論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Planning, Execution, Representation, and their Integration for Multiple Moving Agents
著者(和文)	奥村圭祐
Author(English)	Keisuke Okumura
出典(和文)	学位:博士（工学）, 学位授与機関:東京工業大学, 報告番号:甲第12460号, 授与年月日:2023年3月26日, 学位の種別:課程博士, 審査員:DEFAGO XAVIER,渡部 卓雄,石井 秀明,村田 剛志,下坂 正倫
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12460号, Conferred date:2023/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Doctoral Dissertation

Planning, Execution, Representation, and their Integration for Multiple Moving Agents

by
Keisuke Okumura

Graduate Major in Artificial Intelligence
School of Computing
Tokyo Institute of Technology

written under the direction of
Prof. Xavier Défago

March 2023

Abstract

Navigating a team of agents without colliding with each other plays a crucial role in the modern and coming automation era, including fleet operations in warehouses, as well as collaborative robotic construction, to name just a few. Aiming at providing foundations for such broad applications from computational aspects, this dissertation devotes to developing quick, scalable, near-optimal, robust, domain-independent, and end-to-end multi-agent navigation technologies. Multi-agent navigation is a very complicated art based on compositing many technical components from artificial intelligence and robotics. Therefore, the dissertation decomposes multi-agent navigation into three perspectives, that is, planning, execution, and representation, and respectively overcomes current limitations of cutting-edge technologies.

The first part studies planning, which asks how to determine a sequence of actions for agents. The corresponding problem of multi-agent navigation is often formulated as multi-agent pathfinding (MAPF) which asks for a list of collision-free paths on graphs for multiple agents. The primary challenge of MAPF is to maintain solvability and quality while suppressing computational effort. On one hand, state-of-the-art optimal MAPF algorithms have difficulty in solving instances from the grid MAPF benchmark, containing a few hundred agents, within realistic timeframes. On the other hand, sub-optimal algorithms can cope with massive instances in a short time (e.g., less than 30 s), meanwhile, such algorithms often lack completeness. Indeed, they often fail condensed or cluttered MAPF instances, even if less than ten agents are involved. Aiming at breaking this tradeoff, the dissertation first presents algorithms with short horizon planning called PIBT and TSWAP, respectively developed for solving MAPF iteratively, and, simultaneous target assignment and collision-free pathfinding. Then, the LaCAM algorithm is presented, which uses short-horizon planning like PIBT and TSWAP as a sub-procedure. LaCAM is complete for MAPF, furthermore, it eventually converges to optima, provided that solution cost is accumulative transition costs. As another direction, the dissertation also presents the framework of iterative refinement, enabling us to improve the quality of arbitrary MAPF solutions. Empirically, the dissertation demonstrates that these proposed methods have excellent performance in success rate, computation time, and solution quality, significantly outperforming existing MAPF technologies.

The second part studies execution, which asks how to achieve robust plan execution by agents under various uncertainties in the real world. The primary challenge here is, at runtime, how to ensure safety (i.e., no collision) and liveness (i.e., eventually reaching destinations) when something bad happens, unexpected from the planning phase. To this end, the dissertation studies a novel integration style of planning and execution, namely, deliberative offline planning assuming that agents reactively execute the plan at runtime. Two types of example studies are presented, called the OTIMAPP and MAPPCF problems, respectively for timing uncertainties and crash faults. For both proposed problems, theoretical foundations, such as computational complexities, as well as practical approaches to solving the problems are provided. As proofs-of-concept, demonstrations of decentralized execution with real robots are also included, while ensuring liveness, without any central intervention at runtime, and without any global interac-

tions. Such things can be achieved neither by conventional centralized execution styles that rely on global monitoring systems nor by decentralized execution styles that lack centralized planners.

The third and last part studies representation, which asks how to model the world for agents from infinite design choices. Considering representation issues is necessary to realize end-to-end multi-agent navigation. The primary challenge here is how to construct small but effective search spaces for subsequent planning. It is necessary to construct a sparse representation of the workspace (i.e., roadmaps), otherwise, it becomes dramatically difficult to find a combination of plausible paths because of having to manage a higher number of inter-agent collisions. Nevertheless, roadmaps should be sufficiently dense to ensure high planning solvability and better solutions. To break this tradeoff, the dissertation provides two directions. The first approach is learning to construct sparse roadmaps from planning demonstrations, in short, data-driven roadmap construction. The second approach is combining roadmap construction and multi-agent search (i.e. collision-free path planning), making it possible to develop a small but effective search space such that the multi-agent search is willing to use. Both concepts are extensively tested in various scenarios, revealing their power, i.e., solving more planning instances much faster compared to existing methods.

Putting everything together, the dissertation presents a consistent story to realize multi-agent navigation technologies applicable to various domains.

Acknowledgments

I deeply, deeply appreciate everyone who has supported and encouraged me either directly or indirectly.

Not to mention, this dissertation did not exist without my advisor, Prof. Xavier Défago. He was an excellent guide for my journey, in the sense of an academic mentor, research collaborator, supporter, and friend. I have no words to thank him.

Many thanks to Dr. Yasumasa Tamura, who made significant contributions throughout the dissertation. He gave me important advice and a lot of encouragement, along with nice drink recommendations.

Many thanks to Dr. François Bonnet. He also made significant contributions throughout the dissertation. I learned a lot from him, especially about thinking styles in theoretical computer science.

I would like to thank all the members of the lab, past and present, who have provided many intellectual comments. I also appreciate Mayuko Takano, a secretary at the lab. I definitely declare that she played a crucial role in promoting my research smoothly.

Dr. Ryo Yonetani mentored me for half a year during my internship at OMRON SINIC X (OSX), jointly with Mai Nishimura, Kazumi Kasaura at OSX, and Prof. Asako Kanezaki at Tokyo Institute of Technology. The internship was truly funtastic! The collaboration was successful, I learned a lot, and we did unique research together. I thank them.

Prof. Sebastien Tixeuil hosted me in LIP6, Sorbonne University in Paris, France for half a year. I enjoyed the research collaboration with him. Moreover, he sincerely supported my first long stay abroad. Without his support, I could not enjoy my staying. I thank him.

I appreciate Manao Machida at NEC Corporation, Prof. Tsukada Manabu, Mai Hirata at the University of Tokyo, Jean-Marc Alkazzi at IDEALworks. The research collaboration with them were invaluable.

I appreciate Prof. Yoshihiro Miyake and Dr. Hiroki Ora at Tokyo Institute of Technology. The content of the dissertation is irrelevant to them, however, they sincerely advised me on my first research project. That experience was indeed my academic starting point.

I appreciate many researchers who have made codes public. Maybe I do not have direct interactions, but I learned a lot from not only the papers but also from the codes. This is what I like about the communities of artificial intelligence.

I appreciate the anonymous reviewers of the past submissions to AAAI, AAMAS, AIJ, ICAPS, ICRA, IJCAI, IROS, T-RO, RA-L, and SODA (in alphabetical order). Their constructive and rigorous reviews certainly guided and encouraged my research. Of course, I was upset every time I got the rejection! But I know now that this is a very important process in scientific discovery. I deeply appreciate the reviewers' invaluable work.

This work was fully supported by JSPS KAKENHI Grant Number 20J23011, and partly supported by JSPS Overseas Challenge Program for Young Researchers (for staying in Paris) and JST ACT-X Grant Number JPMJAX22A1. Moreover, the work was partially supported by JSPS KAKENHI Grant Number 21K11748, 21H03423 and ANR project SAPPORO (ref. 2019-CE25-0005-1).

Throughout the three years of my Ph.D. duration, I have received financial support from the Yoshida Scholarship Foundation and the JSPS Research Fellowship for Young Scientists. I could not pursue my study without their support.

Last but not least, I would like to thank my family and friends for their support.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

The following list is in alphabetical order.

BFS	breadth-first search
CBS	conflict-based search
CTRM	cooperative timed roadmaps
CVAE	conditional variational autoencoder
DAG	directed acyclic graph
DBS	deadlock-based search
DCRF	decoupled conflict resolution framework
DFS	depth-first search
LaCAM	lazy constraints addition search for MAPF
MAPD	multi-agent pickup and delivery
MAPF	multi-agent pathfinding
MAPP	multi-agent path planning
MAPPCF	multi-agent path planning with crash faults
MDD	multi-valued decision diagram
ML	machine learning
MRMP	multi-robot motion planning
OTIMAPP	offline time-independent multi-agent path planning
PIBT	priority inheritance with backtracking
PP	prioritized planning
PRM	probabilistic roadmap
RRT	rapidly-exploring random tree
SBMP	sampling-based motion planning
SSSP	simultaneous sampling-and-search planning

Chapter 1

Introduction

> *Navigation:*
> *the act of directing a ship, aircraft, etc. from one place to another, or the science of finding a way from one place to another*
>
> — Cambridge Dictionary

Navigating a team of agents efficiently is a holy grail technology in the modern and coming automation era.

A prominent example can be seen in logistics, that is, fleet operations of mobile robots in warehouses [Wurman *et al.*, 2008]. In such systems, robots are continuously assigned tasks to convey packages. This problem is non-trivial because, when multiple robots operate in a shared workspace, *coordination* is required. It is obvious, but each robot has a physical body and hence exclusively occupies a certain region in the workspace. Consequently, without coordination, "something bad" can be easily triggered. For instance, robots may bump into each other and crash. In another case, robots may block each other, leading to deadlock situations where they cannot complete tasks permanently. If such bad things are triggered, packages never reach end-users. Moreover, coordination is not only necessary but also brings "something good." In the example of warehouse systems, system designers are eager to maximize the number of conveyed packages per a certain time (i.e., throughput). Without good-enough coordination strategies, the system throughput may be unsatisfactory even if the system is operated safely.

In the above example, *coordination is embodied as navigating a team of agents without colliding with each other, while guaranteeing that each robot eventually reaches its destination, moreover, minimizing losses of robot motions.* Herein, this kind of coordination is shortly referred to as *multi-agent navigation*. Throughout the dissertation, there is no special difference between the terms "agent" and "robot."

Not limited to logistics applications, the necessity to maneuver multiple moving agents is getting common. We can see the following examples:

- Modern factories for material handling to produce silicon wafers utilize thousands of autonomous vehicles [Intel Corporation, 2020].
- Displaying information by multiple mobile robots [Alonso-Mora *et al.*, 2012; Le Goc *et al.*, 2016] is another prominent example that needs coordination to form meaningful patterns by robots, nowadays commonly seen in drone shows as entertainment.
- Animated agents that do not have actual physical bodies may also require coordination, as examples seen in video games [Silver, 2005]; to make them realistic, it is ideal to prohibit two animated agents pass through each other.

(a) fleet operation **(b)** drone show **(c)** manufacturing

Figure 1.1: Example applications of multi-agent navigation. Images are retrieved from: 1.1a: Techwords (2017)/CC BY 4.0; 1.1b: Preetam.choudhury (2018)/CC BY 4.0; 1.1c: Steve Jurvetson (2012)/CC BY 2.0.

Moreover, multi-agent navigation is expected to play a crucial role in future high-impacted automated systems:

- Cooperative intelligent transportation systems are a prominent example. Considering extensive attention and investments in the developments of self-driving cars as of 2023, in near future, autonomous vehicles are expected to be deployed on city roads. If so, we need to manage traffic at intersections to avoid collisions between vehicles, as example studies seen in [Dresner and Stone, 2008; Hirata *et al.*, 2021]. This is where coordination is required.
- Assuming deployments of self-driving cars, another interesting application is automated parking [Okoso *et al.*, 2019], posing how to pack unmanned multiple vehicles into tight spaces to maximize space utility.
- Automating airport surface operations [Morris *et al.*, 2016] is an attractive application of multi-agent navigation, where various heterogeneous mobile entities (aircraft, track, etc) are needed to be operated while following time constraints.
- Collective robotic construction [Petersen *et al.*, 2019] is an exciting and promising research topic that automates building construction (e.g., towers or domes) by multi-robot systems. Doing so can naturally receive benefits of the nature of multi-robot systems such as concurrency and robustness. A bunch of studies in this field are emerging, e.g., assembling buildings by piling small brick-like parts [Augugliaro *et al.*, 2014; Werfel *et al.*, 2014] and 3D printing systems with multiple ground/aerial robots [Zhang *et al.*, 2018; Zhang *et al.*, 2022b]. Of course, without coordination, such construction is impossible.

Some of the above examples visually appear in Fig. 1.1. Other applications include scheduling on railway networks [Laurent *et al.*, 2021], multi-robot exploration in unexplored regions [Okumura *et al.*, 2018], and robot soccer [MacAlpine *et al.*, 2015], to name just a few. All these examples are based on multi-agent navigation technologies.

Despite its significance, the realization of multi-agent navigation entails many difficulties. Specifically, it is very challenging to design agents' behaviors to achieve efficient coordination (i.e., *planning*). For instance, *multi-agent path planning (MAPP)* poses a computational problem that seeks collision-free trajectories of multiple robots. This is the foundation of multi-agent navigation, however, even finding sub-optimal solutions are tremendously challenging in general [Hopcroft *et al.*, 1984; Spirakis and Yap, 1984; Hearn and Demaine, 2005]. The intractability of MAPP is relaxed when the environment is explicitly represented as graphs and two agents collide only when sharing vertices or edges simultaneously, which is often referred to as the *multi-agent pathfinding (MAPF)*

problem. Indeed, MAPF is sub-optimally solvable in polynomial time based on graph analytic approaches [Kornhauser *et al.*, 1984; De Wilde *et al.*, 2014; Yu and Rus, 2015]. Meanwhile, solving MAPF optimally remains computationally intractable in various criteria [Yu and LaValle, 2013b], even when restricting fields in grid structures [Banfi *et al.*, 2017; Geft and Halperin, 2022] or approximating solution quality [Ma *et al.*, 2016].

Another example of difficulties in multi-agent navigation lies in *execution*. Even a precise plan execution of *one* physical robot itself entails significant challenges, moreover, we need to care about *multiple* agents. Therefore, the potential that something unexpected happens dramatically increases as the number of agents increases. In addition, multi-robot systems are inherently *distributed systems* wherein an interconnected collection of autonomous agents is spatially deployed. Therefore, system designers need to be aware of difficulties stemming from the nature of distributed systems, such as the nonexistence of exact consensus about global time [Sheehy, 2015], unreliable communication, and robot faults.

To this end, this dissertation is dedicated to developing the foundations and practices for multi-agent navigation technologies, aiming to overcome the limitations of cutting-edge studies.

1.1 Multi-Agent Navigation Problem

The core problem of multi-agent navigation is abstracted as follows, which is partially influenced by [Pecora *et al.*, 2018; Mannucci *et al.*, 2021].

Definition 1.1 (multi-agent navigation problem)**.** *We consider a problem of trajectory planning and execution for a team of agents in the shared workspace, controllable by a central unit, subject to the following assumptions.*

- *Each agent is operated following its own motion constraints while occupying a certain region in the workspace.*
- *The workspace may contain static obstacles, each having a volume. Then, an agent region cannot overlap with obstacle regions.*
- *Two agents cannot overlap their occupying regions.*
- *The central unit can wirelessly communicate with each agent. It is assumed that any message is eventually delivered, however, communication delays are inevitable.*
- *Each agent is autonomous in the sense that it takes actions spontaneously while sensing surrounding situations.*
- *There is no reliable wall time clock, namely, each agent and the central unit can use its own clock, but it might be unsynchronized due to communication delays (e.g., clock shift or drift happens).*

The objective is to synthesize a sequence of instructions, issued by the central unit, to derive all agents to be at their destination.

Several details of Def. 1.1 are complemented below.

- The problem assumes a *centralized* controller, but of course, it is possible to take *decentralized* approaches without intervention by the central unit.
- The workspace is typically a three-dimensional closed space; at least, the dissertation assumes such spaces.
- An agent team may comprise *heterogeneous* agents. Those agents differ in their shapes and motion constraints.

- Motion constraints are particularly called *kinodyanamic constraints* in robotics. We see such examples in wheeled robots that cannot move directly in lateral directions, as well as high-speed cars that are impossible to stop instantly.
- Depending on applications, it is convenient to estimate an agent occupying region larger than its physical body. For instance, quadrocopters need to consider the downwash effect [Hönig *et al.*, 2018b].
- Communication between actors happens while preserving FIFO manner (first in, first out), namely, two messages sent by an actor in order are delivered to another actor in that order. This is because making non-FIFO protocol to FIFO is not difficult [Cachin *et al.*, 2011].
- Destinations may be assigned to each agent *a priori*, otherwise, *target assignment*, determining which agent goes where, is required. The necessity of target assignment occurs when systems do not care about task-executing agents, as seen in warehouse automation.

1.2 Ultimate Goal

Observe that Def. 1.1 covers, abstracts, and dominates applications of multi-agent navigation discussed at the beginning. Therefore, the dissertation ultimately aims at solving this problem in "good" manners from *computational* perspectives. More precisely, the objective is elaborated as follows.

Ultimate Goal of the Dissertation

Establish quick, scalable, near-optimal, robust, domain-independent, and end-to-end methodologies to solve the multi-agent navigation problem, specified in Def. 1.1.

The above description involves several key notions, further elaborated below.

Quickness. "Quick" refers to planning speed assessed by the amount of time required for computation. This is significant because most applications demand *real-time* planning where deliberation time is limited. For instance, in warehouse applications, path planning must be completed by the timing counting backward from the user-specified delivery date; trivially, slow planners spending a month are useless. Moreover, quick planners can play *replanning* in a feedback loop. As a general thing of control systems, one-shot execution style without intervention at runtime is vulnerable to reality gaps unpredictable in the planning phase. Therefore, it is desirable to update planning according to real-time data obtained through sensing. Slow planners are unable to play the replanning part because they are inevitably based on outdated sensing results.

Scalability. Modern factories often operate hundreds or more agents [Wurman *et al.*, 2008; Intel Corporation, 2020]. Here, unscalable methods that can handle only a few agents are useless. Therefore, the dissertation pursues scalable methods that can handle that size, i.e., hundreds or more agents.

Near-Optimality. Even with sufficient speed and scalability, navigation methods are useless if the quality of the outcome is terrible. In multi-agent navigation, we are interested in minimizing losses in agents' motions, typically represented by total travel time, makespan, or sum of moving distance related to energy consumption. It is ideal to achieve optimal coordination, however, such optimization problems may incur an unrealistic computational burden. On the other hand, in practice, optimal solutions are

not mandatory. Rather, it is sufficient to use *near-optimal* solutions. This is where the dissertation is heading.

Robustness. Building robust and resilient multi-robot systems is an emerging and important topic [Prorok *et al.*, 2021], since those systems are expected to be infrastructures of logistics or product lines. To build reliable systems, multi-agent navigation should be tolerant of a slight deviation from the perfect behaviors of agents; otherwise, the system is very vulnerable because it asks "all" agents to take actions precisely following instructions. This is a ridiculous assumption when the system involves hundreds or more agents.

Domain-Independence. Multi-agent navigation technologies are applicable to various domains (e.g., warehouse automation, animated agents, intelligent transportation systems), therefore, we should not restrict the discussion to one specific robotic system. Therefore, the dissertation puts aside application-specific optimization, rather, we seek domain-independent methods. The domain-independence poses a necessity to abstract many aspects of the navigation problem, such as agent shapes, motion constraints, and communication models; formulating these components itself is not trivial.

End-to-End Style. Finally, the dissertation aims at developing end-to-end approaches. As a counter-example, conventional MAPF studies assume a graph abstraction of the workspace, however, how to define such graphs is a remarkable problem. Moreover, MAPF regards the manner of executing solution paths as a blackbox, however, there are many significant challenges in robust plan execution as discussed earlier. The dissertation aims at removing such blackbox assumptions or manually designed technical components as much as possible.

1.3 Global Strategy

Multi-agent navigation is a very complicated art, based on compositing many technical components from fields of artificial intelligence and robotics. Studying it as it is is intractable. Instead, to obtain better outlooks, the dissertation uses a decomposition of multi-agent navigation into three perspectives, namely, *planning*, *execution*, and *representation*. Each perspective is compactly explained as follows:

- *Planning* poses how to determine a sequence of actions for agents.
- *Execution* poses how to execute planning by agents under various uncertainties that lurk in the real world.
- *Representation* poses how to model the world for agents from infinite design choices.

Figure 1.2 illustrates these three perspectives.

As a result, the "global" strategy to achieve the dissertation goal is overcoming the current limitations of cutting-edge technologies in each perspective, and then, integrating the developed techniques.

1.4 Contributions – Local Challenges and Strategies

Let me now explain the "local" challenges and strategies that address each perspective, constituting a series of contributions of the dissertation. Here, the minimum descriptions are provided. Further details will be explained in Chap. 3.6, after providing the preliminary and background knowledge of multi-agent navigation (Chap. 2 and 3).

