,\kappa}$ appears complemented in one clause of size two, appears uncomplemented in one clause of size two and appears a third time in a clause of size three. Moreover, the transformation requires only polynomial time. □

We now use our unified NP-hardness proof structure to reduce $2/\overline{2}/3$-SAT to 2-PERR with a construction similar to that in the proof of Theorem 4.1. The only difference is that a clause C_k of size three has paths of length three only in the form $\langle c_k, \alpha_k, x_{iT}/x_{iF}, d_k\rangle$ (via the private paths of the literals X_i it contains) and thus must send its package to goal vertex d_k if a makespan of three is to be achieved. This prevents any clause of size two from sending its package to d_k. Also, any clause of size two has no paths of length three along which it can send its package to the goal vertex of another clause of size two and thus must send its package to its own goal vertex if a makespan of three is to be achieved. We now prove the theorem formally.

Theorem 4.8. For any $\epsilon > 0$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to 2-PERR for makespan minimization.

Proof. We construct a 2-PERR problem instance that has a solution with makespan three if and only if a given $2/\overline{2}/3$-SAT problem instance is satisfiable. Figure 4.2 shows an example. Then, the remainder of the proof of Theorem 4.1 applies without change.

We follow the construction in the proof of Theorem 4.1, with the exception of making every clause path for clauses c_j of size two of the form $\langle c_j, w_{iT}/w_{iF}, b_j, d_j\rangle$ (vertex α_j is not introduced in this case) and every clause path for clauses c_k of size three of the form $\langle c_k, \alpha_k, x_{iT}/x_{iF}, d_k\rangle$. We distinguish only two package types, namely literal and clause packages. Every vertex s_{iT} and s_{iF} (or t_{iT} and t_{iF}) is a start (or goal) vertex of a literal package, and every vertex c_j (or d_j) is a start (or goal) vertex of a clause package.

Suppose that a satisfying assignment to the $2/\overline{2}/3$-SAT problem instance exists. Then, a solution with makespan three is obtained, as in the proof of Theorem 4.1, by sending literal packages of true literals along their shared paths, the other literal packages along their private paths, and clause packages along the clause paths corresponding to one of the true literals in those clauses.

Conversely, suppose that a solution with makespan three exists. Then, the only paths of length three from start vertex s_{iT} or s_{iF} to a goal vertex of a literal package are the shared or private paths that end at goal vertex t_{iT} or t_{iF}. The shared paths of packages p_{iT} and p_{iF} intersect at their penultimate vertices. Since the two packages cannot occupy the same vertex at time step $t = 2$, at most one of them can traverse its shared path if a makespan of three is to be achieved. Package p_{iT} cannot arrive at t_{iF} since then p_{iF} has no path of length three to a goal vertex of a literal available, and vice versa for p_{iF}. Now consider an arbitrary clause C_k of size three. The only paths of length three from start vertex c_k to a goal vertex of a clause package have the form $\langle c_k, \alpha_k, x_{iT}/x_{iF}, d_k \rangle$. Clause package p_k thus must arrive at goal vertex d_k. Finally, consider an arbitrary clause C_j of size two. The only paths of length three from start vertex c_j to a goal vertex of a clause package have the form $\langle c_j, w_{iT}/w_{iF}, b_j, d_j \rangle$ or $\langle c_j, w_{iT}/w_{iF}, x_{iT}/x_{iF}, d_{k'} \rangle$, where the clause $C_{k'}$ of size three shares a literal with clause C_j. Paths of the latter form cannot be used because goal vertex $d_{k'}$ must receive clause package $p_{k'}$. Clause package p_j thus must arrive at goal vertex d_j. The situation is now identical to that in the proof of Theorem 4.1 because every package p_{iT} (or p_{iF}, p_j, or p_k) travels from its start vertex s_{iT} (or s_{iF}, c_j, or c_k) to goal vertex t_{iT} (or t_{iF}, d_j, or d_k). A solution to the 2-PERR problem instance with makespan three thus yields a satisfying assignment to the $2/\overline{2}/3$-SAT problem instance when assigning *True* to every literal whose package uses its shared path, as explained in the proof of Theorem 4.1. □

Figure 4.2 shows a 2-PERR problem instance reduced from the $2/\overline{2}/3$-SAT problem instance $(X_1 \vee \overline{X}_2) \wedge (\overline{X}_1 \vee X_3) \wedge (X_2 \vee \overline{X}_3) \wedge (X_1 \vee X_2 \vee \overline{X}_3)$. Consider any solution with makespan three. The only paths of length three from start vertex c_4 to a goal vertex of a clause package have the form $\langle c_4, \alpha_4, x_{iT}/x_{iF}, d_4 \rangle$. Clause package p_4 thus must arrive at goal vertex d_4. The only paths of length three from start vertex c_1 to a goal vertex of a clause package have the form $\langle c_1, w_{iT}/w_{iF}, b_1, d_1 \rangle$ or $\langle c_1, w_{1T}, x_{1T}, d_4 \rangle$. Paths of the latter form cannot be used because d_4 must receive p_4. Clause package p_1 thus must arrive at goal vertex d_1. The colored (directed) edges represent one optimal solution to the 2-PERR problem instance with makespan three, which yields the satisfying assignment $(X_1, X_2, X_3) = (\textit{True}, \textit{True}, \textit{True})$. The

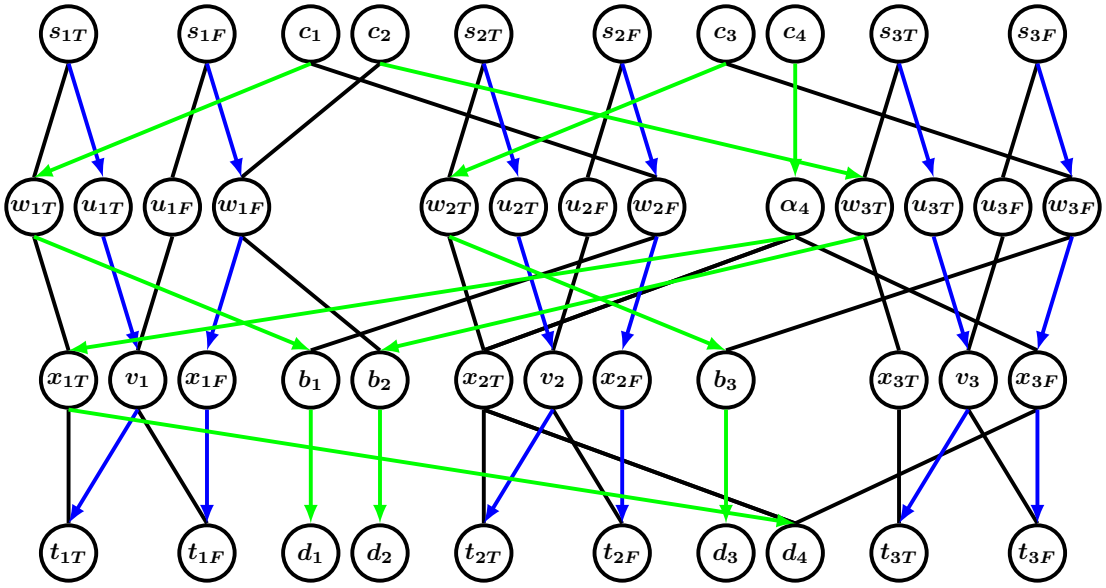


Figure 4.2: Example of the reduction from a $2/\overline{2}/3$-SAT problem instance to a 2-PERR problem instance.

blue edges represent paths of the literal packages, and the green edges represent paths of the clause packages.

The argument for proving Corollary 4.2 yields the following corollary:

Corollary 4.9. It is NP-hard to find an optimal solution to 2-PERR for flowtime minimization.

Theorem 4.8 and Corollary 4.9 hold not only for 2-PERR but also for K-PERR for all $K = 3, \ldots, M$ because one can pad the graph of the constructed 2-PERR problem instance with additional vertices, each being both the start and goal vertex of a package of a different type. This padding constructs a K-PERR problem instance for any $K = 3, \ldots, M$ but leaves the proof unchanged.

Theorem 4.10. For any $\epsilon > 0$ and $K > 1$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to K-PERR for makespan minimization.

Corollary 4.11. For any $K > 1$, it is NP-hard to find an optimal solution to K-PERR for flowtime minimization.

4.3.4 Complexity Results for TAPF

Our unified NP-hardness proof structure also applies to TAPF (see Section 2.3). Specifically, we first construct a 2-PERR problem instance for a given $2/\overline{2}/3$-SAT problem instance as described in the proof of Theorem 4.8. We then obtain a TAPF problem instance with two teams of agents that corresponds to the resulting 2-PERR problem instance as described in Section 2.4.3. The TAPF problem instance has a solution with makespan three if and only if the $2/\overline{2}/3$-SAT problem instance is satisfiable. Also, the TAPF problem instance cannot have a solution with makespan smaller than three and always has a solution with makespan four, even if the $2/\overline{2}/3$-SAT problem instance is unsatisfiable. This yields the following theorem:

Theorem 4.12. For any $\epsilon > 0$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to TAPF with two teams of agents for makespan minimization.

Using the same argument for generalizing Theorem 4.8 to Theorem 4.10, we generalize Theorem 4.12 to the theorem below:

Theorem 4.13. For any $\epsilon > 0$ and $K > 1$, it is NP-hard to find a $4/3 - \epsilon$ approximate solution to TAPF with K teams of agents for makespan minimization.

4.3.5 Additional Generalizations

Our unified NP-hardness proof structure also applies to many other variants of PERR, including but not limited to cases where

1. packages and agents disappear upon delivery;
2. agents can carry more than one package; or
3. agents exchange packages more slowly or more quickly than moving along an edge.

Recent research (Ma et al., 2018a) also uses the unified NP-hardness proof structure to derive similar NP-hardness results for a MAPF variant that maximizes the number of agents that can reach their given goal vertices within a given deadline (see also Section 3.1.3.1).

Our unified NP-hardness proof structure can potentially establish reductions from other NP-complete versions of the Boolean satisfiability problem to different target-assignment and path-planning problems, which could result in different inapproximability ratios for them. For example, the unified NP-hardness proof structure can potentially replace $\leq$3,=3-SAT with the general version of the Boolean satisfiability problem for PERR by adding an appropriate number of vertices for each clause and making the shared and private paths of the variables longer in the construction. This reduction, however, yields only an NP-hardness result but no inapproximability result for makespan minimization because the optimal makespan of the PERR problem instance resulting from the construction is not a constant.

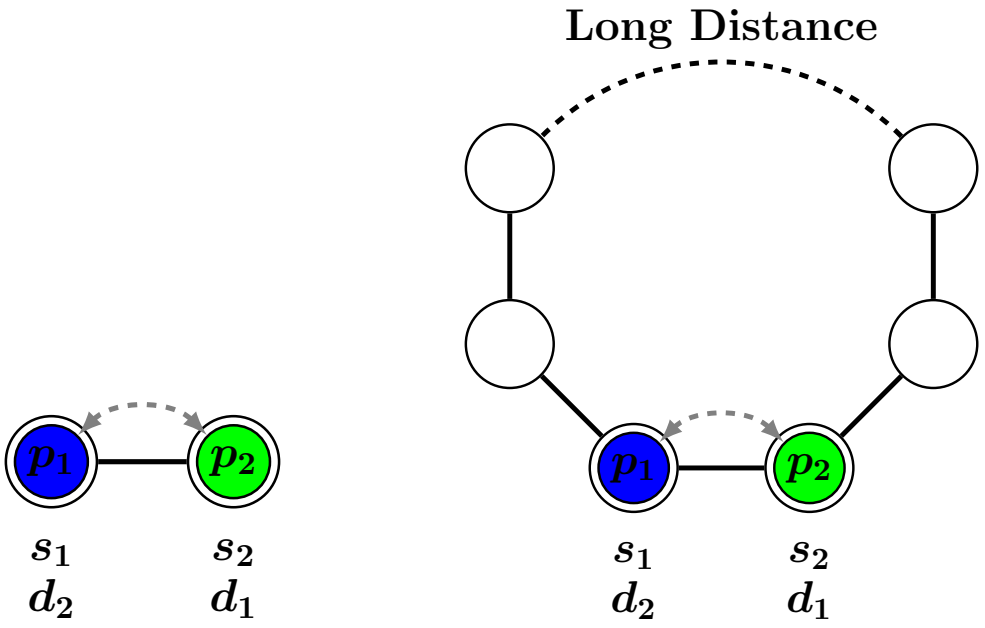
4.4 Feasibility of PERR

In Section 2.4.1, we established the correspondence between a PERR problem instance and a MAPF problem instance on the same graph. We now show one of the key differences between the combinatorial structures of PERR and MAPF.

We consider both the standard objective of minimizing the makespan and the objective of minimizing the flowtime for both PERR and MAPF. Figures 4.3(a) and 4.3(b) illustrate that exchange operations can make a PERR problem instance solvable or improve the makespan and flowtime by a large factor in contrast to its corresponding MAPF problem instance that does not permit exchange operations. In particular, Figure 4.3(a) shows that some MAPF problem instances (with no exchange operations permitted) are unsolvable. The following theorem shows that, on the contrary, all PERR problem instances are solvable:

Theorem 4.14. All PERR problem instances are solvable. Solutions with polynomial makespans and flowtimes can be found in polynomial time.

Proof. Without loss of generality, we assume that all vertices are initially occupied by agents carrying packages. (In the following, if neither of two adjacent vertices is occupied by an agent, then an exchange operation involving them is replaced by a no-op operation. If only one of them is occupied by an agent, then the exchange operation is replaced by the agent moving to the

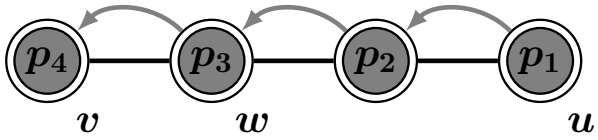


(a) A PERR problem instance on a graph with two vertices. Two agents carrying packages p_1 and p_2 have to deliver them from their start vertices s_1 and s_2 to their goal vertices g_1 and g_2, respectively. They can exchange their packages (indicated by the dashed gray edge). If exchange operations are not allowed (as in the corresponding MAPF problem instance), then no solution exists.

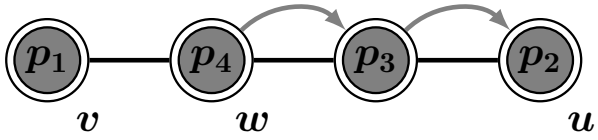
(b) A PERR problem instance on a cyclic graph. Two agents carrying packages p_1 and p_2 have to deliver them from their start vertices s_1 and s_2 to their goal vertices g_1 and g_2, respectively. They can exchange their packages (indicated by the dashed gray edge). If exchange operations are not allowed (as in the corresponding MAPF problem instance), then one agent must take the long detour.

Figure 4.3: Motivating examples of PERR that demonstrate the power of exchange operations.

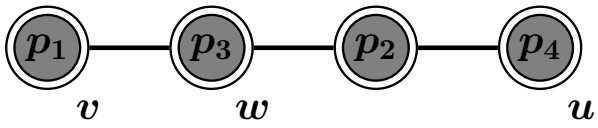
adjacent vertex.) Then, any two packages can switch vertices without affecting the vertices of the other packages. To see this, consider two packages p_i and p_j and their current vertices u and v, respectively. Let $\langle u, \ldots, w, v \rangle$ be a shortest path from u to v, where w is the vertex on the path directly before v. A series of exchange operations along this path moves p_i to v and every other package on this path against the path one edge closer to u (in at most $|V| - 1$ time steps). In particular, it moves p_j to w. A series of exchange operations against the path $\langle u, \ldots, w \rangle$ then moves p_j to u and every other package on this path back to its original vertex (in at most $|V| - 2$ time steps), hence proving the claim. This property allows one to route all packages to their goal



(a) Moving package p_1 from vertex u to vertex v.



(b) Moving package p_4 from vertex w to vertex u.



(c) Resulting positions of the packages.

Figure 4.4: Example of the construction used in the proof of Theorem 4.14.

vertices one at a time. Our algorithm performs only a polynomial number of operations, which implies that the makespans and flowtimes of the resulting solutions are also polynomial. □

Figure 4.4 shows an example of the construction used in the above proof of Theorem 4.14: Package p_1 can be moved from vertex u to vertex v via a series of exchange operations along the path $\langle u, \ldots, v \rangle$. Then, package p_4 can be moved from vertex w to u via a series of exchange operations along the path $\langle w, \ldots, u \rangle$, restoring the original positions of packages p_2 and p_3.

Letting any two packages switch vertices, as done in the proof of Theorem 4.14, takes at most $(|V| - 1) + (|V| - 2) = 2|V| - 3$ time steps. This operation needs to be repeated at most $M - 1$ times since one additional package reaches its goal vertex each time. An upper bound on the makespan of the resulting solution is thus

$$\mathcal{U}_{\textit{makespan}} = (M - 1)(2|V| - 3). \tag{4.1}$$

Since each package reaches its goal vertex by time $\mathcal{U}_{makespan}$, an upper bound on the flowtime of the resulting solution is

$$\mathcal{U}_{flowtime} = M \cdot \mathcal{U}_{makespan} = M(M-1)(2|V|-3). \tag{4.2}$$

The proof of Theorem 4.14 applies unchanged to K-PERR if all packages of the same type are first assigned arbitrary different goal vertices of the same type, yielding the following corollary:

Corollary 4.15. All K-PERR problem instances are solvable. Solutions with polynomial makespans and flowtimes can be found in polynomial time.

4.5 PERR Algorithms

From Section 2.4.1, we know that PERR can be viewed as a relaxed version of MAPF by omitting the edge collisions (Definition 2.2). Therefore, many MAPF algorithms can be directly adapted to solving PERR by removing the constraints of avoiding edge collisions. We now demonstrate how the two optimal MAPF algorithms, CBS and the ILP-based MAPF algorithm, introduced in Section 3.2 can be adapted to solving PERR optimally for makespan minimization. We also discuss K-PERR and the special case of 1-PERR.

4.5.1 Adapted CBS

We can easily adapt CBS (see Section 3.2.2) from MAPF to PERR, which requires only the addition of exchange operations. Therefore, the resulting adapted CBS algorithm does not attempt to detect edge collisions on Line 12 in Algorithm 3.2 and thus never adds edge constraints on the high level. Its low-level space-time A* search for agent a_i uses $\mathcal{U}_{makespan}$ given by Equation 4.1 as the upper bound on the arrival time T_i. The adapted CBS algorithm is correct, complete, and optimal for makespan minimization for PERR, as can be shown with arguments similar to those in (Sharon et al., 2015). The adapted CBS algorithm can also be used to compute optimal

solutions for a version of PERR with the flowtime objective if the flowtime of the plan of a node is assigned to the cost of the node on Lines 7 and 22 in Algorithm 3.2.

4.5.2 ILP-Based PERR Algorithm

Similar to the ILP-based MAPF algorithm (see Section 3.2.3), the ILP-based PERR algorithm first reduces PERR to the integer multi-commodity flow problem on a time-expanded flow network and then uses this reduction to solve PERR optimally for makespan minimization.

4.5.2.1 Reducing PERR to Multi-Commodity Flow

We now describe the reduction used by the ILP-based PERR algorithm.

Given a PERR problem instance on undirected graph $G = (V, E)$ and a limit T on the number of time steps, we construct a T-step *time-expanded flow network* $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V} = \bigcup_{v \in V}(\{v_0^{out}\} \cup \bigcup_{t=1}^{T}\{v_t^{in}, v_t^{out}\})$ and directed edges $\mathcal{E}$ with unit capacity. Each vertex $v \in V$ is translated to a vertex $v_t^{out} \in \mathcal{V}$ for all $t = 0 \dots T$ (which represents vertex v at the end of time step t) and a vertex $v_t^{in} \in \mathcal{V}$ for all $t = 1 \dots T$ (which represents vertex v in the beginning of time step t). For each package p_i, we set a supply of one at (start) vertex $(s_i)_0^{out}$ and a demand of one at (goal) vertex $(g_i)_T^{out}$, both for commodity type i (corresponding to package p_i). Each vertex $v \in V$ is translated to an edge $(v_t^{out}, v_{t+1}^{in}) \in \mathcal{E}$ for all time steps $t = 0 \dots T-1$ (which represents a package waiting at vertex v between time steps t and $t+1$). Each vertex $v \in V$ is also translated to an edge $(v_t^{in}, v_t^{out}) \in \mathcal{E}$ for all time steps $t = 1 \dots T$ (which prevents vertex collisions of the form $\langle *, *, v, t \rangle$ among all packages since only one package can occupy vertex v between time steps t and $t+1$). Each edge $(u, v) \in E$ is translated to edges $(u_t^{out}, v_{t+1}^{in})$ and $(v_t^{out}, u_{t+1}^{in})$ for all time steps $t = 0 \dots T-1$ to allow a package to move along the edge or exchange with another package along the edge.

The above reduction is similar to reducing MAPF to the integer multi-commodity flow problem (see Section 3.2.3.1). The construction of the flow network for a PERR problem instance is the same as that for the corresponding MAPF problem instance except that it uses edges

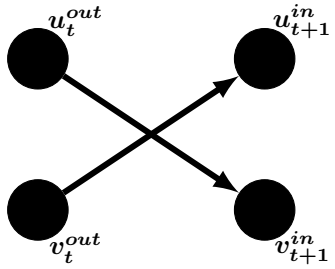


Figure 4.5: Example of the construction of $\mathcal{N}$ for edge $(u, v) \in E$ and time step t.

$(u_t^{out}, v_{t+1}^{in})$ and $(v_t^{out}, u_{t+1}^{in})$ for each edge $(u, v) \in E$ and each time step $t = 0 \dots T-1$ to allow for exchange operations instead of a gadget (shown in Figure 3.2) to prevent edge collisions. Figure 4.5 shows an example of the construction for an exchange operation. The construction of the flow network implies the following theorem:

Theorem 4.16. There is a correspondence between all feasible integer multi-commodity flows on the T-step time-expanded flow network of a number of units equal to the number of agents and all solutions of the PERR problem instance with makespans of at most T.

The proof of this theorem mirrors the one for the reduction of MAPF to the integer multi-commodity flow problem (Yu & LaValle, 2013b).

4.5.2.2 Example

Figure 4.6 shows a feasible integer multi-commodity flow in the 3-step time-expanded flow network reduced from the PERR problem instance shown in Figure 2.7. Solid colored circles are (start) vertices with a supply. Hatched colored circles are (goal) vertices with a demand. The blue edges represent a unit flow of commodity type 1, corresponding to a path for package p_1. The green edges represent a unit flow of commodity type 2, corresponding to a path for package p_2. The feasible integer multi-commodity flow corresponds to a solution $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle\}$.

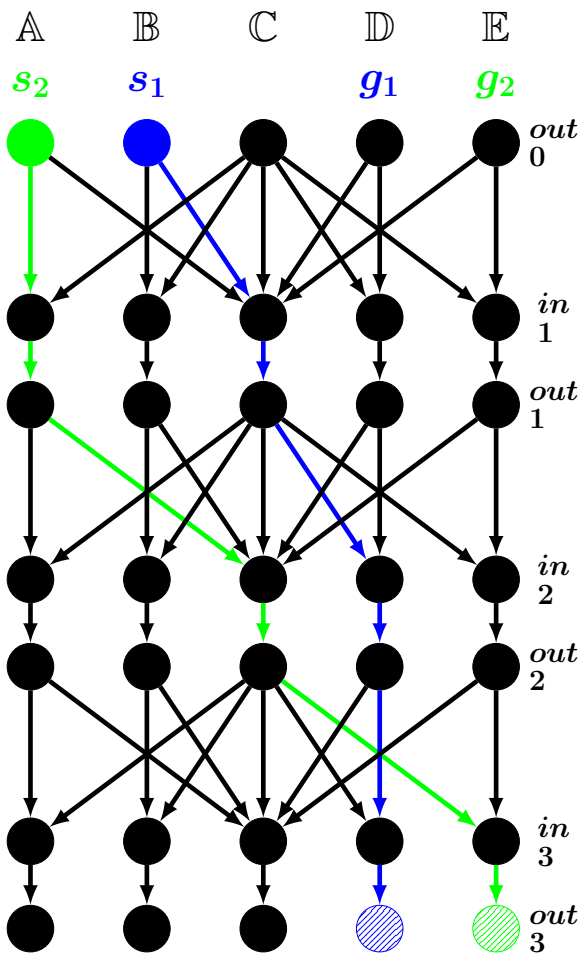


Figure 4.6: A feasible integer multi-commodity flow for the PERR problem instance shown in Figure 2.7.

4.5.2.3 Solving PERR via ILP

The ILP-based PERR algorithm then employs the reduction to solve PERR optimally for makespan minimization. An integer multi-commodity flow problem can be expressed as ILP using the standard formulation (also shown in Section 3.2.3.3). Let $\delta^+(v)$ (respectively $\delta^-(v)$) be the set of incoming (respectively outgoing) edges of v. Let $x_i[e]$ be the Boolean variable representing the amount of flow of commodity type i on edge e. An integer multi-commodity flow problem on the T-step time-expanded flow network can be written as an ILP using the standard formulation shown in Section 3.2.3.3. An optimal solution can thus be found by starting with a

lower bound on T and iteratively checking for increasing values of T whether a feasible integer multi-commodity flow of M units exists in the corresponding T-step time-expanded flow network (which is an NP-hard problem). We can use the maximum over $i \in [M]$ of the length of a shortest path from s_i to g_i in G as a lower bound on T. Each T-step time-expanded flow network is translated into an ILP in the above way, which is then solved with an ILP solver. A feasible integer multi-commodity flow for the smallest value of T corresponds to a PERR solution with the smallest makespan. Unlike the ILP-based MAPF algorithm, the ILP-based PERR algorithms does not require an upper bound on T since every PERR problem instance is solvable.

If an upper bound on T is available, then binary search on T can also be used to the smallest value of T for which a feasible integer multi-commodity flow of M units exists. One such upper bound is $\mathcal{U}_{\textit{makespan}}$ given by Equation 4.1. Thus, the binary search requires at most $O(\log \mathcal{U}_{\textit{makespan}}) = O(\log(M|V|)) \leq O(\log |V|^2) = O(\log |V|)$ iterations. The binary search requires fewer iterations than repeatedly incrementing T if the smallest value of T is much larger than the lower bound.

4.5.2.4 Alternative ILP Formulation for Minimizing Makespan

Given an upper bound on T, for example, $\mathcal{U}_{\textit{makespan}}$ given by Equation 4.1, we also develop a way to minimize the makespan by solving one ILP only. To do so, we add a single auxiliary variable z to the standard ILP formulation for the integer multi-commodity flow problem on the $\mathcal{U}_{\textit{makespan}}$-step time-expanded flow network. We minimize z subject to $z \geq t \cdot x_i[(u_{t-1}^{out}, v_t^{in})]$ for all $i \in [M]$, $(u, v) \in E$, and $t = 1, \ldots, \mathcal{U}_{\textit{makespan}}$. The last movement of any package sets the value of z, thereby minimizing the makespan. Let $\mathcal{L}$ be a lower bound on T, for example, the maximum over $i \in [M]$ of the length of a shortest path from s_i to g_i in G. Since only the last movement of some package p_i on the incoming edges of g_i no earlier than time step $\mathcal{L}$ can

possibly set the value of z for any feasible flow, we only need to keep the above constraints for all $t = \mathcal{L}, \ldots, \mathcal{U}_{\textit{makespan}}$ and edges in $\delta^+((g_i)_t^{in})$. The complete ILP formulation is as follows.

$\text{minimize}\, z,\ \text{subject to:}$

$$0 \leq \sum_{i \in [M]} x_i[e] \leq 1 \qquad \forall e \in \mathcal{E}.$$

$$\sum_{e \in \delta^+(v)} x_i[e] - \sum_{e \in \delta^-(v)} x_i[e] = 0 \qquad \forall i \in [M], \\ \forall v \in \mathcal{V} \text{ such that } v \neq (s_i)_0^{out} \text{ and } v \neq (g_i)_T^{out}.$$

$$\sum_{e \in \delta^-((s_i)_0^{out})} x_i[e] = \sum_{e \in \delta^+((g_i)_T^{out})} x_i[e] = 1 \qquad \forall i \in [M].$$

$$x_i[e] \in \{0, 1\} \qquad \forall i \in [M], \forall e \in \mathcal{E}.$$

$$z \geq t \cdot x_i[e] \qquad \forall i \in [M], \forall t \in \{\mathcal{L}, \ldots, \mathcal{U}_{\textit{makespan}}\}, \forall e \in \delta^+((g_i)_t^{in}).$$

This formulation can also be used to solve MAPF optimally for makespan minimization using an $O(|V|^3)$ upper bound on the smallest makespan.

4.5.2.5 ILP Formulation for Minimizing Flowtime

We also consider a version of PERR with the flowtime objective. A similar formulation allows us to minimize the flowtime by solving one ILP only, namely by using an upper bound for T and adding auxiliary variables z_i for all $i \in [M]$. An upper bound on the smallest flowtime, for example, $\mathcal{U}_{\textit{flowtime}}$ given by Equation 4.2, is also an upper bound on T, namely an upper bound on the makespan of any solution with the smallest flowtime. We minimize $\sum_{i \in [M]} z_i$ subject to $z_i \geq t \cdot x_i[(u_{t-1}^{out}, v_t^{in})]$ for all $i \in [M]$, $(u, v) \in E$, and $t = 1, \ldots, \mathcal{U}_{\textit{flowtime}}$. The last movement of package p_i sets the value of z_i, thereby minimizing the flowtime. Let $\mathcal{L}_i$ be the length of a shortest path from s_i to g_i in G. Since only the last movement of package p_i on the incoming edges of g_i no earlier than time step $\mathcal{L}_i$ can possibly set the value of z_i for any feasible flow, we

only need to keep the above constraints for all $t = \mathcal{L}, \ldots, \mathcal{U}_{\textit{flowtime}}$ and edges in $\delta^+((g_i)_t^{in})$ for each i. The complete ILP formulation is as follows.

$$\text{minimize} \sum_{i \in [M]} z_i, \text{ subject to:}$$

$$0 \leq \sum_{i \in [M]} x_i[e] \leq 1 \qquad \forall e \in \mathcal{E}.$$

$$\sum_{e \in \delta^+(v)} x_i[e] - \sum_{e \in \delta^-(v)} x_i[e] = 0 \qquad \forall i \in [M], \; \forall v \in \mathcal{V} \text{ such that } v \neq (s_i)_0^{out} \text{ and } v \neq (g_i)_T^{out}.$$

$$\sum_{e \in \delta^-((s_i)_0^{out})} x_i[e] = \sum_{e \in \delta^+((g_i)_T^{out})} x_i[e] = 1 \qquad \forall i \in [M].$$

$$x_i[e] \in \{0, 1\} \qquad \forall i \in [M], \forall e \in \mathcal{E}.$$

$$z_i \geq t \cdot x_i[e] \qquad \forall i \in [M], \forall t \in \{\mathcal{L}_i, \ldots, \mathcal{U}_{\textit{flowtime}}\}, \forall e \in \delta^+((g_i)_t^{in}).$$

This formulation can also be used to solve MAPF optimally for flowtime minimization using an $O(|V|^3)$ upper bound on the optimal flowtime.

4.5.3 Solving K-PERR Optimally

We can also use the construction described in Section 4.5.2.1 to reduce K-PERR to the integer multi-commodity flow problem. Given a K-PERR problem instance on undirected graph $G = (V, E)$ and a limit T on the number of time steps, we construct a T-step time-expanded flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ in the same way as described in Section 4.5.2.1 with the following change: There are K instead of M commodity types. There are a supply of one unit at vertex $(s_i)_0^{out}$ and a demand of one unit at vertex $(g_i)_T^{out}$ for all $i \in [M]$, both for commodity type k where $k \in [K]$ is the type of package p_i. We then use the standard ILP formulation to solve K-PERR optimally for makespan minimization as described in Section 4.5.2.3, since $\mathcal{U}_{\textit{makespan}}$ given by Equation 4.1 is also an upper bound on the optimal makespan for K-PERR.

4.5.3.1 Special Cases of 1-PERR

In the special case of 1-PERR, the integer multi-commodity flow problem becomes a regular feasible circulation problem, which is easily converted to a maximum flow problem. Since all supply and demand values are one, any polynomial or pseudopolynomial time algorithm for maximum flow determines the feasibility of 1-PERR for any particular T in polynomial time. This reduction is similar to reducing Anonymous MAPF (see Section 3.4.2) to a max-flow problem but permits exchange operations. Binary search on T thus yields the following theorem:

Theorem 4.17. 1-PERR can be solved optimally for makespan minimization in polynomial time.

We can derive a tighter upper bound on the optimal makespan for 1-PERR. We recall that there is a correspondence between a 1-PERR problem instance and an Anonymous MAPF problem instance on the same graph (see Section 2.4.4). We further show the following theorem:

Theorem 4.18. Any solution to an Anonymous MAPF problem instance can be interpreted as a solution to its corresponding 1-PERR problem instance with the same makespan and flowtime. Any solution to a 1-PERR problem instance can be transformed into a solution to its corresponding Anonymous MAPF problem instance with the same makespan and flowtime.

Proof. The first statement is obviously true since any solution to an Anonymous MAPF problem instance does not contain any exchange operations and thus is also a solution to its corresponding 1-PERR problem instance with the same makespan and flowtime.
For any solution to a 1-PERR problem instance that possibly contains exchange operations, we construct a 1-PERR solution with the same makespan and flowtime that does not contain exchange operations as follows. For every two packages p_i and p_j and each exchange operation that moves package p_i from vertex u to vertex v and package p_j from vertex v to vertex u between time steps t and $t+1$, we let both packages wait at their current vertices for one time step and exchange their paths from time step $t+1$ on (and thus their goal vertices and arrival times), resulting in the new path of package p_i $\pi_i' = \langle \pi_i(0), \ldots, \pi_i(t) = u, \pi_j(t+1) = u, \ldots, \pi_j(T_j) \rangle$

and the new path of package p_j $\pi'_j = \langle \pi_j(0), \ldots, \pi_j(t) = v, \pi_i(t+1) = v, \ldots, \pi_i(T_i) \rangle$. The resulting 1-PERR solution has the same makespan and flowtime as the original one but does not contain exchange operations. It thus is also a solution to the corresponding Anonymous MAPF problem instance with the same makespan and flowtime. □

Therefore, exchange operations do not improve the makespan or the flowtime for 1-PERR. The above theorem implies the following corollary:

Corollary 4.19. The optimal makespan of any 1-PERR problem instance is equal to the optimal makespan of its corresponding Anonymous MAPF problem instance.

Proof. The optimal makespan of any 1-PERR problem instance is at most the optimal makespan of its corresponding Anonymous MAPF problem instance since any solution with the optimal makespan to the Anonymous MAPF problem instance can be interpreted as a solution to the 1-PERR problem instance with the same makespan. The optimal makespan of the 1-PERR problem instance is at most the optimal makespan of the Anonymous MAPF problem instance since any solution with the optimal makespan to the 1-PERR problem instance can be transformed into a solution to the Anonymous MAPF problem instance with the same makespan. The corollary thus follows. □

Since the optimal makespan of any Anonymous MAPF problem instance is bounded from above by $M + |V| - 1$ (Yu & LaValle, 2013a) (which is a tighter upper bound than the one given by Equation 4.1), the optimal makespan of any 1-PERR problem instance is also bounded from above by $M + |V| - 1$.

4.5.3.2 Open Questions for Flowtime Minimization

We now consider a version of 1-PERR and a version of Anonymous MAPF that minimize the flowtime. An argument similar to that for proving Corollary 4.19 yields the following corollary:

Corollary 4.20. The optimal flowtime of any 1-PERR problem instance is equal to the optimal flowtime of its corresponding Anonymous MAPF problem instance.

There exists an Anonymous MAPF solution with the optimal flowtime whose makespan is bounded from above by $\frac{(M-1)(M-2)}{2} + |V|$ (Yu & LaValle, 2013a). Such an Anonymous MAPF solution can be interpreted as a 1-PERR solution with the same makespan and flowtime due to Theorem 4.18, which is also optimal for 1-PERR with respect to the flowtime objective due to Corollary 4.20. Therefore, there also exists a 1-PERR solution with the optimal flowtime whose makespan is bounded from above by $\frac{(M-1)(M-2)}{2} + |V|$.

Therefore, both 1-PERR and Anonymous MAPF can be solved optimally with respect to the flowtime objective by using the ILP formulation introduced in Section 4.5.2.5 with one type of commodity and setting $\mathcal{U}_{\textit{flowtime}} = \frac{(M-1)(M-2)}{2} + |V|$ for their respective $\mathcal{U}_{\textit{flowtime}}$-step time-expanded flow networks. $\mathcal{L}_i$ can be set as the minimum of the length of a shortest path from $s_{i'}$ to g_i in G over all $i' \in [M]$. The complete ILP formulation is as follows.

$$\text{minimize} \sum_{i \in [M]} z_i, \text{ subject to:}$$

$$\sum_{e \in \delta^+(v)} x[e] - \sum_{e \in \delta^-(v)} x[e] = 0 \qquad \forall i \in [M], \\ \forall v \in \mathcal{V} \text{ such that } v \neq (s_i)_0^{out} \text{ and } v \neq (g_i)_T^{out}.$$

$$\sum_{e \in \delta^-((s_i)_0^{out})} x[e] = \sum_{e \in \delta^+((g_i)_T^{out})} x[e] = M \qquad \forall i \in [M].$$

$$x[e] \in \{0, 1\} \qquad \forall e \in \mathcal{E}.$$

$$z_i \geq t \cdot x[e] \qquad \forall i \in [M], \forall t \in \{\mathcal{L}_i, \ldots, \mathcal{U}_{\textit{flowtime}}\}, \\ \forall e \in \delta^+((g_i)_t^{in}).$$

However, the complexities of optimally solving 1-PERR and Anonymous MAPF for flow time minimization are not known. Their complexities are equivalent due to Theorem 4.18 and Corollary 4.20. We do not address their complexities in this dissertation but post the following open questions:

- Can 1-PERR and Anonymous MAPF be solved optimally in polynomial time for flowtime minimization?

- Are 1-PERR and Anonymous MAPF NP-hard to solve optimally for flowtime minimization?

4.6 Summary

In this chapter, we introduced a unified NP-hardness proof structure that stems from formalizing and studying a novel MAPF variant, called PERR. Our unified NP-hardness proof structure reduces NP-complete versions of the Boolean satisfiability problem to target-assignment and path-planning problems. We used our unified NP-hardness proof structure to prove that PERR is NP-hard to approximate within any constant factor less than $4/3$ for makespan minimization by reducing $\leq$3,=3-SAT to PERR. We also used our proof structure to prove that 2-PERR is NP-hard to approximate within any constant factor less than $4/3$ for makespan minimization by reducing $2/\overline{2}/3$-SAT to 2-PERR. The latter proof generalizes to all cases of K-PERR with $K > 1$. Studying PERR is a first attempt toward establishing a theoretical understanding for one-shot target-assignment and path-planning problems that involve payload exchanges. More importantly, studying PERR also generates new insights into solving other target-assignment and path-planning problems. For example, we demonstrated that our unified NP-hardness proof structure can be used to derive fixed-parameter inapproximability results for the one-shot path-planning problem MAPF, the one-shot combined target-assignment and path-planning problem TAPF, and the long-term combined target-assignment and path-planning problem MAPD and NP-hardness results for many other MAPF variants. Specifically, we proved that MAPF, TAPF with $K > 1$ teams of agents, and MAPD are all NP-hard to approximate within any constant factor less than $4/3$ for makespan minimization. We also described how optimal MAPF algorithms can be used to solve PERR optimally by viewing MAPF as a relaxation of PERR and that the special case of 1-PERR can be solved optimally for makespan minimization in polynomial time. Finally, we posted open questions for optimally solving 1-PERR and Anonymous MAPF with respect to the flowtime objective.

To summarize, in this chapter, we validated the hypothesis that formalizing and studying new variants of MAPF can result in new theoretical insights into the one-shot and the long-term combined target-assignment and path-planning problems for teams of agents. The theorems and corollaries derived in this chapter suggest that it is NP-hard to solve the one-shot and the long-term combined target-assignment and path-planning problems optimally for makespan minimization in general, but computing solutions with the optimal makespan for the special case of one team of agents for the one-shot combined target-assignment and path-planning problem can be done in polynomial time. These results lay a theoretical foundation for studying the combined target-assignment and path-planning problems that we introduce in Chapters 5 and 6.

Chapter 5

One-Shot Target Assignment and Path Planning

In this chapter, we present the second major contribution of this dissertation. Specifically, we formalize and study a new variant of MAPF, called *Combined Target Assignment and Path Finding (TAPF)*, that models the one-shot coordination of autonomous target-assignment and path-planning operations of teams of agents. We demonstrate how TAPF can be solved using a flow-based ILP formulation. We also present a novel TAPF algorithm, called *Conflict-Based Min-Cost Flow (CBM)*, that exploits the combinatorial structure of TAPF. Both algorithms are optimal for TAPF. Experimental results show that CBM can compute optimal TAPF solutions for more than 400 agents in minutes of runtime. We describe how the existing polynomial-time procedure MAPF-POST can be used in a post-processing step to transform TAPF solutions into plan-execution schedules that can be safely executed by teams of real-world agents. Therefore, these results validate the hypothesis that formalizing and studying new variants of MAPF can result in new algorithms for the one-shot combined target-assignment and path-planning problem that can benefit real-world applications of multi-agent systems. We follow our formalization of TAPF in Section 2.3.

The remainder of this chapter is structured as follows. In Section 5.1, we reiterate the motivation behind formalizing and studying TAPF. In Section 5.2, we establish the relationship between

This chapter is based on Ma, H., & Koenig, S. (2016). Optimal target assignment and path finding for teams of agents. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 1144–1152).

TAPF and flow problems and present an optimal TAPF algorithm based on a flow-based ILP formulation. In Section 5.3, we present CBM and prove that it is correct, complete, and optimal. In Section 5.4, we experimentally evaluate the scalability of CBM on TAPF problem instances with dozens of teams and hundreds of agents, compare it with the ILP-based TAPF algorithm, and adapt CBM to a warehouse map. In Section 5.5, we describe how TAPF algorithms can take kinematic constraints into account. Finally, we summarize the contributions of this chapter in Section 5.6.

5.1 Introduction

In this chapter, we develop algorithmic techniques for the one-shot coordination of autonomous target-assignment and path-planning operations of teams of agents by studying TAPF (see Section 2.3). TAPF generalizes both Anonymous MAPF and (Non-Anonymous) MAPF. In TAPF, M agents are given and are partitioned into K teams. Each team is given the same number of unique targets as there are agents in the team. The problem is to assign targets to agents and plan collision-free paths for the agents from their given start vertices to their assigned targets in a way such that each agent moves to exactly one target given to its team and each target of each team is reached by an agent in the same team. Any agent in a team can be assigned any target of the team, and the agents in the same team are thus exchangeable. However, agents in different teams are not exchangeable. For example, in an Amazon Robotics automated warehouse system (see Figure 1.1 for an example), warehouse robots delivering inventory pods to K inventory stations are partitioned into K teams, one for each inventory station. The problem is to assign positions in (the entrance queue of) each inventory station to warehouse robots carrying inventory pods that store the products needed by the inventory station and plan collision-free paths for the warehouse robots from their current locations to their assigned positions in the inventory stations.

TAPF also models the one-shot coordination of both the target-assignment and the path-planning operations for many other real-world applications of multi-agent systems, including

teams of video game characters (Ma, Yang, et al., 2017) or swarms of differential-drive robots and quadcopters (Hönig, Kumar, Ma, et al., 2016; Hönig, Preiss, et al., 2018; Li, Sun, et al., 2020; Preiss et al., 2017; Turpin, Michael, & Kumar, 2014) that assign targets among themselves and move to their assigned targets to form a designed formation together, groups of soccer robots (MacAlpine, Price, & Stone, 2015) that assign positions of a soccer field among themselves and move to their assigned positions to fill different roles in a soccer game of RoboCup, and teams of service robots (Jiang, Yedidsion, Zhang, Sharon, & Stone, 2019) that assign different tasks among themselves and move to the task locations (where their assigned tasks can be executed) in an indoor office environment to provide service to the office workers.

As we previously discussed in Section 1.1, solving the one-shot combined target-assignment and path-planning problem TAPF requires solving both the one-shot target-assignment and the one-shot path-planning sub-problems for these teams of agents. However, research so far has concentrated on the two extreme cases of Anonymous MAPF and (Non-Anonymous) MAPF. Yet, many real-world applications fall between the extreme cases because the number of teams is larger than one but smaller than the number of agents ($1 < K < M$). Straightforward ways of generalizing (Non-Anonymous) MAPF algorithms have difficulties with either scalability (due to the resulting large state spaces), such as searching over all assignments of targets to agents to find optimal solutions, or solution quality, such as assigning targets to agents with target-assignment algorithms such as (Tovey, Lagoudakis, Jain, & Koenig, 2005; Zheng & Koenig, 2009) and then planning collision-free paths for the agents with (Non-Anonymous) MAPF algorithms (perhaps followed by improving the assignment and iterating (Wagner, Choset, & Ayanian, 2012)) to find sub-optimal solutions. It is also unclear how to generalize algorithms for solving Anonymous MAPF, which is a polynomial-time solvable problem, to the NP-hard problem of TAPF with multiple teams of agents.

Therefore, we study algorithmic techniques for optimally solving TAPF to bridge the gap between algorithms for the extreme cases of Anonymous MAPF and (Non-Anonymous) MAPF. We first demonstrate how TAPF can be solved by generalizing the flow-based ILP formulation for MAPF (see Section 3.2.3) to TAPF. We then present a TAPF algorithm, called CBM, that

solves TAPF optimally by simultaneously assigning targets to agents and planning collision-free paths for them. It exploits the combinatorial structure of TAPF by reducing TAPF to the polynomial-time solvable sub-problems for single teams and the NP-hard sub-problem of coordinating different teams. Specifically, CBM is a hierarchical algorithm that combines ideas from both Anonymous and (Non-Anonymous) MAPF algorithms. It uses a min-cost max-flow algorithm (Goldberg & Tarjan, 1987) on a time-expanded flow network on the low level to assign targets to and find paths for agents in single teams and CBS (see Section 3.2.2) on the high level to resolve any collisions among the paths of agents from different teams. Theoretically, we prove that CBM is correct, complete, and optimal. Experimentally, we compare CBM to solving the flow-based ILP formulation of TAPF directly with an ILP solver.

5.2 ILP-Based TAPF Algorithm

Similar to the ILP-based MAPF algorithm (see Section 3.2.3), the ILP-based TAPF algorithm first reduces TAPF to the integer multi-commodity flow problem on a time-expanded flow network and then uses this reduction to solve TAPF optimally.

5.2.1 Reducing TAPF to Multi-Commodity Flow

We now describe the reduction used by the ILP-based TAPF algorithm.

Given a TAPF problem instance on undirected graph $G = (V, E)$ and a fixed number of time step T, we construct a T-step *time-expanded flow network* $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V} = \bigcup_{v \in V}(\{v_0^{out}\} \cup \bigcup_{t=1}^{T}\{v_t^{in}, v_t^{out}\})$ and directed edges $\mathcal{E}$ with unit capacity. Each vertex $v \in V$ is translated to a vertex $v_t^{out} \in \mathcal{V}$ for all $t = 0, \ldots, T$ (which represents vertex v at the end of time step t) and a vertex $v_t^{in} \in \mathcal{V}$ for all $t = 1, \ldots, T$ (which represents vertex v in the beginning of time step t). For each agent a_j^k, we set a supply of one unit of commodity type k at (start) vertex $(s_j^k)_0^{out}$. For each target g_j^k, we set a demand of one unit of commodity type k at (target) vertex $(g_j^k)_T^{out}$. Each vertex $v \in V$ is also translated to an edge $(v_t^{out}, v_{t+1}^{in}) \in \mathcal{E}$ for all time steps $t = 0, \ldots, T-1$ (which represents an agent waiting at vertex v between time steps t and $t+1$).

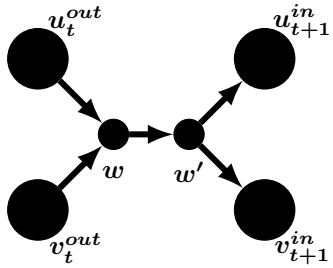


Figure 5.1: Example of the construction of the gadgets in $\mathcal{N}$ for edge $(u, v) \in E$ and time step t.

Each vertex $v \in V$ is translated to an edge $(v_t^{in}, v_t^{out}) \in \mathcal{E}$ for all time steps $t = 1, \ldots, T$ (which prevents vertex collisions of the form $\langle *, *, v, t \rangle$ among all agents since only one agent can occupy vertex v between time steps t and $t+1$). Each edge $(u, v) \in E$ is translated to a *gadget* of vertices in $\mathcal{V}$ and edges in $\mathcal{E}$ for all time steps $t = 0, \ldots, T-1$, which consists of two auxiliary vertices $w, w' \in \mathcal{V}$ that are unique to the gadget (but have no superscripts/subscripts here for ease of readability) and the edges $(u_t^{out}, w), (v_t^{out}, w), (w, w'), (w', u_{t+1}^{in}), (w, v_{t+1}^{in}) \in \mathcal{E}$. This gadget prevents edge collisions of the forms $\langle *, *, u, v, t \rangle$ and $\langle *, *, v, u, t \rangle$ among all agents since only one agent can move along the edge (u, v) in any direction between time steps t and $t+1$. Figure 5.1 shows an example of the construction of the gadgets.

The above reduction is similar to reducing MAPF to the integer multi-commodity flow problem (see Section 3.2.3). The construction of the flow network for a TAPF problem instance is similar to that for a MAPF problem instance except that the supplies at all vertices $(s_j^k)_0^{out}$ and the demands at all vertices $(g_j^k)_T^{out}$ have the same commodity type k for each $k \in [K]$. The construction implies the following theorem:

Theorem 5.1. There is a correspondence between all feasible integer multi-commodity flows on the T-step time-expanded network of a number of units equal to the number of agents and all solutions of the TAPF problem instance with makespans of at most T.

The proof of this theorem mirrors the one for the reduction of MAPF to the integer multi-commodity flow problem (Yu & LaValle, 2013b).

5.2.2 Solving TAPF via ILP

The ILP-based TAPF algorithm employs the reduction to solve TAPF optimally. Let $\delta^+(v)$ (respectively $\delta^-(v)$) be the set of incoming (respectively outgoing) edges of v. Let $x_k[e]$ be the Boolean variable representing the amount of flow of commodity type k on edge e. An integer multi-commodity flow problem on the T-step time-expanded network can be written as an ILP using the following standard formulation.

$$0 \leq \sum_{k=1}^{K} x_k[e] \leq 1 \qquad \forall e \in \mathcal{E}.$$

$$\sum_{e \in \delta^+(v)} x_k[e] - \sum_{e \in \delta^-(v)} x_k[e] = 0 \qquad \forall k \in [K], \forall j \in [M_k], \forall v \in \mathcal{V} \text{ such that } v \neq (s_j^k)_0^{out} \text{ and } v \neq (g_j^k)_T^{out}.$$

$$\sum_{j \in [M_k]} \sum_{e \in \delta^-((s_j^k)_0^{out})} x_k[e] = \sum_{j \in [M_k]} \sum_{e \in \delta^+((g_j^k)_T^{out})} x_k[e] = M_k \qquad \forall k \in [K].$$

$$x_k[e] \in \{0, 1\} \qquad \forall k \in [K], \forall e \in \mathcal{E}.$$

An optimal solution can thus be found by starting with $T = 0$ and iteratively checking for increasing values of T whether a feasible integer multi-commodity flow of a number of units equal to the number of agents exists for the corresponding T-step time-expanded network (which is an NP-hard problem), until an upper bound on T is reached (such as the one provided in Section 2.3.1). Each T-step time-expanded network is translated into an ILP, which is then solved with an ILP solver. We evaluate this ILP-based TAPF algorithm experimentally in Section 5.4.1.1.

5.2.3 Example

We now demonstrate how to solve the TAPF problem instance shown in Figure 2.5 via the reduction to the integer multi-commodity flow problem. There is no feasible integer multi-commodity flow on the 0, 1, or 2-step time-expanded flow network constructed from the problem

instance. Figure 5.2 shows the construction of the 3-step time-expanded flow network from the problem instance and a feasible integer multi-commodity flow on it. There is a supply of one unit of commodity type 1 at vertex $\mathbb{C}_0^{out}$ and a demand of one unit of commodity type 1 at vertex $\mathbb{E}_3^{out}$. There is a supply of one unit of commodity type 2 at vertices $\mathbb{A}_0^{out}$ and $\mathbb{B}_0^{out}$ each and a demand of one unit of commodity type 2 at vertices $\mathbb{D}_3^{out}$ and $\mathbb{F}_3^{out}$ each. The blue edges represent a flow for commodity type 1, which corresponds to a path for the agent in $team_1$. The green edges represent a flow for commodity type 2, which corresponds to paths for the two agents in $team_2$. This feasible integer multi-commodity flow flow thus corresponds to the assignments of target $g_1^1 = \mathbb{E}$ to agent a_1^1 of $team_1$ and target $g_1^2 = \mathbb{D}$ to agent a_1^2 and target $g_2^2 = \mathbb{F}$ to agent a_2^2, respectively, of $team_2$, namely $\sigma^1(a_1^1) = g_1^1$, $\sigma^2(a_1^2) = g_1^2$, and $\sigma^2(a_2^2) = g_2^2$, and represents the optimal solution $\{\pi_1^1 = \langle \mathbb{C}, \mathbb{D}, \mathbb{E} \rangle, \pi_1^2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{B}, \mathbb{D} \rangle, \pi_2^2 = \langle \mathbb{B}, \mathbb{B}, \mathbb{D}, \mathbb{F} \rangle\}$ with makespan 3.

5.2.4 Special Case of One Team

Anonymous MAPF results from TAPF if only one team exists (that consists of all agents) (see also Section 3.4.2) and can be solved with a max-flow algorithm in polynomial time.

Corollary 5.2. TAPF with one team of agents can be solved optimally for makespan minimization in polynomial time.

5.3 Conflict-Based Min-Cost Flow

In this section, we present Conflict-Based Min-Cost-Flow (CBM), a hierarchical algorithm that solves TAPF optimally. On the high level, CBM considers each team to be a meta-agent. It uses CBS (see Section 3.2.2) to resolve collisions among meta-agents, that is, agents in different teams. Similar to CBS, the high-level search of CBM performs a best-first search on a constraint tree, where each node contains a set of constraints and paths for all agents that obey these constraints, move all agents to unique targets of their teams, and result in no collisions among agents in the same team. On the low level, CBM uses a polynomial-time min-cost max-flow algorithm (Goldberg & Tarjan, 1987) on a time-expanded network to assign all targets of a single team to

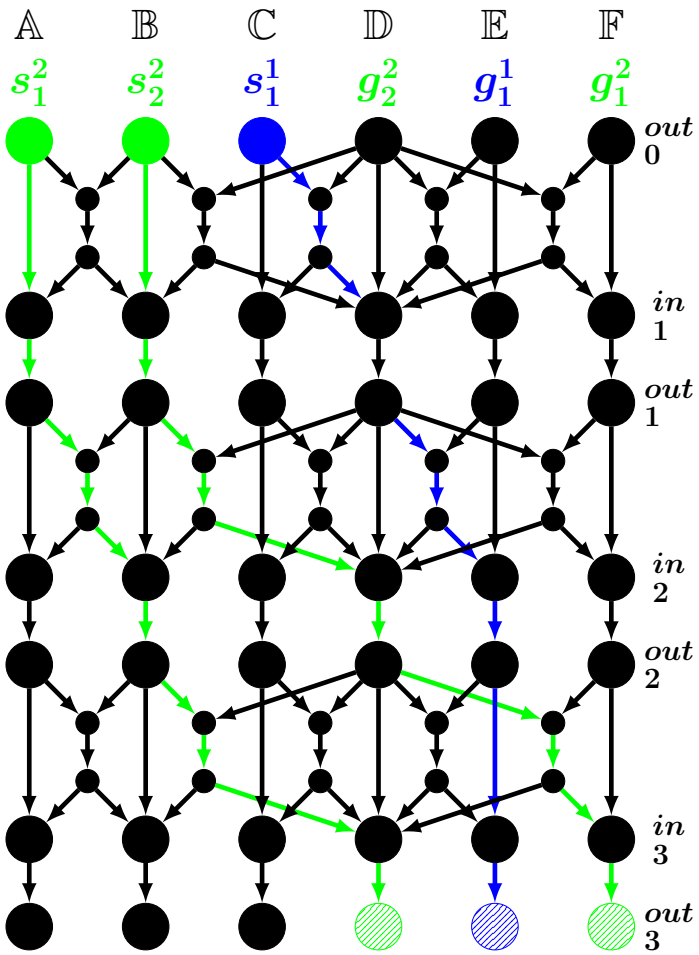


Figure 5.2: A feasible integer multi-commodity flow for the TAPF problem instance shown in Figure 2.5.

unique agents in the same team and plan paths for the agents that obey the constraints imposed by the currently considered high-level node and result in no collisions among the agents in the team. Since the runtime of CBS on the high level can be exponential in the number of collisions that need to be resolved (Sharon et al., 2015), CBM uses edge weights on the low level to bias the paths of agents in the same team so as to reduce the possibility of creating collisions with agents in different teams.

The idea of biasing the search on the low level has been used before for solving (Non-Anonymous) MAPF with CBS (as described in Section 3.2.2.2). Similarly, the idea of grouping

some agents into a meta-agent on the high level and planning paths for each group on the low level has been used before for solving (Non-Anonymous) MAPF with CBS (as described in Section 3.2.2.5) but faces the difficulty of having to identify good groups of agents (Sharon et al., 2015). The best way to group agents can often be determined only experimentally and varies significantly among MAPF problem instances. On the other hand, grouping all agents in a team into a meta-agent for solving TAPF is a natural way of grouping agents since the assignments of targets to agents in the same team and the paths of these agents strongly depend on each other and should thus be planned together on the low level. For example, if an agent is assigned a different target, then many of the agents in the same team typically need to be assigned different targets as well and have their paths re-planned. Also, the lower level can then use a polynomial-time max-flow algorithm on a time-expanded network to assign all targets of a single team to agents in that team and find paths for the agents, due to the polynomial-time complexity of the corresponding TAPF sub-problem for one single team (see Section 5.2.4).

5.3.1 High-Level Search of CBM

Similar to the high-level search of CBS (see Section 3.2.2.1), CBM performs a best-first search on the high level to resolve collisions among agents from different teams and builds a constraint tree.

Definition 5.1. A *constraint* for team $team_k$ is either a *vertex constraint* $\langle team_k, v, t \rangle$, that prohibits any agent in team $team_k$ from occupying a vertex v at timestep t, or an *edge constraint* $\langle team_k, u, v, t \rangle$, that prohibits any agent in $team_k$ from moving from vertex u to vertex v between timesteps t and $t + 1$.

Each node N contains a set of constraints $N.constraints$, a plan (paths for all agents) $N.plan$ that obeys these constraints, moves all agents to unique targets of their teams, and results in no collisions among agents in the same team, and a cost $N.cost$ equal to the makespan of its plan. The list OPEN stores all generated but unexpanded nodes. Algorithm 5.1 shows the high-level search of CBM. CBM starts with the root node, that has an empty set of constraints [Line 1].

Algorithm 5.1: High-Level Search of CBM

Input: TAPF problem instance

1 *Root.constraints* $\leftarrow \emptyset$;
2 *Root.plan* $\leftarrow \emptyset$;
3 **foreach** $k \in [K]$ **do**
4 **if** *LowLevel*($team_k$, *Root*) returns no paths **then**
5 **return** "No Solution";
6 Add the returned paths to *Root.plan*;
7 *Root.cost* $\leftarrow$ *Makespan*(*Root.plan*);
8 OPEN $\leftarrow \{Root\}$;
9 **while** OPEN $\neq \emptyset$ **do**
10 $N \leftarrow \arg\min_{N' \in \text{OPEN}} N'.cost$;
11 OPEN $\leftarrow$ OPEN $\setminus \{N\}$;
12 **if** $N.plan$ has no collision **then**
13 **return** $N.plan$;
14 *collision* $\leftarrow$ a vertex or edge collision $\langle team_k, team_{k'}, \dots \rangle$ in $N.plan$;
15 **foreach** $team_k$ involved in *collision* **do**
16 $N' \leftarrow$ new node;
17 $N'.plan \leftarrow N.plan$;
18 $N'.constraints \leftarrow N.constraints$;
19 $N'.constraints \leftarrow N'.constraints \cup \{\langle team_k, \dots \rangle\}$;
20 **if** *LowLevel*($team_k$, N') returns paths **then**
21 Update $N'.plan$ with the returned paths;
22 $N'.cost \leftarrow$ *Makespan*($N'.plan$);
23 OPEN $\leftarrow$ OPEN $\cup \{N'\}$;
24 **return** "No Solution";

It performs a low-level search to find paths for all agents in each team that move all agents to unique targets of the team and result in no collisions among agents in the team. It terminates unsuccessfully if the low-level search for any team returns no path. Otherwise, the plan of the root node contains paths for all agents [Line 2-6]. The cost of the root node is the makespan of its plan [Line 7]. CBM inserts the root node into OPEN [Line 8]. If OPEN is empty, then CBM terminates unsuccessfully [Lines 9 and 24]. Otherwise, CBM expands a node N in OPEN with the smallest cost [Line 10] and removes the node from OPEN [Line 11]. The cost of a node is the makespan of its plan. Similar to the high-level search of CBS (see Section 3.2.2.1), ties are broken in favor of the node whose plan has the smallest number of pairs of colliding teams.

If the plan of node N have no colliding agents, then they are a solution and CBM terminates successfully with this plan [Lines 12-13]. Otherwise, CBM finds a collision that it needs to resolve [Line 14]. CBM then generates two child nodes N_1 and N_2 of node N [Lines 15-16]. Each child node inherits the plan and all constraints from node N [Lines 17-18]. If the collision is a vertex collision $\langle team_k, team_{k'}, v, t\rangle$, then CBM adds the vertex constraint $\langle team_k, v, t\rangle$ to the constraints of node N_1 and the vertex constraint $\langle team_{k'}, v, t\rangle$ to the constraints of node N_2 [Line 19]. If the collision is an edge collision $\langle team_k, team_{k'}, u, v, t\rangle$, then CBM adds the edge constraint $\langle team_k, u, v, t\rangle$ to the constraints of node N_1 and the edge constraint $\langle team_{k'}, v, u, t\rangle$ to the constraints of node N_2 [Line 19]. For node N_1 (respectively N_2), CBM performs a low-level search to assign targets of team $team_k$ (respectively $team_{k'}$) to agents in the same team and find paths for the agents that obey all constraints in N_1.*constraints* (respectively N_2.*constraints*) relevant to team $team_k$ (respectively $team_{k'}$) and result in no collisions among the agents in the team. If the low-level search successfully returns such paths, CBM replaces the old paths of all agents in $team_k$ (respectively $team_{k'}$) in N_1.*plan* (respectively N_2.*plan*) with the returned ones [Line 20], updates the cost of N_1 (respectively N_2) accordingly, and inserts N_1 (respectively N_2) into OPEN [Lines 22-23]. Otherwise, it discards the node.

5.3.2 Low-Level Search of CBM

On the low level, *LowLevel*($team_k$,N) assigns the targets of team $team_k$ to agents in the same team and finds paths for the agents that obey all constraints of node N (namely all vertex constraints of the form $\langle team_k, *, *\rangle$ and all edge constraints of the form $\langle team_k, *, *, *\rangle$) and result in no collisions among the agents in the team.

Given a fixed number of time steps T, CBM constructs the T-step time-expanded flow network from Section 5.2 with the following changes: (a) There is only a single commodity type k since CBM considers only the single team $team_k$. There are a supply of one unit of this commodity type at vertex $(s_j^k)_0^{out}$ and a demand of one unit of this commodity type at vertex $(g_j^k)_T^{out}$ for all $j \in [M_k]$. (b) To obey the vertex constraints, CBM removes the edge (v_t^{in}, v_t^{out}) from $\mathcal{E}$ for each vertex constraint of the form $\langle team_k, v, t\rangle$. (c) To obey the edge constraints,

CBM removes the edges $((u)_t^{out}, w)$ and $(w', (v)_{t+1}^{in})$ from $\mathcal{E}$ for all gadgets that correspond to edge $(u, v) \in E$ for each edge constraint of the form $\langle team_k, u, v, t \rangle$. Let $\mathcal{V} = \mathcal{V}'$ be the set of (remaining) vertices and $\mathcal{E}'$ be the set of (remaining) edges.

Similar to the procedure from Section 5.2, CBM iteratively checks for increasing values of T whether a feasible integer single-commodity flow of M_k units exists for the corresponding T-step time-expanded flow network, which can be done with the polynomial-time max-flow algorithm that finds a feasible maximum flow. CBM can start with T being a lower bound on the cost of node N. One such lower bound is the cost of the parent node of node N since the constraints of node N is a superset of those of its parent node, which results in a plan with makespan no smaller than that of its parent node. (For $N = Root$, CBM starts with $T = 0$.) During the earliest iteration when the max-flow algorithm finds a feasible flow of M_k units, the call returns successfully with the paths for the agents in the team that correspond to the flow. If T reaches an upper bound on the makespan of an optimal solution (such as the one provided in Yu and Rus (2015)) and no feasible flow of M_k units was found, then the call returns unsuccessfully with no paths.

5.3.3 Example

We now demonstrate how to solve the TAPF problem instance shown in Figure 2.5 using CBM. Figure 5.3 shows the construction of the root node *Root* in the constraint tree on the left and the 2-step time-expanded flow network for the low-level search for each team on the bottom right with a feasible integer flow for the team. *Root.constraints* of the root node *Root* is an empty set of constraints [Line 1]. *LowLevel*($team_1$, *Root*) returns path $\pi_1^1 = \langle \mathbb{C}, \mathbb{D}, \mathbb{E} \rangle$, and *LowLevel*($team_2$, *Root*) returns paths $\pi_1^2 = \langle \mathbb{A}, \mathbb{B}, \mathbb{D} \rangle, \pi_2^2 = \langle \mathbb{B}, \mathbb{D}, \mathbb{F} \rangle$, that result in no collisions among agents in $team_2$ [Lines 3-6]. The resulting cost *Root.cost* of the root node *Root* is 2 [Line 7]. In the first iteration, the root node $N = Root$ is popped from OPEN [Line 10]. There is a vertex collision $\langle team_1, team_2, \mathbb{D}, 1 \rangle$ in its plan [Line 14]. For team $team_1$, CBM generates the left child node N' and adds a new constraint $\langle team_1, \mathbb{D}, 1 \rangle$ [Lines 16-19]. Figure 5.4 shows the construction of the left child node $N' = N_1$ in the constraint tree on the left and the 3-step

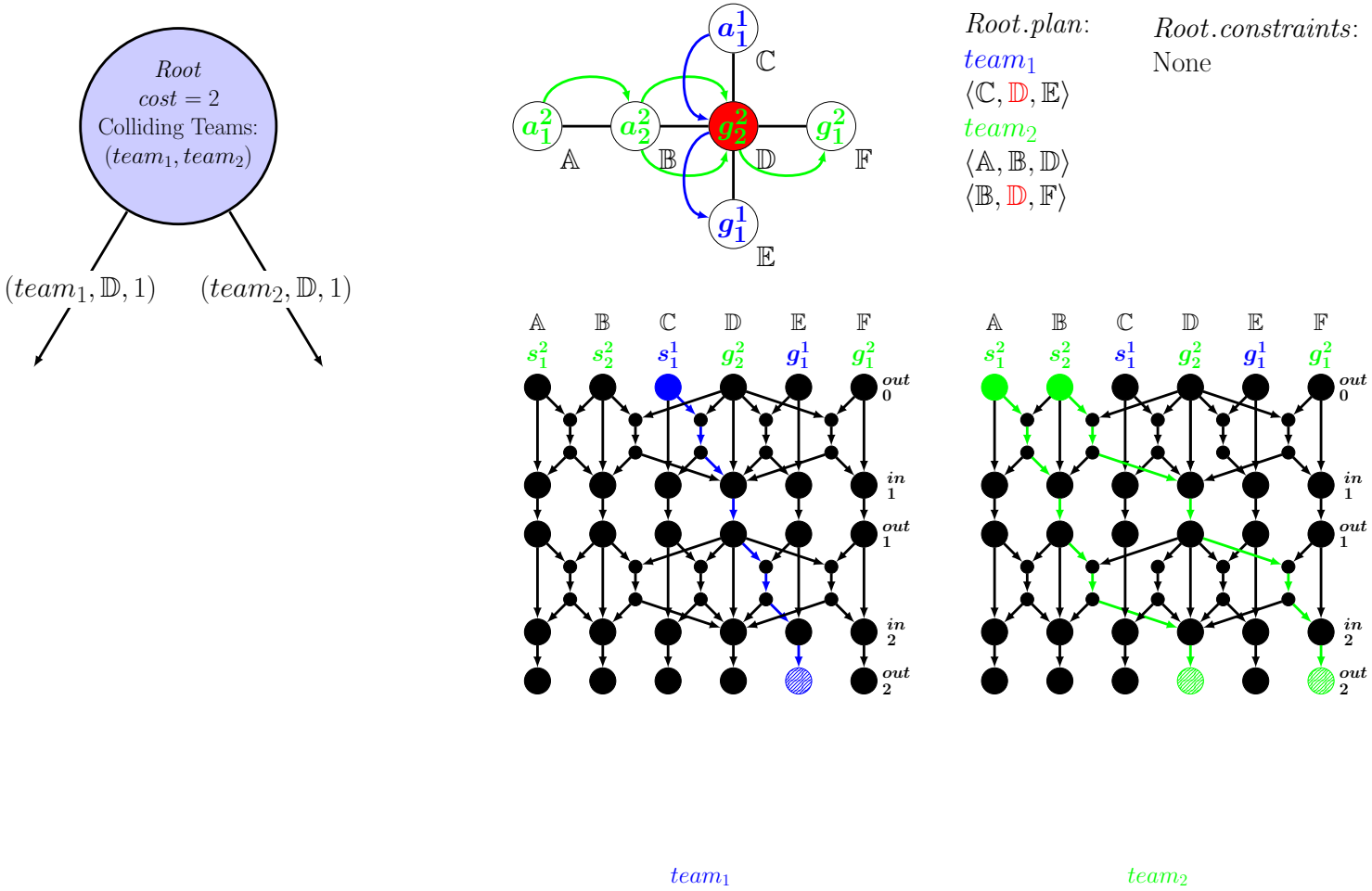


Figure 5.3: Construction of the root node *Root* and the low-level search for $team_1$ and $team_2$. The constraints and the plan of *Root* are shown on the top right. The colliding teams of each node are shown to explain the tie breaking.

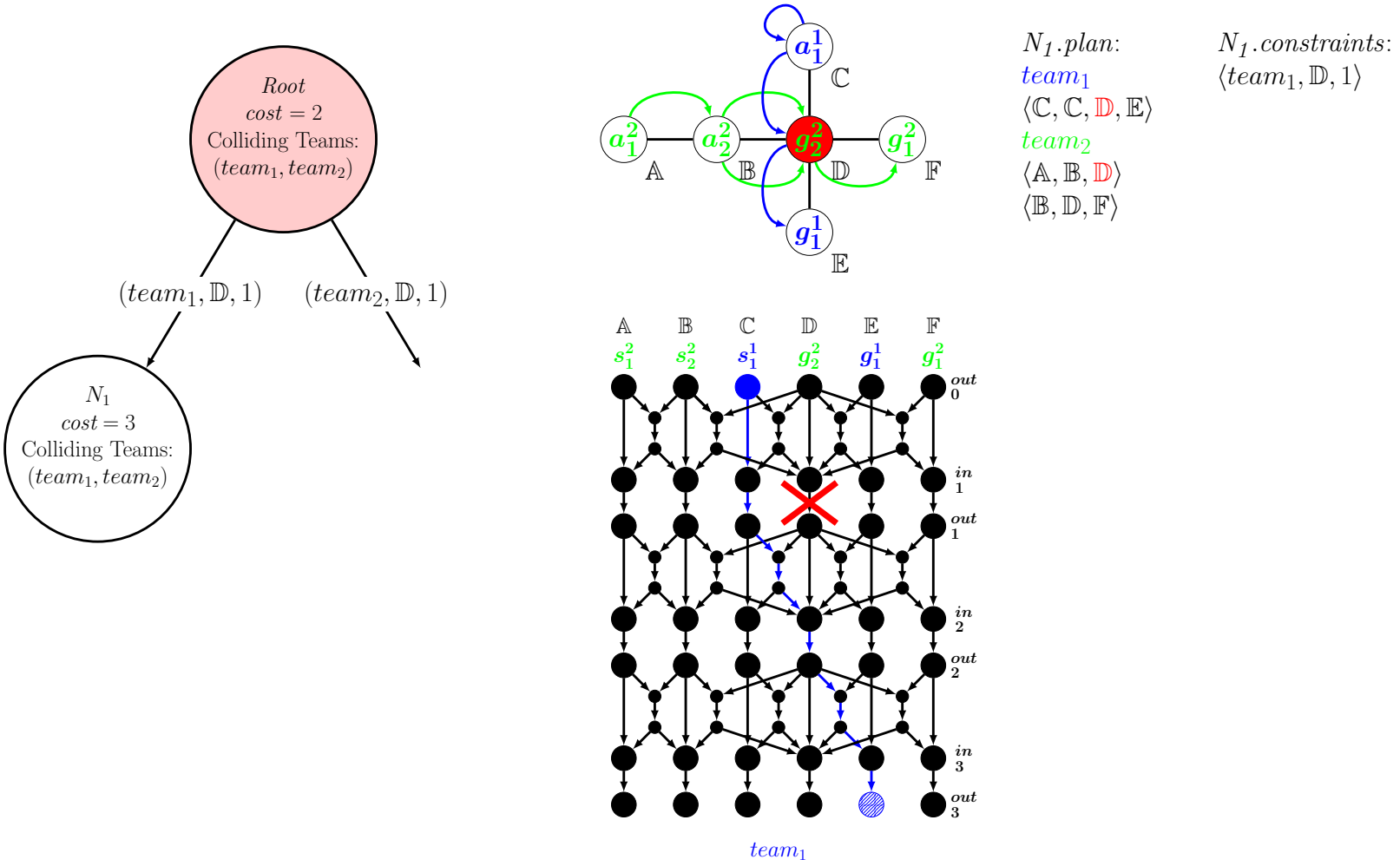


Figure 5.4: Construction of the left child node N_1 and the low-level search for $team_1$. The constraints and the plan of N_1 are shown on the top right. The constraints and the plan of the other nodes are not shown. The colliding teams of each node are shown to explain the tie breaking.