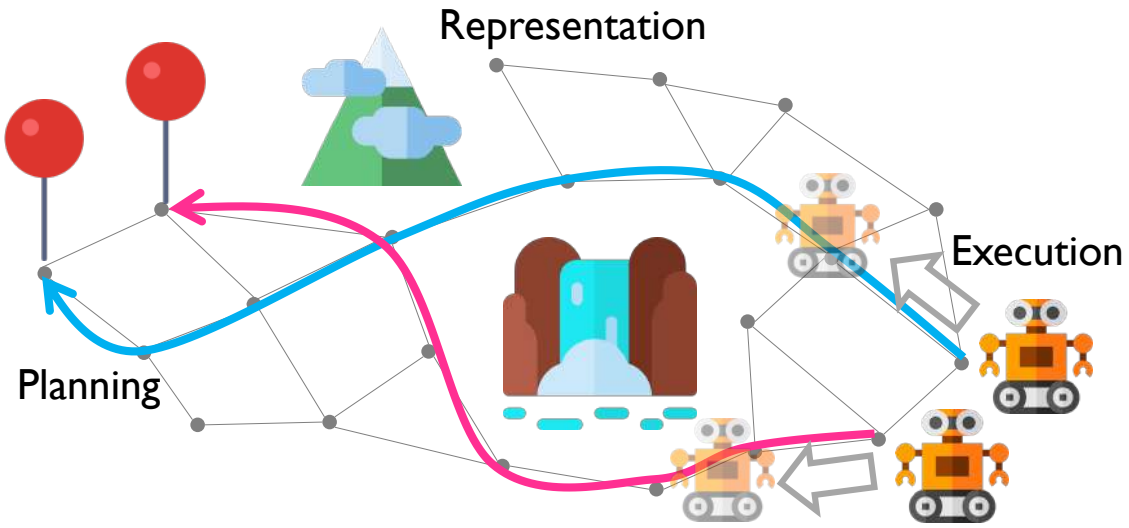


Figure 1.2: Illustration of planning, execution, and representation.

1.4.1 Planning

In planning for multi-agent navigation, *multi-agent pathfinding (MAPF)* plays a crucial role, which is a problem that assigns collision-free paths to each agent, given a graph and a list of start-goal pairs for agents. Therefore, the planning part challenges this compute-demanding problem, while following "good" manners discussed in Chap. 1.2. *The primary challenge of MAPF is to maintain solvability and quality, while suppressing planning efforts to secure speed and scalability.* On one hand, even with state-of-the-art optimal algorithms, planning with a few hundred agents in several minutes is still challenging [Li *et al.*, 2021b; Lam *et al.*, 2022]; they lack quickness and scalability. On the other hand, existing sub-optimal algorithms can solve massive instances, however, they lack good theoretical properties such as completeness [Silver, 2005; Ma *et al.*, 2019a; Li *et al.*, 2022]. Indeed, these algorithms often fail condensed or cluttered MAPF instances, even if less than ten agents are involved. Some graph analysis-based algorithms [Surynek, 2009; De Wilde *et al.*, 2014] can provide completeness, however, they drop domain independence; these algorithms do not go beyond puzzle solvers and are never leveraged to various multi-agent navigation problems.

To this end, the dissertation presents four primary contributions to breaking the current limitations of MAPF technologies.

- **Contribution-1 (Chap. 4):** The *PIBT* algorithm solving MAPF iteratively is presented. It ensures that all agents reach their destinations within a finite time, provided that all pairs of adjacent vertices in a graph belong to a simple cycle (e.g., biconnected).
- **Contribution-2 (Chap. 5):** The *TSWAP* algorithm solving simultaneous target assignment and path planning is presented. The problem is called *unlabeled-MAPF*. TSWAP is complete for unlabeled-MAPF.
- **Contribution-3 (Chap. 6):** The *LaCAM* algorithm solving MAPF is presented. LaCAM uses existing MAPF algorithms as a sub-procedure, such as PIBT or TSWAP. It is complete, furthermore, eventually converges to optimal solutions, provided that a solution cost is accumulative transition costs.
- **Contribution-4 (Chap. 7):** The *iterative refinement* framework for arbitrary MAPF solutions is presented. It uses existing MAPF algorithms as a sub-procedure to find appropriate neighborhood solutions.

All of them are designed with quickness, scalability, near-optimality, as well as domain-independence in mind. Especially, without a doubt, LaCAM has developed a new horizon in MAPF studies. Combined with PIBT, it sub-optimally solved 99% of instances retrieved from the MAPF benchmark [Stern *et al.*, 2019] in 10 s, while ensuring eventual convergence to optima. This performance significantly outperforms existing MAPF algorithms.

1.4.2 Execution

The primary challenge of execution is, at runtime, how to ensure safety (i.e., no collision) and liveness (i.e., eventually reaching destinations) when something bad happens, unexpected at the planning phase. The dissertation fights two types of something bad: *timing uncertainties* and *robot faults*. Specifically, the former considers how to overcome various types of timing uncertainties stemming from *robot internal factors*, such as kinematic constraints, slips, and battery consumption, as well as *distributed environmental factors*, such as communication delays, clock shift/drift, or uncaptured individual differences between robots. Meanwhile, the latter considers how to overcome robot faults happening at runtime, potentially blocking others and compromising the liveness.

To realize robust execution overcoming such something bad, the dissertation explores a novel relationship between planning and execution, namely, *offline planning assuming that agents "reactively" execute (or change) the plan at runtime*. This concept is exemplified in the following two contributions.

- **Contribution-5 (Chap. 9):** Aiming at overcoming timing uncertainties, the *offline time-independent multi-agent path planning (OTIMAPP)* problem is studied. This is a novel planning problem wherein agents spontaneously act without any timing assumptions. The problem requires a list of paths, ensuring that all agents eventually reach their destinations without permanently blocking each other.
- **Contribution-6 (Chap. 10):** Aiming at overcoming crash faults, the *multi-agent path planning with crash faults (MAPPCF)* problem is studied. In this problem, the crashed agents forever block part of the workspace. Correct agents (i.e., non-crashed ones) can detect crashes through local observations and then switch their executing path on the fly. The objective is to find a list of paths and their switching rules for each agent, such that all correct agents can reach their destinations regardless of unforeseen crash patterns.

For both proposed problems, both theoretical foundations (e.g., analysis of computational complexity) and practical approaches to solving the problems are provided. The dissertation also presents demonstrations with real robots as proofs-of-concept.

1.4.3 Representation

The representation issue arises when considering path planning in *continuous spaces*, namely, how to construct discretized representations of the workspace for subsequent planning. The representation often takes the form of a *roadmap*, which is a graph approximating the workspace for agents. Then, *the primary challenge of representation is constructing roadmaps suitable for planning algorithms.* On one hand, there is the necessity of constructing sparse roadmaps; otherwise, it would be dramatically difficult to find a combination of plausible paths due to the necessity to manage a higher number of inter-agent collisions. On the other hand, roadmaps should be sufficiently dense to ensure a high planning solvability and to derive near-optimal trajectories. In short, there is a tradeoff regarding roadmap density.

To break the tradeoff in representation, the dissertation provides the following two directions.

- **Contribution-7 (Chap. 11):** The first approach is *learning to construct sparse roadmaps from planning demonstrations*, i.e., data-driven roadmap construction. The constructed roadmaps are called *cooperative timed roadmaps (CTRM)* and exploited by arbitrary MAPF algorithms.
- **Contribution-8 (Chap. 12)**: The second approach is *building roadmaps while planning*. Combining roadmap construction and multi-agent search (i.e. collision-free planning) makes it possible to develop a small but effective search space such that the multi-agent search is willing to use. This idea is fruited as the *simultaneous sampling-and-search planning (SSSP)* algorithm, solving *multi-robot motion planning (MRMP)*.

Although multi-agent path planning in continuous spaces is tremendously challenging, both approaches contribute to reducing planning effort significantly, resulting in quick, scalable, near-optimal planning. Furthermore, SSSP is an example of domain-independence; it can do planning for various scenarios with diverse degrees of freedom and kinematic constraints of agents. Moreover, the study includes a demonstration of end-to-end multi-agent navigation using 32 ground robots in a dense situation.

1.5 Dissertation Outline

The rest of the dissertation begins with providing the preliminary knowledge in Chap. 2, e.g., the basis of algorithmic properties, search, and motion planning. Then, Chap. 3 broadly reviews studies on multi-agent navigation, followed by detailed challenges and strategies to achieve the dissertation goal. After that, the remaining dissertation divides into three parts: Part I for planning, Part II for execution, and Part III for representation. Finally, Chap. 13 concludes the dissertation while discussing future interesting directions.

The online material is summarized at `https://kei18.github.io/phd-dissertation/`.

Chapter 2

Preliminaries

Before starting topics of multi-agent navigation, this chapter provides preliminary knowledge used throughout the dissertation. The chapter begins with descriptions of notations and mathematical styles (Chap. 2.1), followed by terminologies of graphs (Chap. 2.2), basic algorithm properties such as completeness and optimality (Chap. 2.3), quick overview of computational complexity theory (Chap. 2.4), basis of planning and search (Chap. 2.5), and robotic motion planning (Chap. 2.6). Readers who are familiar with these notations can skip reading.

2.1 Notations and Mathematical Styles

The dissertation aligns with the following notations and mathematical styles, as much as possible.

- For convenience, the dissertation uses $\perp$ as "undefined" or "not found" sign.
- Bold letters represent vectors, e.g., $\boldsymbol{x} = [0, 1, 2]^\top$.
- Script capital letters represent collections of objects such as sets or tuples.
- A *set* $S = \{a, b, c, \dots\}$ is a collection of objects, where a, b, c are objects such as numerical values.
- A *tuple* $T = (a, b, c, \dots)$ is also a collection of objects but the objects are ordered. That is, given an integer k, it is valid to consider the k-th element in tuples. Meanwhile, the k-th element for sets is indefinable.
- $|S|$ (or $|T|$) denotes the number of elements in S (resp. T).
- $T[k]$ denotes the k-th element in a tuple T, where $1 \leq k \leq |T|$. Inspired by some programming languages like Python, let $T[-1]$ denote the last element of T, i.e., $T[-1] := T[|T|]$.
- The dissertation uses a simplified notation of the asymptotic complexity like $O(S)$ rather than $O(|S|)$, where S is a collection of objects.

Caution of Start Indexes in Collections. *An index of a tuple usually starts from one*, however, there are exceptions. To align with notations in the literature, *it sometimes starts at zero*. This is particularly when the index has the meaning of time. For instance, in MAPF, it is common to represent an initial location of agent-i as $\Pi_i[0] = a$, where $\Pi_i = (a, b, c \dots)$ is a timed path on a graph and $a, b, c, \dots$ are locations. For clarification, the dissertation complements the starting index as necessary.

2.2 Graph

Computer science cannot disjoint with *graph* representation; throughout the dissertation, graphs play a crucial role. This section provides the very basic terminologies of graph theory. For further details of graph theory, please refer to textbooks such as [Diestel, 2017].

A *graph*, or, an *undirected graph* G is a tuple (V, E). The elements of V are *vertices*. The elements of E are *edges*, a set of unordered pairs of vertices. A *digraph*, or, a *directed graph* D is a tuple (V, A). The elements of V are *vertices*. The elements of A are *arcs*, a set of ordered pairs of vertices. Given a digraph D, its *underlying graph*, denoted as $\mathcal{G}(D)$, is the undirected graph ignoring the direction of the arcs of D. This dissertation often does not distinguish between arcs and edges; simply calls both "edges."

An arc (or edge) is *self-loop* when it connects a vertex to itself, that is, (v, v) where $v \in V$. A directed graph is a *multigraph* when it has two or more edges with both the same tail and the same head vertices. An undirected graph is a *multigraph* when it has two or more edges that are incident to the same two vertices. More precisely, a multigraph is similar to a graph but edges E (or arcs for digraphs) are a multiset instead of just a set. A directed/undirected graph is *simple* when it is not a multigraph, moreover, it has no self-loops. Without explicit mentions, the dissertation assumes simple graphs.

A *path* in a digraph $D = (V, A)$ (or a graph $G = (V, E)$) is a non-empty sequence of vertices and arcs (resp. edges) of the form $v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k$ such that $v_i \in V$ for all $1 \leq i \leq k$, $e_j = (v_j, v_{j+1}) \in A$ (resp. $\{v_j, v_{j+1}\} \in E$) for all $1 \leq j \leq k-1$. A *simple path* is a path such that $v_i \neq v_j$ for all $i \neq j$. A *cycle* is a path such that $v_1 = v_k$, $k \geq 4$, and the repeated vertex is only the first/last vertex. Given a path Π, its i-th vertex is denoted as $\Pi[i]$. Its index starts from one. $|\Pi|$ is the number of vertices in Π. A *length* of a path Π is its number of edges, denoted by $\texttt{length}(\Pi) := |\Pi| - 1$. A path is often denoted by omitting edges, e.g., $\Pi = (v_1, v_2, \ldots, v_k)$. Here, $\Pi[1] = v_1$, $\Pi[-1] = v_k$, $|\Pi| = k$, and $\texttt{length}(\Pi) = k - 1$.

An (undirected) graph is *connected* when a path exists between each pair of distinct vertices. A graph is *biconnected* when the graph will remain connected if any one vertex is removed. A digraph D is *weakly connected* when $\mathcal{G}(D)$ is connected. A digraph is *strongly connected* when a path exists between each pair of distinct vertices. A digraph D is *strongly biconnected* when it is strongly connected and $\mathcal{G}(D)$ is biconnected. A *cycle graph* is a graph that consists of a single cycle. A directed graph without cycles is called a *directed acyclic graph (DAG)*. An undirected graph without cycles is called a *forest*. A forest is called a *tree* when it is connected. A graph is called *planar* when it is possible to draw in a plane so that none of the edges cross each other.

A vertex $v \in V$ is *adjacent* or *neighboring* to another vertex $u \in V$ when there is an edge connecting from u to v. Formally, the set of neighboring vertices of u is defined as follows.

$$\texttt{neigh}(u) := \{v \mid (u, v) \in A\} \qquad \text{(directed graph)} \tag{2.1}$$

$$\texttt{neigh}(u) := \{v \mid \{u, v\} \in E\} \qquad \text{(undirected graph)} \tag{2.2}$$

The *degree* of a vertex u is the number of outgoing edges from u, denoted by $\deg(u) := |\texttt{neigh}(u)|$. The maximum degree of G is denoted by $\Delta(G) := \max_{u \in V} \deg(u)$. Given two vertices u and v in a directed/undirected graph, the *distance* from u to v, denoted by $\texttt{dist}(u, v)$, is the shortest path length from u to v;

$$\texttt{dist}(u, v) := \operatorname*{argmin}_{\Pi \in \mathbf{\Pi}} \texttt{length}(\Pi) \tag{2.3}$$

$$\mathbf{\Pi} := \{ \text{ paths on } G \text{ starting from } u \text{ and ending at } v \ \} \tag{2.4}$$

The distance is undefinable when there is no path connecting u and v. The *diameter* of G, denoted by $\mathtt{diam}(G)$, is the greatest distance between any pair of vertices;

$$\mathtt{diam}(G) := \underset{u,v \in V}{\operatorname{argmax}}\ \mathtt{dist}(u,v) \tag{2.5}$$

2.3 Algorithm Properties

An *algorithm* is a pre-designed procedure to solve specific problems without human intervention at runtime. Designing appropriate algorithms is a central dogma of automation, and of course, that of the dissertation. This section briefly explains general things about algorithm properties.

2.3.1 Decision Problem

Given a problem instance I, a *decision problem* poses a yes-no question regarding I. For instance, a problem instance of *graph pathfinding* is specified by $I = (G, s, t)$, where $G = (V, E)$ is either a directed or undirected graph, $s \in V$ is a start vertex, and $t \in V$ is a goal vertex. The graph pathfinding problem asks existence of paths from s to t on G. In this case, a path from s to t is evidence for the yes-answer of I. This kind of evidence is called a *solution*. A solution is sometimes decorated as a *feasible* solution, to distinguish it from a *infeasible* solution that is unsatisfied with the problem specification. For instance, in graph pathfinding, a path that does not end at the goal vertex is an infeasible solution. *Solvable instances* are instances that have solutions (usually, they correspond to yes-answer instances), and *unsolvable instances* are instances without solutions.

2.3.2 Completeness

For a given decision problem, an algorithm is called *complete* when it is guaranteed to answer "yes" for all yes-answer instances and "no" for all no-answer instances, within a finite time. Otherwise, the algorithm is called *incomplete*.

2.3.3 Optimization Problem

In many practical situations, beyond decision problems, we are interested in optimizing specific criteria regarding solutions. For instance, in graph pathfinding, finding the shortest path is clearly important because the path length is usually related to, e.g., the time consumption of schedules or energy consumption of robot motions. This is an *optimization problem*. Given a problem instance and a cost function $\mathtt{cost}$ that maps a solution to a real value, the optimization problem asks to find a solution that minimizes $\mathtt{cost}$. In *the shortest pathfinding problem*, the cost function is defined as $\mathtt{cost}(\Pi) := \mathtt{length}(\Pi)$, where Π is a path on G that starts from s and ends at t. A solution x^* is *optimal* when no solution x exists such that $\mathtt{cost}(x) < \mathtt{cost}(x^*)$. Otherwise, the solution is *sub-optimal*. An algorithm is called *optimal* when it outputs optimal solutions for all solvable instances; otherwise, it is called *sub-optimal*. An algorithm is *complete and optimal* only if, within finite time, it outputs an optimal solution for all solvable instances, and reports unsolvability for all unsolvable instances. For instance, the CBS algorithm [Sharon *et al.*, 2015] for MAPF is optimal but incomplete because CBS cannot identify unsolvable instances. On the other hand, the M* algorithm [Wagner and Choset, 2015] for MAPF is complete and optimal.

It is worth notably that optimization problems are often compatible with decision problems. For instance, given the shortest pathfinding problem, we can consider the corresponding decision problem that asks whether a path from start to goal exists with a

length of exactly k. It is possible to solve the optimization problem by repeatedly solving the decision problem while incrementing k starting from zero.

2.3.4 Relaxations of Optimal Algorithms

Optimization problems are important, however, finding optimal solutions is sometimes computationally intractable. Therefore, algorithmic concepts that relax optimization have been developed. For instance, given a parameter $w \geq 1$, an algorithm is called *bounded sub-optimal* when it outputs a solution x such that $\mathtt{cost}(x) \leq w \cdot \mathtt{cost}(x^*)$ for all solvable instances, where x^* is an optimal solution. An *anytime algorithm* [Zilberstein, 1996] is first finding a sub-optimal solution, and then gradually improving the solution quality. The anytime property is desirable in *real-time* planning where deliberation time for algorithms is limited. This is because, until the deadline, algorithms can improve the solution quality while guaranteeing that at least one solution is available. We will encounter both properties in this dissertation.

2.3.5 Safety and Liveness

When designing algorithms, it is valuable to consider two properties attached to algorithms, namely, *safety* and *liveness*. Roughly speaking, an algorithmic property defines *safety* when the algorithm never does "anything bad," while an algorithm property defines *liveness* when the algorithm eventually does "something good." As an example, consider multi-robot navigation in a shared workspace. Inter-robot collisions are "anything bad" because robots might crash due to bumping each other; hence, safety includes collision-free guarantees, together with other safety conditions such as that robots do not go out of the workspace. Meanwhile, robots need to reach their destinations eventually; this is "something good" and specifies the liveness condition. For instance, in lifelong MAPF [Li *et al.*, 2021d], the PIBT algorithm [Okumura *et al.*, 2022b] provides the safe condition (i.e., collision-free), moreover, it also provides the live condition (i.e., every agent eventually reaches its assigned goal) when the workspace is represented as a biconnected graph.

2.4 Computational Complexity

Evaluating how computational problems are difficult is the foundation of computer science, taking the form of *computational complexity theory*. The theory is worth understanding because it provides an outlook of the "limitations" of algorithms. This section describes its very basic concepts and terminologies. For the formal definitions, please refer to textbooks such as [MacCormick, 2018; Cormen *et al.*, 2022]. The objective here is to cover Fig. 2.1.

2.4.1 Time Complexity

Given a computational problem (e.g., decision or optimization problems), most computer scientists, as well as practitioners, are first interested in whether the problem can be solved in *polynomial* time with respect to the problem size. This is because, for such problems, we can have a relatively high expectation to solve them in realistic timeframes. The explanation of "polynomial" is as follows. The problem is said to be solvable in polynomial time when a polynomial time algorithm that solves the problem exists. An algorithm is said to be *polynomial time* if its running time is upper bounded by a polynomial expression in the size of the input to the algorithm. Such an algorithm is often denoted as $O(\mathtt{poly}(n))$, where n is the size of the input and $\mathtt{poly}$ denotes a polynomial

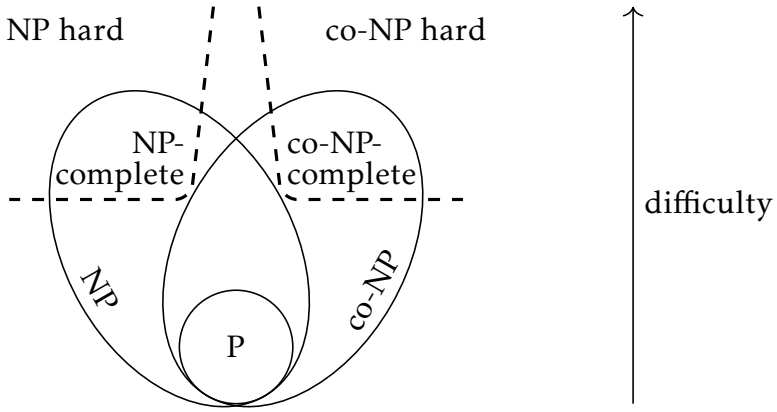


Figure 2.1: Complexity classes and hierarchies.

expression of n (e.g., n^k where $k \in \mathbb{N}$). Roughly speaking, the big-O notation means that, if n is sufficiently large, the algorithm runtime is no worse than $\mathtt{poly}(n)$.

2.4.2 Complexity Class

Using the concept of polynomial time, the difficulty of a decision problem can be categorized into various *complexity classes*. A decision problem is in *class P* when it is solvable in polynomial time. A decision problem is in *class NP* when a solution candidate, might be incorrect (i.e., infeasible), is *verifiable* in polynomial time. The *verification* asks whether the solution candidate is correct (i.e., feasible) for the problem. The problem is in NP if a polynomial time algorithm exists to answer this question. It is known that P $\subseteq$ NP, and, most people believe as P $\neq$ NP, as of 2023.

The computationally difficult problems are often categorized into *class NP-hard*. Informally, a decision problem is in NP-hard when it is at least as hard as the hardest problems in NP. Here, the hardness refers to how much compute-demanding the problems are; "hard" problems are expected to require much computation time. A problem is in *class NP-complete* when it is in both NP and NP-hard. An algorithm that solves the NP-complete problem can solve all other NP problems with polynomial-time overhead.

In this dissertation, another class will appear called *class co-NP*. A decision problem is in co-NP when its *complement* problem is in NP. In the complement problem, any yes-answer instances of the original problem must be answered as "no," and vice versa. Counterintuitively, whether co-NP is equivalent to NP is unknown as of 2023.[1] The corresponding class of NP-hard for co-NP is called *class co-NP-hard*; the problem is in co-NP-hard when it is at least as hard as the hardest problems in co-NP. A problem is in *class co-NP-complete* when it is in both co-NP and co-NP-hard.

Figure 2.1 visualizes relationships between P, NP, NP-hard, NP-complete, and their counterparts of co-NP. The figure will be a "guide" to the difficulty of problems that we will encounter.

It is worth mentioning that more computationally demanding classes exist than NP. For instance, a problem is in *class PSPACE* if it can be solved using an amount of memory that is polynomial in the input length. *Class EXPTIME* refers to problems that can be decided in time $O\left(2^{\mathtt{poly}(n)}\right)$. *Class PSPACE-hard*, *class PSPACE-complete*, *class EXPTIME-hard*, and *class EXPTIME-complete* are defined in the identical schemes to classes NP-hard and NP-complete against class NP. It is known that P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME.

In this dissertation, the critical border lies between P and NP, or, P and co-NP.

[1] If you prove either NP = co-NP or NP $\neq$ co-NP, you will get a million-dollar prize since it is contraposition of the P versus NP problem.

2.5 Planning and Search

As appeared in the title, *planning* plays a king role throughout the dissertation, which is typically computed by (combinatorial) *search*. This section quickly provides the basis for planning and search. For further details, I recommend referring to [Edelkamp and Schrodl, 2011; Ghallab *et al.*, 2016].

2.5.1 Planning

Planning is a computational procedure that determines a *sequence of actions* to lead an agent to ideal states. An action transits the agent's state from one to another. Therefore, planning is also regarded as a procedure that determines a *sequence of states*, subject to that two consecutive states are transitionable by an action. Unless explicitly mentioned, *this dissertation assumes planning to determine a sequence of states.*

For instance, navigating a (holonomic) robot in a cluttered workspace requires *path planning*. Here, the state of the robot is a location in the workspace. Planning then determines a sequence of locations that the robot will visit in order, where it starts from the current location of the robot and ends at a specified destination. If any two consecutive locations in the path are linearly travelable without encountering obstacles, the robot can reach the destination as long as following the path. Without planning, the robot may get lost and may fail to reach its destination.

When all states of an agent are represented in discrete manners, *planning is inherently equivalent to graph pathfinding*. In this case, a digraph $G = (V, A)$ can be either explicitly or implicitly constructed, where each state is a vertex and each action defines an edge. The converted graph pathfinding problem requests a path that starts from a vertex corresponding to the agent's initial state, and ends at one of the vertices corresponding to ideal states. Then, a solution, i.e., path $\Pi = (v_1, v_2, \ldots, v_k)$ on G, indeed specifies a sequence of states $v_1, v_2, \ldots, v_k \in V$, as well as a sequence of actions $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) \in A$. This graph is often called *search space*.

2.5.2 General Search Schema

Using the graph representation, planning is typically solved by *search*. Algorithm 2.1 presents a procedural schema of the search. We will meet many variations of this schema throughout the dissertation.

Algorithm 2.1 General search schema.

input: directed graph $G = (V, A)$, start $s \in V$, goals $V^{\text{goal}} \subset V$

output: path on G from s to $v \in V^{\text{goal}}$ or `FAILURE`

```
1: Open ← ⟦⟨state : s, parent : ⊥⟩⟧          ▷ ⟦...⟧ can be stack, queue, priority queue, etc
2: while Open ≠ ∅ do
3:     N ← Open.pop()                        ▷ selecting and removing one node from Open
4:     if N.state ∈ V^goal then return backtrack(N)
5:     for v ∈ neigh(N.state) do
6:         if ¬prunable(v,...) then          ▷ prunable returns TRUE or FALSE
7:             Open.push(⟨state : v, parent : N⟩)
8: return FAILURE
```

The crux of Alg. 2.1 is building a *search tree*, consisting of *search nodes*. A search node, herein simply called a *node*, typically consists of *state* that represents one state of the agent, and *parent* that points to another node that generated the node. From the view of a parent node, generated nodes (i.e., children) are often called *successors*. The search tree is a tree that has an edge between two nodes when they have a parent-child relationship.

The search tree is gradually constructed by maintaining an *Open* list that stores nodes. *Open* is implemented by data structures of stack, queue, or priority queue, etc. For each iteration of Lines 2–7, Alg. 2.1 picks one node n from *Open* and removes the node from *Open* (Line 3). If n satisfies the goal condition, a solution is obtained by backtracking the search tree (i.e., following *parent*) from n until reaching the initial state s (Line 4). Otherwise, it generates successors of n (Lines 5–7). In general, this process involves *pruning* (Line 6), aiming at reducing planning effort. Pruning refers to a process that avoids creating condition-matched successors. For instance, this process prunes nodes already appearing in the search tree. In the pseudocode, the `prunable` function plays this role.

By specifying *Open* and `prunable`, Alg. 2.1 can represent many of the existing search schemes. For instance, assume that the `prunable` function returns `TRUE` if and only if the state has been already included in the search tree. Then, when *Open* is implemented by a data structure of stack, Alg. 2.1 is called *depth-first search (DFS)*. When *Open* is implemented by a queue, it is called *breadth-first search (BFS)*. When *Open* is implemented by a priority queue following a node scoring function $f : N \mapsto \mathbb{R}$, where N is a set of all possible nodes, it is called *best-first search*.

The search effort (i.e., amount of required time) is largely determined by the average *branching factor*, which is the number of successors at each node.

2.5.3 The A* Algorithm

One of the important forms of best-first search is the A^* algorithm [Hart *et al.*, 1968], presented in Alg. 2.2. A* can solve a generalized version of the shortest pathfinding problem wherein each edge is weighted by a non-negative real value. Formally, given an edge cost function $\mathtt{cost}_e : V \times V \mapsto \mathbb{R}_{\geq 0}$, this optimization problem requests a path $(v_1, v_2, \ldots, v_k)$ from the start s (i.e., $v_1 = s$) to one of the goals V^{goal} (i.e., $v_k \in V^{\text{goal}}$) that minimizes the accumulated edge cost along the path: $\sum_{i=1}^{k-1} \mathtt{cost}_e(v_i, v_{i+1})$. The shortest pathfinding problem described in Chap. 2.3 is a special case where $\mathtt{cost}_e$ is a constant function that always returns one. In the rest of the dissertation, a *path cost* refers to the accumulated edge cost along the path, usually unweighted (i.e., $\mathtt{cost}_e(\cdot) = 1$). The `dist` function is defined by the minimum path cost between two vertices.

The procedural flow of A* is as follows. A* selects a node from *Open* according to *f-value*, which is a sum of *cost-to-come* (aka. *g-value*) and estimation of *cost-to-go* (aka. *h-value*). Cost-to-come is a path cost to reach the corresponding vertex from the start vertex. Cost-to-go is the minimum path cost to reach one of the goal vertices from the corresponding vertex. The estimation of cost-to-go is often called *heuristic* and provided via a heuristic function $\mathsf{h} : V \mapsto \mathbb{R}_{\geq 0}$. The h function is *admissible* when, for every state $v \in V$, $\mathsf{h}(v)$ is smaller than or equal to the actual cost-to-go. A* is guaranteed to be optimal for the shortest pathfinding problem when h is admissible. For instance, $\mathsf{h}(v) := 0$ is admissible. In this case, A* is equivalent to the celebrated *Dijkstra* algorithm [Dijkstra, 1959].

Instead of the f-value, it is possible to prioritize the h-value by making $\mathtt{cost}_e$ always return zero. This search scheme is called *greedy (best-first) search*. In general, there is no optimal guarantee for the greedy search, but it has the potential to speed up the search if the h-value is a good estimation of the actual cost-to-go.

Algorithm 2.2 The A* search. The same parts from Alg. 2.1 are gray-colored.

input: $G = (V, A)$, $s \in V$, $V^{\text{goal}} \subset V$, $\texttt{cost}_e : V \times V \mapsto \mathbb{R}_{\geq 0}$, $\mathsf{h} : V \mapsto \mathbb{R}_{\geq 0}$

output: path or FAILURE

```
Open ← [[⟨state : s, parent : ⊥, g : 0, f : h(s)⟩]]
while Open ≠ ∅ do
    N ← Open.pop()                ▷ priority queue, sorted by ascending order of f-value
    if N.state ∈ V^goal then return backtrack(N)
    for v ∈ neigh(N.state) do
        g ← N.g + cost_e(N.state, v)
        f ← g + h(v)
        if node M s.t. M.state = v ∧ M.f ≤ f has not appeared in the search tree then
            Open.push(⟨state : v, parent : N, g : g, f : f⟩)
return FAILURE
```

2.6 Motion Planning

Motion planning is a computational procedure that seeks a trajectory in continuous spaces to lead a physical object on ideal states. The previous section (Chap. 2.5) refers to planning in discretized spaces, while this section provides basic concepts of *planning in continuous spaces*. Part III, studying representation for planning, relies on the concepts presented in this section.

Motion planning has been developed in the literature of robotics. For further details, I recommend reading [Choset *et al.*, 2005; LaValle, 2006].

2.6.1 Configuration Space

To begin with, consider a circle-shaped robot that can move in an arbitrary direction, working in the 2D *workspace* $\mathcal{W} \subset \mathbb{R}^2$. A pair of two real values $(x, y) \in \mathbb{R}^2$ can specify a *configuration* of the robot, which corresponds to a state of planning in discretized domains. That is, a robot is abstracted by a single point in $\mathbb{R}^2$. The space where configurations are laid is called *configuration space*, or *C-space*, and denoted as $\mathcal{C}$. In this example, $\mathcal{C} = \mathbb{R}^2$. The number of *degrees of freedom* is the dimension of configuration space (e.g., two in this example). The dimension of workspace and configuration space is not necessarily to be identical. For instance, manipulators are operated in the 3D workspace while their degrees of freedom are linear to the number of joints.

In general, it is not allowed for a robot to take an arbitrary configuration $q \in \mathcal{C}$ because there are *obstacles*; the robot's body and obstacles cannot be overlapped. Let a set of points occupied by the robot at a configuration $q \in \mathcal{C}$ denote $\mathcal{R}(q) \subset \mathcal{W}$. Similarly, let $\mathcal{O} \subset \mathcal{W}$ be occupied regions by obstacles. An *obstacle space* for the robot is $\mathcal{C}^{\text{obs}} := \{q \in \mathcal{C} \mid \mathcal{R}(q) \cap \mathcal{O} \neq \emptyset\}$. Then, a *free space* for the robot is $\mathcal{C}^{\text{free}} := \mathcal{C} \setminus \mathcal{C}^{\text{obs}}$. That is, every feasible state of the robot is "packed" into a single point in the free space. Figure 2.2 visualizes a free space of the circle-shaped robot. For simplicity, the dissertation assumes that configurations leading to self colliding of robots are included in the obstacle space, as seen in manipulators.

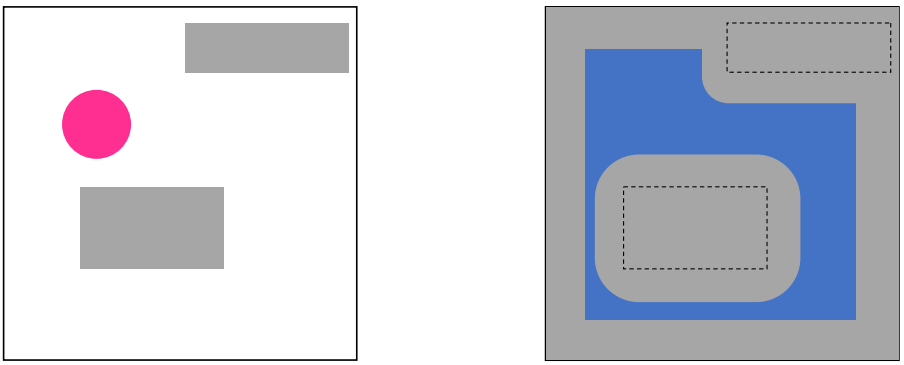


Figure 2.2: Illustration of free space. *left*: A robot is represented by a pink-filled circle. Gray zones are obstacle regions. A wall is represented by a black-line rectangle. *right*: The free space is a blue-filled region.

2.6.2 Decision Problem

In a nutshell, motion planning aims at finding a possibly twisting and turning line that connects the start and goal points (i.e., configuration), fitting in the free space. By continuously transiting the robot's configuration along that line, the robot can reach the ideal configuration without colliding with obstacles.

Formally, given a workspace $\mathcal{W}$, a configuration space $\mathcal{C}$, a set of obstacles $\mathcal{O}$, an initial configuration $q^{\text{init}} \in \mathcal{C}$, and a set of goal configurations $\mathcal{Q}^{\text{goal}} \subset \mathcal{C}$, the *motion planning* problem poses whether a collision-free trajectory exists that starts from q^{init} and ends in $\mathcal{Q}^{\text{goal}}$. A *trajectory* is defined by a continuous mapping $\sigma : [0,1] \mapsto \mathcal{C}$. It is *collision-free* when $\sigma(\tau) \in \mathcal{C}^{\text{free}}$ for $0 \leq \tau \leq 1$.

2.6.3 Roadmap

The primary challenge of motion planning is the manner of representing $\mathcal{C}^{\text{free}}$. It is not given *a priori*, rather, motion planning includes a process of building representation.

A popular method to represent $\mathcal{C}^{\text{free}}$ is using a (directed) graph $G = (V, E)$, specially called a *roadmap* in the literature. Each vertex of the roadmap corresponds to one configuration $q \in \mathcal{C}^{\text{free}}$ of the robot. Each edge $(q^{\text{from}}, q^{\text{to}}) \in E$ represents an existence of a collision-free trajectory from q^{from} to q^{to}. This trajectory is typically *local*, in the sense that it is easy to check whether two configurations are travelable. In motion planning studies, this connectivity checking process is treated as a blackbox procedure, sometimes called *local planner*. Once an appropriate roadmap is constructed, the motion planning problem is reduced to the graph pathfinding problem that seeks a path on the roadmap connecting to the initial and goal configurations. We can then obtain a trajectory that satisfies the solution conditions by applying the local planner that interpolates between consecutive points in the path.

Exact methods exist [Canny, 1988] to build a roadmap without resorting to approximations. However, they are too theoretical, and as stated in [LaValle, 2006], "it might be close to impossible to implement." Instead, approximating $\mathcal{C}^{\text{free}}$ is a practical approach. One of the common approaches is called *sampling-based motion planning (SBMP)* [Elbanhawi and Simic, 2014], explained in next.

2.6.4 Sampling-Based Motion Planning (SBMP)

To construct a roadmap, the SBMP methods roughly comprise the following steps.

1. Initiate the roadmap $G = (V, E)$.

2. Sample one configuration q^{rand} from $\mathcal{C}$.
3. Obtain $q^{\text{new}} \in \mathcal{C}^{\text{free}}$ using q^{rand}.
4. Add q^{new} to V. Update E with reference to q^{new} using the local planner.
5. Go back to Step 2 and repeat the above procedures until satisfying certain conditions.

Details of each step have design choices. Indeed, many SBMP algorithms exist. Prominent examples are probabilistic roadmap (PRM) [Kavraki *et al.*, 1996] and rapidly-exploring random trees (RRT) [LaValle, 1998].

Several important notions of SBMP are below.

- SBMP methods are often *probabilistically complete*, that is, the probability of finding a solution increases with more sampling trials, eventually converging to one.
- Moreover, with appropriate sampling and edge connection rules, they are sometimes *asymptotically optimal*; with more sampling trials, a solution trajectory approaches optimal.
- A constructed roadmap by RRT is not a general graph, rather, it is a tree. Such a SBMP algorithm is sometimes called a *tree-based planner*.
- Not limited to sampling uniformly at random from $\mathcal{C}$, many studies use *biased sampling* such that a configuration is sampled from "important" regions of $\mathcal{C}$, aiming to derive a better solution quickly, e.g., [Gammell *et al.*, 2014; Gammell *et al.*, 2015].

2.6.5 Kinodynamic Planning

Thus far, we assume that a robot can go in any direction in the configuration space, as long as it does not encounter obstacles. Such planning is sometimes called *geometric planning*. However, real robots are subject to kinematic constraints, as well as dynamics constraints. These motion constraints are typically expressed in terms of differential equations, governing the state of the robot. Consequently, we need to consider *planning under differential constraints*.

Kinematic constraints restrict the local directions of motion available to a robot from a given configuration. For instance, wheeled robots cannot translate sideways. Non-integrable kinematic constraints are often referred to as *nonholonomic constraints*. *Dynamics constraints* are governed by the time derivatives, such as velocity and acceleration. For instance, cars cannot stop instantly. Kinematic and dynamics constraints are collectively called *kinodynamic constraints*. Motion planning under kinodynamic constraints is called *kinodynamic planning* [Donald *et al.*, 1993; Schmerling and Pavone, 2019].

Instead of planning in configuration space $\mathcal{C}$, kinodynamic planning is performed in a *state space* $\mathcal{X} \in \mathbb{R}^n$. A *state* $x \in \mathcal{X}$ of the robot is a composition of a configuration $q \in \mathcal{C}$ and its derivative $\dot{q}$, i.e., $x := (q, \dot{q})$. Assume a *control space* of the robot, denoted as $\mathcal{U} \in \mathbb{R}^m$. For instance, the control space of vehicles is defined by a handle, an accel pedal, and a brake pedal. The state transition under kinodynamic constraints is then given by $\dot{x} = f(x, u)$, where $x \in \mathcal{X}$ and $u \in \mathcal{U}$. Kinodynamic planning requests a sequence of control inputs, namely, a continuous mapping $\mathbb{R}_{\geq 0} \mapsto \mathcal{U}$, such that the robot ends in the ideal states. Note that, given an initial state, the mapping of control input specifies the trajectory of the robot.

Fortunately, most SBMP methods are applied to kinodynamic planning with small modifications of how to construct roadmaps in the state space. For instance, edges of roadmaps are defined with two vertices, representing states, when there exist control inputs that enable a robot to traverse those two. Moreover, random samplings of the

state space are realized by, direct sampling from the state space, or, "simulating" robot behavior with a sampled control input from the control space, given a specific state. For this reason, it is not necessary to explicitly distinguish geometric and kinodynamic planning, at least in the context of SBMP.

2.7 Others

Chapter 11 requires basic knowledge of *machine learning (ML)*. Especially, it is desirable for readers to be familiar with the foundation of *deep learning*, however, no deep understanding of deep learning is required.[2] It is sufficient to know that neural networks can approximate "blackbox" functions from empirical data. For further details, please refer to textbooks such as [Bishop and Nasrabadi, 2006; Goodfellow *et al.*, 2016].