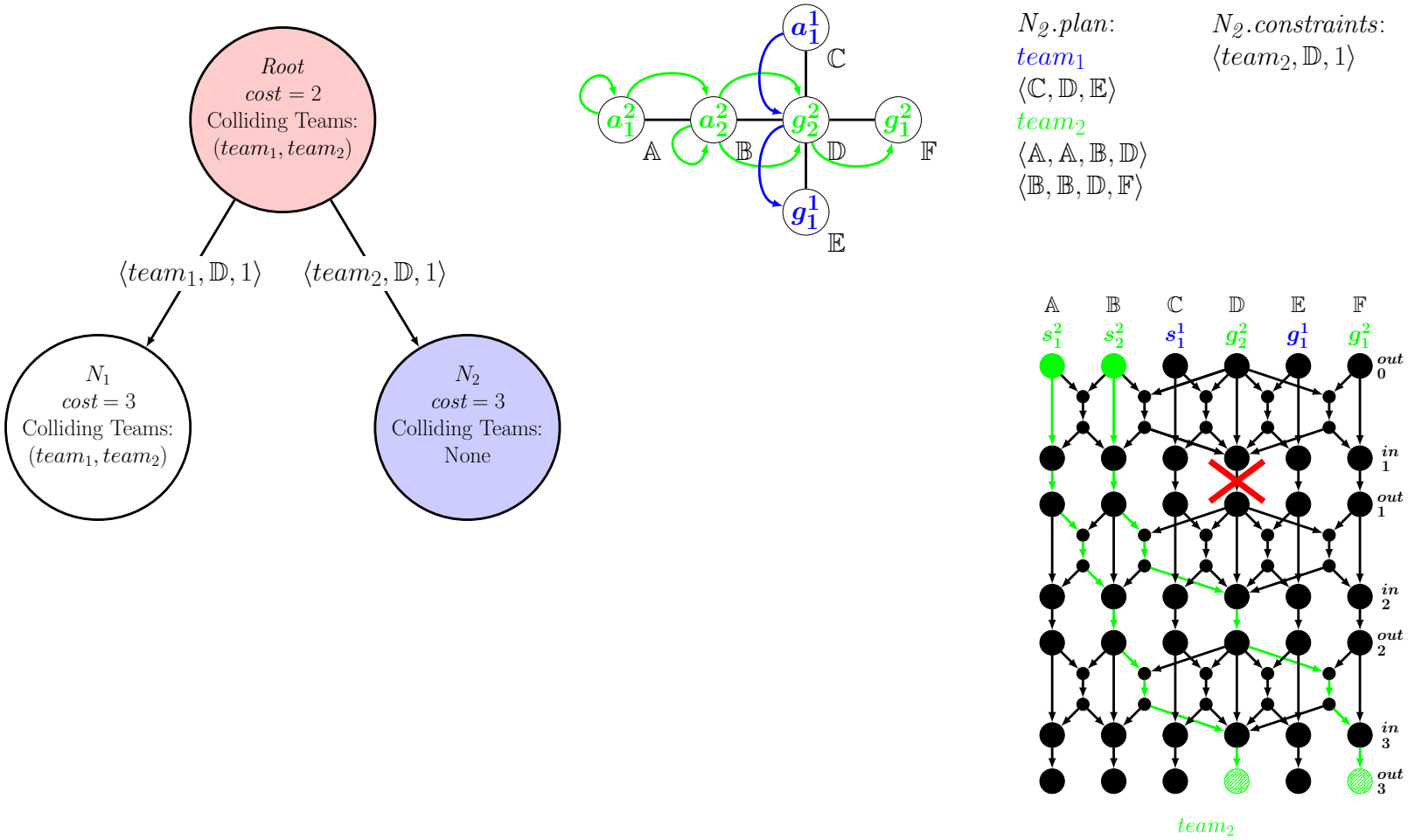


Figure 5.5: Construction of the right child node N_2 and the low-level search for $team_2$. The constraints and the plan of N_2 are shown on the top right. The constraints and the plan of the other nodes are not shown. The colliding teams of each node are shown to explain the tie breaking.

time-expanded flow network for the low-level search for $team_1$ on the bottom right, where edge $(\mathbb{D}_1^{in}, \mathbb{D}_1^{out})$ is removed to enforce the vertex constraint $\langle team_1, \mathbb{D}, 1\rangle$. *LowLevel*($team_1$, N_1) returns path $\pi_1^1 = \langle \mathbb{C}, \mathbb{C}, \mathbb{D}, \mathbb{E}\rangle$, that obeys N'.*constraints* and results in no collisions among agents in $team_1$. The plan N'.*plan* and the cost N'.*cost* of the child node $N' = N_1$ are updated accordingly [Lines 20-23]. Then, the right child node $N' = N_2$ is constructed in a similar way (Figure 5.5). In the next iteration, the two nodes N_1 and N_2 in OPEN have the same cost. The right child node $N = N_2$ is popped from OPEN because its plan has the smallest number of pairs of colliding teams. Since its plan N.*plan* has no collision, CBM returns its plan as the solution [Lines 12-13].

5.3.4 Avoiding to Create Collisions with Other Teams in the Low-Level Search

CBM actually implements *LowLevel*($team_k$,N) in a more sophisticated way to avoid creating collisions between agents in team $team_k$ and agents in other teams by adding edge weights to the T-step time-expanded flow network. CBM sets the weights of all edges in $\mathcal{E}'$ to zero initially and then modifies them as follows: (a) To reduce vertex collisions, CBM increases the weight of edge $(v_t^{in}, v_t^{out}) \in \mathcal{E}'$ by one for each vertex $v = \pi_{j'}^{k'}(t) \in V$ in the paths of node N with $k' \neq k$ to reduce the possibility of an agent of team $team_k$ occupying the same vertex at the same time step as an agent from a different team. (b) To reduce edge collisions, CBM increases the weight of edge $(v_t^{in}, w) \in \mathcal{E}'$ by one for each edge $(u = \pi_{j'}^{k'}(t), v = \pi_{j'}^{k'}(t+1)) \in E$ in the paths of node N with $k' \neq k$ (where w is the auxiliary vertex of the gadget that corresponds to edge (u, v) and time step t) to reduce the possibility of an agent of team $team_k$ moving along the same edge in a different direction but at the same time step as an agent from a different team.

CBM uses the procedure described above, except that it now uses a min-cost max-flow algorithm (instead of a max-flow algorithm) that finds a **flow of minimal weight among all feasible maximum flows**. In particular, it uses the successive shortest path algorithm (Goldberg & Tarjan, 1987), a generalization of the Ford-Fulkerson algorithm (Ford & Fulkerson, 1956) that uses Dijkstra's algorithm (Dijkstra, 1959) to find a path of minimal weight for one unit of flow. The complexity of the successive shortest path algorithm is $O(U(|\mathcal{E}'| + |\mathcal{V}'| \log |\mathcal{V}'|))$, where

$O(|\mathcal{E}'| + |\mathcal{V}'| \log |\mathcal{V}'|)$ is the complexity of Dijkstra's algorithm and U is the value of the feasible maximum flow, which is bounded from above by M_k. The number of times that the successive shortest path algorithm is executed is bounded from above by the chosen upper bound on the makespan of an optimal solution, which in turn is bounded from above by $O(|V|^3)$. Thus, each low-level search runs in polynomial time.

The low-level space-time A* search of CBS (see Section 3.2.2.2) uses a similar idea to return a time-minimal path with the fewest collisions with the paths of other agents.

5.3.5 Analysis of Properties

We use the following properties to prove that CBM is correct, complete, and optimal.

Property 5.1. There is a correspondence between all feasible integer flows of M_k units on the T-step time-expanded flow network constructed for team $team_k$ and node N and all paths for agents in team $team_k$ that a) obey the constraints of node N, b) move all agents in team $team_k$ from their start vertices to unique targets of their team, c) result in no collisions among agents in team $team_k$, and d) result in a team cost of team $team_k$ of at most T.

Proof. The property holds by construction and can be proved in a way similar to the one for the reduction of the (non-anonymous) MAPF problem to the integer multi-commodity flow problem (Yu & LaValle, 2013b):

Left to right: Assume that a flow is given that has the stated properties. Each unit flow from a source to a sink corresponds to a path through the time-expanded flow network from a unique source to a unique sink. Thus, it can be converted to a path for an agent such that all such paths together have the stated properties: Properties a and d hold by construction of the time-expanded flow network; Property b holds because a flow of M_k units uses all sources and sinks; and Property c holds since the flows neither share vertices nor edges.

Right to left: Assume that paths are given that have the stated properties. If necessary, we extend the paths by letting the agents wait at their targets. Each path now corresponds to a path through the time-expanded network (due to Properties a and d) from a unique source to a unique sink

(due to Property b) that does not share directed edges with the other such paths (due to Property c). Thus, it can be converted to a unit flow such that all such unit flows together respect the unit capacity constraints and form a flow of M_k units. □

Property 5.2. CBM generates only finitely many nodes.

Proof. The constraint added on Line 19 to a child node is different from the constraints of its parent node since the paths of its parent node do not obey it. Also, only finitely many different vertex and edge constraints exist since the graph is finite and there is a finite limit (bounded from above by $O(|V|^3)$) on the number of time steps. Overall, CBM creates a binary tree of finite depth and thus generates only finitely many nodes. □

Property 5.3. Whenever CBM inserts a node into OPEN, its cost is finite.

Proof (by induction). The property holds for the root node since the graph is connected. Assume that it holds for the parent node of some child node. The cost of the child node is the maximum of the cost of the parent node and the team cost of the team whose paths are re-planned for the child node. The cost of the parent node is finite due to the induction assumption. The team cost resulting from any returned paths by the low level search is finite as well. □

Property 5.4. Whenever CBM chooses a node on Line 10 with no collisions in its plan, then CBM correctly terminates with a solution with finite makespan of at most its cost.

Proof. The cost of the node is finite according to Property 5.3, and the makespan of its plan is at most its cost due to Line 22. □

Property 5.5. CBM chooses nodes on Line 10 in non-decreasing order of their costs.

Proof. CBM performs a best-first search on the high level, and the cost of a parent node is at most the cost of any of its child nodes due to Line 22 since the team cost of the team whose paths are re-planned (with an additional constraint) for the child node cannot decrease and the team costs of all other teams remain unchanged in the child node. □

Property 5.6. The smallest makespan of any solution that obeys the constraints of a parent node is at most the smallest makespan of any solution that obeys the constraints of any of its child nodes.

Proof. The solutions that obey the constraints of a parent node are a superset of the solutions that obey the constraints of any of its child nodes since the constraints of the parent node are a subset of the constraints of any of its child nodes. □

Property 5.7. The cost of a node is at most the makespan of any solution that obeys its constraints.

Proof (by induction). The paths for each team in the root node have the smallest possible team cost for that team. The property thus holds for the root node. Assume that it holds for the parent node N of any child node N' and that the paths for team $team_k$ were updated in the child node. Let x be the smallest makespan of any solution that obeys the constraints of the parent node and y be the smallest makespan of any solution that obeys the constraints of the child node. We show in the following that the cost of the parent node and the team costs of all teams for the paths of the child node are all at most y. Then, the cost of the child node is also at most y since it is the maximum of all these quantities, and the property holds. First, consider the cost of the parent node. The cost of the parent node is at most x due to the induction assumption, which in turn is at most y due to Property 5.6. Second, consider any team different from team $team_k$. Then, the team cost of the team for the paths of the child node is equal to the team cost of the team for the paths of the parent node (since the paths were not updated in the child node and are thus identical), which in turn is at most the cost of the parent node (since the cost of the parent node is the maximum of several quantities that include the team cost of the team for the paths of the parent node), which in turn is at most y (as shown directly above). Finally, consider team $team_k$. When the low level finds new paths for team $team_k$, it starts with T being the cost of the parent node, which is at most y (as shown directly above). Thus, the min-cost max-flow algorithm on a T-step time-expanded network constructed for team $team_k$ and the constraints of the child node finds a feasible integer flow of M_k units for $T \leq y$ since there exists a solution with makespan y

that obeys the constraints of the child node. The team cost of the corresponding paths for team $team_k$ is at most T due to Property 5.1. The cost of the child node is thus at most y, the smallest makespan of any solution that obeys its constraints. □

Theorem 5.3. CBM is correct, complete, and optimal.

Proof. *LowLevel*($team_k$, *Root*) on Line 4 solves an Anonymous MAPF sub-problem for $team_k$ for which a solution always exists (Yu & LaValle, 2013a). CBM thus never terminates unsuccessfully on Line 5.

Assume that no solution to a TAPF problem instance exists. Then, whenever CBM chooses a node on Line 10, the paths of the node have colliding agents (because otherwise a solution would exist due to Property 5.4). Thus, OPEN eventually becomes empty and CBM terminates unsuccessfully on Line 24 since it generates only finitely many nodes due to Property 5.2.

Now assume that a solution exists and the makespan of an optimal solution is x. Assume, for a proof by contradiction, that CBM does not terminate with a solution with makespan x. Thus, whenever CBM chooses a node on Line 10 with a cost of at most x, the paths of the node have colliding agents (because otherwise CBM would correctly terminate with a solution with makespan at most x due to Property 5.4). A node whose constraints the optimal solution obeys has a cost of at most x due to Property 5.7. The root note is such a node since the optimal solution trivially obeys the (empty) constraints of the root node. Whenever CBM chooses such a node on Line 10, the paths of the node have colliding agents (as shown directly above since its cost is at most x). CBM thus generates the child nodes of this parent node, the constraints of at least one of which the optimal solution obeys and which CBM thus inserts into OPEN with a cost of at most x. Since CBM chooses nodes on Line 10 in non-decreasing order of their costs due to Property 5.5. it chooses infinitely many nodes on Line 10 with costs of at most x, which is a contradiction with Property 5.2. □

5.4 Experiments

In this section, we describe the results of four experiments on a 2.50 GHz Intel Core i5-2450M PC with 6 GB RAM. First, we compare CBM to four other TAPF or MAPF algorithms. Second, we study how CBM scales with the number of agents in each team. Third, we study how CBM scales with the number of agents. Fourth, we test CBM on a warehouse map.

5.4.1 Experiment 1: Alternative Algorithms

We compared our optimal TAPF algorithms CBM to two optimal (Non-Anonymous) MAPF algorithms, namely (1) an implementation of CBS for makespan minimization provided by the authors of Sharon et al. (2015) and (2) an implementation of the ILP-based MAPF algorithm provided by the authors of Yu and LaValle (2013b), and two optimal TAPF algorithms, namely (1) an unweighted version of CBM that runs the polynomial-time max-flow algorithm on a time-expanded network without edge weights (instead of the min-cost max-flow algorithm on a time-expanded network with edge weights) on the low level and (2) an ILP-based TAPF algorithm that casts a TAPF problem instance as a series of integer multi-commodity flow problems as described in Section 5.2, each of which it models as an ILP and solves with the ILP solver Gurobi 6.0 (www.gurobi.com).

For Experiment 1, each team consists of 5 agents but the number of agents varies from 10 to 50 in increments of 5, resulting in 2 to 10 teams. For each number of agents, we generated 50 TAPF problem instances from 50 different 30×30 2D 4-neighbor grids with 10% randomly blocked cells by randomly assigning unique start cells to agents and unique targets to teams. For the MAPF algorithms, we converted each TAPF problem instance to a (Non-Anonymous) MAPF problem instance by randomly assigning targets of each team to agents in the same team.

Table 5.1 reports the average makespans and runtimes (in seconds) as well as the success rates over the problem instances that were solved within a runtime limit of 5 minutes each. Gray entries indicate that some problem instances were not solved within the time limit, while dashed

Table 5.1: Results for different TAPF and MAPF algorithms on 30×30 4-neighbor grids with randomly blocked cells for different numbers of agents.

agents	CBM (TAPF)			Unweighted CBM (TAPF)			ILP (TAPF)			CBS (MAPF)			ILP (MAPF)		
	makespan	time (s)	success	makespan	time (s)	success	makespan	time (s)	success	makespan	time (s)	success	makespan	time (s)	success
10	22.34	0.34	1	22.08	0.41	0.72	22.34	18.24	1	36.36	0.03	1	36.36	8.66	1
15	23.88	0.57	1	24.64	1.06	0.44	23.88	35.44	1	37.32	0.05	1	37.32	15.31	1
20	25.06	0.78	1	23.73	2.06	0.22	24.74	62.85	0.94	39.84	0.55	1	39.84	30.30	1
25	25.20	1.07	1	22.25	1.58	0.08	24.76	88.55	0.82	40.44	0.12	1	40.44	43.76	1
30	26.26	1.71	1	31	6.73	0.02	24.70	108.75	0.66	41.92	0.21	1	41.92	65.86	1
35	26.50	1.92	1	-	-	0	24.65	121.99	0.46	42.50	1.55	1	42.50	81.83	1
40	27.60	2.95	1	-	-	0	25.29	152.98	0.14	43.69	4.82	0.98	43.53	115.53	0.98
45	27.20	3.66	1	-	-	0	24.29	161.52	0.14	42.41	2.60	0.92	42.37	133.47	0.98
50	27.90	5.32	1	-	-	0	24.50	161.95	0.04	43.96	7.95	0.96	42.86	166.99	0.86

entries indicate that all problem instances were not solved within the time limit. CBM solved all TAPF problem instances within the time limit.

5.4.1.1 CBS and the ILP-Based MAPF Algorithm

Both MAPF algorithms solve most of the MAPF problem instances within the time limit. The runtimes of CBM and CBS are similar because, on the low level, both the min-cost max-flow algorithm of CBM (for a single team) and the A* algorithm of CBS (for a single agent) are fast. Optimal solutions of the TAPF problem instances have smaller makespans than optimal solutions of the corresponding MAPF problem instances due to the freedom of being able to assign targets to agents optimally for the TAPF problem instances rather than assigning them randomly for the MAPF problem instances.

5.4.1.2 Unweighted CBM

Unweighted CBM solves less than half of all TAPF problem instances within the time limit if the number of agents is larger than 10 due to the large number of collisions among agents in different teams produced by the max-flow algorithm on the low level in tight spaces with many agents, which results in a large number of node expansions by CBM on the high level. We conclude that biasing the search on the low level is important for CBM to solve all TAPF problem instances within the time limit.

5.4.1.3 ILP-Based TAPF Algorithm

The ILP-based TAPF algorithm solves less than half of all TAPF problem instances within the runtime limit if the number of agents is larger than 30, and its runtime is much larger than that of CBM. The success rates and runtimes of the the ILP-based TAPF algorithm tend to be larger than those of the ILP-based MAPF algorithm. The flow conservation constraints for the sources and sinks in the ILP formulation of the MAPF problem instance are for a flow of one unit, while those in the ILP formulation of the TAPF problem instance are for a flow of M_k units (for each team $team_k$), which make the ILP for TAPF harder to solve than for MAPF.

Table 5.2: Results for CBM on 30×30 4-neighbor grids with randomly blocked cells for different team sizes.

teams	team size	makespan	time (s)	success
1	100	7.58	0.76	1
2	50	11.10	1.37	1
4	25	15.90	2.94	1
5	20	17.12	2.10	1
10	10	23.04	3.93	1
20	5	29.32	5.33	1
25	4	30.88	6.49	1
50	2	39.76	12.27	1
100	1	46.72	22.83	1

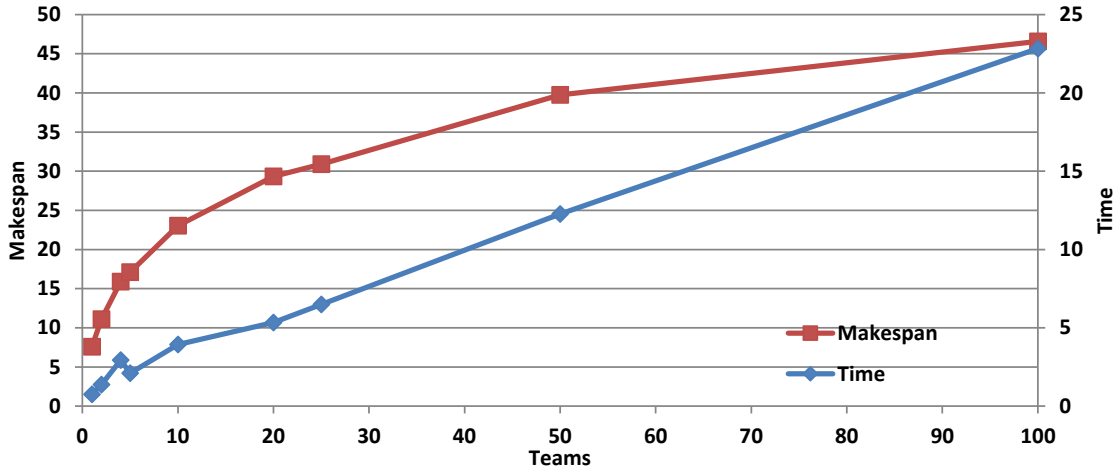


Figure 5.6: Makespans and runtimes for CBM on 30×30 4-neighbor grids with randomly blocked cells for different team sizes.

5.4.2 Experiment 2: Team Size

We experimented with CBM on TAPF problem instances with a fixed total number of agents but different numbers of teams. For Experiment 2, there are 100 agents but the number of agents in a team (team size) varies from 100 to 1, resulting in 1 to 100 teams. For each number of agents, we generated 50 TAPF problem instances from 50 different 30×30 2D 4-neighbor grids with 10% randomly blocked cells by randomly assigning unique start cells to agents and unique targets to teams.

Table 5.2 reports the average makespans and runtimes (in seconds) as well as the success rates over the problem instances that were solved within a runtime limit of 5 minutes each. CBM solved all TAPF problem instances within the time limit. For large team sizes and thus small numbers of teams, the makespans are small because CBM has more freedom to assign targets to agents. The runtimes are also small because the runtime of CBM is exponential in the number of collisions it resolves on the high level (Sharon et al., 2015) and CBM tends to resolve fewer collisions on the high level for smaller numbers of teams. Thus, it is advantageous for teams to consist of as many agents as possible. Figure 5.6 shows a spectrum that spans from one single team of 100 agents on the left side of the x-axis (where CBM solves Anonymous MAPF problem instances) to 100 teams of one agent each (100 single agents) on the right side of the x-axis (where CBM is solving MAPF problem instances). The average runtimes tend to grows linearly in the number of teams, which indicates that CBM exploits the problem structure of TAPF problem instances well.

5.4.3 Experiment 3: Number of Agents and Scalability

We experimented with CBM on TAPF problem instances with large numbers of agents to evaluate how CBM scale in the number of agents. For Experiment 3, each team consists of 5 agents but the number of agents varies from 100 to 550 in increments of 50, resulting in 20 to 110 teams. For each number of agents, we generated 50 TAPF problem instances from 50 different 30×30 2D 4-neighbor grids with 10% randomly blocked cells by randomly assigning unique start cells to agents and unique targets to teams.

Table 5.3 reports the average makespans and runtimes (in seconds) as well as the success rates over the problem instances that were solved within a runtime limit of 5 minutes each. Gray entries indicate that some problem instances were not solved within the time limit, while dashed entries indicate that all problem instances were not solved within the time limit. Figure 5.7 shows the number of agents on the x-axis and the success rate on the y-axis. For 400 agents or fewer, the success rate is larger than 90%. As the number of agents increases, the success rate decreases and the makespan and runtime both increase due to the increasing number of collisions among

Table 5.3: Results for CBM on 30×30 4-neighbor grids with randomly blocked cells for different numbers of agents.

agents	teams	makespan	time (s)	success
100	20	30.10	6.79	1
150	30	29.78	11.47	1
200	40	32.12	16.48	1
250	50	31.38	26.19	0.96
300	60	32.40	38.94	0.96
350	70	32.61	53.39	0.98
400	80	33.28	93.01	0.94
450	90	33.15	164.13	0.78
500	100	35.27	277.24	0.22
550	110	-	-	0

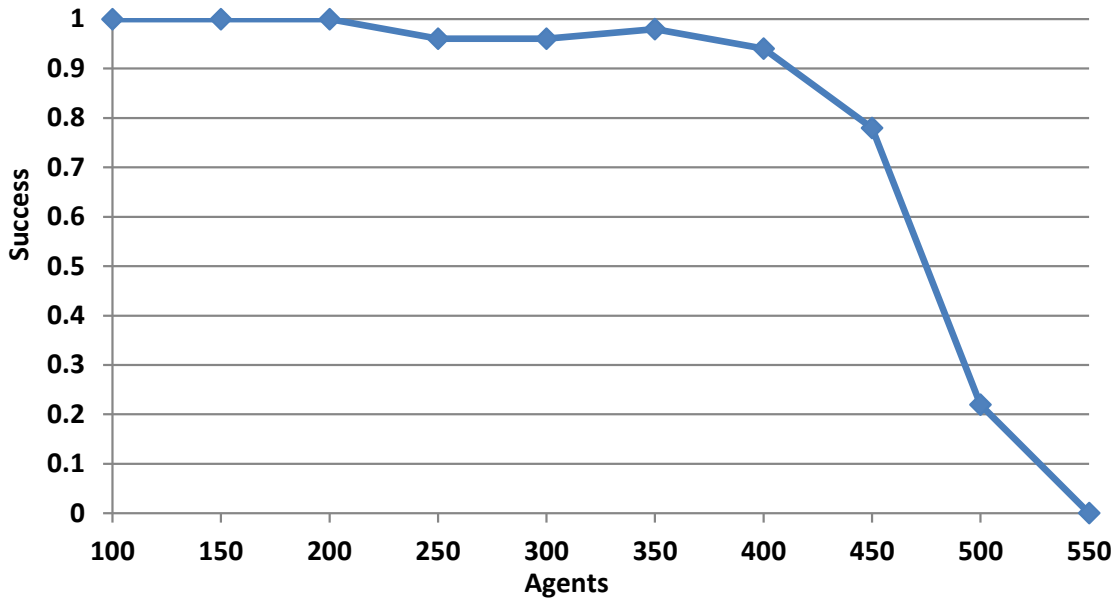


Figure 5.7: Success rates for CBM on 30×30 4-neighbor grids with randomly blocked cells for different numbers of agents.

agents in different teams produced by the min-cost max-flow algorithm on the low level. For 500 agents, for example, more than 60% of the unblocked cells are occupied by agents and thus many start cells of agents are also targets for other agents, resulting in many collisions.

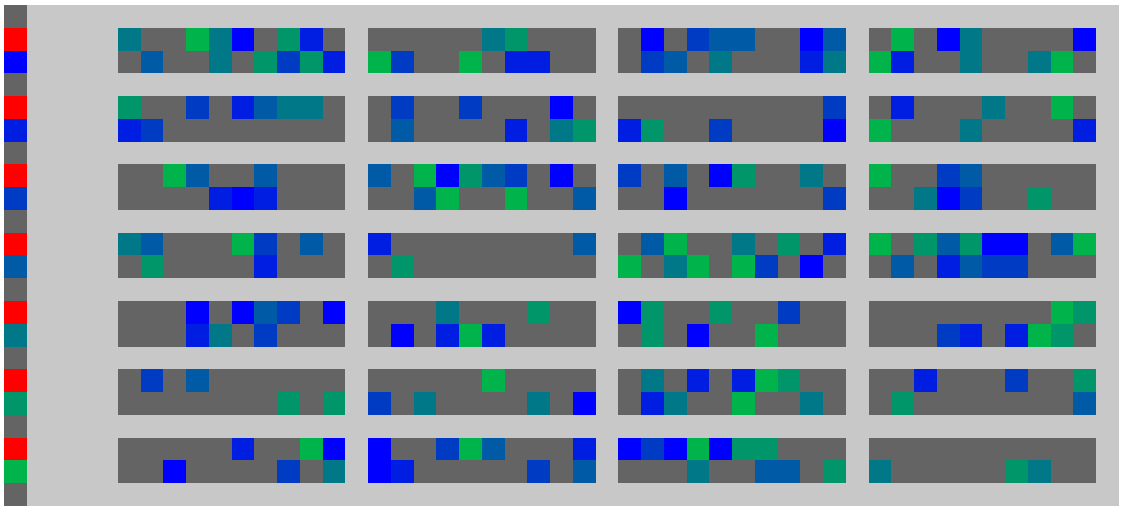


Figure 5.8: A randomly generated TAPF problem instance on the warehouse map.

5.4.4 Experiment 4: Warehouse Map

We experimented with CBM on TAPF problem instances on the warehouse map shown in Figure 1.1 that represents the layout of part of an Amazon Robotics automated warehouse system (Wurman et al., 2008). This map has been used to generate problem instances in recent research on MAPF and TAPF (Cohen et al., 2016; Felner et al., 2018). The TAPF problem instances we used have also been considered as benchmark TAPF problem instances by recent research (Nguyen, Obermeier, Son, Schaub, & Yeoh, 2017). Figure 5.8 shows a randomly generated TAPF problem instance on this warehouse map, represented as a 2D 4-neighbor grid. The light gray cells represent free space and are unblocked. The dark gray cells represent storage locations of inventory pods and are blocked. There are 7 inventory stations on the left side. The red cells are their exits, and the other 7 cells with blue-green gradient colors are their entrances. Agents can enter and leave the inventory stations one at a time through their entrances and exits, respectively. The incoming and outgoing queues of the inventory stations are not modeled for simplicity. The cells with blue-green gradient colors in the storage area are occupied by agents. Each agent needs to carry the inventory pod in its current cell to the inventory station of the same color.

For Experiment 4, we generated 50 TAPF problem instances on the warehouse map. Each problem instance has 420 agents. 210 "incoming" agents start at randomly determined storage locations: 30 agents each need to move their inventory pods to the 7 inventory stations. In order to create difficult simulated warehouse problem instances, we generated the start cells of these agents randomly among all storage locations rather than clustering them according to their target inventory stations. 210 "outgoing" agents start at the inventory stations: 30 agents each need to move their inventory pods from the 7 inventory stations to the storage locations vacated by the incoming agents. The problem is to assign the vacated storage locations to the 210 outgoing agents and plan collision-free paths for all 420 agents in a way such that the makespan is minimized. The incoming agents that have the same inventory station as target are a team (since they can arrive at the inventory station in any order), and all outgoing agents are a team.

So far, we have assumed that, for any TAPF problem instance, all start vertices are unique, all targets are unique, and each one of the teams is given the same number of targets as there are agents in the team, but these assumptions are not satisfied here: (a) The outgoing agents that start at the same inventory station all start at its exit. In this case, we change the construction of the T-step time-expanded network for the team of outgoing agents so that there is a supply of one unit at vertex $v_t^{out} \in \mathcal{V}'$ for all $t = 0, \ldots, 29$ and all vertices $v \in V$ that correspond to exits of inventory stations. This construction forces the outgoing agents that start at the same inventory station to leave it one after the other during the first 30 time steps. No further changes are necessary. (b) The incoming agents that have the same inventory station as target all end at its entrance. In this case, we change the construction of the T-step time-expanded network for each team of incoming agents so that there is an auxiliary vertex with a demand of 30 units and vertex $v_t^{out} \in \mathcal{V}'$ for all $t = 0, \ldots, T$ is connected to the auxiliary vertex with an edge with unit capacity and zero edge weight, where $v \in V$ corresponds to the entrance of the inventory station. This construction forces the incoming agents to enter the inventory station at different time steps. No further changes are necessary. (c) There could be more empty storage locations than outgoing agents. In this case, no changes are necessary.

For the TPAF problem instance shown in Figure 5.8, CBM returned a solution with makespan 65 in 25.97 seconds. The cost of the shortest path for both green agents near the top-right corner to the green inventory station in the bottom-left corner is 64. Thus, at least one of them has to wait for at least one time step to enter the green inventory station (thus also validating that the solution found by CBM is optimal).

In general, CBM found solutions for 40 of the 50 simulated warehouse problem instances within a runtime limit of 5 minutes each, yielding a success rate of 80%. The average makespan over the solved simulated warehouse problem instances is 63.73, and the average runtime is 39.26 seconds. Since even bounded-suboptimal (Non-Anonymous) MAPF algorithms that were specifically designed for simulated Amazon Robotics automated warehouse systems do not scale well to hundreds of agents (Cohen, Uras, & Koenig, 2015), we conclude that CBM is a promising TAPF algorithm for applications of real-world scale.

5.5 Including Kinematic Constraints

Since any solution of a TAPF problem instance is also a solution of a MAPF problem instance on the same graph for a suitable assignment of targets to agents, we can directly use MAPF-POST (see Section 3.3) to transform a TAPF solution returned by any TAPF algorithm into a plan-execution schedule that can be safely executed by teams of real-world agents. Specifically, our recent research (Hönig, Kumar, Ma, et al., 2016) (not covered as a contribution in this dissertation) uses MAPF-POST in a post-processing step to transform a TAPF solution into a plan-execution schedule that works for the coordination of target-assignment and path-planning operations of teams of differential-drive robots. The resulting plan-execution schedule takes kinematic constraints of these robots into account and provides a guaranteed safety distance between them. The results validate the hypothesis that our TAPF algorithms have the potential to be eventually deployed in real-world applications of multi-agent systems for the one-shot coordination of the autonomous target-assignment and path-planning operations of teams of agents.

5.6 Summary

In this chapter, we formalized and studied TAPF, a new variant of MAPF that models the one-shot combined target-assignment and path-planning problem for multiple teams of agents. We developed optimal TAPF algorithms to bridge the gap between the extreme cases of Anonymous MAPF (one team of agents) and (Non-Anonymous) MAPF (teams of one agent each). We demonstrated how TAPF can be solved by generalizing the flow-based ILP formulation for MAPF (see Section 3.2.3). We then presented CBM, that simultaneously assigns targets to agents and plans collision-free paths for the agents. CBM exploits the combinatorial structure of TAPF by breaking it down to the polynomial-time solvable sub-problems of coordinating different agents in single teams and the NP-hard sub-problem of coordinating different teams. It uses a min-cost max-flow algorithm (Goldberg & Tarjan, 1987) on a time-expanded network for the polynomial-time solvable sub-problems and CBS for the NP-hard sub-problem. Theoretically, we proved that CBM is correct, complete, and optimal. Experimentally, we compared CBM to the ILP-based algorithm. We also demonstrated the scalability of CBM for TAPF problem instances with dozens of teams and hundreds of agents and adapted it to TAPF problem instances on a warehouse map. MAPF-POST can be used in a post-processing step to transform TAPF solutions, like MAPF solutions, into plan-execution schedules that can be safely executed by teams of real-world agents.

To summarize, in this chapter, we validated the hypothesis that formalizing and studying new variants of MAPF can result in new algorithms for the one-shot combined target-assignment and path-planning problem for teams of agents which can potentially be applied to and thus benefit real-world applications of multi-agent systems.

Chapter 6

Long-Term Target Assignment and Path Planning

In this chapter, we present the third major contribution of this dissertation. Specifically, we formalize and study a new variant of MAPF, called *Multi-Agent Pickup and Delivery (MAPD)*, that models the long-term coordination of autonomous target-assignment and path-planning operations of teams of agents. We demonstrate how environmental characteristics can be utilized to ensure long-term robustness for well-formed MAPD problem instances, a class of MAPD problem instances that are realistic for many real-world applications of multi-agent systems. We present three novel MAPD algorithms that can utilize such environmental characteristics of well-formed MAPD problem instances. Theoretically, we proved that they are long-term robust for all well-formed MAPD problem instances. Experimentally, we test them with up to 500 agents and 1,000 tasks. Therefore, these results validate the hypothesis that formalizing and studying new variants of MAPF can result in new algorithms for the long-term target-assignment and path-planning problem. We follow our formalization of MAPD in Section 2.5.

The remainder of this chapter is structured as follows. In Section 6.1, we reiterate the motivation behind formalizing and studying MAPD. In Section 6.3, we introduce the concept of well-formedness that allows for long-term robustness. In Section 6.4, we present two decoupled MAPD algorithms that are long-term robust for all well-formed MAPD problem instances

This chapter is based on Ma, H., Li, J., Kumar, T. K. S., & Koenig, S. (2017). Lifelong multi-agent path finding for online pickup and delivery tasks. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 837–845).

and discuss their extensions. In Section 6.5, we present a centralized MAPD algorithm that is also long-term robust for all well-formed MAPD problem instances and discuss its extensions. In Section 6.6, we experimentally compare these MAPD algorithms in a simulated warehouse environment. Finally, we summarize the contributions of this chapter in Section 6.7.

6.1 Introduction

In this chapter, we develop algorithmic techniques for the long-term coordination of autonomous target-assignment and path-planning operations of teams of agents by studying MAPD (see Section 2.5). MAPD generalizes the one-shot problems MAPF, Anonymous MAPF, and TAPF to a long-term problem. In MAPD, M agents have to attend to a stream of incoming tasks. Each task enters the system at an unknown time and is characterized by two targets, namely a pickup vertex and a delivery vertex. A free agent (Definition 2.25), namely one that is currently not executing any task, can be assigned an unexecuted task. To execute the task, the agent has to first move from its current vertex to the pickup vertex of the task, become occupied (Definition 2.25) and start to execute the task upon reaching the pickup vertex of the task, and then move from the pickup vertex to the delivery vertex of the task, while avoiding collisions with other agents. For example, in an Amazon Robotics automated warehouse system (see Figure 1.1 for an example), warehouse robots have to attend to tasks of relocating inventory pods from their storage locations to inventory stations or vice versa. A warehouse robot that is not carrying an inventory pod can be assigned any task of relocating an inventory pod. To execute the task, it then has to move from its current location to the pickup location of the inventory pod and from there to the delivery location of the inventory pod, while avoid collisions with other warehouse robots.

There are three benefits of modeling "pickup-and-delivery" tasks that have an intermediate target (the pickup vertex) and a final target (the delivery vertex) each instead of "navigation" tasks that have only one target each: (a) Modeling "pickup-and-delivery" tasks results in a mix

of both anonymous and non-anonymous agents (see Section 2.5.1), while modeling "navigation" tasks results in only anonymous agents, which limits its generalizability. (b) The resulting MAPD algorithms still apply directly to "navigation" tasks because each such task is a special case of a "pickup-and-delivery" task with the pickup and delivery vertices being the same vertex. (c) Modeling "pickup-and-delivery" tasks also makes it easy to explain the resulting MAPD algorithms, even though these algorithms can easily be generalized to cases where the tasks have multiple (ordered) intermediate targets and a subsequent final target each (but it is much harder to generalize algorithms for "navigation" tasks to these cases).