[2]Indeed, I definitely declare that I do not have such knowledge.

Chapter 3

Background

This chapter provides the background for presenting the contributions of this dissertation. That is, the chapter presents the basics of multi-agent navigation, followed by the current limitations of cutting-edge technologies, as well as strategies to overcome these limitations. The chapter starts with general things about multi-agent navigation to get a global picture.

3.1 Characterizing Multi-Agent Navigation

The objective of multi-agent navigation technologies is to solve Def. 1.1. In short, it is to ensure the eventual occurrence of "something good," such as all agents reaching their destinations, while never triggering "anything bad," such as inter-agent collisions. The former defines liveness while the latter defines safety. Throughout this chapter, we will see thorough reviews of how to achieve both these properties.

Recall that, since multi-agent navigation is very complicated, the dissertation relies on a decomposition into three perspectives, that is, *planning*, *execution*, and *representation*. These perspectives are briefly summarized as follows:

- *Planning* poses how to determine a sequence of actions for agents.
- *Execution* poses how to execute planning by agents under various uncertainties that lurk in the real world.
- *Representation* poses how to model the world for agents from infinite design choices.

The reviews in this chapter also exploits the three perspectives. Among the three, planning plays a king role. This is because execution without planning does not make sense, and, representation itself does not make sense. Both execution and representation must be combined with planning. Of course, planning without execution drops its significance, and, planning without representation is impossible. In this sense, the three perspectives cannot be considered separately.

We next see two fundamental axes to categorize approaches to multi-agent navigation. The first axis is whether an approach is *centralized* or *decentralized*. Another axis is whether an approach is *reactive* or *deliberative*. The two axes are based on both planning and execution perspectives, while representation does not affect them. The representation perspective appears when considering path planning in continuous spaces (i.e., motion planning).

3.1.1 Centralized vs. Decentralized

Multi-agent/robot systems are inherently *distributed* because agents are spatially deployed. Consequently, the following questions must explicitly be addressed: who does

planning, and who manages execution? This is a characteristic of multi-agent navigation that differs from single-agent navigation. It is important to distinguish between planning and execution regarding this point, elaborated as follows.

- *Centralized planning* refers to a planning style such that one component performs planning for all agents.
- *Decentralized planning* refers to a planning style such that each agent performs its own planning while negotiating with each other.
- *Centralized execution* refers to an execution style in which one component is responsible for the entire execution by monitoring the entire configuration and issuing instructions to agents as needed.
- *Decentralized execution* refers to an execution style in which each agent is responsible for its own execution by observing situations and performing actions spontaneously.

Characteristics in Planning. In general, people phrase as follows: centralized planning styles have good theoretical advantages over decentralized ones, such as completeness and optimality. On the other hand, it compromises the planning speed and scalability for the number of agents. Meanwhile, decentralized planning styles often have the advantage of scalability, while compromising theoretical properties. Note, this is somewhat correct but inaccurate. When a centralized component is available, in most cases, it is possible to "emulate" decentralized planning. Then, centralized planning is indeed scalable. The true thing about planning is that *it is difficult to achieve both good theoretical properties and scalability (or speed)*.

Characteristics in Execution. Unlike planning, execution is straightforward to discuss. Centralized execution styles have the advantage of theoretical properties such as safety and liveness, while the implementations are sometimes costly or impossible. For instance, consider deploying a thousand agents with centralized execution. It then requires real-time monitoring systems that can track a thousand agents without delays. It is not trivial at all how to realize such systems. In this sense, centralized execution is subject to the limitation of scalability. Decentralized execution can overcome this issue, however, provably safe and live systems are difficult to realize.

In Chap. 3.6.5, the relationships between centralized and decentralized execution styles will be refined to provide strategies of the dissertation.

3.1.2 Reactive vs. Deliberative

Aside from who manages control of multiple agents, the relationship between planning and execution is characterized by how the two interplay with each other. Specifically, we can distinguish between *reactive* and *deliberative* approaches.

Reactive Approaches repeatedly alternate planning and execution in a short duration, until agents reach ideal states. The planning used in reactive approaches is sometimes referred to as *online planning*. In multi-agent navigation, reactive approaches make agents continuously react to situations at runtime to avoid collisions while heading to their destination. Such examples can be seen in [Van Den Berg *et al.*, 2011; Lalish and Morgansen, 2012; Senbaslar *et al.*, 2018], to name just a few. This class is computationally inexpensive and suitable for decentralized planning/execution. However, provably deadlock-free

systems are difficult to realize owing to the shortsightedness of time evolution (i.e., lacking liveness). Moreover, reactive approaches may require rich and no-delay observations, such as accurate positions and velocities of the surrounding agents for each agent.

Deliberative Approaches first compute the entire planning, then execute it, and done; hence this is typically a one-short style. The planning used in deliberative approaches is sometimes referred to as *offline planning*. The main advantage of deliberative approaches is providing good theoretical properties, such as completeness and optimality. On the other hand, deliberative approaches need to manage uncertainties in execution, either by planning assumming uncertainties or runtime interventions during execution, as later discussed in Chap. 3.5. Moreover, similar to the centralized vs. decentralized approaches, deliberative approaches often suffer from the size of problem instances. This is because planning demands a large amount of computation time for large instances. In other words, planning used in deliberative approaches is typically slow compared to that used in reactive approaches.

When considering a strategy of planning in Chap. 3.6.3, we will see deeper insights into the two approaches.

3.1.3 Chapter Organization

Based on the perspectives provided the above, the rest of the chapter conducts an extensive literature review. Specifically, it is organized as follows.

- As mentioned earlier, planning plays a king role among the three perspectives. Centralized planning for multiple agents used in deliberative approaches is often embodied as a path planning problem for multiple agents on graphs, called *multi-agent pathfinding (MAPF)* [Stern *et al.*, 2019]. The MAPF problem plays a crucial role in the dissertation, hence the detailed review are provided in Chap. 3.2.
- *Classical* MAPF is convenient as a starting point, however, it over-simplifies many aspects of real multi-agent navigation, such as discretization of space and time. In Chap. 3.3, all simplifications in classical MAPF are removed to discuss a general form of planning in multi-agent navigation. The introduced problem is very challenging, called *multi-robot motion planning (MRMP)*. Solving MRMP efficiently is one of the ultimate goals of the dissertation. Since MRMP is planning in continuous spaces, we here need to consider the representation perspective.
- Next, in Chap. 3.4, another sort of multi-agent navigation will be discussed, namely, *path planning with target assignment*. MAPF and MRMP assume that agents have distinct goals *a priori*, however, real applications often care only about the completion of tasks; they do not care about who does tasks. This is where the target assignment is attractive.
- In Chap. 3.5, the execution perspective is organized, aiming at overcoming uncertainties in the real world. Specifically, timing uncertainties and robot faults at runtime are addressed.
- Based on provided knowledge in these sections, Chap. 3.6 presents non-trivial challenges and strategies of this dissertation.
- Last, in Chap. 3.7, relationships of the rest of the chapters are provided.

3.1.4 Disclaimer

- *Control theory* is sometimes used in reactive approaches, however, it is totally outside of the dissertation.

- The dissertation perspectives are not from roboticists, mathematicians, and control theory experts, but rather, they are from computer science, especially from artificial intelligence.

3.2 Multi-Agent Pathfinding (MAPF)

In a nutshell, MAPF is a problem that assigns a path to each agent on graphs such that any two agents never collide. MAPF plays a crucial role throughout the dissertation. Indeed, Part I devotes to developing powerful MAPF algorithms. The rest of this section consists of problem formulation (Chap. 3.2.1), known complexity analysis results (Chap. 3.2.3), reviews of existing algorithms (Chap. 3.2.4), and MAPF variants (Chap. 3.2.5).

3.2.1 Problem Formulation

Below, MAPF formalization is presented for undirected graphs. An MAPF problem on digraphs is similar to the undirected case.

Definition 3.1 (MAPF instance)**.** *An* MAPF instance *is defined by an undirected graph* $G = (V, E)$*, a set of agents* $A = \{1, \ldots, n\}$*, an injective initial state function* $s : A \mapsto V$*, and an injective goal state function* $g : A \mapsto V$*.*

As simplified notations, the dissertation uses s_i for $s(i)$, and g_i for $g(i)$, respectively.

Herein, as a problem formulation of MAPF, two types of representation are provided, whether it is based on *paths* or *configurations*. Both representations are equivalent. However, the representations are closely connected to concepts behind each MAPF algorithm, as we will see later. The dissertation uses both representations.

Formalization by Paths

Given an MAPF instance, let $\Pi_i[t] \in V$ denote the location of agent $i \in A$ at discrete time $t \in \mathbb{N}_{\geq 0}$. At each timestep t, agent-i can move to an adjacent vertex, or can stay at its current vertex, i.e., $\Pi_i[t+1] \in \mathsf{neigh}(\Pi_i[t]) \cup \{\Pi_i[t]\}$. Therefore, Π_i is a path on a "modified" graph of G such that each vertex has a self-loop. For clarification, the dissertation sometimes refers to this path as *timed path*. To align with the literature, its index starts from zero.

Timed paths for two agents $i, j \in A, i \neq j$, have a *vertex collision* when there exists $t \in \mathbb{N}_{\geq 0}$ such that $\Pi_i[t] = \Pi_j[t]$. Similarly, the two paths have an *edge collision* when there exists $t \in \mathbb{N}_{\geq 0}$ such that $\Pi_i[t] = \Pi_j[t+1] \wedge \Pi_i[t+1] = \Pi_j[t]$. The two paths are *collision-free* when neither vertex nor edge collisions exist.

Definition 3.2 (MAPF problem; representation by paths)**.** *Given an MAPF instance, an* MAPF problem *is a decision problem that asks existence of a tuple of timed paths* $\mathbf{\Pi} = (\Pi_1, \Pi_2, \ldots, \Pi_n)$*, where* $\Pi_i = (\Pi_i[0], \Pi_i[1], \ldots, \Pi_i[k])$ *is a timed path,* $k \in \mathbb{N}_{\geq 0}$ *is common between agents, while satisfying the following conditions:*

- $\Pi_i[0] = s_i$ *for all* $i \in A$*.*
- $\Pi_i[k] = g_i$ *for all* $i \in A$*.*
- *Any two paths in* $\mathbf{\Pi}$ *are collision-free.*

A *solution* to MAPF is $\mathbf{\Pi}$ that satisfies the condition.

Formalization by Configurations

We next see another representation of MAPF formulation. A *configuration* is a function that maps A to V. For convenience, the dissertation uses a tuple representation of the configuration, that is, $\mathcal{Q} = (v_1, v_2, \ldots, v_n) \in V^{|A|}$, where $\mathcal{Q}[i] = v_i$ is the location of agent $i \in A$. Then, the *start configuration*, or sometimes called the *initial configuration*, is $\mathcal{S} = (s_1, s_2, \ldots, s_n)$. The *goal configuration*, or *terminal configuration*, is $\mathcal{G} = (g_1, g_2, \ldots, g_n)$.

A configuration $\mathcal{Q}$ has a *vertex collision* when there is a pair of agents $i, j \in A, i \neq j$ such that $\mathcal{Q}[i] = \mathcal{Q}[j]$. Two configurations $\mathcal{Q}^{\mathrm{from}}$ and $\mathcal{Q}^{\mathrm{to}}$ has an *edge collision* when there is a pair of agents $i, j \in A, i \neq j$, such that $\mathcal{Q}^{\mathrm{from}}[i] = \mathcal{Q}^{\mathrm{to}}[j] \wedge \mathcal{Q}^{\mathrm{to}}[i] = \mathcal{Q}^{\mathrm{from}}[j]$. Two configurations $\mathcal{Q}^{\mathrm{from}}$ and $\mathcal{Q}^{\mathrm{to}}$ are *connected* when $\mathcal{Q}^{\mathrm{to}}[i] \in \mathtt{neigh}(\mathcal{Q}^{\mathrm{from}}[i]) \cup \{\mathcal{Q}^{\mathrm{from}}[i]\}$ for all $i \in A$, and, there are neither vertex nor edge collisions in $\mathcal{Q}^{\mathrm{from}}$ and $\mathcal{Q}^{\mathrm{to}}$.

Definition 3.3 (MAPF problem; representation by configurations)**.** *Given an MAPF instance, an* MAPF problem *is a decision problem that asks existence of a tuple of configurations* $\mathbf{\Pi} = (\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_k)$*, where* $\mathcal{Q}_t$ *is a configuration, satisfying the following conditions:*

- $\mathcal{Q}_0$ *is the start configuration* $\mathcal{S}$*.*
- $\mathcal{Q}_k$ *is the goal configuration* $\mathcal{G}$*.*
- *Any two consecutive configurations in* $\mathbf{\Pi}$ *are connected.*

A *solution* to MAPF is $\mathbf{\Pi}$ that satisfies the condition. To align with the literature, the index of $\mathbf{\Pi}$ starts from zero (i.e., $\mathbf{\Pi}[0]$ is the initial configuration).

Observe that a solution in Def. 3.2 (i.e., representation by paths) is transformable to a solution in Def. 3.3 (i.e., representation by configurations), and vice versa.

Understanding MAPF as the Graph Pathfinding Problem

The MAPF representation by configurations enables us to regard MAPF as the graph pathfinding problem. Indeed, we can construct a graph consisting of vertices representing configurations and edges representing the connectivity of configurations. The start and goal vertices are the start and goal configurations, respectively. Therefore, any graph pathfinding algorithms, such as A*, can solve MAPF.

Other Collision Types

The above formulations consider two types of collisions (i.e., vertex and edge collisions). Incorporating these two is popular among MAPF studies. Meanwhile, various collision types other than these two have been suggested [Stern *et al.*, 2019]. For instance, a *following collision* occurs when an agent uses the current location of another agent in the next timestep, i.e., $\mathcal{Q}^{\mathrm{from}}[i] = \mathcal{Q}^{\mathrm{to}}[j]$ with the representation by configurations. Imposing following collisions is sometimes useful when considering the asynchronous execution of MAPF.

3.2.2 Optimization Problems

Next, we consider optimization problems of MAPF. Recall that an optimization problem aims at finding a solution that minimizes a given cost function.

Using the MAPF representation by configurations (Def. 3.3), consider a transition cost between two configurations, $\mathtt{cost}_e : V^{|A|} \times V^{|A|} \mapsto \mathbb{R}_{\geq 0}$. Then, many optimization problems of MAPF take the form of the shortest pathfinding problem, that is, minimizing the accumulative transition cost defined along a solution (i.e., equivalent to accumulative edge cost in Chap. 2.5.3). Formally, a solution cost is defined as $\mathtt{cost}(\mathbf{\Pi}) := \sum_{t=0}^{k-1} \mathtt{cost}_e\big(\mathbf{\Pi}[t], \mathbf{\Pi}[t+1]\big)$, where k is the last timestep of the solution $\mathbf{\Pi}$. For instance:

- The *makespan* metric is a solution length, defined by $\texttt{cost}_e(X, Y) := 1$; that is, makespan is $\texttt{length}(\boldsymbol{\Pi})$.
- The *sum-of-fuels* metric (aka. total distance) is how many times agents move from their current locations, representing energy consumption, defined by $\texttt{cost}_e(\mathcal{Q}^{\text{from}}, \mathcal{Q}^{\text{to}}) := |\{i \in A \mid \mathcal{Q}^{\text{from}}[i] \neq \mathcal{Q}^{\text{to}}[i]\}|$.
- The *sum-of-loss* metric counts actions of non-staying at goals, defined by $\texttt{cost}_e(\mathcal{Q}^{\text{from}}, \mathcal{Q}^{\text{to}}) := \left|\{i \in A \mid \neg(\mathcal{Q}^{\text{from}}[i] = \mathcal{Q}^{\text{to}}[i] = g_i)\}\right|$.

Another popular MAPF metric is *sum-of-costs*, also known as *flowtime*. Using the MAPF representation by paths (Def. 3.2), the sum-of-costs is defined by $\sum_{i \in A} t_i$, where t_i is the earliest timestep such that $\Pi_i[t_i] = \Pi_i[t_i + 1] = \ldots = g_i$. This metric is impossible to be framed as the shortest pathfinding form because it is history-dependent on paths of agents.[1] For the same reason, the *maximum-moves* metric is impossible to formulate by the representation by configurations. Formally, given a timed path Π_i for agent i, the number of *moves* is how many times each agent moves to adjacent vertices. The maximum-moves metric takes the maximum over the scores of all agents. Note that its summation version is equivalent to the sum-of-fuels.

Sum-of-costs and makespan have Pareto optimal structure [Yu and LaValle, 2013b]. Generally, they cannot be minimized simultaneously.

3.2.3 Computational Complexity

Next, theoretical limitations of MAPF algorithms are provided from the view of computational complexity theory. The primary observation is that optimizing MAPF is very challenging.

Decision Problems

Undirected Graph. Historically, the complexity studies of MAPF started from solving *the pebble motion (PM) problem*, in which objects are moved on an undirected graph one-at-a-time to rearrange the objects. Hence, rotation of agents along a cycle is impossible, different from MAPF. PM is a generalization of 15 puzzles, illustrated in Fig. 3.1. In [Kornhauser *et al.*, 1984], a polynomial time procedure is shown to answer the solvability of PM, with the time complexity of $O\left(V^3\right)$, which is equal to the solution length (i.e., makespan). That is, PM is in P. The paper was later rediscovered in the MAPF research community [Röger and Helmert, 2012]. Later work by [Yu and Rus, 2015] studied a variant of PM that allows rotation movements, and provided a polynomial time procedure to solve PM with rotations, again with the time complexity of $O\left(V^3\right)$, which is equal to the solution length. In summary, the decision problem of MAPF on undirected graphs is in P.

Digraph. In contrast, the decision problem of MAPF on digraphs is in NP-complete [Nebel, 2020]. The NP-hardness does not hold when limiting graphs to strongly biconnected digraphs; a polynomial time procedure to solve such instances is reported in [Botea *et al.*, 2018]. Recent work in [Ardizzoni *et al.*, 2022] reports a polynomial time procedure for MAPF on strongly connected digraphs.

[1] However, it is worth mentioning that flowtime is definable as the shortest pathfinding problem by introducing virtual goals where once an agent has reached there, it cannot move anywhere in the future.

Figure 3.1: 15 puzzle. The image was retrieved from Wikipedia, ©Micha L. Rieser (2007).

Optimization Problems

What about the optimization of MAPF? In short, MAPF on undirected graphs is known to be an NP-hard problem for various optimization criteria. The known results are summarized in Table 3.1.

	general graph	planar graph	four-connected grid
sum-of-costs (aka. flowtime)	[Yu and LaValle, 2013b]	[Yu, 2015]	[Banfi *et al.*, 2017]
makespan	[Surynek, 2010]	[Yu, 2015]	[Banfi *et al.*, 2017]
sum-of-fuels (aka. total traveling distance)	[Yu and LaValle, 2013b]	[Yu, 2015]	[Geft and Halperin, 2022]
maximum-moves	([Yu, 2015])	[Yu, 2015]	N/A
sum-of-loss	([Yu and LaValle, 2013b])	N/A	N/A

Table 3.1: Known NP-hard MAPF problems. For each entry, the first study that reports the NP-hardness is displayed.

Solving PM optimally with reference to the solution length (i.e., makespan) is known to be NP-hard since the 1980s [Ratner and Warmuth, 1986]. Similarly, solving makespan-optimal MAPF in general graphs is proven to be NP-hard [Surynek, 2010]. The work by [Yu and LaValle, 2013b] shows that optimal MAPF problems about sum-of-costs (aka. flowtime), sum-of-fuels (aka. total distance), as well as makespan, are NP-hard. The proof regarding sum-of-costs is applicable to sum-of-loss without modifications; hence, solving sum-of-loss MAPF optimally is also NP-hard. In [Ma *et al.*, 2016], it is revealed that, even with an approximation of less than 4/3, the makespan-optimal MAPF is NP-complete. This implies that finding good-enough sub-optimal solutions is difficult to compute in general.

Later studies improve the hardness results. For instance, optimal MAPF problems in planar graphs regarding sum-of-costs, makespan, sum-of-fuels, and maximum-moves are NP-hard [Yu, 2015]. In [Banfi *et al.*, 2017], it is shown that solving MAPF optimally in four-connected grids regarding sum-of-costs and makespan is NP-hard. Recently in [Geft and Halperin, 2022], it is proven that optimizing sum-of-fuels in four-connected grids is also NP-hard.

3.2.4 Algorithms

To date, numerous algorithms have been developed to solve MAPF. There are three fundamental questions to characterize MAPF solvers.

- The first question asks about *completeness*. Given an MAPF instance, an algorithm is complete only if, in finite time, it returns a solution if the instance is solvable, otherwise, reports the non-existence of solutions as `NO_SOLUTION`. This property is achievable only if the search space is finite. Of course, complete algorithms are ideal, however, incomplete algorithms have also practical values because they are often scalable.
- The second question asks about *optimality*. Given an MAPF instance and a solution metric, an algorithm is optimal only if, in finite time, it returns an optimal solution if the instance is solvable. On one hand, optimal solvers are attractive, therefore, a lot of effort has been devoted to the development of powerful optimal algorithms. On the other hand, solving MAPF optimally is computationally intractable as seen in Chap. 3.2.3. Therefore, sub-optimal algorithms are also engaging because they are the only possibility to handle large MAPF instances (e.g., hundreds of agents or more).
- The third and last question is a bit vague compared to the previous two. That is, *which type of representation is used, paths or configurations*. This aspect will be explained with the details of several algorithms.

Below, representative MAPF algorithms are explained, fitted into six categories, while discussing completeness and optimality. The summary is available in Table 3.2. I also recommend referring to comprehensive reviews such as [Felner *et al.*, 2017; Stern, 2019; Ma, 2022].

	complete?	optimal?	representation	scalability	examples
graph pathfinding	✓	✓	configurations	poor	A* +OD [Standley, 2010] M* [Wagner and Choset, 2015]
compiling-based	SC	✓	configurations	poor/fair	SAT-based [Surynek *et al.*, 2016] ILP-based [Yu and LaValle, 2016]
two-level	SC	✓	paths	good	CBS [Sharon *et al.*, 2015] BCP [Lam *et al.*, 2022]
prioritized planning			paths	excellent	HCA* [Silver, 2005] RPP [Čáp *et al.*, 2015]
rule-based	✓		algorithm-dependent	outstanding	BIBOX [Surynek, 2009] PR [De Wilde *et al.*, 2014]
learning-based			configurations	excellent	PRIMAL [Sartoretti *et al.*, 2019] GNN-based [Li *et al.*, 2020]

Table 3.2: Categories of MAPF algorithms. 'SC' denotes solution complete, that is, ensuring to return a solution for solvable instances. The scalability with respect to $|V|$ or $|A|$ is also presented, based on my research experience.

Graph Pathfinding Approaches

MAPF can be seen as the graph pathfinding problem (Chap. 3.2.1), therefore, off-the-shelf pathfinding algorithms are available such as A* (Alg. 2.2). However, the branching factor of MAPF is exponential for the number of agents. Consequently, a vanilla A* is

impractical to solve MAPF. Instead, several techniques have been proposed to resolve the exponential size of the branching factor [Standley, 2010; Goldenberg *et al.*, 2014; Wagner and Choset, 2015]. In general, this class relies on the representation by configurations since MAPF is converted to graph pathfinding, regarding a configuration as a vertex. The search space is finite, therefore, algorithms can be complete. Furthermore, they are optimal with admissible heuristics, provided that the solution cost is accumulative transition costs.

Compiling-based Approaches

MAPF is reducible to various computational problems, such as constraint satisfaction problem (CSP) [Ryan, 2010], boolean satisfiability problem (SAT) [Surynek, 2012; Surynek *et al.*, 2016], integer linear programming (ILP) [Yu and LaValle, 2016], and answer set programming (ASP) [Erdem *et al.*, 2013; G'omez *et al.*, 2020]. Thus, one approach to solve MAPF is to compile an MAPF instance to another well-known problem and uses off-the-shelf matured solvers, such as Gurobi [Gurobi Optimization, 2022] or CPLEX [Cplex, 2009]. Algorithm 3.1 presents a general schema. Typically, compiling-based approaches repeatedly build an encoding for makespan-fixed MAPF instance and then ask for its solvability to, e.g., SAT solvers. This consulting continues until a solution is obtained. Although compiling-based approaches can be optimal, the search space is not limited to finite; they are thus incomplete. On other hand, for solvable instances, they are ensured to return a solution. The dissertation calls this property *solution complete*.

Algorithm 3.1 Compiling-based MAPF.

input: MAPF instance I

output: solution

1: $t_0 \leftarrow$ compute lower bound of makespan ▷ *e.g.*, $\max_{i \in A} dist(s_i, g_i)$
2: **for** $t = t_0, t_0 + 1, t_0 + 2, \ldots$ **do**
3: build encoding $\mathcal{E}$ of I with makespan t
4: $\mathbf{\Pi} \leftarrow$ consult about $\mathcal{E}$ by an off-the-shelf solver
5: **if** $\mathbf{\Pi} \neq \perp$ **then return** $\mathbf{\Pi}$ ▷ *Recall that* $\perp$ *means "not found" or "undefined"*

Typically, encodings use a variable that represents whether a location at a certain timestep is used by a specific agent. Therefore, for large MAPF instances (as for $|V|$ or $|A|$), the number of variables is dramatically increasing, bottlenecking for compiling-based approaches. To mitigate this issue, efficient encoding or branching techniques of variables have been actively studied, e.g., [Husár *et al.*, 2022; Achá *et al.*, 2022].

Remarks of Completeness. I acknowledge that the definition of completeness in MAPF is debatable. According to [Yu and Rus, 2015], solvable MAPF instances have a solution of $O\left(V^3\right)$-makespan. Therefore, it is possible to set the makespan upper bound; then compiling-based approaches are complete. However, I present them as "incomplete" because the upper bound analysis is very sensitive to MAPF specifications. For instance, in MAPF-C [Hönig *et al.*, 2018b] where even two agents in distinct vertices can be colliding, which is the starting point of MRMP, the analysis does not make sense at all. For the same reason, I present two-level approaches as incomplete.

Two-level Approaches

Modern and powerful MAPF algorithms take the form of two-level approaches, using the representation by paths. In a nutshell, these algorithms start from infeasible solutions that contain collisions, and then gradually resolve collisions by re-computing individual paths for agents. The term *two-level* comes from that most algorithms have the same algorithmic structure as follows.

- At the high-level, algorithms manage collisions among paths for agents.
- At the low-level, algorithms compute individual paths for agents, under constraints imposed by the high-level.

Representatives of two-level approaches are *conflict-based search (CBS)* [Sharon *et al.*, 2015] and ICTS [Sharon *et al.*, 2013]. In recent years, compiling-based approaches also take the form of two-level approaches, achieving successful results, as seen in SAT [Surynek *et al.*, 2022] or branch-and-cut-and-price (BCP) framework [Lam *et al.*, 2022]. They are optimal but incomplete; the search space is not finite, regarding how to generate constraints at the high-level. Rather, they are solution complete. The most popular approach, CBS, is presented in detail as follows.

Conflict-Based Search (CBS). Algorithm 3.2 provides the pseudocode of CBS. A high-level search of CBS manages collisions between agents. When a collision occurs between two agents at a certain time and location, there are two possible resolutions depending on which agent gets to use the location at that time. Following this observation, CBS constructs a binary tree where each high-level node includes constraints prohibiting the use of space-time pairs for certain agents. In a low-level search, agents find a single path constrained by the corresponding high-level node.

Algorithm 3.2 Conflict-based search (CBS).

input: MAPF instance I

output: solution

1: $Open \leftarrow [\![\langle\, paths : \texttt{get\_initial\_paths}(I),\ constraints : \emptyset \,\rangle]\!]$
2: **while** $Open \neq \emptyset$ **do**
3: $\quad \mathcal{N} \leftarrow Open.\text{pop}()$
4: $\quad C \leftarrow$ get constraints for $\mathcal{N}$ ▷ *constraint specifies where and when is prohibited for i*
5: $\quad$ **if** $C = \emptyset$ **then return** $\mathcal{N}.paths$
6: $\quad$ **for** $(i \in A,\ constraint) \in C$ **do**
7: $\quad\quad \mathcal{N}^{\text{new}} \leftarrow \langle\, paths : \mathcal{N}.paths,\ constraints : \mathcal{N}.constraints \cup \{(i, constraint)\} \,\rangle$
8: $\quad\quad \Pi_i \leftarrow$ compute a path for i following $\mathcal{N}^{\text{new}}.constraints$ ▷ *low-level search*
9: $\quad\quad$ **if** $\Pi_i \neq \bot$ **then**
10: $\quad\quad\quad \mathcal{N}^{\text{new}}.paths[i] \leftarrow \Pi_i$
11: $\quad\quad\quad Open.\text{push}(\mathcal{N}^{\text{new}})$

CBS has a bunch of powerful enhancements, e.g., prioritizing collisions [Boyarski *et al.*, 2015], improving heuristics [Felner *et al.*, 2018; Li *et al.*, 2019a], and posing effective constraints [Li *et al.*, 2021b; Zhang *et al.*, 2022a], to name just a few. The low-level search

is typically performed by A*, but replacing it has been studied, by specialized pathfinding algorithms in dynamic environments such as SIPP [Phillips and Likhachev, 2011] or temporal jump point search [Hu *et al.*, 2021; Hu *et al.*, 2022]. Bounded sub-optimal versions of CBS have also been proposed [Barer *et al.*, 2014; Li *et al.*, 2021c].

Prioritized Planning

This class is incomplete and sub-optimal, however, it is computationally inexpensive and often outputs solutions with acceptable quality. In short, prioritized planning (aka. PP) is practical in real scenarios. The scheme is sequentially planning individual paths for agents, while avoiding collisions with already planned paths [Erdmann and Lozano-Perez, 1987; Silver, 2005]; therefore, PP relies on the representation by paths. Algorithm 3.3 provides the pseudocode. PP is a typical example of *decoupled planning* that avoids planning in a joint space where the actions of all agents are considered.

Algorithm 3.3 Prioritized planning (PP).

input: MAPF instance I

output: solution or `FAILURE`

```
1: initiate Π (i.e., partial solution)
2: for i ∈ A do
3:     Π_i ← compute a path for agent-i without collisions with other paths in Π
4:     if Π_i = ⊥ then return FAILURE
5:     register Π_i to Π
6: return Π
```

There is a sufficient condition that the sequential collision-free solution can always be constructed regardless of planning orders [Čáp *et al.*, 2015]. Such instances are called *well-formed*. The condition of well-formed instances is that, for each pair of start and goal, a path exists that traverses no other starts and goals. Meanwhile, a theoretical study in [Ma *et al.*, 2019a] reveals that there is an instance such that PP fails for any order of priorities. Because priority ordering is crucial for PP in both solvability and solution quality, a bunch of studies addressed this issue [Azarm and Schmidt, 1997; Bennewitz *et al.*, 2002; Van Den Berg and Overmars, 2005; Ma *et al.*, 2019a; Wu *et al.*, 2020; Zhang *et al.*, 2022c].

Rule-based Approaches

This class leverages analytical results of graph topology, and makes agents move step-by-step following ad-hoc rules to solve MAPF. In [Peasgood *et al.*, 2008], a rule-based algorithm relying on spanning tree construction is studied. BIBOX [Surynek, 2009] is a polynomial time complete algorithm for MAPF in biconnected graphs. Representative of this class is the *push and swap (PS)* algorithm [Luna and Bekris, 2011] for arbitrary graphs, relying on two primitives: the "push" operation to move an agent toward its goal and the "swap" operation to allow two agents to swap locations without altering the configuration of other agents. PS was originally presented as a complete algorithm, but the follow-up work [De Wilde *et al.*, 2014] pointed out it is indeed incomplete. Instead, the work presented the push and rotate (PR) algorithm. PR is a polynomial time complete algorithm for PM without rotations when at least two unoccupied vertices exist in the start configuration. PR is also seen as an implementation algorithm of analysis by [Kornhauser *et al.*, 1984].

Rule-based approaches have both advantages and disadvantages. The good points are that they are complete for certain instances. Furthermore, they can solve MAPF in polynomial time, i.e., ensuring excellent scalability. However, they are sub-optimal and the solution quality is often terrible. Furthermore, the rule-based approaches are very sensitive to problem specifications; the most critical flaw in achieving domain independence. For instance, PR cannot solve MAPF instances requiring rotations. Other classes, such as PP, are easy to adapt to changes in MAPF specifications. In contrast, once a specification is slightly changed, the rule-based approaches require substantial changes to algorithm structure, or in the worst case, the algorithms do not make sense at all.

Learning-based Approaches

This class is an emerging area that uses machine learning (ML) techniques, influenced by recent successful results of deep learning in various fields [Mnih *et al.*, 2015; Krizhevsky *et al.*, 2017; Jumper *et al.*, 2021]. Typically, learning-based approaches mimic good agent behaviors in expert data, for example, by combining imitation learning and reinforcement learning [Sartoretti *et al.*, 2019; Damani *et al.*, 2021]. In the planning phase, each agent observes nearby situations and determines the next location, indicated by a trained ML model. Hence, this class relies on the representation by configurations. As ML models, various choices exist to incorporate how agents interact with each other, such as using graph neural networks (GNN) [Li *et al.*, 2020; Li *et al.*, 2021e; Ma *et al.*, 2021] or learning collision avoidance policy with Monte-Carlo simulation [Skrynnik *et al.*, 2021].

They have good potential for scalability with plausible solution quality, however, they do not have any theoretical guarantees in general. In other words, learning-based approaches are incomplete and sub-optimal, furthermore, might produce infeasible solutions (e.g., paths with collisions).

Others

Lastly, approaches not fitted into the aforementioned categories are introduced. The FAR algorithm [Wang *et al.*, 2008] first annotates a vertex so that avoids head-on collisions of agents, and then plans the respective agent's path by A* following the annotated graph, while locally avoiding deadlocks by a heuristic method. The MAPP algorithm [Wang and Botea, 2011] combines PP and rule-based approaches, which is complete in some instances. These two algorithms were pioneers of MAPF algorithms. Influenced by SBMP, in [Cáp *et al.*, 2013], a discretized version of RRT is presented to solve MAPF. In [Bouzy, 2013], an adaptation of Monte-Carlo tree search to MAPF is studied. In [Li *et al.*, 2022], a large neighborhood search [Ahuja *et al.*, 2002] is applied to MAPF, resulting in an excellent solver called LNS2; the algorithm is similar to the two-level approaches like CBS. At the high level, LNS2 selects a subset of agents. At the low level, it replans paths for those agents while avoiding collisions as much as possible. This continues until entire agents have no collisions. Note, a similar concept has already appeared in [Standley and Korf, 2011].

I am aware that many interesting algorithms other than the above have been proposed, but of course, I cannot list all of them; they are endless! So I end up with the introduction of MAPF algorithms here.

3.2.5 Variants of MAPF – One Step before MRMP

Thus far, many variants of MAPF have been studied. Among them, some variants are positioned "one step before" multi-robot motion planning (MRMP), introduced in the next section. Especially, three directions are worth to be mentioned here.

Continuous Space and Time

Relaxing discretized space and time assumptions is one important direction beyond classical MAPF. For instance, MAPF assumes that each edge has a unit cost; however, in reality, edge weight (i.e., traveling time) differs among locations. Motivated by this fact, MAPF on weighted graphs has been studied [Walker *et al.*, 2018; Ren *et al.*, 2021].

Most MAPF studies assume grid environments that allow agents to move in four directions: north, west, south, or east. However, allowing diagonal moves (e.g., northwest and east-southeast) can save the traveling time of agents. The corresponding problem for single-agent is known as any-angle path planning on grids [Daniel *et al.*, 2010], and its MAPF adaptation has also been studied [Yakovlev and Andreychuk, 2017].

Classical MAPF assumes that agents take action aligned with discretized time scale. In contrast, allowing each agent to take action at any time can smooth the planned trajectory. The corresponding problem is known as MAPF with continuous time [Andreychuk *et al.*, 2022; Kasaura *et al.*, 2022].

Kinodynamic Constraints

Making agents close to "physical robots" is also an important direction. Classical MAPF abstracts real robots into agents while dropping many important features such as kinodynamic constraints. Hence, there are huge reality gaps to deploy MAPF with real robots. To reflect the kinodynamic constraints, post-processing classical MAPF solutions has been studied [Hönig *et al.*, 2016; Ma *et al.*, 2019b]. Another approach is incorporating the constraints in the planning phase [Yakovlev *et al.*, 2019; Wen *et al.*, 2022], assuming specific models such as differential drive robots.

Heterogeneous Agents

The third direction is breaking the homogeneity of agents. A team of agents often consists of different types of agents regarding shapes and kinodynamic constraints such as maximum speeds. Thus, allowing planning for a team of heterogeneous agents is appealing. For this purpose, MAPF with heterogeneous agents, especially for heterogeneity of shapes, has been studied [Li *et al.*, 2019c; Atzmon *et al.*, 2020c]. Those studies demonstrate that MAPF algorithms (e.g., CBS) can adapt to heterogeneity.

3.3 Multi-Robot Motion Planning (MRMP)

In a nutshell, *multi-robot motion planning (MRMP)* is a multi-robot version of motion planning, that is, planning trajectories for multiple objects to lead them to ideal states. MRMP is also regarded as a generalized version of MAPF, assuming continuous space and time, incorporating kinodynamic constraints, and allowing a team of heterogeneous agents. Therefore, developing quick and scalable methods to solve MRMP near-optimally is a holy grail for automation. Indeed, MRMP is the end boss monster of the dissertation, studied in Part III.

3.3.1 Problem Formulation

The below definition naturally extends (single-agent) motion planning introduced in Chap. 2.6 to multi-agent scenarios. We first define geometric planning, i.e., MRMP without kinodynamic constraints. To align with the literature, this part uses "robot" instead of "agent," however, note again that these two have no special difference throughout the dissertation.

Geometric MRMP

Geometric MRMP considers a problem of motion planning for a team of n robots $A = \{1, 2, \ldots, n\}$ in the 3D closed *workspace* $\mathcal{W} \subset \mathbb{R}^3$. Each robot i is operated in its own *configuration space* $\mathcal{C}_i \subset \mathbb{R}^{d_i}$ where $d_i \in \mathbb{N}_{>0}$. A set of points occupied by robot i at a configuration $q \in \mathcal{C}_i$ is denoted as $\mathcal{R}_i(q) \subset \mathcal{W}$. The space $\mathcal{W}$ may contain obstacles $\mathcal{O} \subset \mathcal{W}$. An *obstacle space* for robot i is $\mathcal{C}_i^{\text{obs}} = \{q \mid q \in \mathcal{C}_i, \mathcal{R}_i(q) \cap \mathcal{O} \neq \emptyset\}$. A free space for robot i is then $\mathcal{C}_i^{\text{free}} = \mathcal{C}_i \setminus \mathcal{C}_i^{\text{obs}}$. A *trajectory* for robot i is defined by a continuous mapping $\sigma_i : \mathbb{R}_{\geq 0} \mapsto \mathcal{C}_i$.

Definition 3.4 (geometric MRMP instance)**.** *A* geometric MRMP instance *is defined by a tuple* $(\mathcal{W}, A, \mathcal{C}, \mathcal{O}, \mathcal{R}, \mathcal{S}, \mathcal{G})$*, where* $\mathcal{C} = (\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n)$ *and* $\mathcal{R} = (\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n)$*.* $\mathcal{S}$ *is a tuple of initial configurations* $\left(q_1^{init}, q_2^{init}, \ldots, q_n^{init}\right)$*, where* $q_i^{init} \in \mathcal{C}_i$*.* $\mathcal{G}$ *is a tuple of goal regions* $\left(Q_1^{goal}, \ldots, Q_n^{goal}\right)$*, where* $Q_i^{goal} \subseteq \mathcal{C}_i$*.*

Definition 3.5. *Given a geometric MRMP instance, the* geometric MRMP problem *is a decision problem that asks the existence of a tuple of n trajectories* $(\sigma_1, \ldots, \sigma_n)$ *(i.e., solution) and* $t_{end} \in \mathbb{R}_{\geq 0}$ *that satisfies the following conditions:*

- endpoint*:* $\sigma_i(0) = q_i^{init} \wedge \sigma_i(t_{end}) \in Q_i^{goal},\ \forall i \in A$
- obstacle-free*:* $\sigma_i(\tau) \in \mathcal{C}_i^{free},\ \forall i \in A, 0 \leq \tau \leq t_{end}$
- inter-robot collision-free*:* $\mathcal{R}_i\left(\sigma_i(\tau)\right) \cap \mathcal{R}_j\left(\sigma_j(\tau)\right) = \emptyset,\ \forall i, j \in A, i \neq j, 0 \leq \tau \leq t_{end}$

Kinodynamic MRMP

Recall that kinodynamic planning requests a sequence of control inputs that determines a trajectory in the state space.