MAPD also models the long-term coordination of both target-assignment and path-planning operations for many other real-world applications of multi-agent systems, including autonomous aircraft-towing vehicles (Morris et al., 2016) that constantly assign incoming towing tasks and tow aircraft between airport terminal gates and runways and other multi-robot systems that use fleets of forklift robots (Pecora et al., 2018; Salvado et al., 2018) or teams of service robots (Ahmadi & Stone, 2006; Khandelwal et al., 2017; Veloso et al., 2015) to relocate items between different locations. These real-world applications require the assignment of tasks to agents and the planning of collision-free paths both in a *lifelong* and *online* setting. In a lifelong setting, agents have to attend to a stream of tasks. Therefore, agents cannot stay idly at their targets after they finish executing tasks. In an online setting, tasks can enter the system at any time. Therefore, assigning tasks to and planning paths for the agents cannot be done in advance but rather need to be done online. Solving the combined target-assignment and path-planning problem optimally in a lifelong setting, even when all the tasks are known a priori, is computationally challenging as the number of tasks becomes large. For example, a recent CBS-based algorithm requires two minutes of runtime on average to compute optimal solutions for 4 agents and 4 tasks (Henkel, Abbenseth, & Toussaint, 2019), assuming that the tasks are known a priori. Moreover, solving the combined target-assignment and path-planning problem optimally in an online setting is impossible, as indicated by the competitive analysis result for the online target-assignment problem (Azar et al., 1995; Kalyanasundaram & Pruhs, 1993). Therefore, unlike the research on optimal algorithms for one-shot coordination problems, such as MAPF, PERR, and TAPF, we focus on

developing MAPD algorithms that are suboptimal but long-term robust (Definition 2.29) in this chapter.

As we have previously discussed in Section 1.1, solving the long-term combined target-assignment and path-planning problem MAPD requires repeatedly solving both the one-shot target-assignment and the one-shot path-planning sub-problems as new tasks are added to the system. However, existing research has considered only either the one-shot coordination problems, such as MAPF, PERR, and TAPF, or the long-term target-assignment problem. In particular, as we have discussed in Section 3.4.3, existing research (Gerkey & Matarić, 2002; Khuller et al., 1994; Parker, 1998; 1999; Werger & Matarić, 2000) in the operations research and robotics communities has studied the long-term target-assignment problem extensively under the names "Online Assignment" or "Iterative Multi-Robot Task Allocation" and reduces it to a sequence of one-shot target-assignment problems. Therefore, one can potentially solves the long-term combined target-assignment and path-planning problem by reducing it to a sequence of one-shot target-assignment and path-planning sub-problems in a similar way and adapt one-shot target-assignment and path-planning algorithms to an online setting to solve these one-shot sub-problems. However, we show in Section 6.2 that a trivial combination of target-assignment and path-planning algorithms can result in deadlocks and is thus not long-term robust. We have also mentioned in Section 3.1.2.3 that some research on Prioritized Planning, including Cooperative A* (see Section 3.2.1), has utilized environmental characteristics to guarantee completeness for well-formed MAPF problem instances (Cáp et al., 2015; Turpin, Mohta, et al., 2014; Wang & Botea, 2011; Yu, 2017), but for the one-shot path-planning problem only. Inspired by the above results, we thus examine how and how well these results can be applied to the long-term combined target-assignment and path-planning problem MAPD and how long-term robustness can be guaranteed, for example, by utilizing environmental characteristics.

Therefore, we demonstrate how MAPD algorithms can utilize environmental characteristics to ensure long-term robustness for *well-formed* MAPD problem instances (Definition 6.1), a class of MAPD problem instances analogous to well-formed MAPF problem instances (Definition 3.1) that are realistic for many real-world multi-agent systems. Well-formed MAPD

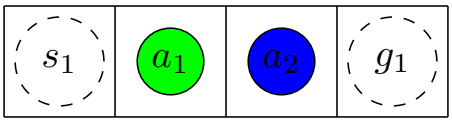


Figure 6.1: Example of an unsolvable MAPD problem instance.

problem instances capture the important graph properties of given environments of many multi-agent systems that allow MAPD algorithms to avoid deadlocks via a simple deadlock-avoidance mechanism: Agents should only be allowed to park (stay for a long period) at vertices where they cannot block other agents. For example, in an automated warehouse system, warehouse robots are only allowed to charge batteries or pick up and drop off inventory pods in locations where they cannot block other robots. We present two decoupled algorithms, Token Passing (TP) and Token Passing with Task Swaps (TPTS), and one centralized MAPD algorithm, CENTRAL. As suggested before, these MAPD algorithms repeatedly apply one-shot target-assignment and path-planning algorithms to a sequence of one-shot sub-problems and utilize environmental characteristics to combine the target-assignment and the path-planning algorithms in a smart way to avoid deadlocks. We show that all of these MAPD algorithms are thus long-term robust for well-formed MAPD problem instances. We compare them experimentally in a simulated warehouse environment with up to 500 agents and 1,000 tasks, thus showcasing their advantages and disadvantages for the long-term coordination of target-assignment and path-planning operations for large-scale multi-agent systems.

6.2 Motivating MAPD Examples

Not every MAPD problem instance is solvable. Figure 6.1 shows an example of a MAPD problem instance on a 2D 4-neighbor grid with two free agents a_1 and a_2 and one task τ_1 with pickup vertex s_1 and delivery vertex g_1 that is added to the system at time step 0. Colored circles are agents. Dashed circles are pickup and delivery vertices. Neither of the agents can execute task τ_1.

Even if a MAPD problem instance is solvable, a trivial concatenation of individual solutions for the one-shot target-assignment and one-shot path-planning sub-problems can result in deadlocks. We consider the MAPD problem instance shown in Figure 2.9, where a direct application of one-shot target-assignment and one-shot path-finding algorithms might result in a deadlock. For example, if the target assignment lets agent a_1 execute task τ_1 and assigns agent a_2 its current vertex, then there are no collision free paths for the agents because, otherwise, the paths end at the same vertex, that is, the resulting one-shot path-planning problem is unsolvable. One possible solution is to (still) assign task τ_1 to agent a_1 at time step 0, move agent a_2 from vertex $\mathbb{E}$ to vertex $\mathbb{B}$, and then move agent a_1 from the pickup vertex $s_1 = \mathbb{A}$ to the delivery vertex $g_1 = \mathbb{E}$ (see Section 2.5.2 for the complete solution). The intuition is to design a mechanism that lets agent a_2 move away from vertex $\mathbb{E}$, which is the delivery vertex of a task, to another vertex v and rest (that is, stay for a long period without an intention to move away) there so that it cannot block agent a_1 from executing the task. Ideally, one can guarantee that agent a_2 can always find such a vertex v and a collision-free path from its current vertex to vertex v and that it should not block agents from executing other tasks while resting at vertex v

6.3 Utilizing Environmental Characteristics: Well-Formedness

We now demonstrate how the environmental characteristics, namely well-formedness, of a class MAPD problem instances can provide a sufficient condition to allow for long-term robustness. Our MAPD algorithms then utilize these environmental characteristics to guarantee long-term robustness.

The definition of well-formed MAPD problem instances is inspired by that of well-formed MAPF problem instances (Definition 3.1). As we explained in the above motivating example, the intuition is that agents should only be allowed to *rest* (that is, stay for a long period without an intention to move away) at vertices, called endpoints, where they cannot block other agents. For example, storage locations of inventory pods are typically placed in a warehouse so as not to block traffic, and office workspaces are also typically placed in office environments so as not to

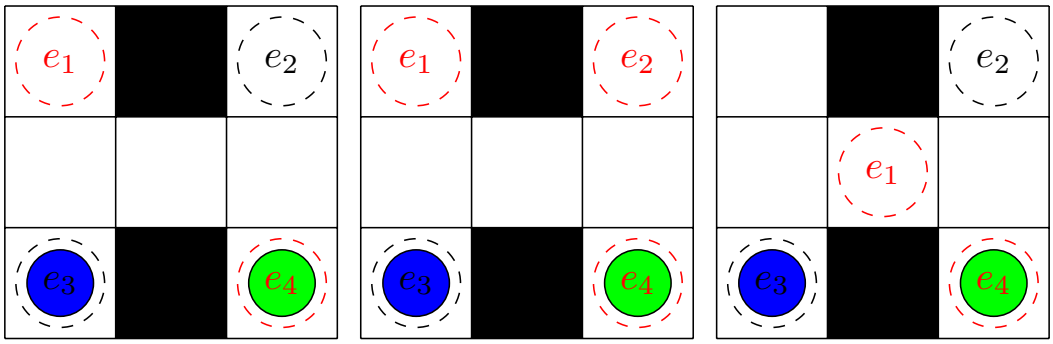


Figure 6.2: Three MAPD problem instances.

block traffic. The set V_{ep} of endpoints of a MAPD problem instance contains all initial vertices of agents, all pickup and delivery vertices of tasks, and perhaps additional designated vertices that allow the agents to rest. Let V_{tsk} denote the set of all pickup and delivery vertices of tasks, called the task endpoints. The set $V_{ep} \setminus V_{tsk}$ is called the set of non-task endpoints.

Definition 6.1. A MAPD problem instance is *well-formed* if and only if:

1. The number of tasks is finite.

2. There are no fewer non-task endpoints than the number of agents.

3. For any two endpoints, there exists a path between them that traverses no other endpoints.

Well-formed MAPD problem instances (with at least one task) have at least $M+1$ endpoints. Figure 6.2 shows three MAPD problem instances. Black cells are blocked. Blue and green circles are the initial vertices of agents. Red dashed circles are task endpoints. Black dashed circles are non-task endpoints. We assume each of the MAPD problem instances has finitely many tasks. The MAPD problem instance on the left is well-formed. The MAPD problem instance in the center is not well-formed because there are two agents but only one non-task endpoint. The MAPD problem instance on the right is not well-formed because all paths between endpoints e_2 and e_3 (or e_3 and e_4) traverse endpoint e_1. We design MAPD algorithms in the following sections that are long-term robust for all well-formed MAPD problem instances.

6.4 Decoupled MAPD Algorithms

In this section, we present first a simple decoupled MAPD algorithm, called Token Passing (TP), and then one of its improved versions, called Token Passing with Task Swaps (TPTS), that is more effective. Similar to decoupled MAPF algorithms (see Section 3.1.2.3), decoupled MAPD algorithms make decisions for one agent at a time, where each agent assigns itself a task and computes a path given some global information about the system.

6.4.1 TP

TP is based on ideas similar to the ones used by an existing greedy algorithm (Kalyanasundaram & Pruhs, 1993) for the long-term target-assignment problem, where targets are greedily assigned to the agents whenever there are agents available to navigate to targets, and Cooperative A* (see Section 3.2.1) for the one-shot path-planning problem, where agents plan their paths one after the other. The task set of TP only contains all tasks that are not assigned to agents. We describe a version of TP that uses token passing and can thus potentially be extended to a fully distributed MAPD algorithm via any token-based mutual exclusion mechanism (Suzuki & Kasami, 1985). The token is a synchronized shared block of memory that contains the current paths of all agents, the task set, and the task assignments that record which tasks are currently assigned to which agent. Similar to the definition of a path for one-shot problems, all MAPD algorithms presented in this chapter, including TP, always assume that an agent rests at the last vertex of its path in the token when it reaches the vertex. The idea of passing a token has previously been used to develop COBRA (Cáp et al., 2015), which is a MAPF-like algorithm that does not take into account that pickup or delivery vertices of tasks can be occupied by agents not executing them and can thus result in deadlocks for MAPD.

Algorithm 6.1 shows the pseudocode of TP, where $loc(a_i)$ denotes the current vertex of agent a_i. Agent a_i finds all paths via space-time A* searches (Section 3.2.1.1) that do not result in it colliding with other agents that move along their paths in the token. Since time-minimal paths need to be found only to endpoints, the arrival times at all endpoints from all vertices (that ignore

Algorithm 6.1: TP

```
   /* system executes now */
1  Initialize token with the (trivial) path ⟨loc(a_i)⟩ for each agent a_i;
2  while true do
3      Add all new tasks, if any, to the task set 𝒯;
4      while agent a_i exists that requests token do
           /* system sends token to a_i - a_i executes now */
5          𝒯' ← {τ_j ∈ 𝒯 | no path of any other agents in token ends at s_j or g_j};
6          if 𝒯' ≠ ∅ then
7              τ ← arg min_{τ_j ∈ 𝒯'} h(loc(a_i), s_j);
8              Assign τ to a_i;
9              Remove τ from 𝒯;
10             Update a_i's path in token with Path1(a_i, τ, token);
11         else if no task τ_j ∈ 𝒯 exists with g_j = loc(a_i) then
12             Update a_i's path in token with the path ⟨loc(a_i)⟩;
13         else
14             Update a_i's path in token with Path2(a_i, token);
           /* a_i returns token to system - system executes now */
15     All agents move along their paths in token for one time step;
       /* system advances to the next time step */
```

collisions) are computed in a preprocessing phase and then used as consistent h-values for all space-time A* searches. TP works as follows: The system initializes the token with the trivial paths where all agents rest at their initial vertices [Line 1]. At each time step, the system adds all new tasks, if any, to the task set [Line 3]. Any agent that has reached the end of its path in the token requests the token once per time step. The system then sends the token to each agent that requests it, one after the other [Line 4]. The agent with the token chooses a task from the candidate task set $\mathcal{T}'$ such that no path of other agents in the token ends at the pickup or delivery vertex of the task [Line 5] because it can then find a path to the pickup vertex and then a path to the delivery vertex of such a task and safely rest at the delivery vertex.

- If there is at least one such task, then the agent assigns itself the one with the smallest h-value from its current vertex to the pickup vertex of the task and removes this task from the task set [Lines 7-9]. The agent then calls Function *Path1* to update its path in the token

with a path that (1) is the concatenation of two time-minimal sub-paths, one that moves from its current vertex to the pickup vertex of the task (and rests there) and one that moves from there to the delivery vertex of the task (and rests there), and (2) does not collide with the paths of other agents stored in the token [Line 10].

- If there is no such task, then the agent does not assign itself a task at the current time step. If the agent is not at the delivery vertex of a task in the task set, then it updates its path in the token with the trivial path where it rests at its current vertex [Lines 11-12]. Otherwise, to avoid deadlocks (where it blocks other agents from executing the task whose delivery vertex is its current vertex), it calls Function *Path2* to update its path in the token with a time-minimal path that (1) moves from its current vertex to an endpoint (and rests there) such that the delivery vertices of all tasks in the task set are different from the chosen endpoint and no path of other agents in the token ends at the chosen endpoint and (2) does not collide with the paths of other agents stored in the token [Line 14]. In either case, the agent rests at an endpoint that is different from the delivery vertex of any task in the task set at the end of its path. (The latter case thus subsumes the former one.)

Finally, the agent returns the token to the system and moves along its path in the token [Line 15].

We now prove that the agent is always able to find a path because it finds a path only when it is at an endpoint and thus has to find only a path from an endpoint to an endpoint.

Property 6.1. Function *Path1* always returns a path successfully for well-formed MAPD problem instances.

Proof. We construct a path from the current vertex $loc(a_i)$ of agent a_i (which is an endpoint) via the pickup vertex s_j of task τ_j to the delivery vertex g_j of task τ_j that does not collide with the paths of other agents stored in the token. Due to Definition 6.1, there exists a path from $loc(a_i)$ via s_j to g_j that traverses no other endpoints. The paths of other agents stored in *token* end at endpoints that are different from $loc(a_i)$ (because those paths do not collide with the path of a_i

stored in *token*), s_j, and g_j. Thus, this path does not collide with the paths of the other agents if a_i moves along it after all other agents have moved along their paths. □

Property 6.2. Function *Path2* always returns a path successfully for well-formed MAPD problem instances.

Proof. Due to Definition 6.1, there exist at least M non-task endpoints and thus at least one non-task endpoint such that no path of agents other than agent a_i in the token ends at the non-task endpoint. The delivery vertices (which are task endpoints) of all tasks in the task set are different from the non-task endpoint as well. We construct a path from the current vertex $loc(a_i)$ of agent a_i (which is an endpoint) to the chosen endpoint that does not collide with the paths of other agents stored in the token. Due to Definition 6.1, there exists a path from $loc(a_i)$ to the chosen endpoint that traverses no other endpoints. The paths of other agents stored in *token* end at endpoints that are different from $loc(a_i)$ and the chosen endpoint. Thus, this path does not collide with the paths of the other agents if a_i moves along it after all other agents have moved along their paths. □

Theorem 6.1. All well-formed MAPD problem instances are solvable, and TP is long-term robust for them.

Proof. We show that each task is eventually assigned to some agent and executed by it. Each agent requests the token after a bounded number of time steps, and no agent rests at the delivery vertex of a task in the task set due to Line 14. Thus, the condition on Line 6 becomes eventually satisfied and some agent assigns itself some task on Line 8. The agent is then able to execute it due to Properties 6.1 and 6.2. □

6.4.1.1 Extensions of TP

The design of TP is kept as simple as possible but has the following benefits:

- TP can be used to constructively prove that all well-formed MAPD problem instances, as shown above.

- TP is still long-term robust even if each agent has to spend a known or unknown number of time steps at the pickup or delivery vertex, for example, to pick up or drop down a shelve of products in an automated warehouse system, because the agent can then plan a path from the pickup (respectively delivery) vertex only when it has finished the pickup (receptively delivery) operation.

- TP can replace the space-time A* search in Functions *Path1* and *Path2* with other one-shot single-agent path/motion-planning algorithms to plan a path for an agent as long as the path avoids collisions with the paths of other agents stored in *token*. For example, in Chapter 7, our extension of TP uses a single-agent path-planning algorithm that takes kinematic constraints of real-world agents into account.

- TP can potentially be generalized to a distributed setting using a token-based mutual exclusion mechanism (Suzuki & Kasami, 1985).

- TP can potentially apply, without many changes, to cases where tasks have more than one intermediate target (instead of only one—the pickup vertex) each.

TP can potentially be made more effective with changes in the following components of its design, which, however, can make TP less general:

1. Task assignment can be made more effective since the cost of assigning a task $\tau_j \in \mathcal{T}$ to a free agent a_i can be defined as (1) $h(loc(a_i), s_j)$ if no paths of other agents ends at s_j or g_j (similar to Line 7) and (2) a sufficiently large constant otherwise. Therefore, on Line 4, the token can thus be sent to the agent a_i with the smallest cost of being assigned any task in the task set over all agents that request the token (instead of an arbitrary agent that requests the token). This procedure thus greedily assigns a task to an agent with the smallest cost but, however, requires significantly more expensive communication between agents to generalize TP to a distributed setting. A centralized task-assignment procedure such as the Hungarian method (Kuhn, 1955) can further improve upon this greedy task assignment to produce one with the smallest total cost but makes it even harder to generalize TP to a

distributed setting. Our centralized algorithm (see Section 6.5) uses the Hungarian method for task assignment.

2. Task assignment can be changed so that new tasks that have just been added to the system or agents that have just become free can be taken into account at each time step. Our extension of TP (see Section 6.4.2) allows reassignment of tasks to agents by transferring a task from one free agent to another. Our centralized algorithm (see Section 6.5) reassigns tasks to agents whenever a new task has been added to the system or an agent has become free. However, additional considerations need to be made to guarantee long-term robustness because the agents may need to plan paths from non-endpoints.

3. Path planning can be made more effective. For example, once all agents that request the token assign themselves (pickup vertices of) tasks (as described in Function *Path1*) or endpoints where they can rest (as described in Function *Path2*), we can use any MAPF algorithms to plan paths for them together. Based on this idea, our centralized algorithm (see Section 6.5) uses CBS to plan paths for these agents together. However, it can be hard to generalize this idea to a distributed setting because a centralized MAPF algorithm is used. Also, additional considerations need to be made to guarantee that there exist paths that move the agents to their assigned targets.

4. Deadlock avoidance at the pickup vertex can be made more effective. Function *Path1* currently plans two time-minimal paths for an agent a_i, the first one from its current vertex to the pickup vertex s_j of its assigned task τ_j and the second one from s_j to g_j. Therefore, the agent can only reach s_j at a time step where no path of other agents goes through s_j at a later time step since the agent need to rest at the last vertex (s_j) of the first path. This guarantees that the second path can be found but can unnecessarily delay the arrival time of the agent at s_j. The space-time A* search can be modified so that it finds a time-minimal path first to s_j and then to g_j. This modification can potentially let agent a_i start to execute its assigned task earlier in Function *Path1* and can even allow agent a_i to consider tasks whose pickup vertex is the last vertex of the path of another agent $a_{i'}$ (as long as

agent a_i can reach the pickup vertex before agent $a_{i'}$) on Line 7. Recent research uses this modification (Grenouilleau, van Hoeve, & Hooker, 2019). However, this modification cannot generalize to cases where each agent has to spend an unknown number of time steps at the pickup vertex to finish the pickup operation.

5. TP can be made more effective if it does not assume that each agent rests at the last vertex of its path since this vertex can then be used by other paths. Recent research uses this idea (Okumura, Machida, Défago, & Tamura, 2019). However, the resulting algorithm is not long-term robust for well-formed MAPD problem instances.

6.4.2 TPTS

We use one of the ideas presented in Section 6.4.1.1 to make TP more effective. The resulting algorithm, TPTS, is similar to TP except that its task set now contains all unexecuted tasks, rather than only unassigned tasks. This means that an agent with the token can assign itself not only a task that is not assigned to any agent but also a task that is already assigned to another agent as long as that agent is still moving to the pickup vertex of the task. This might be beneficial when the former agent can move to the pickup vertex of the task in fewer time steps than the latter agent. The latter agent is then no longer assigned the task and no longer needs to execute it. The former agent thus sends the token to the latter agent so that the latter agent can try to assign itself a new task.

Algorithm 6.2 shows the pseudocode of TPTS. It uses the same main loop [Lines 2-6] and the same Functions *Path1* and *Path2* as TP. The agent a_i with the token executes Function *GetTask* [Line 5], where it tries to assign itself a task in the task set $\mathcal{T}$ and find a path to an endpoint. The call of Function *GetTask* returns success (true) if agent a_i finds a path to an endpoint and failure (false) otherwise.

Algorithm 6.3 shows the pseudo-code of Function *GetTask*. When executing Function *GetTask*, agent a_i considers all tasks τ from the candidate task set $\mathcal{T}'$ such that no path of other agents in the token ends at the pickup or delivery vertex of the task [Line 2], one after the other

Algorithm 6.2: TPTS

```
   /* system executes now */
1  Initialize token with the (trivial) path ⟨loc(a_i)⟩ for each agent a_i;
2  while true do
3      Add all new tasks, if any, to the task set 𝒯;
4      while agent a_i exists that requests token do
           /* system sends token to a_i - a_i executes now */
5          GetTask(a_i, token);          // function always returns true (Property 6.3)
           /* a_i returns token to system - system executes now */
6      All agents move along their paths in token for one time step and remove tasks from
        𝒯 when they start to execute them;
       /* system advances to the next time step */
```

in order of increasing h-values from its current vertex to the pickup vertices of the tasks [Lines 3-5]. If the task is not assigned to any agent, then (as in TP) agent a_i assigns itself the task, updates its path in the token with Function *Path1* and returns success [Lines 6-9]. Otherwise, agent a_i remembers the token with the paths, task set, and task assignments in case the reassignment of the task is not successful [Line 11]. Agent a_i then unassigns the task from the agent $a_{i'}$ that is assigned the task and assigns itself the task [12-13]. It removes the path of agent $a_{i'}$ from the token and updates its own path in the token with Function *Path1* [Lines 14-15]. If agent a_i reaches the pickup vertex of the task with fewer time steps than agent $a_{i'}$, then it sends the token to agent $a_{i'}$, which executes Function *GetTask* to try to assign itself a new task and eventually returns the token to agent a_i [Lines 16-5]. If agent $a_{i'}$ returns success, then agent a_i returns success as well [Lines 19-20]. In all other cases, agent a_i reverses all changes to the paths, the task set, and the task assignments in the token and then considers the next task τ [Lines 11 and 21].

Once agent a_i has considered all tasks τ unsuccessfully, then it does not assign itself a task at the current time step. If it is not at an endpoint (which can happen only during a call of Function *GetTask* on Line 18), then it updates its path in the token with Function *Path2* to move to an endpoint [Line 23]. The call can fail since agent a_i is not at an endpoint. Agent a_i returns success or failure depending on whether it was able to find a path [Lines 30 and 25]. Otherwise, (as in TP) if the agent is not in the delivery vertex of a task in the task set, then it updates its

Algorithm 6.3: Function *GetTask*(a_i, *token*)

```
1  Function GetTask(a_i, token)
2      T' ← {τ_j ∈ T | no path of other agents in token ends at s_j or g_j};
3      while T' ≠ ∅ do
4          τ ← arg min_{τ_j ∈ T'} h(loc(a_i), s_j);
5          Remove τ from T';
6          if τ is not assigned to any agent then
7              Assign τ to a_i;
8              Update a_i's path in token with Path1(a_i, τ, token);
9              return true;
10         else
11             Remember token (with paths, task set, and task assignments);
12             a_i' ← agent that is assigned τ;
13             Unassign τ from a_i' and assign τ to a_i;
14             Remove a_i''s path from token;
15             Path1(a_i, τ, token);
16             Compare the arrival time of a_i at s_j on its path in token to the arrival time of
                a_i' at s_j on its path that has been remembered on Line 11;
17             if a_i reaches s_j earlier than a_i' then
                   /* a_i sends token to a_i' - a_i' executes now */
18                 success ← GetTask(a_i', token);
                   /* a_i' returns token to a_i - a_i executes now */
19                 if success then
20                     return true;
21             Restore token (with paths, task set, and task assignments which have been
                remembered on Line 11);
22     if loc(a_i) is not an endpoint then
23         Update a_i's path in token with Path2(a_i, token);
24         if no path was found then
25             return false;
26     else if no task τ_j ∈ T exists with g_j = loc(a_i) then
27         Update a_i's path in token with the path ⟨loc(a_i)⟩;
28     else
29         Update a_i's path in token with Path2(a_i, token);
30     return true;
```

path in the token with the trivial path where it rests at its current vertex [Line 27]. Otherwise, to avoid deadlocks (where it blocks other agents from executing the task whose delivery vertex is

its current vertex), it updates its path in the token with Function *Path2* [Line 29]. In both cases, it returns success [Line 30].

Finally, the agent returns the token to the system and moves along its path in the token, removing the task that it is assigned (if any) from the task set once it reaches the pickup vertex of the task and thus starts to execute it [Line 6 of Algorithm 6.2].

Property 6.3. Function *GetTask* always returns successfully for well-formed MAPD problem instances when called on Line 5 of Algorithm 6.2.

Proof. Function *GetTask* returns in finite time because (1) the number of tasks in the task set is finite; (2) an agent can unassign a task from another agent only if it reaches the pickup vertex of the task with fewer time steps than the other agent; (3) a task that is assigned to some agent always continues to be assigned to some agent until it has been executed; and (4) Functions *Path1* and *Path2* return in finite time. Properties (1) to (3) make sure that the number of times an agent can unassign any task from another agent is bounded during the execution of Function *GetTask* and the number of recursive calls of Function *GetTask* on Line 18 of Algorithm 6.3 is thus bounded. Properties (1) to (4) thus make sure that the execution time of each call of Function *GetTask* is also bounded.

We now show that an agent that executes Function *GetTask* on Line 5 of Algorithm 6.2 finds a path to an endpoint. The agent is always able to find paths with Functions *Path1* and *Path2* on all lines but Line 23 of Algorithm 6.3 because it is then at an endpoint and thus has to find a path from an endpoint to an endpoint. The proofs are similar to those of Properties 6.1 and 6.2. However, the agent is not guaranteed to find a path with Function *Path2* on Line 23 of Algorithm 6.3 because it is then not at an endpoint and thus has to find a path from a non-endpoint to an endpoint. Since the agent is at an endpoint during the call of Function *GetTask* on Line 5 of Algorithm 6.2, it does not execute Line 23 of Algorithm 6.3, finds a path, and returns success. □

Theorem 6.2. TPTS is long-term robust for all well-formed MAPD problem instances.

Proof. The proof is similar to the one of Theorem 6.1 but uses Property 6.3. □

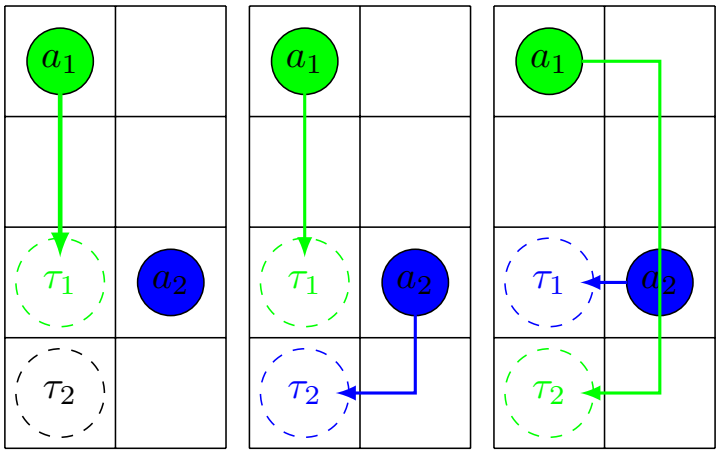


Figure 6.3: Example of a MAPD problem instance for the comparison of TP and TPTS.

TPTS is often more effective than TP but Figure 6.3 shows that this is not guaranteed. The figure shows a MAPD problem instance with two agents a_1 and a_2 and two tasks τ_1 and τ_2 that are added to the system at time step 0. The blue and green circles are the initial vertices of agents. Dashed circles are the pickup/delivery vertices. The pickup vertex is the same as the delivery vertex for both tasks. Assume that both a_1 and a_2 request the token and the system sends it to agent a_1 first. Agent a_1 assigns itself τ_1. Figure 6.3 (left) shows the path of a_1. The system then sends the token to a_2 next. In TP, a_2 assigns itself τ_2. Figure 6.3 (center) shows the paths of a_1 and a_2 for TP. The resulting service time and makespan are both two. In TPTS, however, a_2 assigns itself τ_1 because it can reach the pickup vertex of τ_1 with fewer time steps than a_1. In return, a_1 assigns itself τ_2. Figure 6.3 (right) shows the paths of a_1 and a_2 for TPTS. The service time is three (the average of five and one), and the makespan is five.

6.4.2.1 Extensions of TPTS

TPTS requires only additional local communication between pairs of agents and thus has the same benefits of TP (see Section 6.4.1.1). TPTS can potentially be made more effective by allowing agents on the way to the pickup vertex of its assigned task to consider new tasks that have just been added to the system. This can be achieved by letting all free agents request the token at every time step, which makes TPTS much less efficient, or at time steps where a new

task is added to the system, which, in a distributed setting, requires an additional broadcasting mechanism to inform all free agents (that a new task is added). In both cases, the call of Function *GetTask* on Line 5 of Algorithm 6.2 can return failure for an agent that is currently on the way to the pickup vertex of its assigned task because it is not necessarily at an endpoint, but the agent can still move along its current path. Our centralized algorithm (see Section 6.5) uses a similar idea to allow all free agents to consider all tasks, including the ones that have just been added to the system, in the task set.

6.5 Centralized Algorithm

In this section, we use some of the ideas presented in Sections 6.4.1.1 and 6.4.2.1 to develop a centralized MAPD algorithm, CENTRAL, that can potentially improve the effectiveness of decoupled MAPD algorithms. Unlike the decoupled algorithms TP and TPTS, CENTRAL makes decisions for multiple agents at a time. We expect CENTRAL to be reasonably efficient and effective. But unlike centralized MAPF and TAPF algorithms, we do not require CENTRAL to be optimally effective since (1) state-of-the-art algorithms that can solve MAPD optimally in an offline setting (by making the assumption that all tasks are known a priori) scale to only 4 agents and 4 tasks (Henkel et al., 2019) and (2) there does not exist any optimal online MAPD algorithm (Azar et al., 1995; Kalyanasundaram & Pruhs, 1993).

Similar to TPTS, CENTRAL allows agents that have just become free to consider not only unassigned tasks but also all unexecuted tasks, including the ones that have been assigned to agents, in the task set. Unlike TPTS, CENTRAL uses a centralized target-assignment algorithm, the Hungarian method (Kuhn, 1955), to assign (pickup vertices of) tasks to agents and allows all free agents to consider tasks that have just been added to the system. It uses the centralized MAPF algorithm CBS (Section 3.2.2) to plan paths for multiple agents.

6.5.1 Task/Endpoint-Assignment Procedure

At each time step, CENTRAL executes the task/endpoint-assignment procedure in two phases to assign endpoints to agents.

In the first phase, CENTRAL considers each agent, one after the other, that rests at the pickup vertex of an unexecuted task. If the delivery vertex of the task is currently not assigned to other agents, CENTRAL assigns the corresponding unexecuted task (if it is not assigned to the agent already) and its delivery vertex to the agent. The agent then starts to execute the task and thus becomes occupied. CENTRAL then executes the first stage of the path-planning procedure (see Section 6.5.2) to plan paths for all agents that become occupied to execute their assigned tasks.

In the second phase, if any new tasks are added to the system or any agents finish executing their tasks (and thus become free) at the current time step, CENTRAL assigns either the pickup vertex of an unexecuted task or some other endpoint as *parking endpoint* to each free agent. To make the resulting MAPF problem solvable, the endpoints assigned to all agents must be pairwise different. Agents are assigned pickup vertices of unexecuted tasks in order to execute the tasks afterward. Thus, when CENTRAL assigns pickup vertices of unexecuted tasks to agents, we want the delivery vertices of these tasks to be different from the endpoints assigned to all agents (except for their own pickup vertices in case the pickup and delivery vertices of a task are the same) and from each other. CENTRAL achieves these constraints as follows:

- First, CENTRAL greedily constructs a set of possible endpoints X for the free agents as follows: CENTRAL greedily constructs a subset $\mathcal{T}'$ of unexecuted tasks, starting with the empty set, by checking for each unexecuted task, one after the other, whether its pickup and delivery vertices are different from the delivery vertices of all executed tasks and the pickup and delivery vertices of all unexecuted tasks already added to $\mathcal{T}'$ and, if so, adds it to $\mathcal{T}'$. CENTRAL then sets X to the pickup vertices of all tasks in $\mathcal{T}'$. If the number of free agents is larger than $|X|$, then CENTRAL needs to add endpoints to X as parking endpoints for some free agents. Since it is not known a priori which free agents these parking endpoints will be assigned to, there should be one *good* parking endpoint

available for each free agent, which is possible because there is at least one unique non-task endpoint for each free agent due to Definition 6.1: CENTRAL greedily determines a good parking endpoint for each free agent a_i, one after the other, as the endpoint e that minimizes the cost $c(a_i, e)$ ("is closest to the agent") among all endpoints that are different from the delivery vertices of all executed tasks, the pickup and delivery vertices of all tasks in $\mathcal{T}'$, and the parking endpoints already determined, where $c(a_i, e)$ is the cost of a time-minimal path that moves from the current vertex of free agent a_i to endpoint e (a unique non-task endpoint is thus always a good-parking-endpoint candidate for each free agent). It then adds this endpoint to X.

- Second, CENTRAL assigns each free agent an endpoint in X to satisfy all constraints. It uses the Hungarian Method (Kuhn, 1955) for this purpose with the modified costs $c'(a_i, e)$ for each pair of free agent a_i and endpoint e, where c is the number of free agents, C is a sufficiently large constant (for example, the maximum over all costs $c(a_i, e)$ plus one), and $c'(a_i, e) = c \cdot C \cdot c(a_i, e)$ if e is a pickup vertex of a task in $\mathcal{T}'$ and $c'(a_i, e) = c \cdot C^2 + c(a_i, e)$ if e is a parking endpoint. The modified costs have two desirable properties: (a) The modified cost of assigning a pickup vertex to a free agent is always smaller than the modified cost of assigning a parking endpoint to the same agent. Therefore, assigning pickup vertices is more important than assigning parking endpoints. (b) Assigning a closer pickup vertex to a single free agent that is assigned a pickup vertex reduces the total modified cost more than assigning closer parking endpoints to all free agents that are assigned parking endpoints. Therefore, assigning closer pickup vertices is more important than assigning closer parking endpoints. The Hungarian Method itself handles the case where the number of free agents is smaller than $|\mathcal{T}'|$ (and thus $|X| = |\mathcal{T}'|$) by adding "dummy" free agents (so that there are $|X|$ "dummy"and "real" free agents in total) that have a sufficiently large constant cost for every endpoint in X, which does not affect the assignments of endpoints to "real" free agents.

6.5.2 Path-Planning Procedure

CENTRAL uses a version of CBS to plan collision-free paths for all agents from their current vertices to their assigned endpoints simultaneously. This version of CBS (see Section 3.2.2.3) minimizes the flowtime, that is, the sum of the number of time steps required by all agents to reach their assigned endpoints and stop moving, because minimizing the makespan can unnecessarily delay the arrival times of some agents at their assigned endpoints for which they could have otherwise reached at an earlier time step.

We noticed that CENTRAL becomes significantly more efficient if it plans paths in two stages at each time step:

In the first stage, if any agents become occupied (in the first phase of the task/endpoint-assignment procedure) at the current time step, CENTRAL plans paths for all these agents to their assigned endpoints (using the approach described above but treating all other agents as dynamic obstacles that follow their most recently calculated paths and with which collisions need to be avoided). These occupied agents will then follow the entire planned paths.

In the second stage, if any new tasks are added to the system or any agents become free at the current time step (and the second phase of the task/endpoint-assignment procedure has thus been executed), it plans paths for all free agents to their assigned endpoints (again using the approach described above but treating all other agents as dynamic obstacles that follow their most recently calculated paths and with which collisions need to be avoided). The free agents will not necessarily follow the entire planned agents if they are reassigned tasks at the future time steps.

In general, two smaller MAPF problem instances can be solved much faster than their union due to the NP-hardness of the problem. Also, CENTRAL can then determine a more informed cost $c(a_i, e)$ as the cost of a time-minimal path that (1) moves from the current vertex of agent a_i to endpoint e and (2) does not collide with the paths of the occupied agents (as described for TP).

Property 6.4. Path planning for all agents that became occupied at the current time step returns paths successfully for well-formed MAPD problem instances.

Proof. We construct paths for all agents that became occupied at the current time step from their current vertices to their assigned endpoints that do not collide with the most recently calculated paths of all other agents: Assume that all other agents move along their most recently calculated paths. When all of them have reached the ends of their paths, move all agents that became occupied one after the other to their assigned endpoints, which is possible due to Definition 6.1 since their current vertices are endpoints and their assigned endpoints are different from the endpoints that all other agents now occupy. Any complete MAPF algorithm thus returns paths successfully. □

Property 6.5. Path planning for all free agents returns paths successfully for well-formed MAPD problem instances.

Proof. We construct paths for all free agents from their current vertices to their assigned endpoints that do not collide with the most recently calculated paths of all other agents: In the first step, assume that all (both free and occupied) agents move along their most recently calculated paths. In the second step, when all of them have reached the ends of their paths, move all free agents one after the other to their assigned endpoints, which is possible due to Definition 6.1 since the vertices that they now occupy are endpoints. Directly before an agent moves to its assigned endpoint, check whether this endpoint is blocked by another agent (which must be a free agent that has not executed the second step due to the second phase of the task/endpoint-assignment procedure). If so, move this other agent to an unoccupied endpoint first. Such an endpoint exists since there are at least $M + 1$ endpoints for M agents due to Definition 6.1. The free agents that have executed the second step and all occupied agents never move when other free agents execute the second step because they do not occupy endpoints assigned to free agents that have not executed the second step. The resulting paths thus move all free agents to their assigned endpoints. Any complete MAPF algorithm thus returns paths successfully. □

Theorem 6.3. CENTRAL is long-term robust for all well-formed MAPD problem instances.

Proof. We show that each task is eventually assigned to some agent and executed by it. The last task is added to the system at a bounded time step t_1. After that, for a proof by contradiction, assume that all unexecuted tasks remain unexecuted (any executed task will be finished by the agent that it is assigned to as stated above). If all agents are free at time step t_1, we consider time step $t = t_1$. Otherwise, there must exist an earliest bounded time step $t_2 > t_1$ where all agents have become free, and we consider time step $t = t_2$. At time step t, no unexecuted task is assigned to an agent in the first phase of the task/endpoint-assignment procedure because, otherwise, the task will be executed successfully. The task/endpoint-assignment procedure executes the second phase since a new task is added or an agent becomes free at time step t. Since all tasks are unexecuted, the candidate task set $\mathcal{T}'$ admits at least one task. The pickup vertex of the task is assigned to an agent. This pickup vertex is never re-assigned to another agent afterward because no new task is added and no agent changes from occupied to free (otherwise, an unexecuted task is finished, which contradicts the assumption). Therefore, the agent reaches the pickup vertex of the task due to Property 6.5. Then, the agent is assigned the delivery vertex of the task because no other agents are assigned this vertex as endpoint due to the second phase of the task/endpoint-assignment procedure executed at time step t. The agent eventually reaches the delivery vertex of the task due to Property 6.4, which contradicts the assumption. Therefore, at least one unexecuted task is finished by an agent. We finish the proof by applying the above arguments for the set of remaining unexecuted tasks. □

6.5.3 Extensions of CENTRAL

The design of CENTRAL is kept as simple as possible so that it can easily be extended to different centralized MAPD algorithms.

- Like TP, CENTRAL can be used to constructively prove that all well-formed MAPD problem instances, as shown above.

- Like TP, CENTRAL is also long-term robust even if each agent has to spend a known or unknown number of time steps at the pickup or delivery vertex because CENTRAL can then plan a path for the agent from the pickup (respectively delivery) vertex only when the agent has actually finished the pickup (receptively delivery) operation.
- CENTRAL can replace CBS with any other MAPF algorithm in the path-planning procedure. The resulting MAPD algorithm is still long-term robust as long as the MAPF algorithm is complete.
- CENTRAL can potentially apply, without many changes, to cases where tasks have more than one intermediate target (instead of only one—the pickup vertex) each.

CENTRAL can potentially be made more effective (respectively efficient) with changes in the following components of its design, which, however, can make CENTRAL less efficient (respectively effective), robust, or general:

1. The path-planning procedure can combine the MAPF problems in two stages into a larger MAPF problem (if both stages are executed). Solving such a larger MAPF problem can potentially make CENTRAL more effective but makes CENTRAL less efficient than solving the two smaller MAPF problems in two stages. Also, additional considerations are needed to guarantee long-term robustness.
2. The path-planning procedure can be divided into more stages. For example, in the second stage, paths for agents that are assigned pickup vertices of unexecuted tasks can be planned first, and paths for agents that are assigned parking endpoints, if there are any, can be planned later. This can make CENTRAL more efficient. However, additional considerations are needed to guarantee long-term robustness.
3. Any MAPF algorithms that are more efficient can be used in the path-planning procedure to make CENTRAL more efficient. Complete but suboptimal MAPF algorithms can be used in the path-planning procedure. This can make CENTRAL more efficient but less effective. Incomplete MAPF algorithms can also be used in the path-planning procedure.

However, this can make CENTRAL no longer long-term robust for well-formed MAPD problem instances.

4. Like TP, the task/endpoint-assignment procedure can consider only unassigned tasks and free agents with no task assigned and thus do not reassign tasks to agents that have already been assigned tasks. This can make CENTRAL less effective but makes it more efficient because the target-assignment and MAPF problems are all smaller. This also allows CENTRAL to use space-time A* searches to plan paths for individual agents (like in TP) and still guarantees long-term robustness. Some recent research uses this idea (Grenouilleau et al., 2019; Hönig, Kiesel, Tinka, Durham, & Ayanian, 2019).

5. Similar to an idea of extending TP (see Section 6.4.1.1), deadlock avoidance at the pickup vertex can be made more effective. CENTRAL can assume that agents whose paths end at the pickup vertices of their assigned tasks do not rest at the pickup vertices. Our recent research uses this modification (Liu, Ma, Li, & Koenig, 2019) (not covered as a contribution of this dissertation). However, this cannot generalize to cases where each agent has to spend an unknown number of time steps at the pickup vertex to finish the pickup operation. Also, additional considerations are needed to guarantee long-term robustness.

6.6 Experiments

In this section, we describe the results of four experiments on a 2.50 GHz Intel Core i5-2450M laptop with 6 GB RAM. First, we compare the makespans and service times of the solutions produced by TP, TPTS, and CENTRAL. Second, we compare the runtimes of TP, TPTS, and CENTRAL. Third, we compare the throughputs of TP, TPTS, and CENTRAL. Fourth, we study how TP performs in a large simulated warehouse environment with hundreds of agents.

For Experiments 1 to 3, we ran TP, TPTS, and CENTRAL in the small simulated warehouse environment shown in Figure 6.4. Black cells are blocked. Gray cells in columns of gray cells are potential initial vertices for the agents. Colored circles are the actual initial vertices of agents,

Figure 6.4: The small simulated warehouse environment with 50 agents.

which are drawn randomly from all potential initial vertices and are the only non-task endpoints. Gray cells other than the initial vertices are task endpoints (that would house inventory pods in a warehouse even though we do not model inventory pods here). We generated a sequence of 500 tasks by randomly choosing their pickup and delivery vertices from all task endpoints. We used 6 different task frequencies (numbers of tasks that are added (in order) from the sequence to the system at each time step): 0.2 (one task every 5 time steps), 0.5, 1, 2, 5, and 10. For each task frequency, we used 5 different numbers of agents: 10, 20, 30, 40, and 50. This simulated warehouse environment is inspired by the warehouse map shown in Figure 1.1 that represents the layout of part of an Amazon Robotics automated warehouse system (Wurman et al., 2008). The resulting MAPD problem instances have been considered as benchmark MAPD problem instances by recent research on MAPD (Grenouilleau et al., 2019; Liu, Ma, et al., 2019; Okumura et al., 2019).

Table 6.1 reports the makespans, the service times, the runtimes per time step (in ms), the ratios of the service times of TPTS and TP, and the ratios of the service times of CENTRAL and TP. The measures for a task frequency of 10 tasks per time step are reasonably representative of the case where all tasks are added in the beginning of the operation since the tasks are added over the first 50 time steps only.

Table 6.1: Results for TP, TPTS, and CENTRAL in the small simulated warehouse environment.

task frequency	agents	TP			TPTS				CENTRAL			
		makespan	service time	time (ms)	makespan	service time	time (ms)	ratio	makespan	service time	time (ms)	ratio
0.2	10	2,532	38.54	0.13	2,532	29.33	1.86	0.76	2,513	27.82	37.77	0.72
	20	2,540	39.77	0.26	2,520	25.36	9.82	0.64	2,520	24.77	218.17	0.62
	30	2,546	38.71	0.25	2,527	23.88	21.57	0.62	2,520	23.60	553.20	0.61
	40	2,540	38.88	0.24	2,524	23.50	27.49	0.60	2,516	23.07	1,042.81	0.59
	50	2,540	40.03	0.32	2,524	23.11	47.33	0.58	2,518	22.67	1,711.85	0.57
0.5	10	1,309	132.79	0.24	1,274	131.15	0.33	0.99	1,264	116.79	64.22	0.88
	20	1,094	42.69	1.16	1,038	30.74	12.39	0.72	1,036	27.63	232.52	0.65
	30	1,069	43.97	1.51	1,035	27.14	34.04	0.62	1,032	25.84	712.71	0.59
	40	1,090	43.01	0.70	1,038	25.98	19.85	0.60	1,032	24.94	1,438.38	0.58
	50	1,083	43.66	1.36	1,036	25.22	71.48	0.58	1,031	24.46	2,564.55	0.56
1	10	1,198	311.78	0.20	1,182	301.03	0.37	0.97	1,146	290.52	84.78	0.93
	20	757	95.98	1.03	706	88.25	2.80	0.92	698	75.20	237.68	0.78
	30	607	53.80	2.81	561	42.84	8.45	0.80	560	31.54	394.55	0.59
	40	624	48.80	2.14	563	31.99	39.54	0.66	554	27.85	1,110.96	0.57
	50	597	49.14	3.76	554	30.27	128.13	0.62	555	27.02	2,118.37	0.55
2	10	1,167	407.62	0.19	1,168	407.24	0.37	1.00	1,113	382.25	78.45	0.94
	20	683	190.76	0.96	667	181.03	2.38	0.95	617	163.97	255.19	0.86
	30	529	114.39	2.31	496	102.69	8.39	0.90	479	92.88	516.74	0.81
	40	464	95.32	3.43	425	72.59	7.32	0.76	382	55.88	745.52	0.59
	50	432	75.63	6.28	383	58.06	126.47	0.77	332	38.96	1,101.63	0.52
5	10	1,162	473.78	0.20	1,165	473.18	0.41	1.00	1,128	462.34	80.75	0.98
	20	655	247.08	1.02	645	238.02	1.68	0.96	590	222.69	219.01	0.90
	30	478	170.78	2.22	474	167.66	8.58	0.98	431	147.53	482.46	0.86
	40	418	155.33	4.15	396	131.36	12.31	0.85	345	109.55	802.54	0.71
	50	395	124.59	5.92	343	104.86	59.64	0.84	289	86.68	1,296.44	0.70
10	10	1,163	495.93	0.22	1,172	505.26	0.40	1.02	1,115	474.97	75.55	0.96
	20	643	275.24	1.09	645	258.36	1.87	0.94	589	242.18	223.50	0.88
	30	526	192.01	1.98	491	198.30	10.82	1.03	422	165.13	440.73	0.86
	40	407	154.63	1.65	389	152.49	12.62	0.99	344	128.40	785.88	0.83
	50	333	131.42	5.62	319	126.96	25.32	0.97	304	106.70	1,884.28	0.81

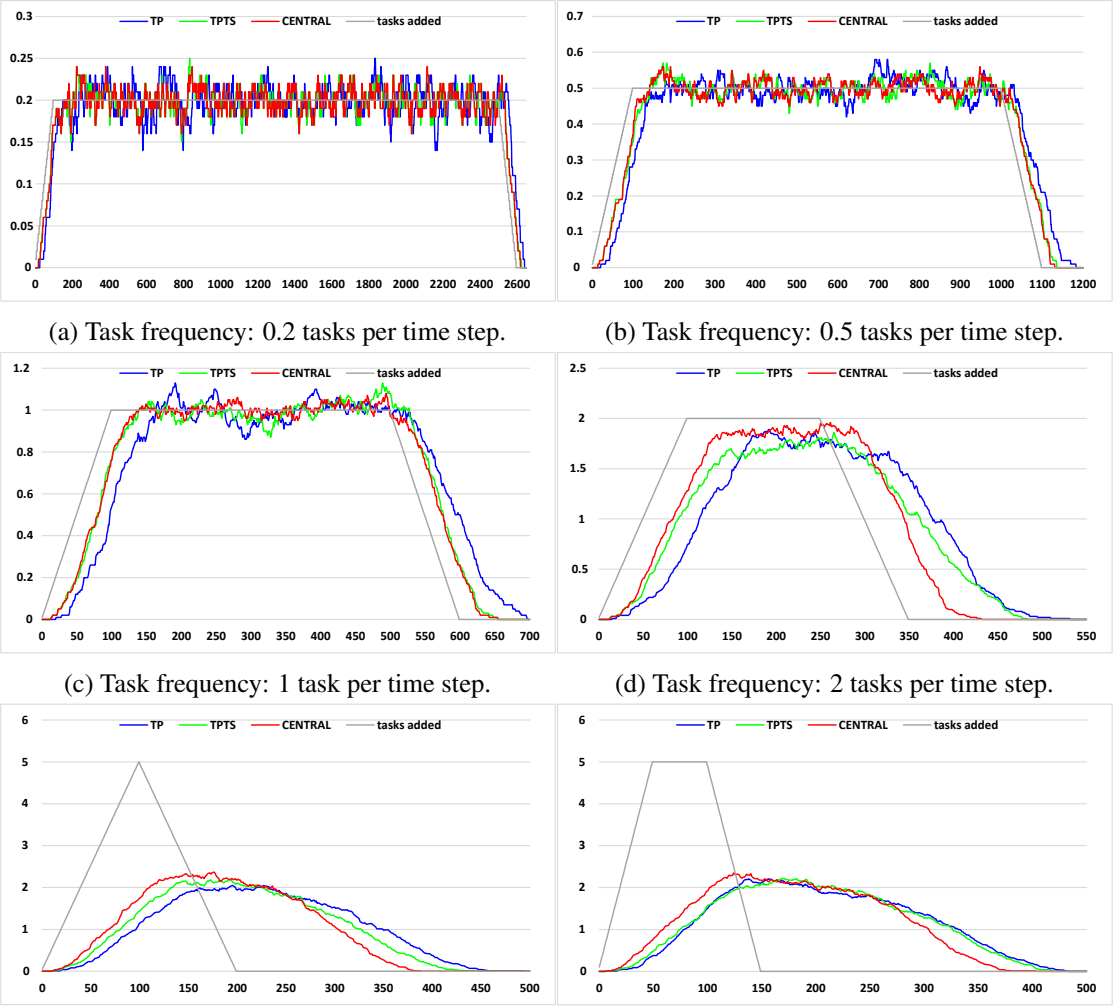
6.6.1 Experiment 1: Makespan and Service Time

The MAPD algorithms, in increasing order of their makespans and service times, tend to be: CENTRAL, TPTS, and TP. For example, the service time of TPTS (and CENTRAL) is up to

about 42 percent (and 48 percent, respectively) smaller than the one of TP for some experimental runs. The makespans tend to be large for low task frequencies and small for high task frequencies because the number of tasks is constant and thus more time steps are needed to add all tasks for low task frequencies. On the other hand, the service times tend to be small for low task frequencies and high for high task frequencies because the agents tend to be able to attend to tasks fast if the number of tasks in the system is small. The makespans and service times tend to be large for small numbers of agents and small for large numbers of agents because the agents tend to be able to attend to tasks fast if the number of agents is large (although congestion increases). The makespans and service times for a task frequency of 0.2 tasks per time step are about the same for all numbers of agents because 10 agents already attend to all tasks as fast as the MAPD algorithms allow. The makespans are similar for all MAPD algorithms and all numbers of agents for the task frequency of 0.2 tasks per time step because 10 agents already execute tasks faster than they are added. The makespans then depend largely on how fast the agents execute the last few tasks. On the other hand, the makespans of MAPD algorithms increase substantially when tasks pile up because the agents execute them more slowly than they are added. This allows us to estimate the smallest number of agents needed for a long-term operation as a function of the task frequency and MAPD algorithm. For example, the makespan of TPTS increases substantially when the number of agents is reduced from 20 to 10 for a task frequency of 1 task per time step. Thus, one needs between about 10 and 20 agents for a long-term operation with TPTS.

6.6.2 Experiment 2: Runtime per Time Step

The MAPD algorithms, in increasing order of their runtimes per time step, tend to be: TP, TPTS, and CENTRAL. For example, the runtime of TPTS (and CENTRAL) is two orders of magnitude larger than the runtime of TP (and TPTS, respectively) for some experimental runs. The runtimes of TP are less than 10 milliseconds, the runtimes of TPTS are less than 200 milliseconds, and the runtimes of CENTRAL are less than 3,000 milliseconds in all experimental runs. We consider runtimes below one second to allow for real-time long-term operation. The runtimes tend not to be correlated with the task frequencies. They tend to be small for small numbers of agents and



(a) Task frequency: 0.2 tasks per time step.

(b) Task frequency: 0.5 tasks per time step.

(c) Task frequency: 1 task per time step.

(d) Task frequency: 2 tasks per time step.

(e) Task frequency: 5 tasks per time step.

(f) Task frequency: 10 tasks per time step.

Figure 6.5: Number of tasks added (gray) and executed by 50 agents per time step in a moving 100-time-step window $[t-99, t]$ for TP, TPTS, and CENTRAL as a function of the time step t for different task frequencies.

large for large numbers of agents because all agents need to perform computations, which are not run in parallel in our experiments.

6.6.3 Experiment 3: Number of Executed Tasks

The service times vary over time since only very few tasks are available at the first and last time steps. The steady state is in between these two extremes. Figure 6.5 visualizes the number of

Table 6.2: Throughputs for TP, TPTS, and CENTRAL in the small simulated warehouse environment.

task frequency	TP	TPTS	CENTRAL
0.2	0.191	0.192	0.192
0.5	0.430	0.446	0.448
1	0.739	0.780	0.780
2	0.967	1.062	1.190
5	1.031	1.152	1.316
10	1.174	1.214	1.259

Table 6.3: Results for TP in the large simulated warehouse environment.

agents	100	200	300	400	500
service time	463.25	330.19	301.97	289.08	284.24
time (ms)	90.83	538.22	1,854.44	3,881.11	6,121.06

tasks added per time step and the number of tasks executed by 50 agents per time step (that is, *throughput* at time step t) during the 100-time-step window $[t-99, t]$ for all MAPD algorithms as a function of the time step t. For low task frequencies, the numbers of tasks added match the numbers of tasks executed closely for all MAPD algorithms. Differences between them arise for higher task frequencies. For example, for the task frequency of 2 tasks per time step, the number of tasks executed by CENTRAL increases faster and reaches a higher level than the numbers of tasks executed by TP and TPTS. The 100-time-step window $[150, 249]$ at time step $t = 249$ is a close approximation of the steady state since all tasks are added at a steady rate during the first 250 time steps. CENTRAL executes more tasks during this 100-time-step window than TP and TPTS and thus has a smaller service time. However, the numbers of tasks executed are smaller for all MAPD algorithms than the number of tasks added, and tasks thus pile up for all of them in the steady state. Table 6.2 reports the average throughput over all time steps t whose numbers of tasks executed during the 100-time-step window $[t-99, t]$ are positive, measured in tasks per time step. The MAPD algorithms, in increasing order of their average throughputs, are: TP, TPTS, and CENTRAL.

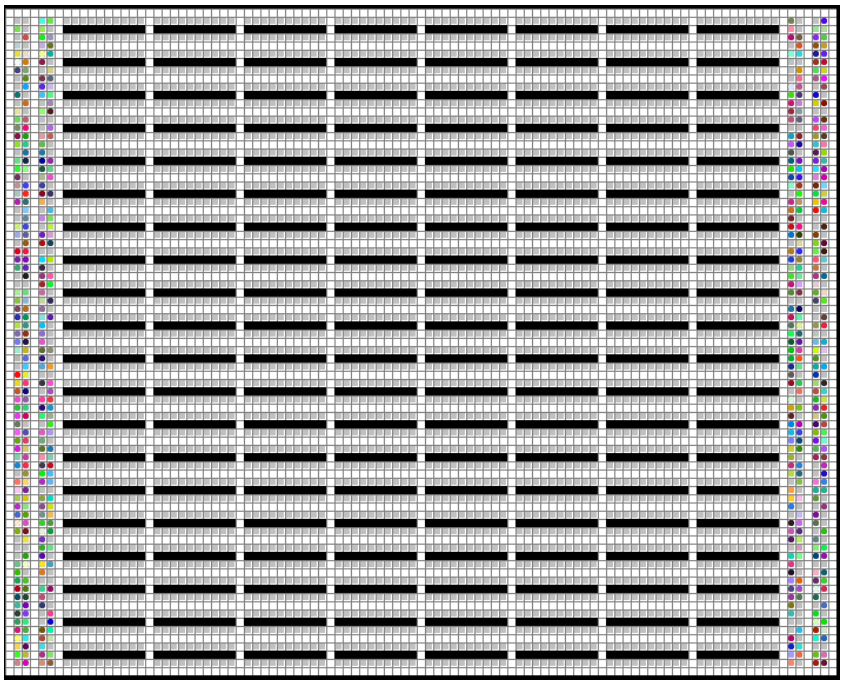


Figure 6.6: The large simulated warehouse environment with 500 agents.

6.6.4 Experiment 4: Scalability

To evaluate how the MAPD algorithms scale in the number of agents, we ran TP, TPTS, and CENTRAL in the large simulated warehouse environment shown in Figure 6.6. We generated a sequence of 1,000 tasks by randomly choosing their pickup and delivery vertices from all task endpoints. The initial vertices of the agents are the only non-task endpoints. We used a task frequency of 50 tasks per time step and 100, 200, 300, 400, and 500 agents. TPTS and CENTRAL did not allow for real-time long-term operation for large numbers of agents. Table 6.3 thus reports, for TP only, the service times and the runtimes per time step (in ms). Figure 6.7 visualizes the numbers of tasks executed per time step during a 100-time-step window for different numbers of agents in the charts. It does not visualize the number of tasks added per time step since the number is much larger than the number of tasks executed per time step for most t. The runtime of TP is smaller than 550 milliseconds for 200 agents, allowing for real-time long-term operation.

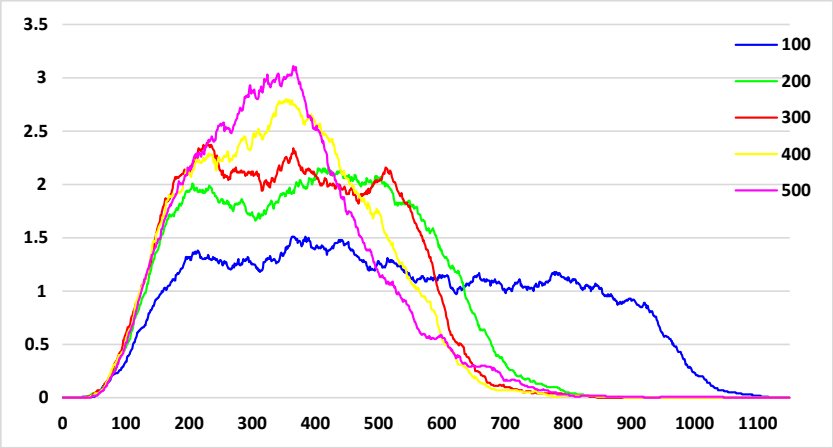


Figure 6.7: Number of tasks executed per time step during the 100-time-step window $[t - 99, t]$ for TP as a function of the time step t for different numbers of agents.

6.7 Summary

In this chapter, we formalized and studied MAPD, a new variant of MAPF that models the long-term combined target-assignment and path-planning problem for teams of agents. We demonstrated how MAPD algorithms can utilize the environmental characteristics of well-formed MAPD problem instances, a realistic subset of MAPD problem instances, to guarantee long-term robust for all well-formed MAPD problem instances. We presented two decoupled MAPD algorithms, TP and TPTS, and one centralized MAPD algorithm, CENTRAL. They reduce MAPD to one-shot target-assignment and path-planning sub-problems for single or multiple agents in different ways and then iteratively apply one-shot target-assignment and path-planning algorithms to solve these sub-problems. Theoretically, we proved that all three MAPD algorithms are long-term robust for well-formed MAPD problem instances. Experimentally, we compared them in a simulated warehouse environment. The MAPD algorithms in increasing order of their makespans and service times tend to be: CENTRAL, TPTS, and TP. The MAPF algorithms in increasing order of their runtimes per time step tend to be: TP, TPTS, and CENTRAL. In particular, TP is the best choice when real-time computation is of primary concern since it remains efficient for MAPD problem instances with hundreds of agents and tasks. Therefore, it

has the potential to address real-time long-term coordination of autonomous target-assignment and path-planning operations for real-world applications of large-scale multi-agent systems.

To summarize, in this chapter, we validated the hypothesis that formalizing and studying new variants of MAPF can result in new algorithms for the long-term coordination of autonomous target-assignment and path-planning operations of teams of agents. In Chapter 7, we conduct a study in a simulated multi-agent system that includes kinematic constraints of real-world agents and thus confirm the benefits of our MAPD algorithms for real-world applications of multi-agent systems.

Chapter 7

MAPD with Kinematic Constraints in a Simulated System

In this chapter, we present the fourth major contribution of this dissertation. Specifically, we conduct a case study of MAPD with kinematic constraints of real-world agents in a simulated system. We demonstrate how MAPD algorithms can take kinematic constraints (see Section 1.1) of real-world agents into account, guarantee a safety distance between the agents, and remain long-term robust for all well-formed MAPD problem instances. Specifically, we demonstrate how to adapt the MAPD algorithm TP (see Section 6.4) to this simulated system to take kinematic constraints of real-world agents into account directly during planning by using a novel one-shot single-agent path-planning algorithm, called *Safe Interval Path Planning with Reservation Table (SIPPwRT)*. The resulting algorithm *TP-SIPPwRT* computes plan-execution schedules that can be safely executed by real-world agents with off-the-shelf controllers. We also compare TP-SIPPwRT against a baseline method that uses the existing existing polynomial-time procedure MAPF-POST (see Section 3.3) in a post-processing step to transform discrete MAPD solutions produced by the MAPD algorithms presented in Chapter 6 into plan-execution schedules. Experimental results on both an agent simulator and a robot simulator validate the hypothesis that

This chapter is based on Ma, H., Hönig, W., Kumar, T. K. S., Ayanian, N., & Koenig, S. (2019). Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *AAAI Conference on Artificial Intelligence* (pp. 7651–7658).

our MAPD algorithms introduced in both Chapters 6 and 7 are potentially applicable to and thus can benefit real-world applications of multi-agent systems.

The remainder of this chapter is structured as follows. In Section 7.1, we reiterate the motivation behind studying MAPD with kinematic constraints in a simulated system. In Section 7.2, we state the assumptions used in this chapter and describe how TP-SIPPwRT uses them to take kinematic constraints into account directly during planning. In Section 7.3, we present SIPPwRT that is used by TP-SIPPwRT to solve one-shot single-agent path-planning problems. In Section 7.4, we analyze the properties of TP-SIPPwRT. In Section 7.5, we provide the specification of the simulated system used in our experiments. In Section 7.6, we experimentally evaluate TP-SIPPwRT on both an agent simulator and a robot simulator and compare it with using MAPF-POST in a post-processing step to transform discrete solutions produced by previous MAPF algorithms into plan-execution schedules. Finally, we summarize the contributions of this chapter in Section 7.7.

7.1 Introduction

In this chapter, we develop algorithmic techniques that take kinematic constraints (see Section 1.1) of real-world agents into account in their solutions of the long-term combined target-assignment and path-planning problem MAPD. Simulated systems that model such constraints provide an important test bed for demonstrating the potential of applying multi-agent coordination algorithms to real-world applications of multi-agent systems. Existing target-assignment and path-planning algorithms do not directly consider kinematic constraints of real-world agents. In Chapter 6, we have developed three MAPD algorithms that are long-term robust for well-formed MAPD problem instances. However, these MAPD algorithms, like MAPF and TAPF algorithms that we have covered in Chapters 3 and 5, assume very simple agent models. Their solutions thus cannot be safely executed by teams of real-world agents. For example, all algorithms we have introduced so far assume discrete agent movements with uniform velocity and thus make it challenging for real-world agents, such as differential-drive robots, to execute

the computed solutions. As we have discussed in Sections 3.3 and 5.5, it has been verified that MAPF and TAPF algorithms can use MAPF-POST in a post-processing step to transform their solutions into plan-execution schedules that take kinematic constraints (sizes and velocities) of real-world agents into account and provide a guaranteed safety distance between them (Hönig, Kumar, Cohen, et al., 2016; Hönig, Kumar, Ma, et al., 2016). However, the resulting plan-execution schedules might then not be effective since planning is oblivious to this transformation. For example, one of the optimal solutions for the MAPF problem instance shown in Figure 2.1 is $\{\pi_1 = \langle \mathbb{B}, \mathbb{C}, \mathbb{D} \rangle, \pi_2 = \langle \mathbb{A}, \mathbb{A}, \mathbb{C}, \mathbb{E} \rangle\}$. The plan-execution schedule produced by MAPF-POST for this MAPF solution enforces a temporal constraint (see Section 3.3) that agent a_1 enters vertex $\mathbb{C}$ before agent a_2 enters it. However, if we assume that agent a_1 always moves with a lower velocity than agent a_2, then both agents might arrive at their goal vertices earlier if the faster agent a_2 enters vertex $\mathbb{C}$ before the slower agent a_1 enters it. The potential loss in effectiveness might even be larger for MAPD since the effect can cascade. It is thus not known how well MAPF-POST applies to MAPD. In general, it remains unclear whether and, if so, how and how well MAPD algorithms can include kinematic constraints of real-world agents in their solutions as their computation is online and their planning horizon is long.

Therefore, we demonstrate how MAPD algorithms can take kinematic constraints of real-world agents into account. Specifically, we focus on TP (see Section 6.4), the only algorithm that remains efficient for MAPD problem instances with hundreds of agents and tasks. TP needs to repeatedly solve one one-shot single-agent path-planning problem for one agent at a time, while avoiding collisions with the paths of the other agents. We show how TP can be made even more efficient by using SIPPwRT, our contribution to improving SIPP (Phillips & Likhachev, 2011) for MAPD and many other problems. Most importantly, we show how TP can be made more general by letting SIPPwRT take kinematic constraints of real-world agents into account directly to compute continuous agent movements. The resulting MAPD algorithm TP-SIPPwRT directly computes plan-execution schedules and remains long-term robust for well-formed MAPD problem instances. Experimentally, we demonstrate that TP-SIPPwRT is more efficient and effective than the baseline method that uses MAPF-POST in a post-processing step to transform MAPD

solutions with discrete agent movements into plan-execution schedules with continuous agent movements. We demonstrate the benefits of our methods in a simulated automated warehouse system using both an agent simulator and a standard robot simulator. For example, we demonstrate that TP-SIPPwRT can compute solutions for hundreds of agents and thousands of tasks in seconds of runtime.

7.2 Assumptions and TP-SIPPwRT

We follow the problem definition of MAPD (see Section 2.5) but make the following assumptions throughout this chapter, even though TP-SIPPwRT could easily be generalized beyond them, mostly because these assumptions have been proven to work for many types of real-world agents, such as differential-drive robots, with off-the-shelf controllers (Hönig, Kumar, Cohen, et al., 2016; Hönig et al., 2019; Hönig, Kumar, Ma, et al., 2016) and make it easier to (1) compare TP-SIPPwRT with using MAPF-POST to post-process MAPD solutions with discrete agent movements and (2) explain TP-SIPPwRT itself:

- The given graph G is a 2D 4-neighbor grid with blocked and unblocked cells of size $L \times L$ each.
- Each agent a_i is a circular agent (disk) with (safety) radius $R_i \leq L/2$. We use its center as its reference point. The radii thus take agent sizes and safety distances into account.
- The *configuration* of an agent is a pair of its cell and orientation (main compass direction).
- Time is continuous.
- Each agent starts from its given initial configuration at time 0 and always moves from the center of its current unblocked cell to the center of a neighboring unblocked cell via the following available actions: a *point turn* of $\pi/2$ rads (ninety degrees) in either the clockwise or counterclockwise direction with a given *rotational velocity*, a *wait* in (the center of) their current unblocked cell, and a *forward movement* to (the center of) the neighboring cell with a given *translational velocity*.

- The agents can accelerate and decelerate infinitely fast.

For the robot simulator used in the experiments, we use controllers for real-world agents that approximate the above assumptions.

In Section 7.3.6, we argue that, for TP-SIPPwRT, the above assumptions on actions are equivalent to using only *turn-and-move* actions, each consisting of a point turn in one of the four compass directions followed by a wait (when necessary) and then a forward movement to a neighboring unblocked cell. A *path* of agent a_i is a sequence of configurations, each associated with the (continuous) time when the agent arrives in it, where any two consecutive configurations are connected by a turn-and-move action. The paths of all agents, when combined with the given rotational and translational velocities, provide sufficient statistics for the agents to follow them during execution under the above assumptions on agent movements and are thus also a plan-execution schedule for the agents. The paths of two agents are free of collisions if and only if the interiors of the agent disks never intersect when they follow their paths.

For ease of exposition, we refer to a sequence of cells as defined in the previous chapters for algorithms that assume discrete agent movements also as a path, as defined in previous chapters.

7.3 SIPPwRT

Recall that TP (Algorithm 6.1) assumes discrete agent movements in the main compass directions with a uniform velocity of typically one cell per time unit (time step). Agents may repeatedly call Functions *Path1* [Line 10] or *Path2* [Line 14], that each plans a time-minimal path for itself from one endpoint to another, considering the other agents as dynamic obstacles that follow their paths in the token and with which collisions need to be avoided. The agents use space-time A* (see Section 3.2.1.1) for these one-shot single-agent path-planning sub-problems.

TP-SIPPwRT replaces space-time A* with SIPPwRT, a version of SIPP that computes continuous forward movements and point turns with given velocities rather than discrete agent movements in the main compass directions with uniform velocity.

7.3.1 A* Search of SIPP

Space-time A* and SIPP are two versions of A* that both plan time-minimal paths for agents from their current cells to given goal cells, considering the other agents as dynamic obstacles that follow their paths and with which collisions need to be avoided. They both assume discrete agent movements in the main compass directions with a uniform velocity of typically one cell per time unit on a grid. Space-time A* operates on pairs of cells and time steps, while SIPP groups contiguous time steps during which a cell is not occupied into safe (time) intervals, that are closed intervals with a lower and upper (time) bound each, for that cell and thus operates on pairs of cells and safe intervals. This affords the A* search of SIPP pruning opportunities because it is always preferable for an agent to arrive at a cell earlier during the same safe interval since it can then simply wait at the cell. Thus, if the A* search of SIPP has already found a path that arrives at some cell at some time during some safe interval and then discovers a path that arrives at the same cell at a later time in the same safe interval, then it can prune the latter path without losing optimality.

SIPP has already been used for robotics applications (Narayanan, Phillips, & Likhachev, 2012; Yakovlev & Andreychuk, 2017). We generalize it to continuous forward movements and point turns with given velocities in the following, where a safe interval for a cell is now a maximal contiguous interval during which the cell is not occupied by dynamic obstacles. Since SIPPwRT, the resulting version of SIPP, is guaranteed to discover collision-free paths (like space-time A*) when used as part of TP, TP-SIPPwRT, like TP, remains long-term robust for all well-formed MAPD problem instances.

7.3.2 Reservation Table and Safe Intervals

SIPP represents the path of each dynamic obstacle as a chronologically ordered sequence of cells occupied by the dynamic obstacle, which is not efficient since SIPP has to iterate through all these sequences to calculate all safe intervals of a given cell. On the other hand, space-time A* maintains a reservation table that is indexed by a cell and a time step, which allows for

the efficient calculation of all safe time steps of a given cell. For example, in Section 3.2.1.3, when Cooperative A* uses a space-time A* search to plan a path for agent a_2 for the MAPF problem instance shown in Figure 2.1, it maintains a reservation table that is indexed by the cell and the time step of the vertex constraints $\langle a_2, \mathbb{B}, 0\rangle$, $\langle a_2, \mathbb{C}, 1\rangle$, $\langle a_2, \mathbb{D}, 2\rangle$, $\langle a_2, \mathbb{D}, 3\rangle$, ... that are imposed by the dynamic obstacles (in this example, only one dynamic obstacle, namely agent a_1, exists). Therefore, when the space-time A* search tries to expand state $\langle \mathbb{A}, 0\rangle$, it can efficiently infer from the reservation table that state $\langle \mathbb{C}, 1\rangle$ is unreachable (removed from the state space). A space-time A* search often uses a similar reservation table that is indexed by an ordered pair of cells (thus an edge traversal) and a time step for edge constraints.

SIPPwRT improves upon SIPP by using a version of a reservation table that handles continuous agent movements with given velocities and is indexed by a cell. A reservation table entry of a given cell is a priority queue that contains all reserved intervals, that are open intervals with a lower and upper (time) bound each, for that cell in increasing order of their lower bounds. A reserved interval for a cell is a maximal contiguous interval during which the cell is occupied by some dynamic obstacle. The reservation table allows SIPPwRT to implement all operations efficiently that are needed by TP-SIPPwRT, namely to (1) calculate all safe intervals of a given cell, (2) add reservation table entries after a new path has been calculated, and (3) delete reservation table entries that refer to irrelevant times in the past in order to keep the reservation table small.

7.3.2.1 Function *GetSafeIntervals*

Function *GetSafeIntervals*(*cell*) returns all safe intervals for cell *cell* in increasing order of their lower bounds. The safe intervals for the cell are obtained as the complements of the reserved intervals for the cell with respect to interval $[0, \infty]$. For safe interval $i = [i.lb, i.ub]$ and a dynamic obstacle departing from cell *cell* at time $i.lb$, *dep_cfg*$[cell, i]$ is the configuration of the dynamic obstacle at time $i.lb$[1]. It is *NULL* if and only if $i.lb \leq$ *current_t*, where *current_t* is the time step when the agent starts at a given start configuration. Similarly, for safe interval

[1]We strictly mean the instant before time $i.lb$. We are not strict when it comes to the boundary cases of a safe or reserved interval because they do not affect our algorithm.