Similarly to geometric MRMP, kinodynamic MRMP considers navigation for a team of robots A in the workspace $\mathcal{W}$, which may contain obstacles $\mathcal{O}$. A *state space* and *control space* of robot $i \in A$ is respectively provided as $\mathcal{X}_i \in \mathbb{R}^{d_i^{\mathcal{X}}}$ and $\mathcal{U}_i \in \mathbb{R}^{d_i^{\mathcal{U}}}$, where $d_i^{\mathcal{X}}, d_i^{\mathcal{U}} \in \mathbb{N}$. A set of points occupied by robot i at a state $x \in \mathcal{X}_i$ is denoted as $\mathcal{R}_i(x) \subset \mathcal{W}$. Then, an obstacle space $\mathcal{X}_i^{\text{obs}}$ and a free space $\mathcal{X}_i^{\text{free}}$ for robot i are defined in the same way as the geometric case, using $\mathcal{R}_i$. The state transition under kinodynamic constraints of robot i is given by $\dot{x} = f_i(x, u)$, where $x \in \mathcal{X}_i$ and $u \in \mathcal{U}_i$. Given a continuous mapping for control inputs of robot i, denoted by $\xi_i : \mathbb{R}_{\geq 0} \mapsto \mathcal{U}_i$, and an initial state $x_i^{\text{init}} \in \mathcal{X}_i$, a trajectory of the state is successively defined as follows.

$$\sigma_i(0) := x_i^{\text{init}} \tag{3.1}$$

$$\sigma_i(t) := \int_0^t f_i\left(\sigma_i(\tau), \xi_i(\tau)\right)\, d\tau + \sigma_i(0) \tag{3.2}$$

A *kinodynamic MRMP instance* is defined by just replacing configuration spaces with state spaces, and adding a tuple of state transition functions for each robot. A *kinodynamic MRMP problem* is then a decision problem that asks the existence of a tuple of continuous mapping for control inputs, respective for each robot, that satisfies the endpoint condition, the obstacle-free condition, and the inter-robot collision-free condition in Def. 3.5.

Discretizing Time and Local Planner

MRMP is defined in continuous time, however, it is realistic for planning to discretize the time. That is, by introducing $\Delta \in \mathbb{R}_{>0}$ as the small amount of time, MRMP aims at

finding a path of configurations (or states) for each robot, such that any consecutive two points are travelable in the amount of Δ time, without encountering obstacles, without inter-robot collisions, and following kinodynamic constraints. For instance, the state trajectory of Eq. (3.2) is successively represented as follows, given a sequence of control inputs $\xi^0, \xi^1, \ldots$.

$$\sigma_i^0 := x_i^{\text{init}} \tag{3.3}$$

$$\sigma_i^k := \sigma_i^{k-1} + f_i\left(\sigma_i^{k-1}, \xi^{k-1}\right) \cdot \Delta \tag{3.4}$$

For this discretization, in motion planning studies, it is convenient to assume that each robot i has a *local planner* [Choset *et al.*, 2005; LaValle, 2006]. The dissertation denotes it $\texttt{connect}_i$. Given two configurations (or states) $q^{\text{from}}, q^{\text{to}} \in \mathcal{C}_i$, this function returns a unique trajectory σ that satisfies the following three conditions:

- $\sigma(0) = q^{\text{from}} \wedge \sigma(\Delta) = q^{\text{to}}$
- $\sigma(\tau) \in \mathcal{C}_i^{\text{free}}$ for $0 \leq \tau \leq \Delta$
- (in kinodynamic MRMP) σ follows f_i

If no such σ is found, $\texttt{connect}_i$ returns $\perp$. For instance, $\texttt{connect}_i\left(q^{\text{from}}, q^{\text{to}}\right)$ may output $(\Delta-\tau)q^{\text{from}}+\tau q^{\text{to}}$ for geometric MRMP, or Dubins paths [Dubins, 1957] for car-like robots.

Observe that the local planner "hides" primary differences between geometric and kinodynamic MRMP, that is, the control input is now treated as a blackbox. As long as the local planner is definable, there is no strong motivation to distinguish geometric and kinodynamic MRMP. For this reason, the dissertation collectively calls both types of problems *MRMP*.

3.3.2 Computational Complexity

As preliminary knowledge, single-robot motion planning itself is intractable in general [Reif, 1979]. Specifically, the problem is in PSPACE-hard. For further details of complexity results of (single-agent) motion planning, please consult reviews in [LaValle, 2006; Solovey, 2020].

Since motion planning itself is intractable, MRMP is at least as hard as this lower bound. Moreover, MRMP is still tremendously challenging even in simplified situations. The prominent example is the warehouseman's problem, illustrated in Fig. 3.2, posing rearrangements of axis-aligned rectangular boxes. It is shown that the warehouseman's problem is in PSPACE-hard [Hopcroft *et al.*, 1984]. In [Hearn and Demaine, 2005], it is revealed that the warehouseman's problem is PSPACE-hard even for 1×2 rectangles packed in a rectangle. Similarly, the many discs problem is MRMP for disc robots in the plain with polygonal obstacles, which is shown to be in NP-hard [Spirakis and Yap, 1984].

3.3.3 Approaches to MRMP

Compared to MAPF, MRMP is still unmatured. Dominant approaches (e.g., PRM or RRT for motion planning; PP or CBS for MAPF) have not been established as of 2023. The below reviews existing approaches to MRMP.

In principle, SBMP such as PRM [Kavraki *et al.*, 1996] or RRT [LaValle, 1998] is applicable to MRMP by considering one composite robot consisting of all robots [Choset *et al.*, 2005]. Indeed, in [Sánchez and Latombe, 2002], PRM is used to solve MRMP. A few studies consider effective sampling strategies from the composite space. For instance, in [Le and Plaku, 2018; Le and Plaku, 2019], solutions to MAPF are used as a heuristic.

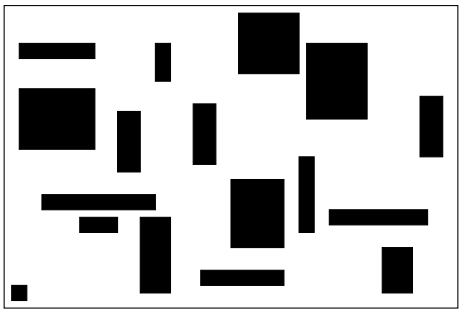


Figure 3.2: Warehouseman's problem. The problem is to seek a finite number of translations for axis-aligned rectangles within a rectangular wokspace.

Despite its applicability, the strategy that samples from the composite space require sampling from the high-dimensional space linear to the number of robots, being a bottleneck even for SBMP [LaValle, 2006]. Consequently, modern MRMP studies mostly take *two-phase planning* that first constructs roadmaps and then performs *multi-agent search*, finding collision-free paths on those roadmaps. In those studies, as the first phase, roadmaps are explicitly prepared via conventional SBMP [Švestka and Overmars, 1998; Gharbi *et al.*, 2009; Wagner *et al.*, 2012b; Solovey *et al.*, 2016; Solis *et al.*, 2021; Dayan *et al.*, 2021] or implicitly embedded as lattice grids [Han *et al.*, 2018; Hönig *et al.*, 2018b; Cohen *et al.*, 2019]. Depending on the heterogeneity, a roadmap is shared among robots, or, robot-wise roadmaps are constructed. The lattice grids are available when the configuration space of each robot is not high-dimensional or state transitions of robots are restricted to a limited number; otherwise, the search space dramatically grows. The second phase uses MAPF algorithms explained in Chap. 3.2.4. Moreover, discretized versions of SBMP are also available [Solovey *et al.*, 2016; Shome *et al.*, 2020], which sample from the composite space and then map the composite configuration to vertices in constructed robot-wise roadmaps in the first phase.

A few studies are beyond this two-phase planning scheme, that is, *simultaneously performing roadmap construction and multi-agent search*. In [Wagner *et al.*, 2012b], RRT is combined with *subdimensional expansion*, which is a concept behind the M* algorithm [Wagner and Choset, 2015] for MAPF. In short, subdimensional expansion aims to gradually couple the search space of multiple robots, which are initially separated. Doing so maintains the search space small. In very recent, tree-based SBMP planners have been used combined with MAPF algorithms, e.g., with PP [Grothe *et al.*, 2022] or with CBS [Kottinger *et al.*, 2022], resulting in promising outcomes.

3.4 Path Planning with Target Assignment

Next, we see another form of multi-agent navigation, i.e., those with target assignment. The above MAPF and MRMP assume that each agent has its own target assigned *a priori*. However, practical applications often do not care about task-executing agents. In other words, goal locations for each agent have room for assignment. We can see such examples in operations of automated warehouses [Wurman *et al.*, 2008], robot soccer [MacAlpine *et al.*, 2015], pattern formation of robot swarms [Turpin *et al.*, 2014], and a robot display [Alonso-Mora *et al.*, 2012], to name just a few. This is where joint optimization of target assignment and collision-free path planning is appealing. The corresponding problem is called a *unlabeled* (or anonymous) version of MAPF/MRMP.

In the rest, this section first formulates unlabled-MAPF (Chap. 3.4.1), and then describes its computational complexities (Chap. 3.4.2), followed by broad reviews of exist-

ing approaches to joint problems (Chap. 3.4.3).

Remarks to Unlabeled-MRMP. This dissertation does not study unlabeled-MRMP. Instead, I put several notes below. Unlike MAPF, MRMP defined in Chap. 3.3.1 assumes that each agent has its respective configuration space. Consequently, it might be impossible to assign goal locations to arbitrary agents. For instance, a goal location laid in a narrow corridor is only reachable by small robots. In this case, a mixed problem of labeled- and unlabeled-MRMP needs to be solved (c.f., Chap. 3.4.3). Meanwhile, unlabeled-MRMP limiting robot shapes to discs have been popularly studied, e.g., [Adler *et al.*, 2015; Solovey and Halperin, 2016].

3.4.1 Problem Formulation

Below, unlabeled-MAPF formalization is presented for undirected graphs. The directed case is similar to the undirected one. Unlabeled-MAPF is defined over a set of *targets*, instead of a goal function of MAPF.

Definition 3.6 (unlabeled-MAPF instance)**.** *An* unlabled-MAPF instance *is defined by an undirected graph $G = (V, E)$, a set of agents $A = \{1, 2, \dots, n\}$, an injective initial state function $s : A \mapsto V$, and a set of target locations $\mathcal{T} = \{g_1, g_2, \dots, g_m\}$, where $g_k \in V$ and $|\mathcal{T}| \leq |A|$.*

The below defines unlabeled-MAPF by the representation by configurations. It is possible to define it by the representation by paths, similar to MAPF.

Definition 3.7 (unlabeled-MAPF problem; representation by configurations)**.** *Given an unlabeled-MAPF instance, an* unlabeled-MAPF problem *asks to find a tuple of configurations $\mathbf{\Pi} = (\mathcal{Q}_0, \mathcal{Q}_1, \dots, \mathcal{Q}_k)$, where Q_t is a configuration, satisfying the following conditions:*

- *$\mathcal{Q}_0$ is the start configuration.*
- *For each $g_l \in \mathcal{T}$, there exists $i \in A$ such that $\mathcal{Q}_k[i] = g_l$.*
- *Any two consecutive configurations in $\mathbf{\Pi}$ are connected.*

For simplicity, the dissertation assumes $|\mathcal{T}| = |A|$ unless explicitly mentioned.

Optimization Problems

The same optimization metrics of MAPF are available to unlabeled-MAPF: *sum-of-costs* (aka. *flowtime*), *makespan*, *sum-of-fuels*, *maximum-moves*, and *sum-of-loss*. See their definitions in Chap. 3.2.1. Similar to MAPF, unlabeled-MAPF has a *Pareto optimal structure* for makespan and sum-of-costs metrics [Yu and LaValle, 2013a]. In other words, there is an instance in which it is impossible to optimize both metrics simultaneously.

3.4.2 Computational Complexity

Unlike conventional MAPF, it is always possible to construct a solution for unlabeled-MAPF in polynomial time when a graph is connected [Kornhauser *et al.*, 1984; Yu and LaValle, 2013a; Ma *et al.*, 2016; Okumura and Défago, 2022b].[2] Furthermore, finding makespan-optimal solutions for unlabeled-MAPF is easier than for MAPF which is known to be NP-hard. Indeed, unlabeled-MAPF has a polynomial-time optimal algorithm based on a reduction to maximum flow [Yu and LaValle, 2013a]. Its detail is available in Appendix C.1. According to the analysis of the paper, a makespan-optimal solution has the makespan of $|A| + |V| - 2$ in the worst case. To my knowledge, other complexity analysis results have not appeared yet, such as flowtime optimization.

[2] This is why Def. 3.7 avoids defining unlabeled-MAPF as a decision problem.

3.4.3 Approaches to Unlabeled-MAPF and Beyond

The unlabeled-MAPF problem consists of two sub-problems: *(i)* target assignment, more generally, task allocation, and *(ii)* collision-free path planning. On one hand, the *multi-robot task allocation* problems are a mature field [Gerkey and Matarić, 2004] with well-known algorithms, such as the Hungarian algorithm [Kuhn, 1955]. On the other hand, path planning for multiple agents, often embodied as MAPF [Stern *et al.*, 2019], has been actively studied since the 2010s. The review below widely explains studies covering both aspects, not limited to unlabeled-MAPF.

Similarly to MAPF algorithms, the existing approaches can be discussed with the categories presented in Table 3.2, that is, *graph pathfinding*, *compiling-based*, *two-level*, *prioritized planning*, and *rule-based* approaches.

For instance, graph pathfinding, compiling-based, and two-phase approaches have been seen in solving the *combined target assignment and pathfinding (TAPF)* problem [Ma and Koenig, 2016] (aka. *colored MAPF* [Barták *et al.*, 2021]). TAPF generalizes both MAPF and unlabeled-MAPF by partitioning the agents into teams. To solve the pathfinding part, the studies for TAPF use existing MAPF algorithms such as M* (graph pathfinding) [Wagner *et al.*, 2012a], answer set programming (compiling-based) [Nguyen *et al.*, 2019], and CBS (two-level) [Ma and Koenig, 2016; Hönig *et al.*, 2018a; Henkel *et al.*, 2019]. The target assignment part mostly uses off-the-shelf target assignment algorithms, or, exploits the makespan-optimal algorithm for unlabeled-MAPF.

The unlabeled version of pebble motion has been studied [Kornhauser *et al.*, 1984; Călinescu *et al.*, 2008; Goraly and Hassin, 2010]. These studies use graph analysis results. Hence, they are categorized into rule-based approaches.

Prioritized planning (PP) is used in [Turpin *et al.*, 2014] to solve unlabeled-MAPF in continuous spaces. The method first solves the lexicographic bottleneck assignment [Burkard and Rendl, 1991; Sokkalingam and Aneja, 1998] and then plans trajectories by PP, together with the delay offset about when agents start moving to avoid collisions. PP has been also used to solve the *multi-agent pickup and delivery (MAPD)* problem [Ma *et al.*, 2017b; Liu *et al.*, 2019], a joint problem of target assignment and path planning that assumes applications of fleet operations in warehouses [Wurman *et al.*, 2008]. In MAPD, agents continuously need to convey packages, issued in an online manner, from their pickup location to their delivery location. The formal definition of MAPD will be provided in Chap. 4.3.2.

3.5 Execution with Uncertainties

Thus far, we have seen what is planning for multiple moving agents. This section shifts the topic to the *execution* perspective, namely, issues raised in plan execution for multiple agents under uncertainties. The section begins with the discussion of *timing uncertainties*.

3.5.1 Timing Uncertainties – What are They?

Recall how to characterize approaches to multi-agent navigation. On one side, there are reactive approaches. On another side, there are deliberative approaches. Below, we first examine deliberative approaches to pose often overlooked problems of execution.

Deliberative approaches consist of preparing a list of collision-free trajectories before robots start moving, and then letting each robot follow the prepared trajectory *precisely* at runtime. The preparation phase is commonly formulated as MAPF (or MRMP). In MAPF, a plan (i.e., a solution of offline planning; paths) specifies the locations that each agent can use and the time at which it can be used. If an MAPF plan is followed precisely, all

robots are guaranteed to reach their destinations because no two robots use overlapping spatiotemporal points.

The main drawback of the above MAPF-based strategy is that robots must precisely follow the planned trajectories both spatially and temporally, despite the existence of reality gaps in real robots. Owing to trajectory tracking being a mature field (e.g., see [Ben-Ari and Mondada, 2017]), it is reasonable to assume the use of trajectory tracking techniques that suppress "spatial" error within a reasonable amount at runtime. However, the "temporal" side is complicated.

In general, plan execution on multiple robots is subject to *timing uncertainties*. Hence, a perfect on-time execution cannot be expected. For instance, robots are often delayed in starting their actions from a prespecified wall-clock time. This is caused by *robot internal factors*, such as kinodynamic constraints, slips, and battery consumption, as well as *distributed environmental factors*, such as communication delays, clock shift/drift, or uncaptured individual differences between robots. More specifically, the latter corresponds to the non-existence of a reliable wall-clock global time that all robots follow because each robot ultimately takes and finishes actions at its own timings independently and unpredictably. Furthermore, when human workers involve in system operations together with mobile robots, as seen in fleet management systems, the time factors become much more unpredictable.

When one robot is delayed from the original plan because of such timing uncertainties, the robot may collide with another robot and crash if they have common regions in their trajectories. Since every motion requires time, it may be impossible to compensate for the delay before bumping into each other. Even worse, such negative interference exponentially increases with the number of agents because the actions of the agents temporally depend on each other.

We, therefore, require a strategy to cope with the timing uncertainties of plan execution on real robots.

3.5.2 Approaches to Timing Uncertainties

Overcoming the issues stemming from timing uncertainties needs both planning and execution perspectives. Existing approaches can be categorized into the following four categories (of course, there are exceptions). As is usual in engineering, all of them have pros and cons.

Deliberative Approaches with Robust Offline Planning

The first approach prepares *robust* offline planning against execution errors, and then performs one-shot execution of the plan, as seen in [Wagner and Choset, 2017; Mansouri *et al.*, 2019; Peltzer *et al.*, 2020; Atzmon *et al.*, 2020a]. This class models timing uncertainties based on the probabilities of travel time or maximum delay assumptions. As long as the system state is within predefined uncertainty models at runtime, these approaches provide safety and liveness. However, once failing to maintain the system status in models owing to black swan events (i.e., events that are unpredictable but bring fatal consequences), the system behavior becomes neither predictable nor controllable.

Deliberative Approaches with Online Intervention

The second approach relies on *online intervention* at runtime that enforces robots to follow the prepared offline planning, as seen in [Čáp *et al.*, 2016; Ma *et al.*, 2017a; Hönig *et al.*, 2019; Atzmon *et al.*, 2020b]. Taking an MAPF plan computed offline as input, at the execution phase, this approach uses a central controller that monitors the status of

all robots in real-time and continuously issues instructions on the manner in which the robots move. However, this approach suffers from the weakness of centralized execution, that is, requiring significant runtime effort. In addition, it requires additional and costly infrastructure, such as steady networks and monitoring systems, to manage all robots' statuses in real-time. The realization of such schemes in large systems is not trivial.

Reactive Centralized Approaches

When real-time intervention is possible by a central component, it is valuable to consider a *reactive* approach, that is, continuously alternating state observation and planning for the next several actions. We can see such styles in [Van Den Berg *et al.*, 2011; Zhou *et al.*, 2017; Senbaslar *et al.*, 2018; Luis *et al.*, 2020; Park *et al.*, 2022], often combined with control theory rather than planning. The primary advantage is the limited effect of timing uncertainties to plan execution because the system can immediately perform online planning, based on rich and no-delay observations. The disadvantages come from the nature of reactive approaches and centralized execution. In other words, ensuring liveness (i.e., each robot eventually reaches its goal) is difficult to realize, moreover, it is challenging to build large systems with this scheme.

Reactive Decentralized Approaches

The fourth and last approach uses entirely decentralized architecture, as seen in [Lalish and Morgansen, 2012; Chen *et al.*, 2017; Tordesillas and How, 2021]. That is, each robot owes its planning and execution. If all robots take a reactive approach that relies only on local interactions (i.e., observation and communication), the negative effect of timing uncertainties is moderately suppressed while achieving excellent scalability. On the other hand, providing some good theoretical properties such as optimality and liveness becomes very challenging.

3.5.3 Against Faults

Timing uncertainties are trouble sources, but they are not the only sources posing execution issues. We also need to care about *robot faults*. Building robust and resilient multi-robot systems against faults is an emerging and important topic [Prorok *et al.*, 2021] since those systems are expected to be infrastructures of logistics or product lines.

Robot faults are not rare and are inevitable in practice due to sensor/motor errors, battery consumption, or other unexpected events. Here, "faults" are beyond delays of robot motions as stated in previous sections. Rather, they are critical events, such as agents forever stopping their motions due to crashes, or, misbehaving from the planning. For instance, according to publicly available data from one warehouse system, the MTBF (mean time between failure) of a robot is 125 days [AutoStore, 2021]. This means that if such a system operates with 125 robots concurrently, we encounter one fault per day on average. Without countermeasures for robot faults, *cascade failures* may be triggered even by one robot fault, and eventually, the entire system may stop its operation. Such disasters actually happened in reality [Financial Times, 2021] and may affect our daily life.