$i = [i.lb, i.ub]$ and the dynamic obstacle arriving at cell *cell* at time $i.ub$, *arr_cfg*$[cell, i]$ is the configuration of a dynamic obstacle at time $i.ub$. It is *NULL* if and only if $i.ub = \infty$.

7.3.3 Time Offsets

The safe intervals of a cell represent the times during which the cell is not occupied. However, this does not mean that an agent can arrive at any of the times at the cell since the agent might still collide with a dynamic obstacle that has just departed from the cell or is about to arrive at the cell. Thus, the lower and upper bounds of a safe interval have to be tightened using the following time offsets.

Function *Offset*(*cfg1*, *cfg2*) returns the time offset ΔT that expresses the minimum amount of time (the center of) some unknown agent a_1 with safety radius R_1 and translational velocity $\boldsymbol{v}_{trans,1}$ needs to depart from (the center of) a cell $cfg1.cell = l$ with configuration *cfg1* before (the center of) some unknown agent a_2 with safety radius R_2 and translational velocity $\boldsymbol{v}_{trans,2}$, coming from some cell l', arrives at (the center of) the same cell $cfg2.cell = l$ with configuration *cfg2* to avoid a collision. The time offset is zero if and only if either $cfg1 = NULL$ or $cfg2 = NULL$, meaning that either agent a_1 or agent a_2 does not exist. The calculation of the time offset requires only knowledge of the configurations, safety radii, and velocities of both agents.[2]

Assume that agent a_2 departs from cell l' at time 0 (and thus arrives at cell l at time $t' = \frac{L}{\boldsymbol{v}_{trans,2}}$) and agent a_1 departs from cell l at time $t_d \leq \frac{L}{\boldsymbol{v}_{trans,2}}$. $D(t)$ is the distance between the (centers of the) agents, where t is the amount of time elapsed after agent a_2 departs from cell l'. It must hold that $D(t) \geq R_1 + R_2$ to avoid that the two agents collide. We distinguish three cases to calculate the time offset ΔT:

[2] In the pseudocode of SIPPwRT, we show how to keep track of the configurations but do not include the safety radii and velocities in the configurations for ease of readability (although this needs to be done in case they are not the same for all agents and times).

7.3.3.1 Same Direction

Both agents move in the same direction (meaning that the orientations of configurations *cfg1* and *cfg2* are equal), see Figure 7.1 (left), where gray lines connect the centers of cells. In this case, $D(t) = L - \boldsymbol{v}_{trans,2}t + \boldsymbol{v}_{trans,1}(t - t_d)$ for $t \geq t_d$. We now distinguish two sub-cases to show that the time offset is $\Delta T = \frac{R_1+R_2}{\min(\boldsymbol{v}_{trans,1}, \boldsymbol{v}_{trans,2})}$.

(1) Case $\boldsymbol{v}_{trans,1} < \boldsymbol{v}_{trans,2}$. This case is shown in Figure 7.1 (middle). $D(t)$ decreases as the time t increases. $D(t)$ thus reaches its minimum at the time $t = t'' = \frac{L}{\boldsymbol{v}_{trans,2}}$ when agent a_2 arrives at cell l. Substituting $t = \frac{L}{\boldsymbol{v}_{trans,2}}$ back into $D(t) \geq R_1 + R_2$, we have

$$D(t) = L - \boldsymbol{v}_{trans,2}t + \boldsymbol{v}_{trans,1}(t - t_d) = \boldsymbol{v}_{trans,1}(\frac{L}{\boldsymbol{v}_{trans,2}} - t_d) \geq R_1 + R_2.$$

Therefore, $t_d \leq \frac{L}{\boldsymbol{v}_{trans,2}} - \frac{R_1+R_2}{\boldsymbol{v}_{trans,1}}$. The time offset ΔT is thus

$$\Delta T = t' - \max t_d = \frac{L}{\boldsymbol{v}_{trans,2}} - (\frac{L}{\boldsymbol{v}_{trans,2}} - \frac{R_1 + R_2}{\boldsymbol{v}_{trans,1}}) = \frac{R_1 + R_2}{\boldsymbol{v}_{trans,1}}.$$

(2) Case $\boldsymbol{v}_{trans,1} \geq \boldsymbol{v}_{trans,2}$. This case is shown in Figure 7.1 (right). $D(t)$ decreases before agent a_1 starts to move and then increases as the time t increases. $D(t)$ thus reaches its minimum at the time $t = t_d$ when agent a_1 starts to move. Substituting $t = t_d$ back into $D(t) \geq R_1 + R_2$, we have

$$D(t) = L - \boldsymbol{v}_{trans,2}t + \boldsymbol{v}_{trans,1}(t - t_d) = L - \boldsymbol{v}_{trans,2}t_d \geq R_1 + R_2.$$

Therefore, $t_d \leq \frac{L-(R_1+R_2)}{\boldsymbol{v}_{trans,2}}$. The time offset is thus

$$\Delta T = t' - \max t_d = \frac{L}{\boldsymbol{v}_{trans,2}} - \frac{L - (R_1 + R_2)}{\boldsymbol{v}_{trans,2}} = \frac{R_1 + R_2}{\boldsymbol{v}_{trans,2}}.$$

7.3.3.2 Orthogonal Directions

Both agents move in orthogonal directions, see Figure 7.2. In this case, $D(t) = \sqrt{(\boldsymbol{v}_{trans,1}(t - t_d))^2 + (L - \boldsymbol{v}_{trans,2}t)^2}$. We determine the time t at which $D(t) \geq 0$ reaches

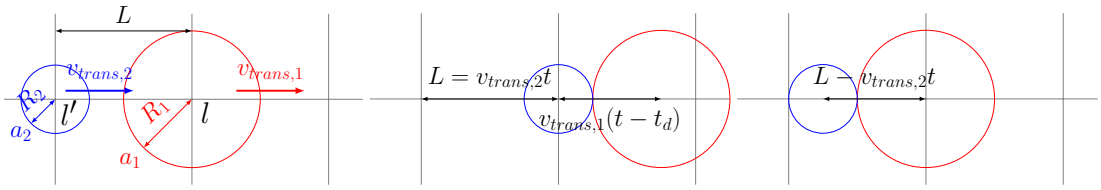


Figure 7.1: Left: Two agents move in the same direction. Middle: D is at its minimum for the $\boldsymbol{v}_{trans,1} < \boldsymbol{v}_{trans,2}$ case. Right: D is at its minimum for the $\boldsymbol{v}_{trans,1} \geq \boldsymbol{v}_{trans,2}$ case.

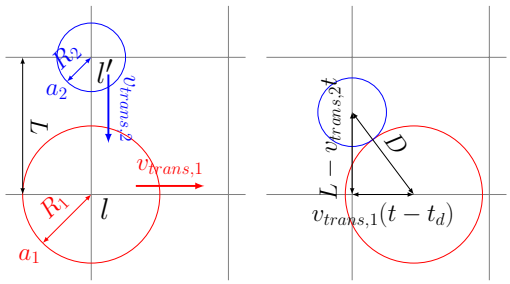


Figure 7.2: Left: Two agents move in orthogonal directions. Right: D is at its minimum.

its minimum by solving $\frac{\partial D^2}{\partial t} = 0$. Substituting the result $t = \frac{\boldsymbol{v}_{trans,1}^2 t_d + \boldsymbol{v}_{trans,2} L}{\boldsymbol{v}_{trans,1}^2 + \boldsymbol{v}_{trans,2}^2}$ into $D^2 \geq (R_1 + R_2)^2$, we have

$$D^2(t) = (L - \boldsymbol{v}_{trans,2}t)^2 + (\boldsymbol{v}_{trans,1}(t - t_d))^2 = \frac{(\boldsymbol{v}_{trans,1}(\boldsymbol{v}_{trans,2}t_d - L))^2}{(\boldsymbol{v}_{trans,1}^2 + \boldsymbol{v}_{trans,2}^2)} \geq (R_1 + R_2)^2.$$

Since $L \geq \boldsymbol{v}_{trans,2}t_d$, we have $\boldsymbol{v}_{trans,1}(L - \boldsymbol{v}_{trans,2}t_d) \geq \sqrt{\boldsymbol{v}_{trans,1}^2 + \boldsymbol{v}_{trans,2}^2}(R_1 + R_2)$. Therefore, $t_d \leq \frac{\boldsymbol{v}_{trans,1}L - \sqrt{\boldsymbol{v}_{trans,1}^2 + \boldsymbol{v}_{trans,2}^2}(R_1 + R_2)}{\boldsymbol{v}_{trans,1}\boldsymbol{v}_{trans,2}}$. The time offset is thus

$$\begin{aligned}\Delta T = t' - \max t_d &= \frac{L}{\boldsymbol{v}_{trans,2}} - \frac{\boldsymbol{v}_{trans,1}L - \sqrt{\boldsymbol{v}_{trans,1}^2 + \boldsymbol{v}_{trans,2}^2}(R_1 + R_2)}{\boldsymbol{v}_{trans,1}\boldsymbol{v}_{trans,2}} \\ &= \frac{\sqrt{\boldsymbol{v}_{trans,1}^2 + \boldsymbol{v}_{trans,2}^2}(R_1 + R_2)}{\boldsymbol{v}_{trans,1}\boldsymbol{v}_{trans,2}}.\end{aligned}$$

7.3.3.3 Opposite Directions

Both agents move in opposite directions, that is, agent a_1 moves from cell l to cell l' and agent a_2 moves from cell l' to cell l. The time offset is set to allow agent a_1 to arrive at cell l' even before agent a_2 departs from cell l'. In this case, the time offset is

$$\Delta T = \frac{L}{\boldsymbol{v}_{trans,1}} + \frac{L}{\boldsymbol{v}_{trans,2}},$$

which is the sum of the times that agent a_1 needs to move from cell l to cell l' and that agent a_2 needs to move from cell l' to cell l. We show in Section 7.4 that SIPPwRT avoids collisions when it uses all bounds simultaneously.

7.3.4 Increased/Decreased Bounds

The algorithm calls the following functions to tighten the lower and upper bounds of safe interval i during which an agent can stay at cell $l = cfg.cell$ safely.

The algorithm calls Function *GetLB1*(cfg, i) for an agent a_2 to return $\max_j(j.lb +$ *Offset*$(dep\_cfg[cfg.cell, j], cfg))$. Here, $j.lb +$ *Offset*$(dep\_cfg[cfg.cell, j], cfg)$ is the increased lower bound for each safe interval j in *GetSafeIntervals*$(cfg.cell)$ with $j.lb \leq i.lb$. For agent a_2 that arrives at cell $l = cfg.cell$ from another cell l' with configuration *cfg*, the idea is to prevent it from colliding with any dynamic obstacle a_1 that departs from cell l with configuration $dep\_cfg[cfg.cell, j]$ before a_2 arrives at cell l.

The algorithm calls Function *GetUB1*(cfg, i) for an agent a_1 to return $\min_j(j.ub -$ *Offset*$(cfg, arr\_cfg[cfg.cell, j]))$. Here, $j.ub -$ *Offset*$(cfg, arr\_cfg[cfg.cell, j])$ is the decreased upper bound for each safe interval j in *GetSafeIntervals*$(cfg.cell)$ with $j.ub \geq i.ub$. For agent a_1 that departs from cell $l = cfg.cell$ with configuration *cfg*, the idea is to prevent it from colliding with any dynamic obstacle a_2 that arrives at cell l from another cell l' with configuration $arr\_cfg[cfg.cell, j]$ after a_1 departs from cell l.

The algorithm calls Function *GetLB2*(cfg, i) for an agent a_1 to return $\max_j(j.lb + \frac{L}{\boldsymbol{v}'_{trans}} - \frac{L}{\boldsymbol{v}_{trans}})$. Here, $j.lb + \frac{L}{\boldsymbol{v}'_{trans}} - \frac{L}{\boldsymbol{v}_{trans}}$ is the increased lower bound for each safe interval j in

GetSafeIntervals$(cfg.cell)$ where the orientation of *dep_cfg*$[cfg.cell, j]$ is the same as that of *cfg* and $j.lb \leq i.lb$. For agent a_1 that departs from cell $l = cfg.cell$ with configuration *cfg* and translational velocity $\boldsymbol{v}_{trans}$ and moves also toward cell l', the idea is to prevent it from arriving at cell l' earlier than (and thus "passing through") any dynamic obstacle a_2 that departs from cell l before agent a_1 with configuration *dep_cfg*$[cfg.cell, j]$ and translational velocity $\boldsymbol{v}'_{trans}$ and moves also toward cell l'.

The algorithm calls Function *GetUB2*(cfg, i) for an agent a_1 to return $\min_j(j.lb + \frac{L}{\boldsymbol{v}'_{trans}} - \frac{L}{\boldsymbol{v}_{trans}})$. Here, $j.lb + \frac{L}{\boldsymbol{v}'_{trans}} - \frac{L}{\boldsymbol{v}_{trans}}$ is the decreased upper bound for each safe interval j in *GetSafeIntervals*$(cfg.cell)$ where the orientation of *dep_cfg*$[cfg.cell, j]$ is the same as that of *cfg* and $j.lb \geq i.ub$. For agent a_1 that departs from cell $l = cfg.cell$ with configuration *cfg* and translational velocity $\boldsymbol{v}_{trans}$ and moves also toward cell l', the idea is to prevent it from arriving at cell l' later than (and thus "being passed through" by) any dynamic obstacle a_2 that departs from cell l after agent a_1 with configuration *dep_cfg*$[cfg.cell, j]$ and translational velocity $\boldsymbol{v}'_{trans}$ and moves toward cell l'.

7.3.5 Admissible h-Values for Multiple Targets

TP-SIPPwRT combines Lines 7-10 of Algorithm 6.1 by letting Function *Path1* directly compute a time-minimal path from its current configuration to a configuration whose cell is the pickup cell of any task in the candidate task set $\mathcal{T}'$, thus solving the one-shot single-agent target-assignment and path-planning problems jointly for a free agent a_i, and then a time-minimal path from there to the delivery cell of the task after the agent becomes occupied. Experimentally, combining target assignment and path planning for each free agent produces solutions that are as effective as, if not more effective than, using the pre-computed h-values for target assignment, which does not take other paths in the token into account, and then planning paths to execute the assigned tasks.

Therefore, Function *Path1* of TP-SIPPwRT requires an agent to use SIPPwRT twice, namely (1) to plan a time-minimal path from its current configuration to a candidate set of endpoints (pickup cells of tasks in the candidate task set $\mathcal{T}'$) and (2) to plan a time-minimal path from the

resulting configuration to a particular endpoint (a delivery cell). TP-SIPPwRT uses the same Function *Path2* (Line 14 of Algorithm 6.1), that requires the agent to use SIPPwRT once to plan a time-minimal path from its current configuration to a candidate set of endpoints (to avoid deadlocks). The agent always moves along each path with given (fixed) translational and rotational velocities (unless it waits). Therefore, SIPPwRT has to plan only paths to a given set V_{Goal} of one or more endpoints.

Like in TP, by ignoring the dynamic obstacles, we determine the admissible h-values needed for the A* search of SIPPwRT to plan time-minimal paths as follows: We calculate a time-minimal path (that excludes waiting) for the agent from each configuration *cfg* to each configuration cfg' whose cell is an endpoint (by searching backward once from each configuration cfg'). We then use the minimum heuristic (Stern, Goldenberg, & Felner, 2017) $h(cfg, V_{Goal}) = \min_{cfg'.cell \in V_{Goal}} h(cfg, cfg')$ as admissible h-value of configuration *cfg*, where $h(cfg, cfg')$ is the calculated time of the time-minimal path from *cfg* to cfg'. In practice, if set V_{Goal} is large and endpoints are densely distributed across the grid, it is more efficient to use $h(cfg, V_{Goal}) = 0$ (as we do for Function *Path2* of TP-SIPPwRT) since it can be calculated faster even though SIPPwRT might expand more nodes.

7.3.6 Pseudocode of SIPPwRT

Algorithm 7.2 shows the pseudocode of SIPPwRT, which plans a time-minimal path for an agent with translational velocity $\boldsymbol{v}_{trans}$ and rotational velocity $\boldsymbol{v}_{rot}$ from its configuration *start_cfg* at time *current_t* to a cell in set V_{Goal}. SIPPwRT performs a regular A* search with nodes that are pairs of configurations of the agent and safe intervals. The g-value $g[n]$ of a node $n = \langle n.cfg, n.int \rangle$ with configuration $n.cfg$ and safe interval $n.int = [n.int.lb, n.int.ub]$ is the earliest discovered time in $n.int$ when the agent can be in configuration $n.cfg$. The start node is $n = \langle start\_cfg, [current\_t, \infty] \rangle$ with $g[n] = current\_t$. The safe interval $n.int$ of the start node expresses that the agent can wait forever in its current configuration. A node n is a goal node if

and only if the cell of its configuration is in set V_{Goal} and the agent can wait forever in its configuration ($n.int.ub = \infty$), that is, the agent can rest in the cell of its configuration, as required by TP.

In our implementation of SIPPwRT, each action is a *turn-and-move* action. Since only forward movements define the temporal constraints between safe intervals of neighboring cells, the state space of our search remains unaffected by the use of turn-and-move actions instead of separate point turn, wait, and move actions independently.

Algorithm 7.1: Function *GetSuccessors*

```
1  Function GetSuccessors(n)
2      successors ← ∅;
3      foreach legal turn-and-move action in n do
4          cfg_t ← configuration resulting from executing the point turn of action in n.cfg
             (cfg_t.cell = n.cfg.cell);
5          ub1 ← GetUB1(cfg_t, n.int);
6          lb2 ← GetLB2(cfg_t, n.int);
7          ub2 ← GetUB2(cfg_t, n.int);
8          lb ← max((g[n] + Δt_turn(action, v_rot)), lb2);
9          ub ← min(ub1, ub2);
10         if lb ≤ ub then
11             cfg′ ← configuration resulting from executing the forward movement of
                 action in n.cfg_t;
12             i′.lb ← lb + Δt_move(action, v_trans);
13             i′.ub ← ub + Δt_move(action, v_trans);
14             safeIntervals ← GetSafeIntervals(cfg′.cell);
15             foreach i″ ∈ safeIntervals do
16                 lb1 ← GetLB1(cfg′, i″);
17                 if [lb1, i″.ub] ∩ i′ ≠ ∅ then
18                     t′ ← max(i′.lb, lb1);
19                     n′ ← NewNode(⟨cfg′, i″⟩);
20                     cost[n, n′] ← t′ − g[n];
21                     successors ← successors ∪ {n′};
22     return successors
```

7.3.6.1 Function *GetSuccessors*

Algorithm 7.1 shows the pseudocode of Function *GetSuccessors*(n), which calculates the successors of node n by considering all legal turn-and-move actions *action* available to the agent in configuration $n.cfg$ [Line 3]. Assume that executing the point turn of action *action* takes $\Delta t_{turn}(action, \boldsymbol{v}_{rot})$ time units and results in configuration *cfg_t* with which the agent departs from its current cell [Line 4]. The agent must depart from its current cell no later than *lb* and no earlier than *ub* to avoid colliding with dynamic obstacles that also visit its current cell [Lines 5-9]. If the agent can depart from its current cell [Line 10], then assume that executing the forward movement of action *action* in configuration *cfg_t* takes $\Delta t_{move}(action, \boldsymbol{v}_{trans})$ time units and results in successor configuration cfg' [Line 11]. The agent waits an appropriate amount of time in configuration *cfg_t* after the point turn, then executes the forward movement, and arrives in configuration cfg' in interval $i' = [lb + \Delta t_{move}(action, \boldsymbol{v}_{trans}), ub + \Delta t_{move}(action, \boldsymbol{v}_{trans})]$ [Lines 12-13]. The successors of node n are generated by processing all safe intervals $i'' = [i''.lb, i''.ub]$ for the new cell $cfg'.cell$ of the agent [Lines 14-15]. The lower bound of safe interval i'' is increased from $i''.lb$ to *lb1* to ensure that the agent can arrive at its new cell without colliding with dynamic obstacles that also visit its new cell [Line 16]. The updated safe interval $[lb1, i''.ub]$ is intersected with interval i' [Line 17]. If their intersection is non-empty, then the agent can arrive at its successor configuration during safe interval i''. Only the earliest time t' in the intersection needs to be considered (since the agent can simply wait in its successor configuration and the later times in the intersection can thus be pruned, as argued earlier) [Line 18]. The resulting successor of node n is $n' = \langle cfg', i'' \rangle$ [Line 19], and the cost (here: time) of the transition from node n to node n' is $cost[n, n'] = t' - g[n]$ [Line 20] (consisting of executing the point turn of action *action* for $\Delta t_{turn}(action, \boldsymbol{v}_{rot})$ time units, waiting for $t' - g[n] - \Delta t_{turn}(action, \boldsymbol{v}_{rot}) - \Delta t_{move}(action, \boldsymbol{v}_{trans})$ time units, and then executing the forward movement of action *action* for $\Delta t_{move}(action, \boldsymbol{v}_{trans})$ time units), so that $g[n'] = g[n] + cost[n, n'] = t'$ is the earliest discovered time in $n'.int = i''$ when the agent can be in configuration $n'.cfg = cfg'$.

Algorithm 7.2: SIPPwRT

Input: *start_cfg*, V_{Goal}, *current_t*, $\boldsymbol{v}_{trans}$, $\boldsymbol{v}_{rot}$

```
1  n_start ← NewNode(⟨start_cfg, [current_t, ∞]⟩);
2  g[n_start] ← current_t;
3  OPEN ← {n_start};
4  while OPEN ≠ ∅ do
5      n ← arg min_{n'∈OPEN}(g[n'] + h(n'.cfg, G));
6      OPEN ← OPEN \ {n};
7      if n.cell ∈ V_Goal and n.int.ub = ∞ then
8          return path from start_cfg to n.cfg;
9      successors ← GetSuccessors(n);
10     foreach n' ∈ successors do
11         if g[n'] is undefined then
12             g[n'] ← ∞;
13         if g[n'] > g[n] + cost[n, n'] then
14             parent[n'] ← n;
15             g[n'] ← g[n] + cost[n, n'];
16             if n' ∉ OPEN then
17                 OPEN ← OPEN ∪ {n'};
18 return "No Path Exists" (does not happen for well-formed MAPD problem instances);
```

7.3.6.2 Main Routine

The main routine of SIPPwRT performs a regular A* search. It initializes the g-value of the start node and inserts the node into the OPEN list [Lines 1-3]. It then repeatedly removes a node n with the smallest sum of g-value and h-value $g[n] + h(n.cfg, V_{Goal})$ from the OPEN list [Lines 5-6] and processes it: If the node is a goal node, then it returns the path found by following the parent pointers from the node to the start node [Lines 7-8]. Otherwise, it generates the successors of the node [Line 9]. For each successor, it initializes its g-value to infinity if the g-value is still undefined [Lines 11-12]. It then checks whether the g-value of the successor can be reduced by changing its parent pointer to node n [Line 13]. If so, it changes the parent pointer of the successor, reduces its g-value, and inserts it into the OPEN list (if necessary) [Lines 14-17].

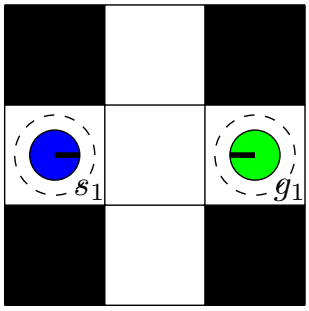


Figure 7.3: Example of a MAPD problem instance.

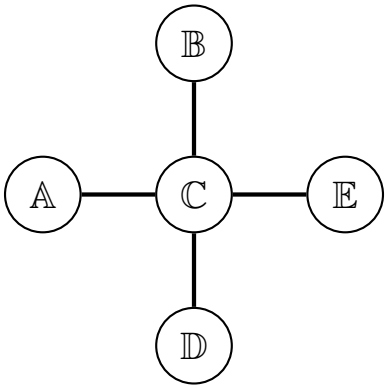


Figure 7.4: Graph representation of the MAPD problem instance shown in Figure 7.3.

7.3.7 Example

We now use an example to describe how SIPPwRT computes a path. Figure 7.3 shows an example of a MAPD problem instance on a 2D 4-neighbor grid with cells of size $1\,\mathrm{m} \times 1\,\mathrm{m}$ each. Colored circles are agents. The bar on each circle represents the orientation of each agent. Dashed circles represent pickup and delivery cells. Figure 7.4 shows the corresponding graph representation of the 2D 4-neighbor grid. Two free agents, a_1 (in blue) with initial configuration $\langle \mathbb{A}, \text{EAST} \rangle$ and a_2 (in green) with initial configuration $\langle \mathbb{E}, \text{WEST} \rangle$, are given. There is only one task τ_1 with $s_1 = \mathbb{A}$ and $g_1 = \mathbb{E}$ which is added to the system at $t = 0$. Cells $\mathbb{B}$ and $\mathbb{D}$ are non-task endpoints. Both agents have a radius of $0.25\,\mathrm{m}$ and always turn with rotational velocity $\boldsymbol{\pi}/2 = 1.57\,\mathrm{rad}$ per time unit. Agent a_1 always moves with transitional velocity $1\,\mathrm{m}$ per time unit, and agent a_2 always moves with transitional velocity $0.5\,\mathrm{m}$ per time unit. We assume that

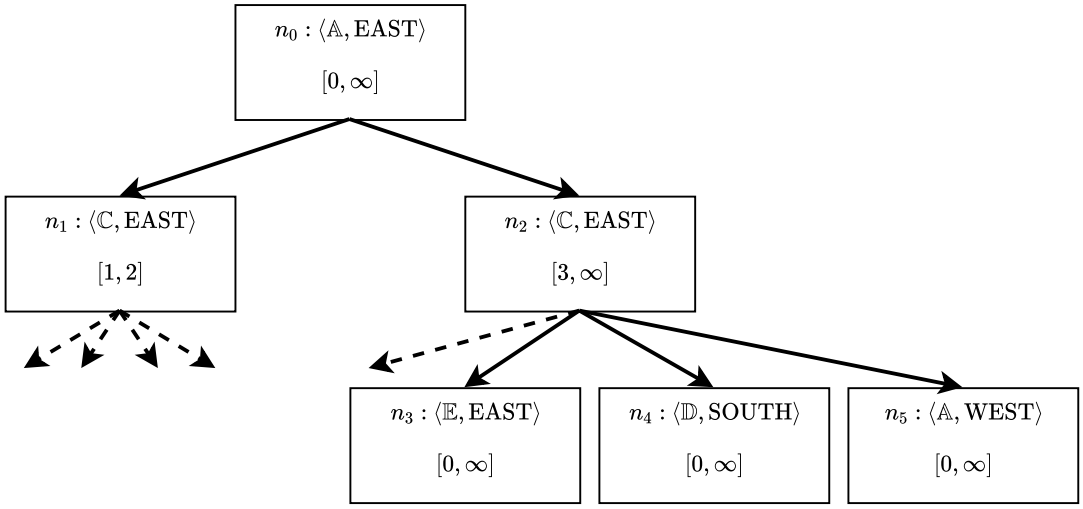


Figure 7.5: The search of SIPPwRT.

agent a_2 is assigned the path $\langle\langle\langle \mathbb{E}, \text{WEST}\rangle, 0\rangle, \langle\langle \mathbb{C}, \text{WEST}\rangle, 2\rangle, \langle\langle \mathbb{B}, \text{NORTH}\rangle, 5\rangle\rangle$ that lets it move to cell $\mathbb{B}$ and rest there. Then, agent a_1 is assigned task τ_1 and needs to compute a path from the pickup cell $\mathbb{A}$ to the delivery cell $\mathbb{E}$.

Figure 7.5 visualizes the search of SIPPwRT for agent a_1. The configuration and safe interval of each node (rectangular box) are shown. Each dashed arrow represents that some legal turn-and-move action (see Line 3 of Algorithm 7.1) does not result in any successors.

The search starts with node $n_0 = \langle\langle \mathbb{A}, \text{EAST}\rangle, [0, \infty]\rangle$ with $g[n_0] = 0$.

When the search expands node n_0, it calls Function *GetSuccessors*(n_0). The only legal turn-and-move action is one that moves to cell $\mathbb{C}$, and the search determines that agent a_1 can safely depart from cell $\mathbb{A}$ between $lb = 0$ and $ub = \infty > lb$ [Lines 8-10] since no dynamic obstacle visits cell $\mathbb{A}$ and affects the bounds. Agent a_1 can thus arrive in cell $\mathbb{A}$ in interval $i' = [1, \infty]$ [Lines 12-13]. The safe intervals of cell $\mathbb{C}$ are $[0, 2]$ and $[3, \infty]$. For safe interval $i'' = [0, 2]$, no dynamic obstacle departs from cell $\mathbb{C}$ before agent a_1 arrives in the cell and affects $lb1$ [Line 16]. The search successfully generates successor n_1 for node n_0 and determines that the earliest time agent a_1 can arrive at cell $\mathbb{C}$ for this safe interval is $t' = 1$ (which is $g[n_1]$ in the main routine) [Lines 17-19]. For safe interval $i'' = [2, 3]$, dynamic obstacle a_2 departs from cell $\mathbb{C}$ at time $t = 3$ in an orthogonal direction before agent a_1 arrives in cell $\mathbb{C}$ from cell $\mathbb{A}$. Function

Offset (the "Orthogonal Directions" case) returns $\Delta T = \frac{\sqrt{5}}{2}$, that increases $lb1$ to $3 + \frac{\sqrt{5}}{2}$ [Line 16]. The search successfully generates successor n_2 for node n_0 and determines that the earliest time agent a_1 can arrive at cell $\mathbb{C}$ for this safe interval is $t' = 3 + \frac{\sqrt{5}}{2}$ (which is $g[n_2]$ in the main routine) [Lines 17-19]. Here, the cost $t' - g[n_0]$ includes waiting at cell $\mathbb{A}$ for $\frac{\sqrt{5}}{2}$ time units.

The search then expands node n_1 in the main routine and calls Function *GetSuccessors*(n_1). The function call does not generate any successors because no legal turn-and-move action results in any successors from the configuration of node n_1. For example, for the turn-and-move action that moves agent a_1 to cell $\mathbb{B}$, no dynamic obstacle affects $lb2$ or $ub2$, but dynamic obstacle a_2, which arrives in cell $\mathbb{C}$ at time $t = 2$ in an orthogonal direction before agent a_1 departs from cell $\mathbb{C}$ for cell $\mathbb{B}$ decreases $ub1$ to $2 - \frac{\sqrt{5}}{2}$ due to the "Orthogonal Directions" case of Function *Offset* [Lines 4-7]. The search determines that agent a_1 can safely depart from cell $\mathbb{C}$ only between $lb = 2$ (including a point turn for 1 time unit) and $ub = 2 - \frac{\sqrt{5}}{2} < lb$, which is not possible [Lines 8-10]. For the turn-and-move action that moves agent a_1 to cell $\mathbb{E}$, no dynamic obstacle affects $lb2$ or $ub2$, but dynamic obstacle a_2, which arrives in cell $\mathbb{C}$ at time $t = 2$ in the opposite direction before agent a_1 departs from cell $\mathbb{C}$ for cell $\mathbb{E}$ decreases $ub1$ to $2 - 3 = -1$ due to the "Opposite Directions" case of Function *Offset* [Lines 4-7] (which implies that agent a_1 has to arrive in cell $\mathbb{E}$ before dynamic obstacle a_2 departs from the cell at time 0 to avoid colliding with the dynamic obstacle along the edge between the cells). The search determines that agent a_1 can safely depart from cell $\mathbb{C}$ only between $lb = 1$ and $ub = -1 < lb$, which is not possible [Lines 8-10].

The search then expands node n_2 in the main routine and calls Function *GetSuccessors*(n_2). The function call successfully generates successors n_3 with $g[n_3] = 4 + \frac{\sqrt{5}}{2}$, n_4 with $g[n_4] = 5 + \frac{\sqrt{5}}{2}$, and n_5 with $g[n_5] = 6 + \frac{\sqrt{5}}{2}$. And when the search expands node n_3, it determines that the node is a goal node and returns the path $\langle\langle\langle\mathbb{A}, \text{EAST}\rangle, 0\rangle, \langle\langle\mathbb{C}, \text{EAST}\rangle, 3 + \frac{\sqrt{5}}{2}\rangle, \langle\langle\mathbb{E}, \text{EAST}\rangle, 4 + \frac{\sqrt{5}}{2}\rangle\rangle$. Figure 7.6 visualizes agent a_1 moving from cell $\mathbb{A}$ to cell $\mathbb{C}$ along this path.

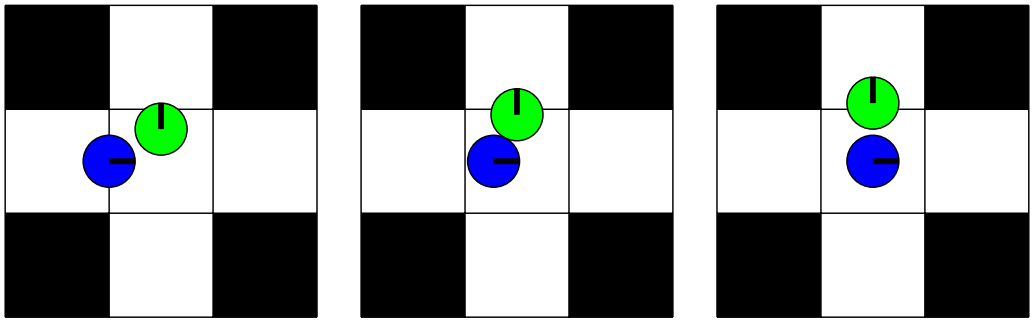


Figure 7.6: Snapshots of agent a_1 following its path. Left [$t = 2 + \frac{\sqrt{5}+1}{2} = 3.62$]: 0.5 time units after agent a_1 departs from cell $\mathbb{A}$. Middle [$t = 3 + \frac{2\sqrt{5}}{5} = 3.89$]: The distance between agents a_1 and a_2 is at its minimum ($0.5\,\mathrm{m}$). Right [$t = 3 + \frac{\sqrt{5}}{2} = 4.12$]: Agent a_1 arrives at cell $\mathbb{C}$.

7.4 Analysis of Properties

We use the following theorem to prove that TP-SIPPwRT is correct (returns collision-free paths) and long-term robust for well-formed MAPD problem instances.

Theorem 7.1. The path returned by SIPPwRT from the start configuration to a goal is free of collisions.

Proof (by induction). Consider a path returned by SIPPwRT. By definition, the agent starts in the first configuration $n_{start}.cfg$ at time *current_t* without any collisions. Assume that the agent arrives at cell $n.cfg.cell$ with configuration $n.cfg$ all the way from the first configuration $n_{start}.cfg$ without any collisions. Let $n'.cfg$ be the (successor) configuration next to configuration $n.cfg$ in the path. As the agent moves from cell $l = n.cfg.cell$ to the next cell $l' = n'.cfg.cell$, we use the following arguments:

(1) Line 6 (7) of Algorithm 7.1, when node n is being expanded, guarantees that any dynamic obstacle, which departs from cell l at a time earlier (later) than when the agent departs from cell l and which then moves in the same direction as the agent, must arrive at the next cell l' earlier (later) than when the agent arrives at cell l'.

(2) Line 5 of Algorithm 7.1, when node n is being expanded, guarantees that the agent does not collide with any dynamic obstacle which arrives at cell l in a non-opposite direction at a time later than when the agent departs from cell l, until the dynamic obstacle has arrived at cell l.

(3) Line 16 of Algorithm 7.1, when node n is being expanded and node n' is being generated, guarantees that the agent does not collide with any dynamic obstacle which departs from cell l' in a non-opposite direction at a time earlier than when the agent arrives at cell l', until the agent has arrived at cell l'. The line also guarantees that, if any dynamic obstacle departs from cell l' in the opposite direction at a time earlier than when the agent arrives at cell l', it also arrives at cell l at a time earlier than when the agent departs from cell l and thus also departs from cell l at a time earlier than when the agent arrives at cell l because its corresponding reserved interval does not intersect with the safe interval $n.\mathit{int}$ for cell l. Therefore, in both cases, the agent does not collide with such a dynamic obstacle (that departs from cell l' at a time earlier than when the agent arrives at cell l') as it moves from cell l to cell l' due to the induction assumption.

(4) For a non-goal node n', Line 5 of Algorithm 7.1, when node n' is being expanded, guarantees that the agent does not collide with any dynamic obstacle which arrives at cell l' in a non-opposite direction at a time later than when the agent departs from cell l', until the dynamic obstacle has arrived at cell l'.

From the induction assumption, it suffices to prove that the agent does not collide with any given dynamic obstacle as the agent departs from cell l until it arrives at cell l'. Assume, for a proof by contradiction, that the agent collides with the dynamic obstacle.

(i) If the dynamic obstacle arrives at cell l at a time earlier than when the agent departs from cell l, it also departs from cell l at a time earlier than when the agent departs from cell l because its corresponding reserved interval does not intersect with the safe interval $n.\mathit{int}$ for cell l. It can collide with the agent only if it then moves from cell l to cell l' at a fixed

velocity $\boldsymbol{v}'_{trans}$ smaller than the fixed velocity $\boldsymbol{v}_{trans}$ at which the agent moves from cell l to cell l'. It must arrive at cell l' at a time earlier than when the agent arrives at cell l' due to (1) and also depart at a time earlier than when the agent arrives at cell l' because its corresponding reserved interval does not intersect with the safe interval $n'.int$ for cell l'. The collision thus remains when the agent arrives at cell l', which contradicts (3).

(ii.a) If the dynamic obstacle arrives at cell l in a non-opposite direction at a time later than when the agent departs from cell l, it does not collide with the agent until it arrives at cell l due to (2). As a consequence, the dynamic obstacle also departs from cell l at a time later than when the agent departs from cell l. It can collide with the agent only if it then moves from cell l to cell l' at a fixed velocity $\boldsymbol{v}'_{trans}$ larger than the fixed velocity $\boldsymbol{v}_{trans}$ at which the agent moves from cell l to cell l'. It must arrive at cell l' at a time later than when the agent arrives at cell l' due to (1) and also later than when the agent departs from cell l' because its corresponding reserved interval does not intersect with the safe interval $n'.int$ for cell l'. Node n' is thus a non-goal node due to Line 7 of Algorithm 7.2 (after n' is popped from OPEN), and the collision thus remains when the agent departs from cell l', which contradicts (4).

(ii.b) If the dynamic obstacle arrives at cell l in the opposite direction at a time later than when the agent departs from cell l, Line 5 of Algorithm 7.1, when node n is being expanded, guarantees that it departs from cell l' at a time later than when the agent arrives at cell l'. The dynamic obstacle thus also arrives at cell l' at a time later than when the agent departs from cell l' because its corresponding reserved interval does not intersect with the safe interval $n'.int$ for cell l'. Node n' is thus a non-goal node due to Line 7 of Algorithm 7.2 (after n' is popped from OPEN). If the dynamic obstacle arrives at cell l' from another cell l'' in the opposite direction to the one in which the agent departs from cell l' (toward cell l''), Line 5 of Algorithm 7.1 (the "Opposite Directions" case of Function *Offset* in Section 7.3.3), when node n' is being expanded, guarantees that the dynamic obstacle arrives at cell l'' at a time later than when the agent departs from cell l', ensuring that the agent

does not collide with the dynamic obstacle until the agent departs from cell l'. Otherwise, any collision thus remains when the agent departs from cell l', which contradicts (4).

(iii) If the dynamic obstacle does not fall into Case (i) but also arrives at cell l' at a time earlier than when the agent arrives at cell l', it must also depart from cell l' earlier than when the agent arrives at cell l' because its corresponding reserved interval does not intersect with the safe interval $n'.\mathit{int}$ for cell l'. Any collision thus remains when the agent arrives at cell l', which contradicts (3).

(iv.a) If node n' is a non-goal node and the dynamic obstacle does not fall into Case (ii.b) but also arrives at cell l' at a time later than when the agent arrives at cell l', the dynamic obstacle must arrive at cell l' later than when the agent departs from cell l' because its corresponding reserved interval does not intersect with the safe interval $n'.\mathit{int}$ for cell l'. We use the same argument as for Case (ii.b): If the dynamic obstacle arrives at cell l' from another cell l'' in the opposite direction to the one in which the agent departs from cell l' (toward cell l''), Line 5 (the "Opposite Directions" case of Function *Offset* in Section 7.3.3), when node n' is being expanded, guarantees that the dynamic obstacle arrives at cell l'' at a time later than when the agent departs from cell l', ensuring that the agent does not collide with the dynamic obstacle until the agent departs from cell l'. Otherwise, any collision thus remains when the agent departs from cell l', which contradicts (4).

(iv.b) If node n' is the goal node n_{goal}, no dynamic obstacle arrives at cell l' at a time later than when the agent arrives at cell l' due to Line 7 of Algorithm 7.2 (after n' is popped from OPEN). No collision is possible until the agent has arrived at cell l'.

Therefore, the agent arrives at cell $l' = n'.\mathit{cfg}.\mathit{cell}$ with configuration $n'.\mathit{cfg}$ without any collisions. □

Since all heuristics used by SIPPwRT are admissible as argued above, using the argument in (Phillips & Likhachev, 2011) together with Theorem 7.1, it is straightforward to show that SIPPwRT returns a time-minimal path to a given set V_{Goal} of one or more endpoints that does

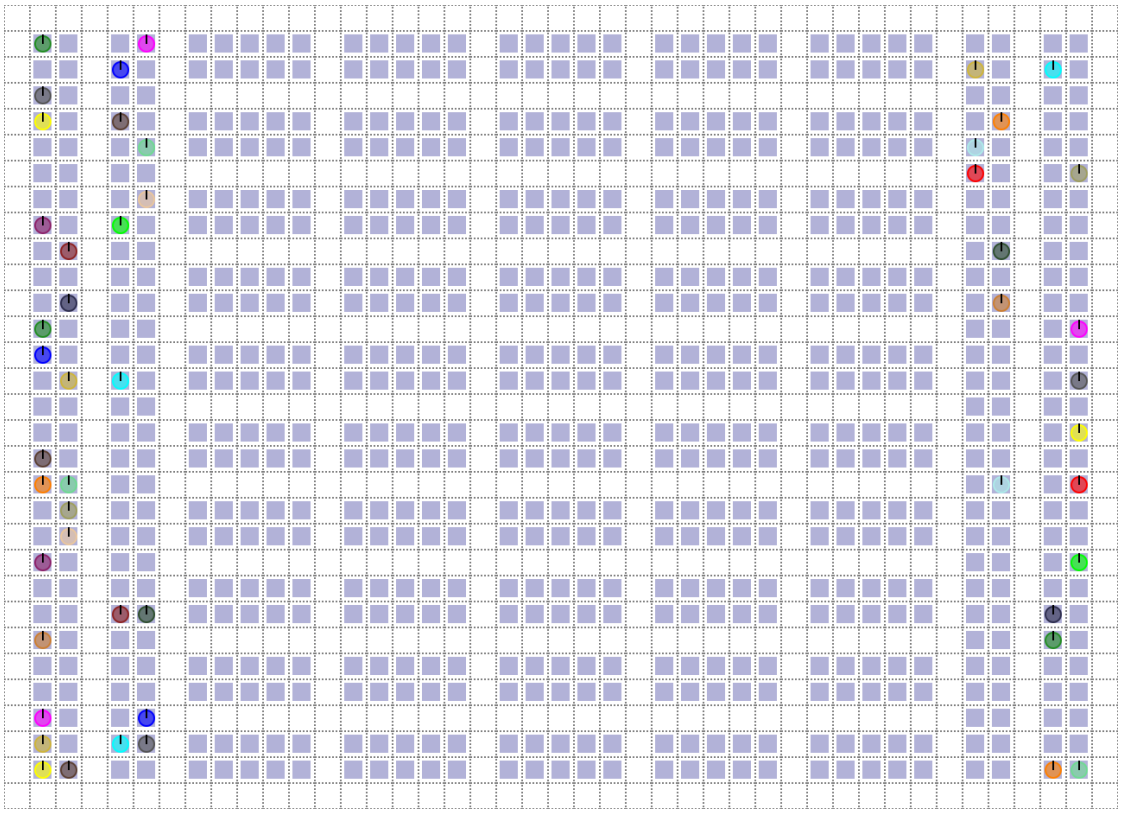


Figure 7.7: The small simulated warehouse environment with 50 agents.

not collide with the paths of other agents in the token and is complete for the single-agent path-planning problems for function calls *Path1* and *Path2*. We can thus rely on the proof of Theorem 6.1 in (Ma, Li, et al., 2017) to show the following theorem.

Theorem 7.2. TP-SIPPwRT is long-term robust for all well-formed MAPD problem instances.

7.5 Simulated Automated Warehouses

We demonstrate the benefits of TP-SIPPwRT for automated warehouses using both an agent simulator with perfect path execution and a standard robot simulator with imperfect path execution resulting from unmodeled high-order dynamic constraints and motion noise by the MAPD algorithms. Figure 7.7 shows an example on the agent simulator with 50 agents and cells of size $1\,\mathrm{m} \times 1\,\mathrm{m}$. Blue gray cells in columns of gray cells are potential initial cells for the agents.

Colored disks are the actual initial cells, which are drawn randomly from all potential initial cells and are the non-task endpoints. All agents face north in their initial cells. Gray cells other than the initial cells are task endpoints (that would house inventory pods in a warehouse even though we do not model inventory pods here). The pickup and delivery cells of all tasks are drawn randomly from all task endpoints. White cells are non-endpoints. This simulator is based on the previous MAPD benchmarks introduced in Section 6.6. But it uses a warehouse environment that is closer to the layout design of a modern real-world Amazon Robotics automated warehouse system (Wurman et al., 2008) and simulates some realistic constraints that are not modeled in the previous MAPD benchmarks. For example, there are no blocked cells between every two rows of task endpoints, robots carrying inventory pods move with a low translational velocity and cannot move through cells that house inventory pods, and robots without inventory pods move with a high translational velocity and are allowed to move through cells that house inventory pods.

We now provide specifications of agent movements in the agent simulator. The agents model circular warehouse robots. All agents use the same rotational velocity $\boldsymbol{v}_{rot}$. The following rules impose restrictions on their legal movements and translational velocities: All free agents can move with a high translational (free) velocity $\boldsymbol{v}_{trans} = \boldsymbol{v}_{free}$ through all cells. All occupied agents can move with a low translational (task) velocity $\boldsymbol{v}_{trans} = \boldsymbol{v}_{task}$ through only the pickup and delivery endpoints of their tasks and all other non-endpoints.

7.6 Experiments

In this section, we describe the results of four experiments on a 2.50 GHz Intel Core i5-2450M laptop with 6 GB RAM. First, we compare TP-SIPPwRT, that directly computes plan-execution schedules during planning, with MAPF-POST to transform the discrete MAPD solutions produced by TP (see Section 6.4) and CENTRAL (see Section 6.5) into plan-execution schedules. Second, we study how TP-SIPPwRT scales with the number of number of agents, task frequency,

Table 7.1: Results for TP-SIPPwRT, CENTRAL, and TP-A* in the small simulated warehouse environment.

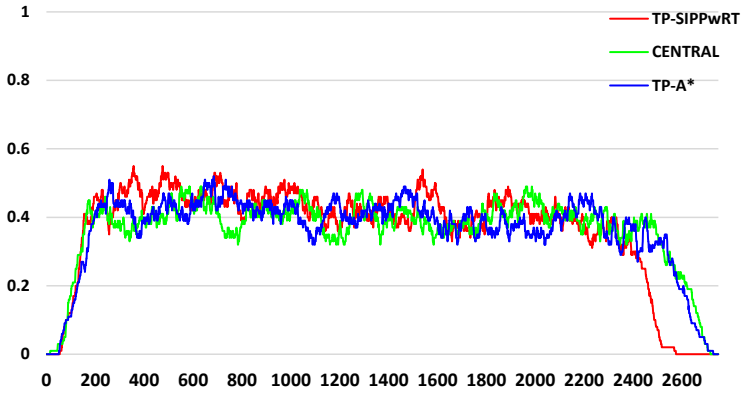
algorithm	v_{task}	discrete service time	discrete makespan	service time	makespan	planning time (s)	post-processing time (s)	throughput
TP-SIPPwRT	0.50	–	–	944.03	2,475.58	0.90	–	0.397
	0.75			601.69	1,755.22	0.92		0.552
	1.00			435.26	1,392.00	0.83		0.689
CENTRAL	0.50	325.28	1,163	1,049.51	2,617.00	1,161.44	264.66	0.370
	0.75			691.90	1,895.68		254.36	0.504
	1.00			520.36	1,553.00		269.91	0.609
TP-A*	0.50	329.83	1,204	1,026.23	2,628.22	1.00	267.38	0.373
	0.75			675.65	1,909.45		295.54	0.508
	1.00			505.81	1,570.77		278.74	0.609

and velocity of occupied agents. Third, we study how TP-SIPPwRT performs in a large simulated automated warehouse system with hundreds of agents. Fourth, we evaluate TP-SIPPwRT using a robot simulator.

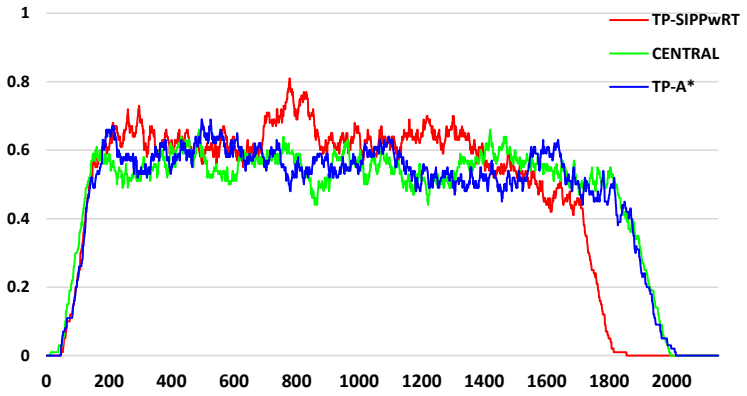
7.6.1 Experiment 1: MAPD Algorithms and Task Velocity

We compared TP-SIPPwRT for $v_{task} = 0.50$, 0.75, and $1.00\,\mathrm{m/s}$ on the agent simulator in the small simulated warehouse environment of Figure 7.7 to two (discrete) MAPD algorithms that both assume discrete agent movements with uniform velocity to the four neighboring cells, namely the original TP (labeled as TP-A*) and CENTRAL (see Chapter 6).

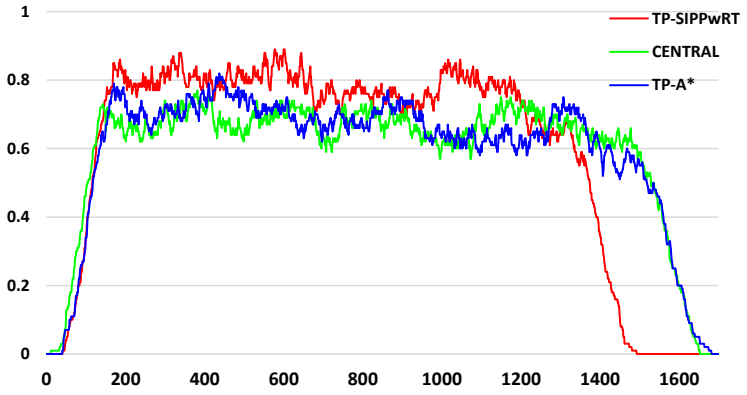
An alternative method to include kinematic constraints for the simulated warehouse environment is to use MAPF-POST (Hönig, Kumar, Cohen, et al., 2016; Hönig, 2019; Hönig, Kumar, Ma, et al., 2016) (see also Section 3.3) in a post-processing step by treating MAPD solutions with discrete agent movements as solutions for one-shot coordination problems such as MAPF and TAPF. To do so, MAPF-POST first converts the discrete MAPD solutions produced by TP and CENTRAL from containing movements in the four compass directions to containing forward movements and point turns and then adapts the discrete MAPD solutions in polynomial



(a) $\boldsymbol{v}_{task} = 0.50\,\mathrm{m/s}$.



(b) $\boldsymbol{v}_{task} = 0.75\,\mathrm{m/s}$.



(c) $\boldsymbol{v}_{task} = 1.00\,\mathrm{m/s}$.

Figure 7.8: Number of tasks executed by 30 agents per second in a moving 100-second window $(t-100, t]$ for TP-SIPPwRT, CENTRAL, and TP-A* as a function of time t for different task velocities.

time to continuous agent movements with given velocities. We point out the differences in the assumptions made by TP-SIPPwRT and MAPF-POST:

- MAPF-POST allows agents to change their translational velocities at an auxiliary location between the centers of two cells but TP-SIPPwRT allows agents to do so only at the center of a cell.

- MAPF-POST allows a free (respectively, occupied) agent to move with a translational velocity lower than $\boldsymbol{v}_{free}$ (respectively, $\boldsymbol{v}_{task}$) and rotate with a rotational velocity lower than $\boldsymbol{v}_{rot}$, but TP-SIPPwRT allows it to move only with translational velocity $\boldsymbol{v}_{free}$ (respectively, $\boldsymbol{v}_{task}$) and rotate with rotational velocity $\boldsymbol{v}_{rot}$.

- MAPF-POST guarantees safety distances of only at most $L/\sqrt{2} = 0.71\,\mathrm{m}$ between agents for a 2D 4-neighbor grid with cells of size $1\,\mathrm{m} \times 1\,\mathrm{m}$ each and, in this case, requires that all agents have the same radius, but TP-SIPPwRT can guarantee safety distances of at most $1\,\mathrm{m}$. We thus used the same radius of $R = 0.5L/\sqrt{2} = 0.35\,\mathrm{m}$ for all agents in this experiment.

For Experiment 1, we used a runtime limit of 5 minutes per instance. We used 30 agents since CENTRAL, the most runtime-intensive of our MAPD algorithms, can handle only slightly more than 30 agents without any timeouts in this environment. We used $\boldsymbol{v}_{free} = 1.00\,\mathrm{m/s}$ and $\boldsymbol{v}_{rot} = \boldsymbol{\pi}/2 = 1.57\,\mathrm{rad/s}$. We generated one sequence of 1,000 tasks by randomly choosing their pickup and delivery vertices from all task endpoints. We used a task frequency of 2 tasks per second (number of tasks that are added (in order) from the sequence to the system in the beginning of every second).

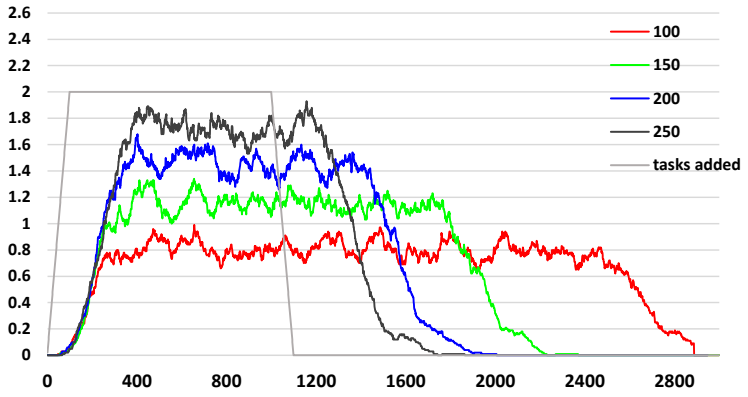
Figure 7.8 visualizes the throughput at time t (number of tasks executed per second in the 100-second window $(t - 100, t]$), measured in tasks per second, as a function of t, measured in seconds. It does not visualize the number of tasks added per second since the number is much larger than the number of tasks executed per second for most t. The throughput at time t of

TP-SIPPwRT decreases earlier than the ones of TP-A* and CENTRAL because fewer still unexecuted tasks are available toward the end for TP-SIPPwRT than for them. Thus, TP-SIPPwRT is more effective than them.

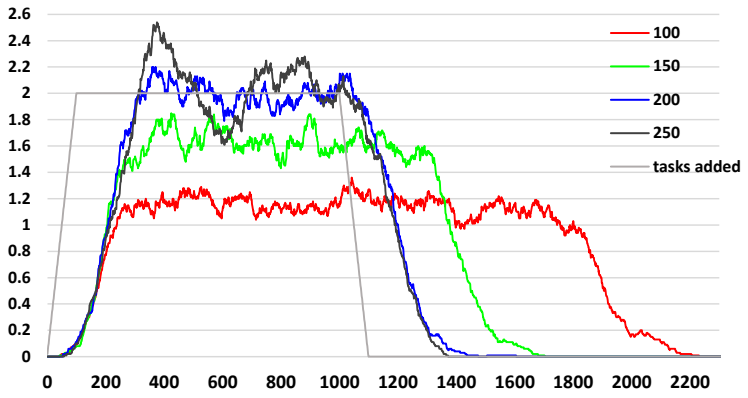
Table 7.1 reports the discrete service time (according to the discrete MAPD solution), discrete makespan (according to the discrete MAPD solution), service time, makespan, planning time (runtime of the MAPD algorithm), and post-processing time (runtime of MAPF-POST), as well as the average throughput over all times t whose throughputs are positive, measured in number of tasks per second. (Inapplicable entries are dashed.) The service time, makespan, and throughput measure effectiveness, while the planning and post-processing times measure efficiency. The planning time of TP-SIPPwRT is less than one second for 30 agents and 1,000 tasks. It is on par with the one of TP-A* and smaller than the one of CENTRAL. Furthermore, TP-SIPPwRT does not have any post-processing time while both TP-A* and CENTRAL have post-processing times of more than 250 seconds. Thus, TP-SIPPwRT is more efficient than them. The service time and makespan of TP-SIPPwRT are smaller than the ones of TP-A* and CENTRAL, while its throughput is larger. Therefore, TP-SIPPwRT is more effective than them.

7.6.2 Experiment 2: Number of Agents, Task Frequency, and Task Velocity

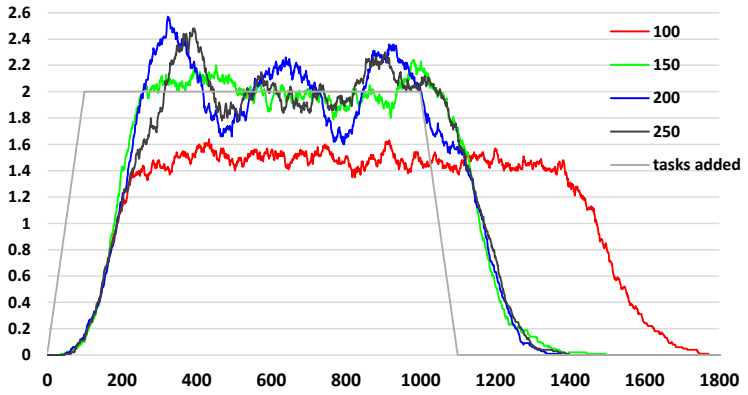
We ran TP-SIPPwRT with the same setup as in Experiment 1 (including the same sequence of 1,000 tasks) for $\boldsymbol{v}_{task} = 0.50$, 0.75, and $1.00\,\mathrm{m/s}$, 10, 20, 30, 40, and 50 agents, and task frequencies of 1, 2, 5, and 10 tasks per second. Table 7.2 shows that the planning time of TP-SIPPwRT is less than one second for up to 50 agents and 1,000 tasks. As expected, the service time decreases as the task frequency decreases; the service time and makespan decrease and the throughput increases as the number of agents increases; and the service time and makespan decrease and the throughput increases as the task velocity increases.



(a) $v_{task} = 0.50\,\mathrm{m/s}$.



(b) $v_{task} = 0.75\,\mathrm{m/s}$.



(c) $v_{task} = 1.00\,\mathrm{m/s}$.

Figure 7.9: Number of tasks added (gray) and executed per second in a moving 100-second window $(t - 100, t]$ for TP-SIPPwRT as a function of time t for different numbers of agents.

Table 7.2: Results for TP-SIPPwRT in the small simulated warehouse environment.

v_{task}		0.50				0.75				1.00			
agents	task frequency	service time	makespan	planning time (s)	throughput	service time	makespan	planning time (s)	throughput	service time	makespan	planning time (s)	throughput
10	1	2,809.72	6,771.00	0.84	0.146	1,834.97	4,764.28	0.86	0.213	1,357.21	3,818.00	0.72	0.270
	2	3,029.59	6,759.41	0.85	0.157	2,077.68	4,768.89	0.84	0.215	1,584.62	3,784.00	0.73	0.274
	5	3,181.97	6,789.41	0.86	0.155	2,185.29	4,748.33	0.86	0.225	1,710.76	3763.71	0.75	0.274
	10	3,215.43	6,775.00	0.84	0.159	2,252.70	4,762.45	0.88	0.219	1,750.19	3,749.00	0.75	0.280
20	1	1,228.35	3,557.58	0.90	0.295	745.48	2,540.33	0.89	0.411	502.50	2,000.71	0.76	0.511
	2	1,450.40	3,503.00	0.89	0.298	966.27	2,493.67	0.91	0.392	714.20	1,980.00	0.79	0.489
	5	1,591.79	3,519.83	0.89	0.292	1,088.86	2,481.85	0.88	0.416	844.32	1,966.00	0.81	0.507
	10	1,661.62	3,502.83	0.88	0.290	1,136.08	2,479.45	0.90	0.417	892.22	1,964.00	0.81	0.507
30	1	723.03	2,482.41	0.94	0.396	389.15	1,763.50	0.91	0.551	222.21	1,431.71	0.82	0.672
	2	944.03	2,475.58	0.90	0.397	601.69	1,755.22	0.92	0.552	435.26	1,392.00	0.83	0.689
	5	1,079.62	2,435.83	0.90	0.398	728.33	1,724.18	0.92	0.555	563.14	1,372.71	0.83	0.688
	10	1,126.47	2,468.00	0.93	0.393	779.67	1,737.00	0.92	0.550	612.06	1,380.00	0.84	0.065
40	1	484.93	2,023.58	0.90	0.484	225.18	1,471.12	0.95	0.657	101.16	1,252.00	0.85	0.765
	2	701.23	1,945.00	0.94	0.503	432.11	1,430.33	0.95	0.674	298.04	1,122.71	0.89	0.847
	5	830.73	2,054.00	0.89	0.470	563.25	1,368.67	0.94	0.693	427.23	1,073.00	0.87	0.870
	10	880.46	1,905.00	0.88	0.506	605.10	1,382.67	0.94	0.686	469.92	1,095.71	0.89	0.853
50	1	331.66	1,680.41	0.98	0.641	122.98	1,262.00	0.98	0.771	63.45	1,140.41	0.96	0.845
	2	557.10	1,676.58	0.97	0.581	335.58	1,192.51	0.96	0.804	219.99	968.00	0.92	0.976
	5	683.56	1,674.41	0.97	0.573	454.42	1,153.51	0.93	0.814	344.44	931.00	0.91	0.992
	10	729.07	1,644.41	0.97	0.582	502.03	1,200.94	0.98	0.784	389.78	926.00	0.93	0.996

Table 7.3: Results for TP-SIPPwRT in the large simulated warehouse environment.

v_{task}	0.50				0.75				1.00			
agents	service time	makespan	planning time (s)	throughput	service time	makespan	planning time (s)	throughput	service time	makespan	planning time (s)	throughput
100	877.94	2,891.58	5.72	0.703	489.46	2,130.67	5.83	0.914	289.80	1,671.00	5.15	1.157
150	525.07	2,269.58	6.49	0.878	253.49	1,602.00	6.67	1.204	122.69	1,396.71	5.57	1.369
200	353.46	1,905.58	7.08	1.026	154.76	1,504.63	7.35	1.314	117.21	1,276.12	9.50	1.504
250	267.07	1,762.24	9.35	1.127	147.90	1,271.67	12.83	1.521	132.45	1,297.00	15.76	1.479

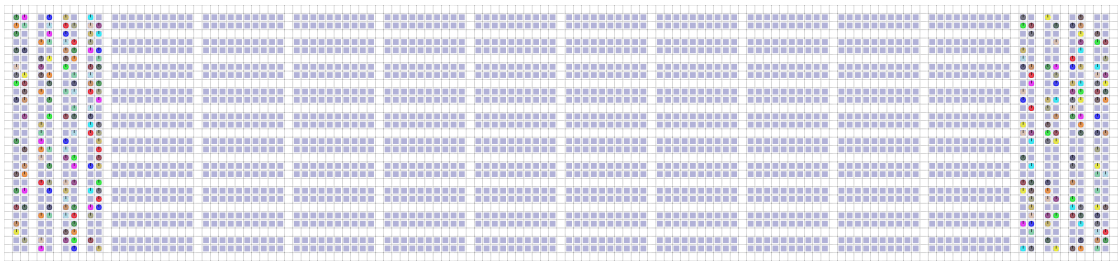


Figure 7.10: The large simulated warehouse environment with 250 agents.

7.6.3 Experiment 3: Scalability, Number of Agents, and Task Velocity

To evaluate how TP-SIPPwRT scales in the number of agents, we ran TP-SIPPwRT with the same setup as in Experiment 1 but in the large simulated warehouse environment of Figure 7.10 for 100, 150, 200, and 250 agents and $\boldsymbol{v}_{task} = 0.50$, 0.75, and $1.00\,\mathrm{m/s}$. We used one sequence of 2,000 tasks and a task frequency of 2 tasks per second. Figure 7.9 visualizes the throughput at time t. Table 7.3 shows that the planning time of TP-SIPPwRT is less than 16 seconds for up to 250 agents and 2,000 tasks, justifying our claim that it can compute paths for hundreds of agents and thousands of tasks in seconds. Similarly to before, the service time and makespan decrease and the throughput and planning time increase as the number of agents increases; and the service time and makespan decrease and the throughput increases as the task velocity increases. There is an exception due to the congestion resulting from many agents for 250 agents and $\boldsymbol{v}_{task} = 1.00\,\mathrm{m/s}$.

7.6.4 Experiment 4: Robot Simulator

We created a custom model of the kinodynamic constraints of a differential-drive Create2 robot from iRobot for the robot simulator V-REP (Rohmer, Singh, & Freese, 2013). Create2 robots have a cylindrical shape with radius $0.175\,\mathrm{m}$ and can reach a translational velocity of $0.5\,\mathrm{m/s}$ and a rotational velocity of $4.2\,\mathrm{rad/s}$. We used $\boldsymbol{v}_{free} = 0.40\,\mathrm{m/s}$, $\boldsymbol{v}_{task} = 0.20\,\mathrm{m/s}$, $\boldsymbol{v}_{rot} = \boldsymbol{\pi} = 3.14\,\mathrm{rad/s}$, and $R = 0.40\,\mathrm{m}$ as conservative values to allow the robots to follow their paths safely despite unmodeled high-order dynamic constraints and motion noise by TP-SIPPwRT. We used an off-the-shelf PID controller (Hönig, 2019), which is also used by recent research to

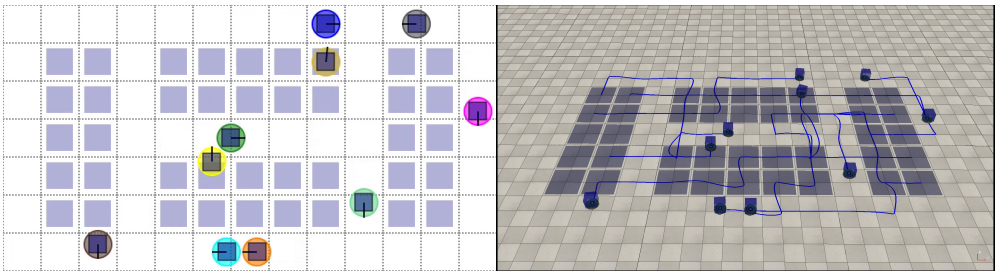


Figure 7.11: Screenshots for Experiment 4 at $t = 35\,\mathrm{s}$. Left: Agent simulator. Right: Robot simulator.

verify MAPF-POST for MAPF (Hönig, Kumar, Cohen, et al., 2016) and TAPF (Hönig, Kumar, Ma, et al., 2016), that uses $[x, y, \theta]^T$ (given by V-REP) as the current state and the desired next cell with the associated desired arrival time (given by TP-SIPPwRT) as the goal state. The PID controller corrects for heading errors by orienting the robot to face the desired next cell while simultaneously adjusting the translational velocity to let the robot arrive at the desired next cell at the desired arrival time. We limit our experiment to the small warehouse environment of Figure 7.11 for 10 robots due to the slow runtime of V-REP. We used one sequence of 20 tasks and a task frequency of 2 tasks per second. The planning time of TP-SIPPwRT is $2\,\mathrm{ms}$. All robots follow their paths safely, resulting in a service time of $90.57\,\mathrm{s}$ and a makespan of $171.16\,\mathrm{s}$.

7.7 Summary

In this chapter, we studied MAPD with kinematic constraints of real-world agents in a simulated system. We demonstrated how to adapt the MAPD algorithm TP to these constraints by using the novel one-shot single-agent path-planning algorithm SIPPwRT that computes paths with continuous agent movements with given velocities. The resulting algorithm TP-SIPPwRT takes kinematic constraints of real-world agents into account directly during planning to compute plan-execution schedules for them and provides guaranteed safety distances between them. Theoretically, we showed that TP-SIPPwRT remains to be long-term robust for well-formed MAPD problem instances. Experimentally, we demonstrated that TP-SIPPwRT can compute a plan-execution schedule for hundreds of agents and thousands of tasks in seconds. We also

described, as a baseline method, how MAPF-POST can be used in a post-processing step to include kinematic constraints of real-world agents to transform MAPD solutions with discrete agent movements into plan-execution schedules. In our experiments, we showed that using MAPF-POST to post-process discrete MAPD solutions is less efficient and effective than letting TP-SIPPwRT compute plan-execution schedules directly. We showcased the benefits of both methods for real-world applications of multi-agent systems in a simulated automated warehouse system using both an agent simulator and a standard robot simulator.

To summarize, in this chapter, we validated the hypothesis that MAPD algorithms can potentially be applied to and thus benefit the long-term coordination of autonomous target-assignment and path-planning operations for real-world applications of multi-agent systems.

Chapter 8

Conclusions

In this chapter, we summarize the contributions of this dissertation and their direct impact on multi-agent target-assignment and path-planning research. We then present potential future research directions.

8.1 Contributions

In many real-world applications of multi-agent systems, teams of autonomous agents must assign targets among themselves (target-assignment operations) and plan collision-free paths to their targets (path-planning operations) in order to finish tasks cooperatively. The coordination of target-assignment and path-planning operations is a fundamental building block for these multi-agent systems. It requires solving either the one-shot or the long-term combined target-assignment and path-planning problem. Recent research in the AI community has focused on tackling only the one-shot path-planning problem MAPF. However, none of the existing results directly contributes to a theoretical understanding of or results in algorithms for either the one-shot or the long-term combined target-assignment and path-planning problem.

While MAPF techniques can potentially be applied to the one-shot and the long-term combined target-assignment and path-planning problems, three central questions remain: Question 1: How hard is it to jointly assign targets to and plan paths for teams of agents? Question 2:

How and how well can one jointly assign targets to and plan paths for teams of agents? Question 3: How do teams of agents execute the computed solutions?

In this dissertation, we addressed the above central questions by evaluating the following hypothesis:

> *Formalizing and studying new variants of MAPF can result in new theoretical insights into or new algorithms for the one-shot and the long-term combined target-assignment and path-planning problems for teams of agents, which can benefit real-world applications of multi-agent systems.*

We presented the following contributions to validate this hypothesis:

- **Contribution 1:** In Chapter 4, we introduced a unified NP-hardness proof structure that can be used to derive complexity results for many target-assignment and path-planning problems. This unified NP-hardness proof structure stems from formalizing and studying PERR, a variant of MAPF, and its generalization K-PERR. We demonstrated how to derive NP-hardness and fixed-parameter inapproximability results for both PERR and K-PERR using this unified NP-hardness proof structure. This unified NP-hardness proof structure lays the theoretical foundation for studying the one-shot and the long-term combined target-assignment and path-planning problems, namely TAPF and MAPD that are addressed in this dissertation, and many other target-assignment and path-planning problems. As a consequence of the theoretical understanding of the relationships between various target-assignment and path-planning problem formulations (see Chapter 2), we demonstrated how the theoretical results for PERR and K-PERR can be carried over to other problems. For example, we improved the state-of-the-art complexity results for makespan minimization for MAPF. We also used the unified NP-hardness proof structure to prove that TAPF is NP-hard to approximate within any constant factor less than 4/3, and that MAPD is NP-hard to solve optimally for service time minimization and NP-hard to approximate within any constant factor less than 4/3 for makespan minimization.

- **Contribution 2:** In Chapter 5, we formalized and studied TAPF, a variant of MAPF that models the one-shot combined target-assignment and path-planning problem for multiple teams of agents. We presented CBM, a hierarchical algorithm that exploits the combinatorial structure of TAPF by breaking it down to the polynomial-time solvable sub-problems of coordinating the agents in every team on the low-level, which are solved by a min-cost max-flow algorithm, and the NP-hard sub-problem of coordinating multiple teams on the high level, which is solved by a best-first tree-search algorithm. We proved that CBM is complete and optimal for TAPF. We also presented an ILP-based TAPF algorithm using a multi-commodity flow-based ILP encoding. Experimental results showed that CBM runs faster than the ILP-based TAPF algorithm and can compute solutions for hundreds of agents in minutes of runtime. These TAPF algorithms can use an existing polynomial-time procedure in a post-processing step to transform their solutions into plan-execution schedules that take kinematic constraints of real-world agents into account.

- **Contribution 3:** In Chapter 6, we formalized and studied MAPD, a variant of MAPF that models the long-term combined target-assignment and path-planning problem. We demonstrated how MAPD algorithms can utilize environmental characteristics of well-formed MAPD problem instances to guarantee long-term robustness. We presented two decoupled MAPD algorithms, TP and TPTS, and one centralized MAPD algorithm, CENTRAL. They reduce, in different ways, the long-term problem MAPD to a sequence of one-shot target-assignment and path-planning sub-problems, which can then be solved by different one-shot target-assignment and path-planning algorithms. Theoretically, we proved that all these MAPD algorithms are long-term robust for well-formed problem instances. Experimentally, we compared them in a simulated warehouse environment and provided guidelines for identifying when each one should be used. In particular, TP remains efficient for MAPD problem instances with hundreds of agents and tasks and thus has the potential to be deployed in large-scale multi-agent systems.

- **Contribution 4:** In Chapter 7, we studied MAPD with kinematic constraints of real-world agents. We demonstrated how MAPD algorithms can take kinematic constraints of real-world agents into account to produce plan-execution schedules for the agents. We presented TP-SIPPwRT, an improvement to TP that uses the novel one-shot single-agent path-planning algorithm SIPPwRT to compute paths with continuous agent movements with given velocities. As a result, TP-SIPPwRT takes kinematic constraints of real-world agents into account directly during planning to compute plan-execution schedules for them and provides guaranteed safety distances between them. We experimentally evaluated TP-SIPPwRT in a simulated automated warehouse system using an agent and a robot simulators and demonstrated that TP-SIPPwRT can compute solutions for hundreds of agents and thousands of tasks in seconds and is more efficient and effective than the baseline method that uses an existing polynomial-time procedure in a post-processing step to transform discrete MAPD solutions into plan-execution schedules.

Therefore, Contribution 1 validates that formalizing and studying new variants of MAPF can result in new theoretical insights into the one-shot and the long-term combined target-assignment and path-planning problems for teams of agents. Contribution 2 validates that formalizing and studying new variants of MAPF can result in algorithms for the one-shot combined target-assignment and path-planning problems for teams of agents that have the potential to be applied to and can thus benefit real-world applications of multi-agent systems. Contribution 3 validates that formalizing and studying new variants of MAPF can result in algorithms for the long-term combined target-assignment and path-planning problems for teams of agents. Contribution 4 validates that the algorithms we presented for the long-term combined target-assignment and path-planning problems for teams of agents have the potential to be applied to and can thus benefit real-world applications of multi-agent systems.