Some early-stage studies against robot faults at runtime can be seen, e.g., target tracking [Zhou *et al.*, 2018], orienteering [Guangyao Shi, 2020], and task assignment [Schwartz and Tokekar, 2020]. In the context of pathfinding, a few studies focus on system designs for potentially non-cooperative agents [Bnaya *et al.*, 2013; Strawn and Ayanian, 2021] where those agents can pretend to be crashed.

3.6 Challenges and Strategies

We are now ready to discuss non-trivial but critical challenges in multi-agent navigation. This section explains what we will see in the rest of the dissertation, again, based on the three perspectives of planning, execution, and representation.

3.6.1 Challenge in Planning

Solving MAPF is the foundation for multi-agent navigation. As explained in Chap. 3.2.4, various algorithms have been proposed for MAPF. These algorithms are exposed to a tradeoff between providing good theoretical properties (e.g., completeness and optimality) and reducing planning efforts (i.e., providing speed and scalability). Therefore, the primary challenge in planning is described as follows.

Challenge in Planning

Develop MAPF algorithms with both good theoretical properties and small planning efforts.

3.6.2 Empirical Results on MAPF Benchmark

To justify this challenge, let me "visualize" the tradeoff. Figure 3.3 is it. This figure shows the number of solved instances (in total 13,900) of representative or state-of-the-art MAPF algorithms, given a time limit up to 30 s. Below, several details for the figure are provided.

Experimental Setup. The instances were retrieved from the MAPF benchmark [Stern *et al.*, 2019]. The benchmark includes 33 four-connected grid maps. For each map, 25 "random" scenarios are available, which is a list of start–goal pairs of agents. The maximum number of agents (i.e., list length) varies between grid maps, up to 1,000 agents. The instances were extracted while increasing the number of agents by 50 up to the maximum. The experiments were run on a desktop PC with Intel Core i7-7820X 3.6 GHz CPU and 32 GB RAM. A maximum of 16 different instances were run in parallel using multi-threading.

Tested Algorithms. Eight algorithms were surveyed, including both representative and state-of-the-art algorithms, covering wide algorithmic properties. The explanation of each algorithm is below.

- **A*** [Hart *et al.*, 1968] as a vanilla search algorithm. It is complete and optimal. The used objective was sum-of-loss.
- **ODrM*** [Wagner and Choset, 2015] as a state-of-the-art optimal and complete algorithm, based on graph pathfinding. The used objective was sum-of-loss. The implementation was obtained from `https://github.com/gswagner/mstar_public`.
- **CBS** [Sharon *et al.*, 2015] with many improvement techniques [Boyarski *et al.*, 2015; Felner *et al.*, 2018; Li *et al.*, 2019a; Li *et al.*, 2019b; Li *et al.*, 2021b; Zhang *et al.*, 2022a] as a state-of-the-art optimal solver. This is representative of two-level approaches using combinatorial search. CBS is solution complete and unable to distinguish unsolvable instances. The used objective was sum-of-costs (aka. flowtime). The implementation was obtained from `https://github.com/Jiaoyang-Li/CBSH2-RTC`.

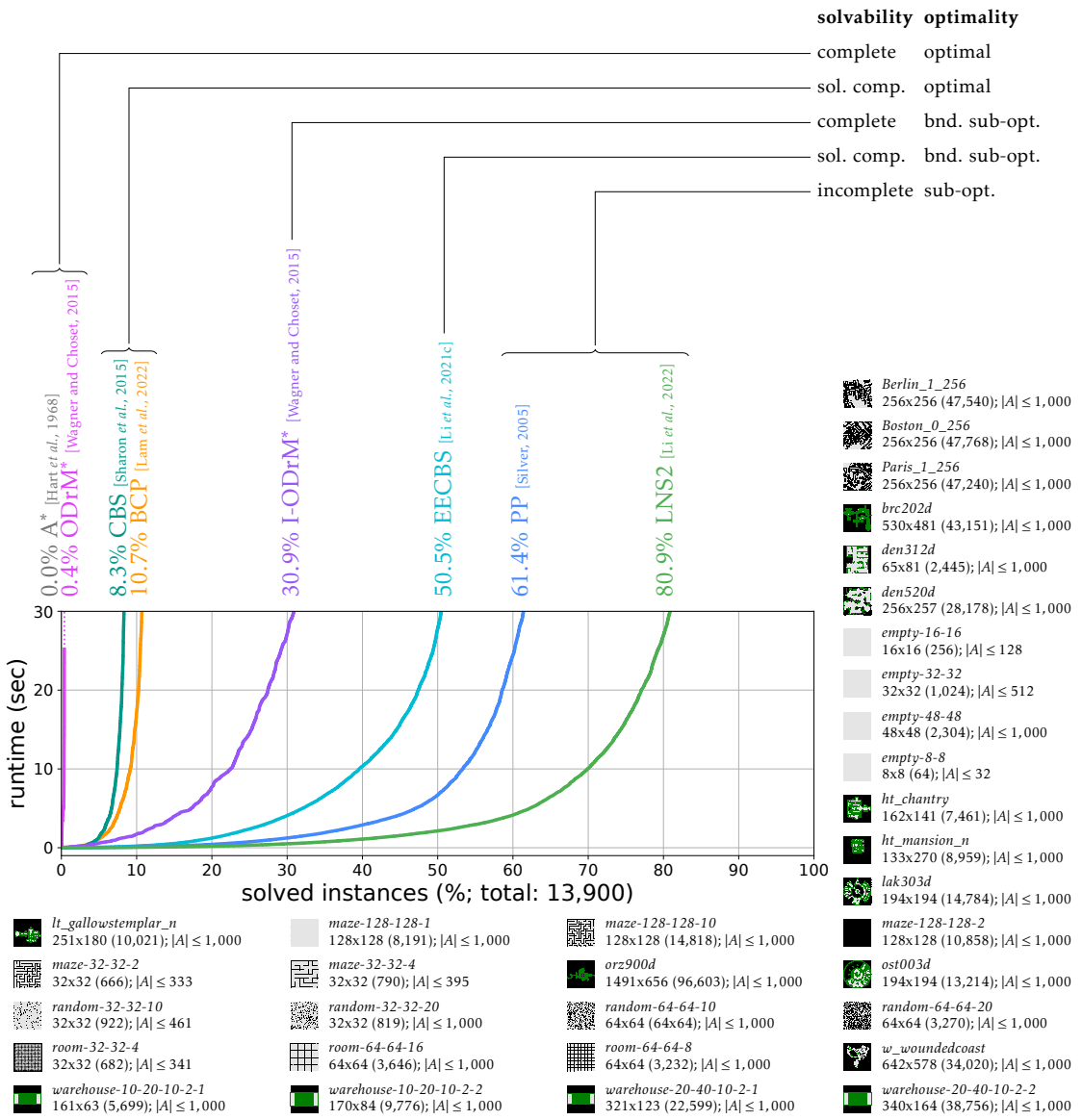


Figure 3.3: Comparison of various algorithms on the MAPF benchmark.

- **BCP** [Lam *et al.*, 2022] as a state-of-the-art optimal solver, which is representative of two-level approaches using reduction to a mathematical optimization problem. BCP is solution complete. For mathematical optimization, CPLEX [Cplex, 2009] was used. The used objective was flowtime. The implementation was obtained from `https://github.com/ed-lam/bcp-mapf`.
- **Inflated ODrM* (I-ODrM*)** [Wagner and Choset, 2015] as a state-of-the-art bounded sub-optimal and complete algorithm. This is a variant of ODrM*. The used objective was sum-of-loss. The sub-optimality was set to five to find solutions as much as possible. The implementation was obtained from `https://github.com/gswagner/mstar_public`.
- **EECBS (EECBS)** [Li *et al.*, 2021c] as a state-of-the-art bounded sub-optimal, but solution complete algorithm. This is a variant of CBS. The used objective was flowtime. The sub-optimality was set to five. The implementation was obtained from `https://github.com/Jiaoyang-Li/EECBS`.

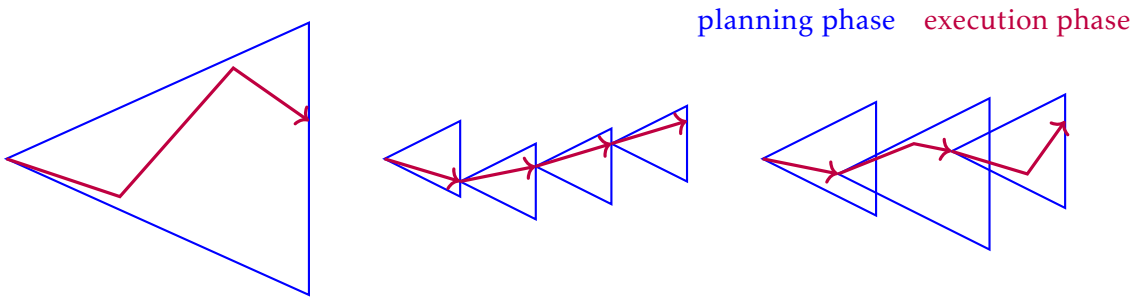


Figure 3.4: Typical relationships between planning and execution. The figure is inspired by [Ghallab *et al.*, 2016].

- **Prioritized Planning (PP)** [Erdmann and Lozano-Perez, 1987; Silver, 2005] as a basic approach for MAPF. This is sub-optimal and incomplete. The implementation used distance heuristics [Van Den Berg and Overmars, 2005] for the planning order and A* for single-agent pathfinding. Furthermore, it involved the repetition of the PP with random priorities until the problem is solved. The implementation was adaptive from `https://github.com/Kei18/pibt2`.
- **MAPF-LNS2 (LNS2)** [Li *et al.*, 2022] as a state-of-the-art sub-optimal and incomplete solver, based on a large neighborhood search. The implementation was obtained from `https://github.com/Jiaoyang-Li/MAPF-LNS2`.

Note that due to implementation issues, two objectives are mixed: sum-of-loss and flowtime, owing to following the original implementations. The dissertation will later complement more detailed results that assess various optimization metrics.

Observations. Figure 3.3 well captures the tradeoff between good theoretical properties and planning efforts. For instance, the complete and optimal solver (ODrM*) solved only 0.4% of the instances. When sacrificing completeness, the optimal state-of-the-art algorithms (CBS and BCP) can solve around 10%. On the other side, a sub-optimal and incomplete algorithm (LNS2), no longer with any good theoretical properties, solves around 80%. However, it still remains unsolved in around 20% of instances.

Can we break this tradeoff? The next outlines the strategies of this dissertation.

3.6.3 Strategy in Planning – Short-Horizon Planning Guides Long-Horizon Planning

Let's rethink the relationship between deliberative and reactive approaches. Figure 3.4 visualizes the deliberative approaches (left) and reactive approaches (middle), as well as hybrid approaches (right). *Understanding this figure is very important to understand the rest of the dissertation.* For each part, the execution phase is represented by red arrows, while the planning phase is illustrated by a blue triangle, assuming that the time flows from left to right. Horizontal lengths of triangles are *planning horizon*, i.e., the amount of time that planning considers. The area of the triangle represents a *search space* for planning, i.e., a set of states that agents can take. The search space roughly approximates the *planning effort*, and is dominantly determined by a planning horizon; a longer horizon incurs a larger search space, and vice versa.

The crucial observation is that deliberative approaches have typically a larger search space, while reactive approaches have a smaller search space.[3] A large search space in-

[3]Of course, it is possible to consider reactive approaches with long planning horizons; however, this is typically *not* suitable for reactive approaches due to their planning speed.

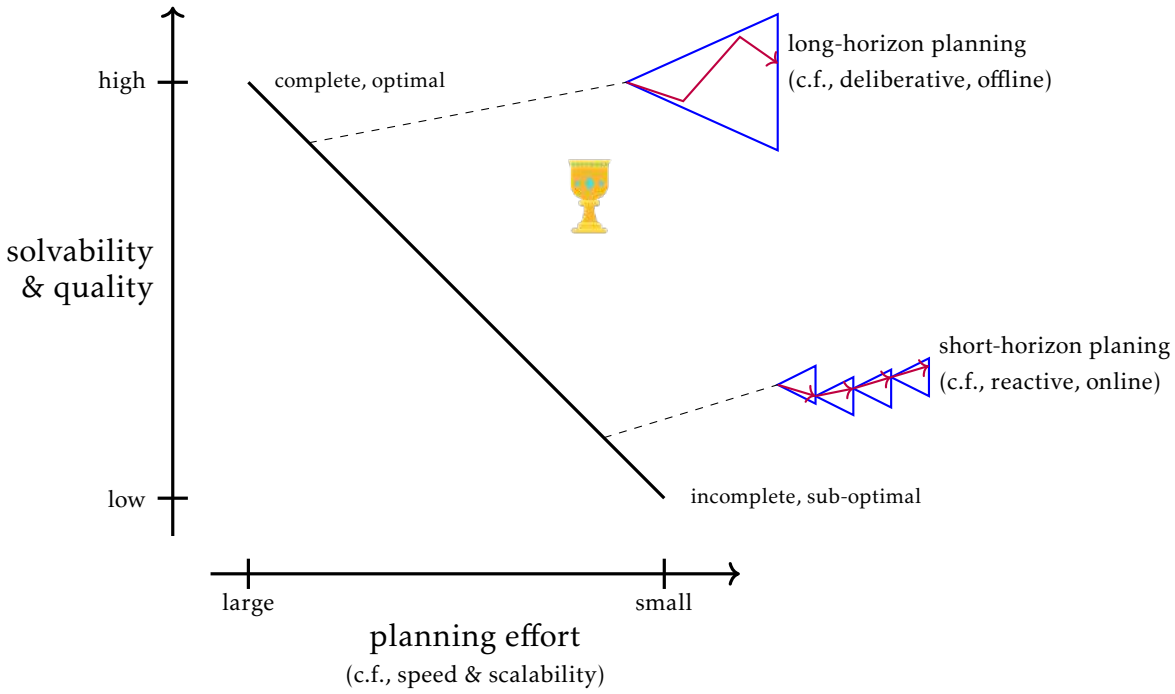


Figure 3.5: Tradeoff in planning.

curs a large planning effort. Therefore, *short-horizon planning* (i.e., planning with short planning horizon) requires less planning effort than *long-horizon planning* (i.e., planning with long planning horizon). In contrast, a large search space enables planning to search for better possibilities. Consequently, long-horizon planning can be attached with good theoretical properties, such as completeness and optimality. This causes the aforementioned tradeoff, as summarized in Fig. 3.5. This is the very basic nature of planning. Ideally, we want to establish methodologies located on the upper right, which is the holy grail of algorithm design.

Basic lessons from the above discussions are as follows.

- *To design complete and optimal algorithms, the long-horizon nature is mandatory* because otherwise the search space will not be entirely explored.
- *To design quick and scalable algorithms, the short-horizon nature is convenient.* Another possible approach is relying on good heuristics; however, it is generally difficult to design such heuristics for multi-agent environments because they must take into account interactions between agents.

According to the lessons, *the best strategy to break the tradeoff seems to integrate long- and short-horizon planning approaches.* This will be a planning style such that *(reactive) short-horizon planning guides (deliberative) long-horizon planning*, as illustrated in Fig. 3.6. Observe first that Fig. 3.6 is significantly different from the hybrid of deliberative and reactive approaches illustrated in Fig. 3.4(right). In the new integrated one, planning and execution do not alternate. Rather, this is likely close to conventional *heuristic search* that uses heuristics, which are estimations of cost-to-go. However, beyond just estimations, the integrated one actually generates successors.

The above concept is the crux of *lazy constraints addition search for MAPF (LaCAM)*, one of the main contributions in Part I that studies planning. LaCAM is described in Chap. 6, together with the eventually optimal version called LaCAM*. Before that, two examples of short-horizon planning are introduced. The first is *priority inheritance with*

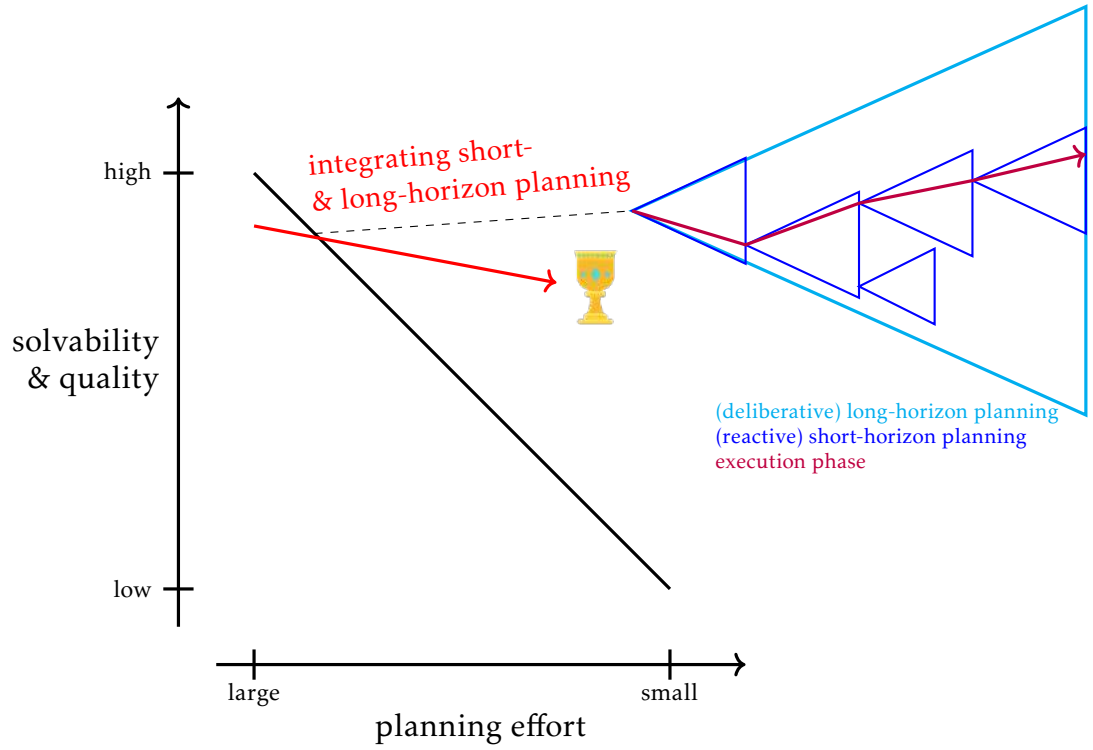


Figure 3.6: Strategy to break the tradeoff in planning.

backtracking (PIBT), suitable for solving MAPF iteratively, presented in Chap. 4. The second is *TSWAP*, a complete algorithm for unlabeled-MAPF, presented in Chap. 5. Both PIBT and TSWAP themselves have good theoretical properties. Not only that, both algorithms can be enhanced by LaCAM$^{(*)}$, providing completeness and the guarantee of convergence to optima. By collectively seeing them, we will see the power of the above concept, that is, successfully breaking the tradeoff!

In short, the strategy in planning is as follows.

Strategy in Planning

Establish a new planning style such that short-horizon planning guides long-horizon planning.

3.6.4 Yet Another Strategy in Planning – Iterative Refinement

As a general thing in optimization problems, once a feasible solution is found, it is not so difficult to improve its quality by *local search* (aka. *iterative refinement*). For instance, a traveling salesman problem (TSP) is a celebrated example of an optimization problem, having such many algorithms to improve solution quality [Rego *et al.*, 2011]. Similar to TSP, it is promising if we can improve the solution quality in MAPF. Though limited to the solution quality aspect, with such methodologies, it is possible to break the tradeoff in planning. This is illustrated in Fig. 3.7.

Therefore, another strategy in planning is as follows, which is studied in Chap. 7, at the end of Part I.

Another Strategy in Planning

Establish iterative refinement methods for MAPF solutions.

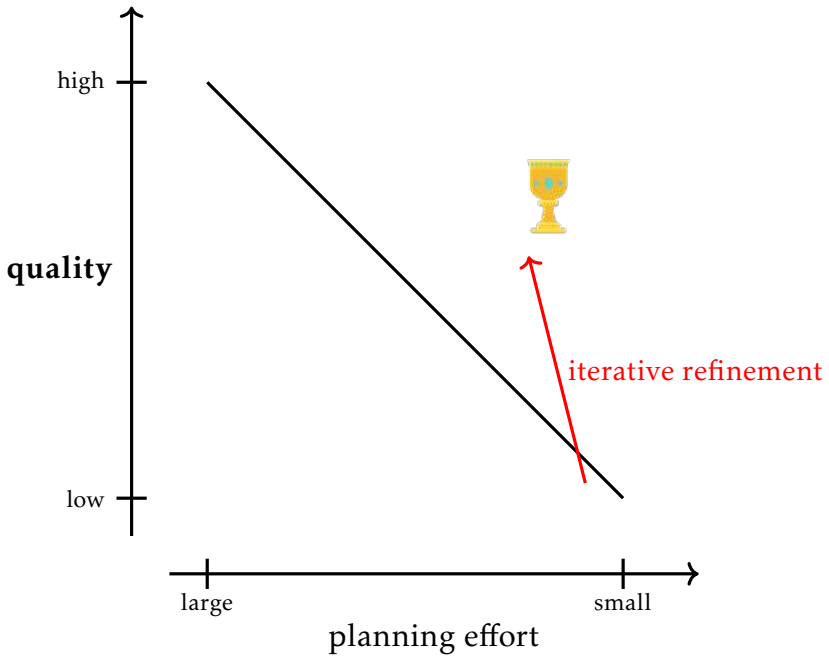


Figure 3.7: Another strategy to break the tradeoff in planning.