8.2 Direct Impact

Prior to this dissertation, the AI community has applied MAPF techniques only to the one-shot path-planning problem. Our recent research on MAPF-POST (Hönig, Kumar, Cohen, et al., 2016) demonstrated the potential benefit of MAPF techniques for real-world applications of multi-agent systems. However, none of the existing results on MAPF directly solve the one-shot or the long-term combined target-assignment and path-planning problem, which models the coordination of both the target-assignment and path-planning operations of the agents as required by many real-world applications of multi-agent systems. This dissertation moves MAPF techniques one-step forward toward establishing theoretical foundations of both the one-shot and the long-term coordination of target-assignment and path planning operations of teams of agents and developing algorithmic solutions for addressing these coordination problems. We now demonstrate how the contributions presented in this dissertation have already had a direct impact on state-of-the-art multi-agent target-assignment and path-planning research. In the following, we list examples of a subset of such research that has been published in top-tier AI and robotics conferences and high-impact robotics journals (AAAI, IJCAI, AAMAS, ICAPS, ICRA, IROS, and RA-L) between 2017 and 2019.

- Research that presents new algorithms for TAPF or MAPD:
  - "Conflict-Based Search with Optimal Task Assignment" (Hönig, Kiesel, Tinka, Durham, & Ayanian, 2018). This work uses the TAPF problem formulation and presents a new algorithm that is optimal for the flowtime objective, It is part of Hoenig's Ph.D. dissertation (Hönig, 2019).
  - "A Multi-Label A* Algorithm for Multi-Agent Pathfinding" (Grenouilleau et al., 2019). This work uses the MAPD problem formulation and improves upon TP by finding a path that moves an agent first to the pickup vertex and then to the delivery vertex of its assigned task in one A* search.

  - "Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding" (Okumura et al., 2019). This work uses the MAPD problem formulation and presents a new prioritized planning algorithm that can solve both MAPF and MAPD.

- Research that studies new variants of TAPF or MAPD:

  - "Generalized Target Assignment and Path Finding Using Answer Set Programming" (Nguyen et al., 2017). This work studies a generalized version of TAPF where each agent may need to visit more than one target. The problem can also be considered as an offline (and thus one-shot) version of MAPD. It can be used for long-term coordination if lookahead computation is allowed.

  - "Target Assignment and Path Planning for Multi-Agent Pickup and Delivery" (Liu, Ma, et al., 2019). Similar to the above work, this work studies an offline version of MAPD where all tasks are known. It presents a new way of utilizing environmental characteristics of well-formed MAPD problem instances.

  - "An Optimal Algorithm to Solve the Combined Task Allocation and Path Finding Problem" (Henkel et al., 2019). This work also studies an offline version of MAPD and presents an optimal search-based algorithm that scales well to four agents and four tasks.

  - "Online Multi-Agent Pathfinding" (Švancara, Vlk, Stern, Atzmon, & Barták, 2019). This work studies the long-term path-planning problem where agents appear at unknown times with preassigned targets.

- Research that studies MAPD with realistic simulations:

  - "A Hierarchical Framework for Coordinating Large-Scale Robot Networks" (Liu, Zhou, et al., 2019). This work studies MAPD in a simulated warehouse system where the solutions need to be executed in a distributed setting and communication between warehouse robots needs to be taken into account.

– "Persistent and Robust Execution of MAPF Schedules in Warehouses" (Hönig et al., 2019). This work studies MAPD on a real-time automated warehouse simulator and develops execution monitoring schemes that handle motion uncertainty. It is also part of Hoenig's Ph.D. dissertation (Hönig, 2019).

8.3 Limitations and Future Directions

Some promising future directions of this dissertation have already been pointed out in the publications mentioned in the above section. In the following, we highlight important directions that result from the limitations of this dissertation and have not yet been addressed by the state of the art.

- This dissertation does not study the computational complexity of versions of Anonymous MAPF and 1-PERR that minimize the flowtime. Instead, their complexities have been posted as open questions in Chapter 4. Future work should thus develop a better theoretical and algorithmic understanding of Anonymous MAPF and 1-PERR for flowtime minimization.

- Few theoretical insights and results are known for MAPD besides the complexity results presented in Chapter 4 and the well-formed properties presented in Chapter 6. In particular, the MAPD algorithms presented in Chapters 6 and 7 do not provide any effectiveness guarantees. Future work should thus develop a better theoretical and algorithmic understanding of MAPD. For example, MAPD can be studied from the point of view of competitive analysis, which might result in algorithms that provide effectiveness guarantees or allow one to conclude that no competitive algorithms exist.

- The MAPD algorithms presented in Chapters 6 and 7 do not consider dependencies between tasks, task constraints, or different priorities of tasks, which makes them unrealistic for some real-world applications. For example, in an automated warehouse system, an inventory pod cannot be simultaneously delivered to multiple inventory stations that need

the pod (that is, some tasks cannot be executed in parallel), some tasks might have deadline constraints (that is, some tasks must be finished before a deadline), and some tasks might have a higher priority than the others (for example, some tasks have a hard deadline but the others do not). Future work should thus investigate how such task dependencies, constraints, and priorities can be incorporated into MAPD.

- In this dissertation, we have assumed that targets/tasks are either completely known (for the one-shot problems MAPF, PERR, and TAPF) or completely unknown (for the long-term problem MAPD). However, in many real-world applications, some information about unknown targets/tasks, for example, the frequency of each inventory pod is needed by an inventory station in an automated warehouse system, can be obtained from historical data using data-driven techniques. Future work should thus investigate how real-world data can be used to predict when and where a future task will appear, which might improve the effectiveness of existing target-assignment and path-planning algorithms.

- Many state-of-the-art algorithmic results for MAPF can be applied to TAPF. For example, in this dissertation, we have focused on formalizing and presenting the first algorithmic results for TAPF. However, we have not focused on improving the efficiency of the TAPF algorithms presented in Chapter 5. Recent research (Felner et al., 2018; Li, Boyarski, et al., 2019; Li, Gange, et al., 2020; Li, Harabor, Stuckey, Felner, et al., 2019; Li, Harabor, Stuckey, Ma, & Koenig, 2019) has presented improved versions of CBS that speed up the high-level search of CBS or bounded-suboptimal versions of CBS (Barer et al., 2014; Cohen et al., 2016) that sacrifice optimality for efficiency. Future work should thus investigate whether and, if so, how these results can make CBM more efficient.

- Many state-of-the-art algorithmic results on MAPF can be applied to MAPD. For example, in Chapter 7, we have extended TP to TP-SIPPwRT that takes kinematic constraints into account directly during planning. However, we have not investigated whether and, if so, how CENTRAL can also take kinematic constraints into account directly during planning. Recent research (Andreychuk, Yakovlev, Atzmon, & Stern, 2019) has presented a version

of CBS that computes MAPF solutions with continuous time on 2^k-neighbor grids (where edges have non-uniform costs). Future work should thus investigate whether and, if so, how this algorithm can be used to let CENTRAL take kinematic constraints into account to compute plan-execution schedules directly.

- In this dissertation, we have only considered some of the kinodynamic constraints (which we refer to as kinematic constraints) of real-world agents (see Section 1.1), including their kinematic and first-order dynamic constraints. We have ignored high-order dynamic constraints, such as finite acceleration and deceleration limits, and have assumed that these constraints are handled by the controllers of real-world agents. While these assumptions have been verified to work on many real-world agents, such as differential-drive robots, they can make it unsafe for real-world agents with complex dynamics, for example, quadcopters, to follow the plan-execution schedules produced by MAPF-POST or TP-SIPPwRT. Future work should thus investigate whether and, if so, how our target-assignment and path-planning algorithms can take high-order dynamic constraints of real-world agents into account.

Bibliography

Agmon, N., Urieli, D., & Stone, P. (2011). Multiagent patrol generalized to complex environmental conditions. In *AAAI Conference on Artificial Intelligence* (pp. 1090–1095). (Cit. on p. 1).

Ahmadi, M., & Stone, P. (2006). A multi-robot system for continuous area sweeping tasks. In *IEEE International Conference on Robotics and Automation* (pp. 1724–1729). (Cit. on pp. 1, 121).

Andreychuk, A., Yakovlev, K., Atzmon, D., & Stern, R. (2019). Multi-agent pathfinding with continuous time. In *International Joint Conference on Artificial Intelligence* (pp. 39–45). (Cit. on p. 195).

Aronson, J. E. (1989). A survey of dynamic network flows. *Annals of Operations Research*, *20*(1-4), 1–66. (Cit. on p. 51).

Atzmon, D., Stern, R., Felner, A., Wagner, G., Barták, R., & Zhou, N. (2018). Robust multi-agent path finding. In *International Symposium on Combinatorial Search* (pp. 2–9). (Cit. on p. 44).

Azar, Y., Naor, J., & Rom, R. (1995). The competitiveness of on-line assignments. *Journal of Algorithms*, *18*(2), 221–237. (Cit. on pp. 6, 8, 15, 121, 137).

Balch, T., & Arkin, R. C. (1998). Behavior-based formation control for multirobot teams. *IEEE Transactions on Robotics and Automation*, *14*(6), 926–939. (Cit. on p. 1).

Banfi, J., Basilico, N., & Amigoni, F. (2017). Intractability of time-optimal multirobot path planning on 2D grid graphs with holes. *IEEE Robotics and Automation Letters*, *2*(4), 1941–1947. (Cit. on p. 39).

Barer, M., Sharon, G., Stern, R., & Felner, A. (2014). Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *International Symposium on Combinatorial Search* (pp. 19–27). (Cit. on pp. 51, 195).

Becker, R., Zilberstein, S., Lesser, V., & Goldman, C. V. (2004). Solving transition independent decentralized Markov Decision Processes. *Journal of Artificial Intelligence Research*, *22*(1), 423–455. (Cit. on p. 44).

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*(1), 87–90. (Cit. on p. 56).

Bennewitz, M., Burgard, W., & Thrun, S. (2002). Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, *41*(2-3), 89–99. (Cit. on pp. 41, 44, 45).

Bertsekas, D. P. (1992). Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, *1*(1), 7–66. (Cit. on pp. 7, 11, 15, 57, 61).

Bnaya, Z., & Felner, A. (2014). Conflict-oriented windowed hierarchical cooperative A*. In *IEEE International Conference on Robotics and Automation* (pp. 3743–3748). (Cit. on p. 42).

Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. In *Conference on Theoretical Aspects of Rationality and Knowledge* (pp. 195–210). (Cit. on p. 44).

Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., & Shimony, S. E. (2015). ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *International Joint Conference on Artificial Intelligence* (pp. 740–746). (Cit. on p. 50).

Brucker, P. (2010). *Scheduling algorithms* (5th). Springer. (Cit. on pp. 6, 13, 62).

Burkard, R. E., Dell'Amico, M., & Martello, S. (2009). *Assignment problems*. Springer. (Cit. on p. 57).

Cáp, M., Vokrínek, J., & Kleiner, A. (2015). Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. In *International Conference on Automated Planning and Scheduling* (pp. 324–332). (Cit. on pp. 5, 8, 45, 122, 126).

Cohen, L., Uras, T., & Koenig, S. (2015). Feasibility study: Using highways for bounded-suboptimal multi-agent path finding. In *International Symposium on Combinatorial Search* (pp. 2–8). (Cit. on p. 117).

Cohen, L., Greco, M., Ma, H., Hernandez, C., Felner, A., Kumar, T. K. S., & Koenig, S. (2018). Anytime focal search with applications. In *International Joint Conference on Artificial Intelligence* (pp. 1434–1441). (Cit. on p. 51).

Cohen, L., Uras, T., Kumar, T. K. S., Xu, H., Ayanian, N., & Koenig, S. (2016). Improved solvers for bounded-suboptimal multi-agent path finding. In *International Joint Conference on Artificial Intelligence* (pp. 3067–3074). (Cit. on pp. 51, 115, 195).

Coltin, B., & Veloso, M. (2014). Ridesharing with passenger transfers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3278–3283). (Cit. on p. 62).

de Wilde, B., ter Mors, A. W., & Witteveen, C. (2013). Push and Rotate: Cooperative multi-agent path planning. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 87–94). (Cit. on p. 40).

Derigs, U., & Zimmermann, U. (1978). An augmenting path method for solving linear bottleneck assignment problems. *Computing*, *19*(4), 285–295. (Cit. on p. 57).

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. (Cit. on p. 104).

Dresner, K., & Stone, P. (2008). A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, *31*, 591–656. (Cit. on p. 1).

Erdem, E., Kisa, D. G., Oztok, U., & Schueller, P. (2013). A general formal framework for pathfinding problems with multiple agents. In *AAAI Conference on Artificial Intelligence* (pp. 290–296). (Cit. on p. 40).

Erdmann, M. A., & Lozano-Pérez, T. (1987). On multiple moving objects. *Algorithmica*, *2*, 477–521. (Cit. on pp. 41, 44, 45).

Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T. K. S., & Koenig, S. (2018). Adding heuristics to conflict-based search for multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling* (pp. 83–87). (Cit. on pp. 51, 115, 195).

Felner, A., Stern, R., Shimony, S. E., Boyarski, E., Goldenberg, M., Sharon, G., Sturtevant, N. R., Wagner, G., & Surynek, P. (2017). Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *International Symposium on Combinatorial Search* (pp. 29–37). (Cit. on pp. 7, 14, 15, 20, 39).

Ford Jr, L. R., & Fulkerson, D. R. (2015). *Flows in networks*. Princeton University Press. (Cit. on p. 56).

Ford, L. R., & Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics*, *8*, 399–404. (Cit. on p. 104).

Fulkerson, D. R., Glicksberg, I., & Gross, O. (1953). *A production line assignment problem*. The Rand Corporation, Paper RM-1102. (Cit. on pp. 11, 57, 61).

Garfinkel, R. S. (1971). An improved algorithm for the bottleneck assignment problem. *Operations Research*, *19*(7), 1747–1751. (Cit. on pp. 7, 11, 15, 57, 61).

Gerkey, B. P., & Matarić, M. J. (2002). Sold!: Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, *18*(5), 758–768. (Cit. on pp. 58, 122).

Gerkey, B. P., & Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, *23*(9), 939–954. (Cit. on pp. 5, 33, 58).

Goldberg, A. V., & Tarjan, R. E. (1987). Solving minimum-cost flow problems by successive approximation. In *Annual ACM Symposium on Theory of Computing* (pp. 7–18). (Cit. on pp. 14, 92, 95, 104, 118).

Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N. R., Holte, R. C., & Schaeffer, J. (2014). Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, *50*, 141–187. (Cit. on p. 42).

Goldman, C. V., & Zilberstein, S. (2004). Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research*, *22*, 143–174. (Cit. on p. 44).

Goldreich, O. (2011). Finding the shortest move-sequence in the graph-generalized 15-puzzle is NP-hard. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation* (pp. 1–5). Springer. (Cit. on pp. 7, 11, 38, 61).

Goraly, G., & Hassin, R. (2010). Multi-color pebble motion on graphs. *Algorithmica*, *58*(3), 610–636. (Cit. on p. 38).

Grenouilleau, F., van Hoeve, W., & Hooker, J. N. (2019). A multi-label A* algorithm for multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling* (pp. 181–185). (Cit. on pp. 132, 144, 145, 192).

Gross, O. (1959). *The bottleneck assignment problem*. The Rand Corporation, Paper P-1630. (Cit. on pp. 7, 11, 15, 57, 61).

Henkel, C., Abbenseth, J., & Toussaint, M. (2019). An optimal algorithm to solve the combined task allocation and path finding problem. *IEEE/RSJ International Conference on Intelligent Robots and Systems*. (Cit. on pp. 121, 137, 193).

Hönig, W., Kumar, T. K. S., Cohen, L., Ma, H., Xu, H., Ayanian, N., & Koenig, S. (2016). Multi-agent path finding with kinematic constraints. In *International Conference on Automated Planning and Scheduling* (pp. 477–485). (Cit. on pp. 9, 55, 155, 156, 178, 186, 192).

Hönig, W. (2019). *Motion coordination for large multi-robot teams in obstacle-rich environments* (Doctoral dissertation, University of Southern California). (Cit. on pp. 55, 56, 178, 185, 192, 194).

Hönig, W., Kiesel, S., Tinka, A., Durham, J. W., & Ayanian, N. (2018). Conflict-based search with optimal task assignment. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 757–765). (Cit. on p. 192).

Hönig, W., Kiesel, S., Tinka, A., Durham, J. W., & Ayanian, N. (2019). Persistent and robust execution of MAPF schedules in warehouses. *IEEE Robotics and Automation Letters*, *4*(2), 1125–1131. (Cit. on pp. 144, 156, 194).

Hönig, W., Kumar, T. K. S., Ma, H., Ayanian, N., & Koenig, S. (2016). Formation change for robot groups in occluded environments. In *IEEE/RSJ International Conference on Intelligent Robots and System* (pp. 4836–4842). (Cit. on pp. 1, 55, 91, 117, 155, 156, 178, 186).

Hönig, W., Preiss, J. A., Kumar, T. S., Sukhatme, G. S., & Ayanian, N. (2018). Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, *34*(4), 856–869. (Cit. on pp. 1, 91).

Jennings, J. S., Whelan, G., & Evans, W. F. (1997). Cooperative search and rescue with a team of mobile robots. In *International Conference on Advanced Robotics* (pp. 193–200). (Cit. on p. 1).

Jiang, Y., Yedidsion, H., Zhang, S., Sharon, G., & Stone, P. (2019). Multi-robot planning with conflicts and synergies. *Autonomous Robots*, *43*(18), 2011–2032. (Cit. on p. 91).

Johnson, W. W., & Story, W. E. (1879). Notes on the "15" puzzle. *American Journal of Mathematics*, *2*(4), 397–404. (Cit. on p. 38).

Kalyanasundaram, B., & Pruhs, K. (1993). Online weighted matching. *Journal of Algorithms*, *14*(3), 478–488. (Cit. on pp. 6, 8, 15, 58, 121, 126, 137).

Kavraki, L. E., & LaValle, S. M. (2016). Motion planning. *Springer Handbook of Robotics*, 139–162. (Cit. on p. 6).

Khandelwal, P., Zhang, S., Sinapov, J., Leonetti, M., Thomason, J., Yang, F., Gori, I., Svetlik, M., Khante, P., Lifschitz, V., et al. (2017). Bwibots: A platform for bridging the gap between AI and human–robot interaction research. *International Journal of Robotics Research*, *36*(5-7), 635–659. (Cit. on pp. 1, 121).

Khorshid, M., Holte, R., & Sturtevant, N. R. (2011). A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *International Symposium on Combinatorial Search* (pp. 76–83). (Cit. on p. 40).

Khuller, S., Mitchell, S. G., & Vazirani, V. V. (1994). On-line algorithms for weighted bipartite matching and stable marriages. *Theoretical Computer Science*, *127*(2), 255–267. (Cit. on pp. 8, 15, 58, 122).

Kloder, S., & Hutchinson, S. (2006). Path planning for permutation-invariant multirobot formations. *IEEE Transactions on Robotics*, *22*(4), 650–665. (Cit. on p. 26).

Kornhauser, D., Miller, G., & Spirakis, P. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Annual Symposium on Foundations of Computer Science* (pp. 241–250). (Cit. on pp. 11, 37, 38, 61).

Korsah, G. A., Stentz, A., & Dias, M. B. (2013). A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, *32*(12), 1495–1512. (Cit. on p. 58).

Kou, N. M., Peng, C., Ma, H., Kumar, T. K. S., & Koenig, S. (2020). Idle time optimization for target assignment and path finding in sortation centers. In *AAAI Conference on Artificial Intelligence* (pp. 9925–9932). (Cit. on p. 1).

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, *2*, 83–97. (Cit. on pp. 7, 11, 15, 57, 61, 130, 137, 139).

Kurniawati, H., Hsu, D., & Lee, W. S. (2008). SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems*. (Cit. on p. 44).

Li, J., Boyarski, E., Felner, A., Ma, H., & Koenig, S. (2019). Improved heuristics for multi-agent path finding with conflict-based search. In *International Joint Conference on Artificial Intelligence* (pp. 442–449). (Cit. on pp. 51, 195).

Li, J., Gange, G., Harabor, D., Stuckey, P. J., Ma, H., & Koenig, S. (2020). New techniques for pairwise symmetry breaking in multi-agent path finding. In *International Conference on Automated Planning and Scheduling* (pp. 193–201). (Cit. on pp. 51, 195).

Li, J., Harabor, D., Stuckey, P. J., Felner, A., Ma, H., & Koenig, S. (2019). Disjoint splitting for conflict-based search for multi-agent path finding. In *International Conference on Automated Planning and Scheduling* (pp. 279–283). (Cit. on pp. 51, 195).

Li, J., Harabor, D., Stuckey, P. J., Ma, H., & Koenig, S. (2019). Symmetry-breaking constraints for grid-based multi-agent path finding. In *AAAI Conference on Artificial Intelligence* (pp. 6087–6095). (Cit. on pp. 51, 195).

Li, J., Sun, K., Ma, H., Felner, A., Kumar, T. K. S., & Koenig, S. (2020). Moving agents in formation in congested environments. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 726–734). (Cit. on pp. 1, 91).

Li, J., Surynek, P., Felner, A., Ma, H., Kumar, T. K. S., & Koenig, S. (2019). Multi-agent path finding for large agents. In *AAAI Conference on Artificial Intelligence* (pp. 7627–7634). (Cit. on p. 44).

Liu, L., & Michael, N. (2016). An MDP-based approximation method for goal constrained multi-MAV planning under action uncertainty. In *IEEE/RSJ International Conference on Robotics and Automation* (pp. 56–62). (Cit. on p. 44).

Liu, M., Ma, H., Li, J., & Koenig, S. (2019). Target assignment and path planning for multi-agent pickup and delivery. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 2253–2255). (Cit. on pp. 144, 145, 193).

Liu, Z., Zhou, S., Wang, H., Shen, Y., Li, H., & Liu, Y. (2019). A hierarchical framework for coordinating large-scale robot networks. In *IEEE International Conference on Robotics and Automation* (pp. 6672–6677). (Cit. on p. 193).

Luna, R., & Bekris, K. E. (2011). Push and Swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conference on Artificial Intelligence* (pp. 294–300). (Cit. on p. 40).

Ma, H., Hönig, W., Cohen, L., Uras, T., Xu, H., Kumar, T. K. S., Ayanian, N., & Koenig, S. (2017). Overview: A hierarchical framework for plan generation and execution in multi-robot systems. *IEEE Intelligent Systems*, *32*(6), 6–12. (Cit. on p. 1).

Ma, H., & Koenig, S. (2016). Optimal target assignment and path finding for teams of agents. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 1144–1152). (Cit. on p. 89).

Ma, H., & Koenig, S. (2017). AI buzzwords explained: Multi-agent path finding (MAPF). *AI Matters*, *3*(3), 15–19. (Cit. on pp. 3, 7, 14, 15, 37).

Ma, H., Kumar, T. K. S., & Koenig, S. (2017). Multi-agent path finding with delay probabilities. In *AAAI Conference on Artificial Intelligence* (pp. 3605–3612). (Cit. on p. 43).

Ma, H., Li, J., Kumar, T. K. S., & Koenig, S. (2017). Lifelong multi-agent path finding for online pickup and delivery tasks. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 837–845). (Cit. on pp. 119, 176).

Ma, H., & Pineau, J. (2015). Information gathering and reward exploitation of subgoals for POMDPs. In *AAAI Conference on Artificial Intelligence* (pp. 3320–3326). (Cit. on p. 44).

Ma, H., Wagner, G., Felner, A., Li, J., Kumar, T. K. S., & Koenig, S. (2018a). Multi-agent path finding with deadlines. In *International Joint Conference on Artificial Intelligence* (pp. 417–423). (Cit. on pp. 7, 43, 73).

Ma, H., Wagner, G., Felner, A., Li, J., Kumar, T. K. S., & Koenig, S. (2018b). Multi-agent path finding with deadlines: Preliminary results. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 2004–2006). (Cit. on p. 43).

Ma, H., Yang, J., Cohen, L., Kumar, T. K. S., & Koenig, S. (2017). Feasibility study: Moving non-homogeneous teams in congested video game environments. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (pp. 270–272). (Cit. on pp. 1, 91).

Ma, H., Harabor, D., Stuckey, P. J., Li, J., & Koenig, S. (2019). Searching with consistent prioritization for multi-agent path finding. In *AAAI Conference on Artificial Intelligence* (pp. 7643–7650). (Cit. on p. 51).

Ma, H., Hönig, W., Kumar, T. K. S., Ayanian, N., & Koenig, S. (2019). Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *AAAI Conference on Artificial Intelligence* (pp. 7651–7658). (Cit. on p. 153).

Ma, H., Koenig, S., Ayanian, N., Cohen, L., Hönig, W., Kumar, T. K. S., Uras, T., Xu, H., Tovey, C., & Sharon, G. (2016). Overview: Generalizations of multi-agent path finding to real-world scenarios. In *IJCAI-16 Workshop on Multi-Agent Path Finding*. (Cit. on pp. 3, 7, 14, 15, 37).

Ma, H., Tovey, C., Sharon, G., Kumar, T. K. S., & Koenig, S. (2016). Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI Conference on Artificial Intelligence* (pp. 3166–3173). (Cit. on p. 60).

MacAlpine, P., Price, E., & Stone, P. (2015). SCRAM: Scalable collision-avoiding role assignment with minimal-makespan for formational positioning. In *AAAI Conference on Artificial Intelligence* (pp. 2096–2102). (Cit. on p. 91).

Masehian, E., & Nejad, A. H. (2009). Solvability of multi robot motion planning problems on trees. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 5936–5941). (Cit. on p. 40).

Mataric, M., Nilsson, M., & Simsarin, K. (1995). Cooperative multi-robot box-pushing. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 556–561). (Cit. on p. 1).

Melo, F. S., & Veloso, M. (2011). Decentralized MDPs with sparse interactions. *Artificial Intelligence*, *175*(11), 1757–1789. (Cit. on p. 44).

Morris, R., Pasareanu, C., Luckow, K., Malik, W., Ma, H., Kumar, T. K. S., & Koenig, S. (2016). Planning, scheduling and monitoring for airport surface operations. In *AAAI-16 Workshop on Planning for Hybrid Systems* (pp. 608–614). (Cit. on pp. 1, 2, 121).

Narayanan, V., Phillips, M., & Likhachev, M. (2012). Anytime safe interval path planning for dynamic environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 4708–4715). (Cit. on p. 158).

Nguyen, V., Obermeier, P., Son, T. C., Schaub, T., & Yeoh, W. (2017). Generalized target assignment and path finding using answer set programming. In *International Joint Conference on Artificial Intelligence* (pp. 1216–1223). (Cit. on pp. 115, 193).

Okumura, K., Machida, M., Défago, X., & Tamura, Y. (2019). Priority inheritance with backtracking for iterative multi-agent path finding. In *International Joint Conference on Artificial Intelligence* (pp. 535–542). (Cit. on pp. 132, 145, 193).

Parker, L. E. (1998). ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, *14*(2), 220–240. (Cit. on p. 122).

Parker, L. E. (1999). Cooperative robotics for multi-target observation. *Intelligent Automation and Soft Computing*, *5*(1), 5–19. (Cit. on pp. 58, 122).

Pecora, F., Andreasson, H., Mansouri, M., & Petkov, V. (2018). A loosely-coupled approach for multi-robot coordination, motion planning and control. In *International Conference on Automated Planning and Scheduling* (pp. 485–493). (Cit. on pp. 1, 121).

Phillips, M., & Likhachev, M. (2011). SIPP: Safe interval path planning for dynamic environments. In *IEEE International Conference on Robotics and Automation* (pp. 5628–5635). (Cit. on pp. 155, 175).

Poduri, S., & Sukhatme, G. S. (2004). Constrained coverage for mobile sensor networks. In *IEEE International Conference on Robotics and Automation* (pp. 165–171). (Cit. on p. 1).

Preiss, J. A., Hönig, W., Ayanian, N., & Sukhatme, G. S. (2017). Downwash-aware trajectory planning for large quadrotor teams. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 250–257). (Cit. on pp. 1, 91).

Röger, G., & Helmert, M. (2012). Non-optimal multi-agent pathfinding is solved (since 1984). In *International Symposium on Combinatorial Search* (pp. 173–174). (Cit. on pp. 11, 38, 61).

Ratner, D., & Warmuth, M. (1986). Finding a shortest solution for the N $\times$ N extension of the 15-puzzle is intractable. In *AAAI Conference on Artificial Intelligence* (pp. 168–172). (Cit. on pp. 7, 11, 38, 61).

Rohmer, E., Singh, S. P. N., & Freese, M. (2013). V-REP: A versatile and scalable robot simulation framework. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1321–1326). (Cit. on p. 185).

Rus, D., Donald, B., & Jennings, J. (1995). Moving furniture with teams of autonomous robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 235–242). (Cit. on p. 1).

Ryan, M. (2008). Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, *31*, 497–542. (Cit. on p. 41).

Ryan, M. (2010). Constraint-based multi-robot path planning. In *IEEE International Conference on Robotics and Automation* (pp. 922–928). (Cit. on p. 41).

Sajid, Q., Luna, R., & Bekris, K. E. (2012). Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *International Symposium on Combinatorial Search*. (Cit. on p. 40).

Salvado, J., Krug, R., Mansouri, M., & Pecora, F. (2018). Motion planning and goal assignment for robot fleets using trajectory optimization. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 7939–7946). (Cit. on pp. 1, 121).

Scharpff, J., Roijers, D. M., Oliehoek, F. A., Spaan, M. T. J., & de Weerdt, M. M. (2016). Solving transition-independent multi-agent MDPs with sparse interactions. In *AAAI Conference on Artificial Intelligence* (pp. 3174–3180). (Cit. on p. 44).

Shapley, L. S., & Shubik, M. (1971). The assignment game I: The core. *International Journal of Game Theory*, *1*(1), 111–130. (Cit. on pp. 7, 11, 15, 57, 61).

Sharon, G. (2015). *Novel search techniques for path finding in complex environment* (Doctoral dissertation, Ben-Gurion University). (Cit. on p. 43).

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, *219*, 40–66. (Cit. on pp. 14, 42, 44, 47–50, 77, 96, 97, 109, 113).

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, *195*, 470–495. (Cit. on p. 42).

Silver, D. (2005). Cooperative pathfinding. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (pp. 117–122). (Cit. on pp. 37, 41, 44–46).

Smith, B. S., Egerstedt, M., & Howard, A. (2009). Automatic generation of persistent formations for multi-agent networks under range constraints. *Mobile Networks and Applications*, *14*(3), 322–335. (Cit. on p. 1).

Solovey, K., & Halperin, D. (2016). On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research*, *35*(14), 1750–1759. (Cit. on p. 26).

Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., & Abbeel, P. (2014). Combined task and motion planning through an extensible planner-independent interface layer. In *IEEE International Conference on Robotics and Automation* (pp. 639–646). (Cit. on p. 8).

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In *AAAI Conference on Artificial Intelligence* (pp. 173–178). (Cit. on p. 42).

Stern, R., Goldenberg, M., & Felner, A. (2017). Shortest path for K goals. In *International Symposium on Combinatorial Search* (pp. 167–168). (Cit. on p. 165).

Stern, R., Sturtevant, N. R., Felner, A., Koenig, S., Ma, H., Walker, T. T., Li, J., Atzmon, D., Cohen, L., Kumar, T. K. S., Boyarski, E., & Barták, R. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. In *International Symposium on Combinatorial Search*. (Cit. on pp. 7, 14, 15).

Sturtevant, N. R., & Buro, M. (2006). Improving collaborative pathfinding using map abstraction. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (pp. 80–85). (Cit. on p. 41).

Surynek, P. (2009). A novel approach to path planning for multiple robots in bi-connected graphs. In *IEEE International Conference on Robotics and Automation* (pp. 3613–3619). (Cit. on p. 40).

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *AAAI Conference on Artificial Intelligence* (pp. 1261–1263). (Cit. on pp. 7, 11, 20, 39, 61, 67).

Surynek, P. (2012). Towards optimal cooperative path planning in hard setups through satisfiability solving. In *Pacific Rim International Conference on Artificial Intelligence* (pp. 564–576). (Cit. on p. 39).

Surynek, P. (2015). Reduced time-expansion graphs and goal decomposition for solving cooperative path finding sub-optimally. In *International Joint Conference on Artificial Intelligence* (pp. 1916–1922). (Cit. on p. 40).

Surynek, P., Felner, A., Stern, R., & Boyarski, E. (2016). Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *European Conference on Artificial Intelligence* (pp. 810–818). (Cit. on p. 40).

Surynek, P., Felner, A., Stern, R., & Boyarski, E. (2017). Modifying optimal SAT-based approach to multi-agent path-finding problem to suboptimal variants. In *International Symposium on Combinatorial Search* (pp. 169–170). (Cit. on p. 40).

Suzuki, I., & Kasami, T. (1985). A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, *3*(4), 344–349. (Cit. on pp. 126, 130).

Švancara, J., Vlk, M., Stern, R., Atzmon, D., & Barták, R. (2019). Online multi-agent pathfinding. In *AAAI Conference on Artificial Intelligence* (pp. 7732–7739). (Cit. on p. 193).

Tanner, H. G., Pappas, G. J., & Kumar, V. (2004). Leader-to-formation stability. *IEEE Transactions on Robotics and Automation*, *20*(3), 443–455. (Cit. on p. 1).

Thurston, T., & Hu, H. (2002). Distributed agent architecture for port automation. In *IEEE Computer Society Signature Conference on Computers, Software and Applications* (pp. 81–87). (Cit. on p. 1).

Tovey, C. (1984). A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, *8*, 85–90. (Cit. on pp. 13, 62, 63, 69).

Tovey, C., Lagoudakis, M., Jain, S., & Koenig, S. (2005). The generation of bidding rules for auction-based robot coordination. In *Multi-Robot Systems. From Swarms to Intelligent Automata* (Chap. 1, Vol. 3, pp. 3–14). Springer. (Cit. on p. 91).

Turpin, M., Michael, N., & Kumar, V. (2014). CAPT: Concurrent assignment and planning of trajectories for multiple robots. *The International Journal of Robotics Research*, *33*(1), 98–112. (Cit. on p. 91).

Turpin, M., Mohta, K., Michael, N., & Kumar, V. (2014). Goal assignment and trajectory planning for large teams of interchangeable robots. *Autonomous Robots*, *37*(4), 401–415. (Cit. on pp. 5, 8, 122).

Veloso, M., Biswas, J., Coltin, B., & Rosenthal, S. (2015). CoBots: Robust symbiotic autonomous mobile service robots. In *International Joint Conference on Artificial Intelligence* (pp. 4423–4429). (Cit. on pp. 1, 62, 121).

Wagner, G. (2015). *Subdimensional expansion: A framework for computationally tractable multirobot path planning* (Doctoral dissertation, Carnegie Mellon University). (Cit. on p. 42).

Wagner, G., & Choset, H. (2015). Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, *219*, 1–24. (Cit. on p. 42).

Wagner, G., & Choset, H. (2017). Path planning for multiple agents under uncertainty. In *International Conference on Automated Planning and Scheduling* (pp. 577–585). (Cit. on p. 43).

Wagner, G., Choset, H., & Ayanian, N. (2012). Subdimensional expansion and optimal task reassignment. In *International Symposium on Combinatorial Search* (pp. 177–178). (Cit. on p. 91).

Wang, K. (2012). *Scalable cooperative multi-agent pathfinding with tractability and completeness guarantees* (Doctoral dissertation, The Australian National University). (Cit. on p. 41).

Wang, K., & Botea, A. (2008). Fast and memory-efficient multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling* (pp. 380–387). (Cit. on p. 40).

Wang, K., & Botea, A. (2011). MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, *42*, 55–90. (Cit. on pp. 5, 9, 40, 122).

Werger, B. B., & Matarić, M. J. (2000). Broadcast of local eligibility for multi-target observation. In *Distributed Autonomous Robotic Systems 4* (pp. 347–356). Springer. (Cit. on pp. 58, 122).

Wurman, P. R., D'Andrea, R., & Mountz, M. (2008). Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, *29*(1), 9–20. (Cit. on pp. 1, 2, 115, 145, 177).

Yakovlev, K., & Andreychuk, A. (2017). Any-angle pathfinding for multiple agents based on SIPP algorithm. In *International Conference on Automated Planning and Scheduling* (pp. 586–593). (Cit. on p. 158).

Yu, J. (2016). Intractability of optimal multi-robot path planning on planar graphs. *IEEE Robotics and Automation Letters*, *1*(1), 33–40. (Cit. on p. 39).

Yu, J. (2017). Expected constant-factor optimal multi-robot path planning in well-connected environments. In *International Symposium on Multi-Robot and Multi-Agent Systems* (pp. 48–55). (Cit. on pp. 5, 9, 40, 122).

Yu, J., & LaValle, S. M. (2013a). Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X, Springer Tracts in Advanced Robotics* (Vol. 86, pp. 157–173). Springer. (Cit. on pp. 7, 8, 13, 14, 22, 57, 58, 61, 85, 86, 108).

Yu, J., & LaValle, S. M. (2013b). Planning optimal paths for multiple robots on graphs. In *IEEE International Conference on Robotics and Automation* (pp. 3612–3617). (Cit. on pp. 39, 40, 44, 51, 52, 54, 79, 93, 105, 109).

Yu, J., & LaValle, S. M. (2013c). Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI Conference on Artificial Intelligence* (pp. 1444–1449). (Cit. on pp. 7, 11, 20, 39, 61, 67).

Yu, J., & LaValle, S. M. (2016). Optimal multi-robot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, *32*(5), 1163–1177. (Cit. on p. 37).

Yu, J., & Rus, D. (2015). Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. In *Algorithmic Foundations of Robotics XI, Springer Tracts in Advanced Robotics* (Vol. 107, pp. 729–746). Springer. (Cit. on pp. 20, 26, 39, 49, 54, 61, 100).

Zheng, X., & Koenig, S. (2009). K-swaps: Cooperative negotiation for solving task-allocation problems. In *International Joint Conference on Artifical Intelligence* (pp. 373–378). (Cit. on p. 91).