3.6.5 Challenge and Strategy in Execution

This primary challenge in execution is simply stated as follows.

Challenge in Execution

Develop scalable methods for execution with good theoretical properties, despite uncertainties in the real world.

How do we tackle this challenge? Recall that Chap. 3.1 explains the two axes to characterize multi-agent navigation: "deliberative vs. reactive" axis, and "centralized vs. decentralized" axis. Here, we focus on the "centralized vs. decentralized" axis.

As stated earlier, it is very important to distinguish planning and execution perspectives in "centralized vs. decentralized" axis. Figure 3.8 summarizes this relationships. In general, only centralized planning can "easily" provide good theoretical properties such as completeness. Meanwhile, *centralized planning itself does not disturb scalability*, contrary to popular belief.[4] This is because centralized planning is mostly able to "emulate" decentralized planning. Rather, the issue of scalability is linked to whether an approach takes centralized or decentralized execution. When the approach takes decentralized execution, this is where decentralized planning achieves excellent scalability. Therefore, the blue line in Fig. 3.8 is not straight, rather, it turns in the middle.

The above discussion provides basic lessons in execution, similar to those of planning with Fig. 3.5.

- *To design scalable methods for execution, some flavor of decentralized execution is mandatory*. Otherwise, a centralized controller needs to care of all agents' statuses in real-time, which will be impossible at some scale.
- *To provide good theoretical properties for execution, centralized planning is convenient*. Another possibility is attaching such properties to decentralized planning, how-

[4]Indeed, we will see excellent scalability of centralized planning in Part I. Centralized path planning for 10,000 agents? Not problematic at all.

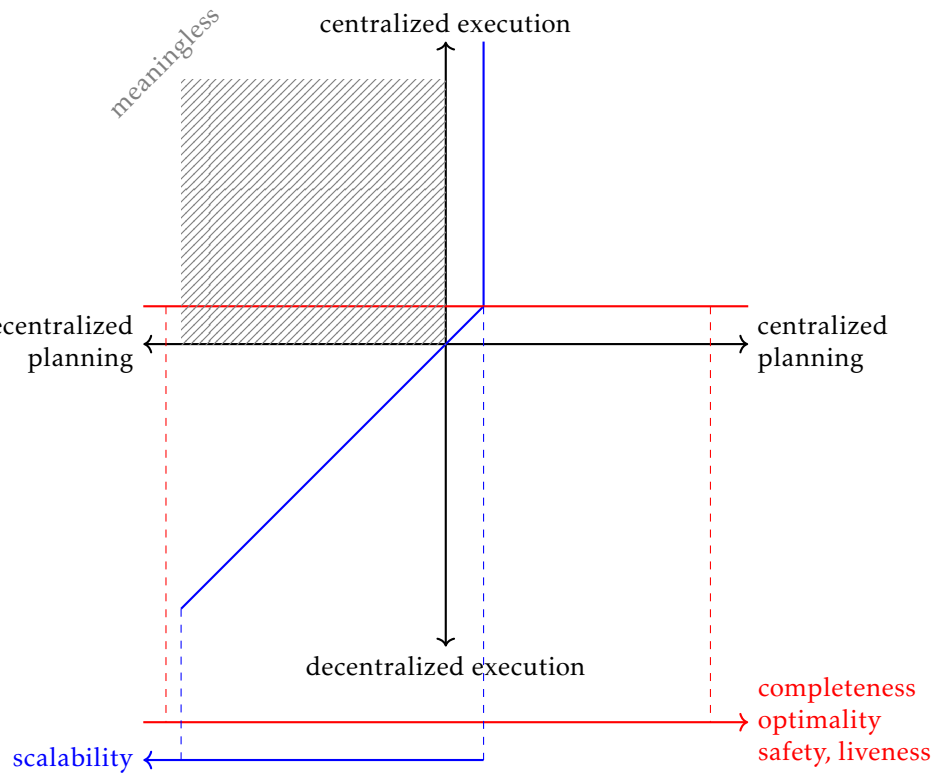


Figure 3.8: Centralized vs. decentralized perspective.

ever, this way requires formal and rigorous proof of domain-dependent specifications; ironically, it is not a "scalable" direction in the sense of the domain-independence of this dissertation goal.

According to the lessons, *the best strategy to achieve the challenge seems to use centralized planning and decentralized execution*. However, this style has already been discussed as "Deliberative Approaches with Robust Offline Planning" in Chap. 3.5.2. As a disadvantage, this style needs uncertainties models. Once the system behavior is outside the models, which is not unrealistic at all, the system behavior is no longer predictable.

To this end, the dissertation explores a novel relationship between planning and execution, that is, *centralized deliberative planning assuming that agents "reactively" execute the plan at runtime*. The keyword is "reactively," meaning that agents react to runtime situations and change/update their executing plan on the fly. Here, offline planning provides multiple options, and agents choose one of them during execution. By considering such styles, there is a possibility to overcome issues of getting off the rails of the uncertainty models, still ensuring some good properties. This concept is illustrated in Fig. 3.9.

The challenge is, "is that possible?" In other words, how do we design centralized planning, with provably good properties, and with room for reactive execution? This is significantly different from strategy in planning, in which short-horizon planning guides long-horizon planning. Rather, the role is reversed, summarized as follows.

Strategy in Execution

Establish execution methods such that offline planning guides reactive execution.

The above concept is embodied in Part II that studies the execution perspective. Specifically, Chap. 9 exemplifies it for timing uncertainties and Chap. 10 exemplifies it for crash faults. Before presenting the two studies, as a "hop" phase, we first see a

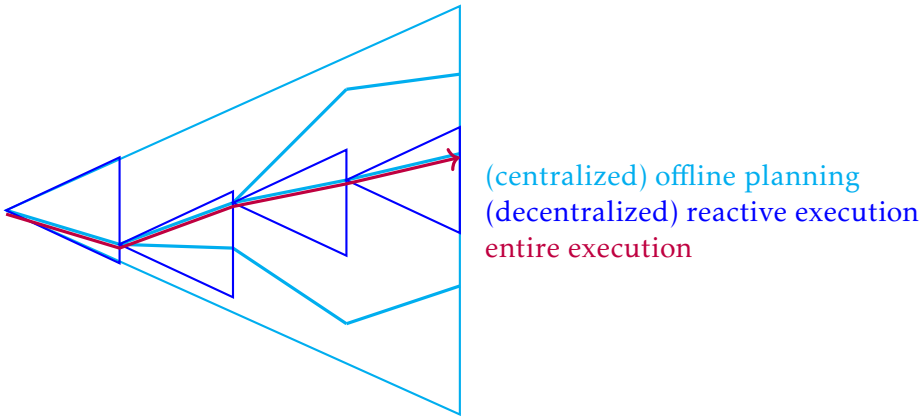


Figure 3.9: Strategy in execution.

reactive centralized approach in Chap. 8. The chapter presents an algorithm tolerant to timing uncertainties, namely, an extension of TSWAP for unlabeled-MAPF; originally presented in Chap. 5. After that, we see the two examples. The first studies a novel problem called *offline time-independent multi-agent path planning (OTIMAPP)*. The second studies a novel problem called *fault-tolerant multi-agent path planning (MAPPCF; with crashes)*. Through these studies, the dissertation provides a direction of (potentially) scalable execution methods having good theoretical properties.

3.6.6 Challenge and Strategy in Representation

Thinking of planning in continuous spaces (aka. motion planning; MRMP with multiple agents) has two benefits over planning in discretized spaces wherein the search space is defined *a priori*, namely, *improving solvability* and *improving solution quality*. Figure 3.10 illustrates these merits.

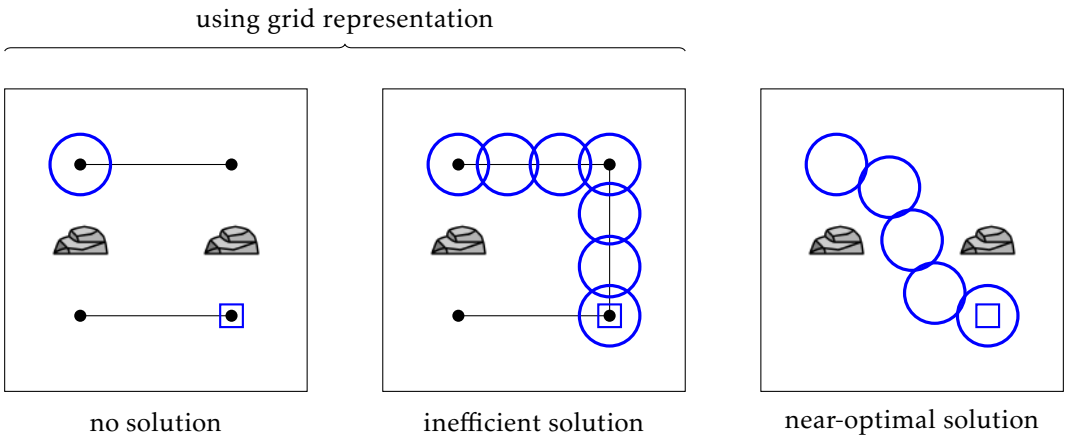


Figure 3.10: Motivations to consider path planning in continuous spaces. A robot and its goal are depicted by blue-edged circles and squares, respectively.

The observations of Fig. 3.10 ask what are good environmental representations, which is a central question of Part III. *To do planning, discretization of the workspace is (almost) mandatory; however, poorly thought-out discretizations cause the problems mentioned above.* In the representation part, we will dive into how to construct an effective representation for planning. Specifically, it takes the form of a *roadmap*, a graph representation that approximates the free space for robots (see Chap. 2.6).

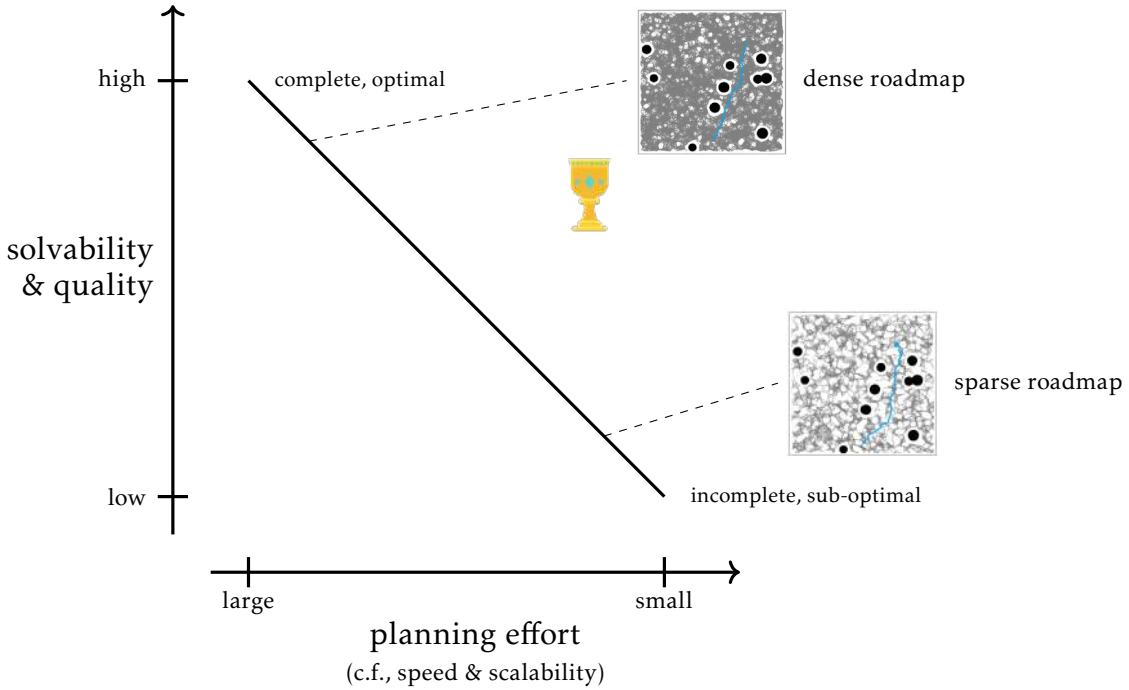


Figure 3.11: Tradeoff in representation. Roadmaps are constructed by PRM [Kavraki *et al.*, 1996]. The blue lines in the roadmaps are solution trajectories.

Note, there are planning methodologies without roadmaps, e.g., using artificial potential fields [Hwang *et al.*, 1992]. However, such approaches suffer from planning in high-dimensional spaces. Similarly, rule-based roadmap construction methods (e.g., cell decomposition [Preparata and Shamos, 1985], lattice grids [Pivtoraiko *et al.*, 2009], or visibility graphs [Lozano-Pérez and Wesley, 1979; Latombe, 1991]) also suffer from high-dimensional spaces. Consequently, *sampling-based motion planning (SBMP)* will be the core to realize domain-independent planning, i.e., a planning style applicable to various domains not limited to specific robot systems. The remaining part of the dissertation, therefore, regards SBMP as central to solving MRMP.

As nature in engineering like we encountered in planning and execution, representation has also a tradeoff. Figure 3.11 visualizes the tradeoff. This time, the effect appears in planning, and again posing the tradeoff between quality (and solvability) vs. planning efforts. The control factor is *density* of the roadmap. For instance, dense roadmaps have a high chance to obtain plausible solutions, meanwhile, such roadmaps incur large planning efforts since the resulting search space is huge. Sparse roadmaps have reverse effects; on one hand, planning efforts are low since the search space is not so huge. On the other hand, it is unlikely to obtain good-enough solutions with the sparse roadmaps.

We, therefore, need to break this tradeoff.

Challenge in Representation

Develop methods to construct small but promising representations for MRMP.

The strategy to tackle this challenge is straightforward.

Strategy in Representation

Establish methods to construct roadmaps only in *important regions* for each agent.

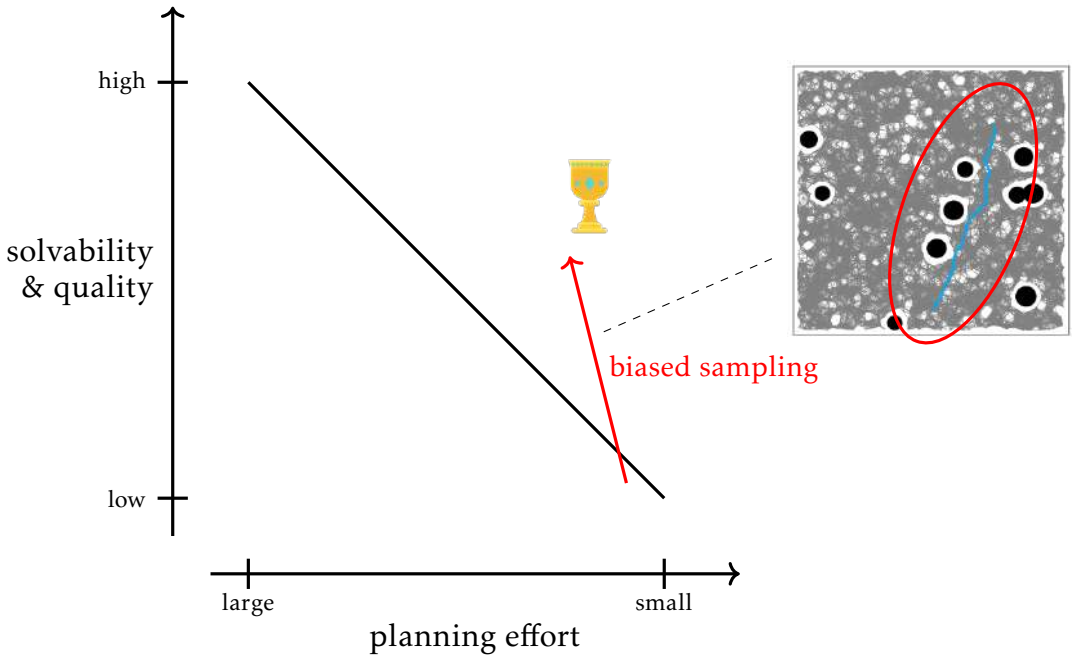


Figure 3.12: Strategy to break the tradeoff in representation.

Doing so keeps the search space small, enabling planners to derive solutions for MRMP quickly. Figure 3.12 annotates the important region for the agent. Intuitively, the roadmap outside of the red-edged ellipse is redundant for planning. We can cut off such regions. Even so, the roadmap still contains plausible solutions. This strategy is related to *biased sampling* of SBMP, which has already been successful in the literature of (single-agent) motion planning, e.g., [Gammell *et al.*, 2014; Gammell *et al.*, 2015].

The remaining issue is how to design a "good bias" that identifies important regions for respective agents, while considering *interactions between agents* because we need to avoid inter-agent collisions. Instead of developing "manual" biases, Chap. 11 presents one approach via *machine learning (ML)* to realize biased sampling methods for MRMP, namely, learning features of good bias from planning demonstrations. The resulting roadmaps are called *cooperative timed roadmaps (CTRM)*. This ML-based approach is influenced by recent successful results of leveraging ML to single-agent motion planning [Ichter *et al.*, 2018; Qureshi *et al.*, 2020; Ichter *et al.*, 2020; Chen *et al.*, 2020; Zhang *et al.*, 2020].

3.6.7 Yet Another Strategy in Representation – Combining Sampling and Search

The previous strategy in representation implicitly assumes *two-phase planning* that decouples roadmap construction and multi-agent search. Then, it tries to develop effective roadmap construction methods. However, *with two-phase planning, it is inevitable to develop search spaces that might not be used by multi-agent search*. Consequently, it is possible to develop large search spaces, even with well-designed biases.

After all, to mitigate the tradeoff in representation, *coupled approaches of roadmap construction and multi-agent search are necessary*. Such approaches perform roadmap construction and multi-agent search simultaneously. Doing so makes it possible to develop a small but effective search space such that the multi-agent search willing to use it. In short, the dissertation stands also on the below concept, which is also visualized in Fig. 3.13.

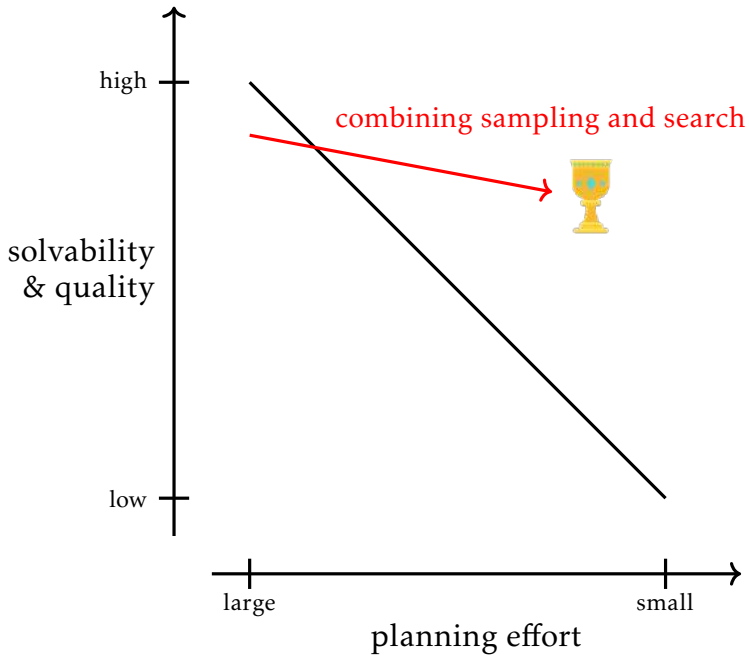


Figure 3.13: Another strategy to break the tradeoff in representation.

Another Strategy in Representation

Establish MRMP algorithms that combine sampling and search.

Chapter 12 presents a proof-of-concept algorithm called *simultaneous sampling-and-search planning (SSSP)* to solve MRMP. The main idea behind the algorithm is to unite techniques developed for SBMP and search techniques for MAPF. Specifically, the SSSP algorithm is directly inspired by two algorithms, respectively for SBMP [Hsu *et al.*, 1997; Hsu, 2000] and MAPF [Standley, 2010]. We will see the power of combining sampling and search; SSSP significantly outperforms two-phase planning. Furthermore, this is indeed an example of the integration of representation and planning.

3.7 Relationships of Chapters

Figure 3.14 illustrates the relationships of the remaining chapters. We will see the planning, execution, and representation perspectives in order.

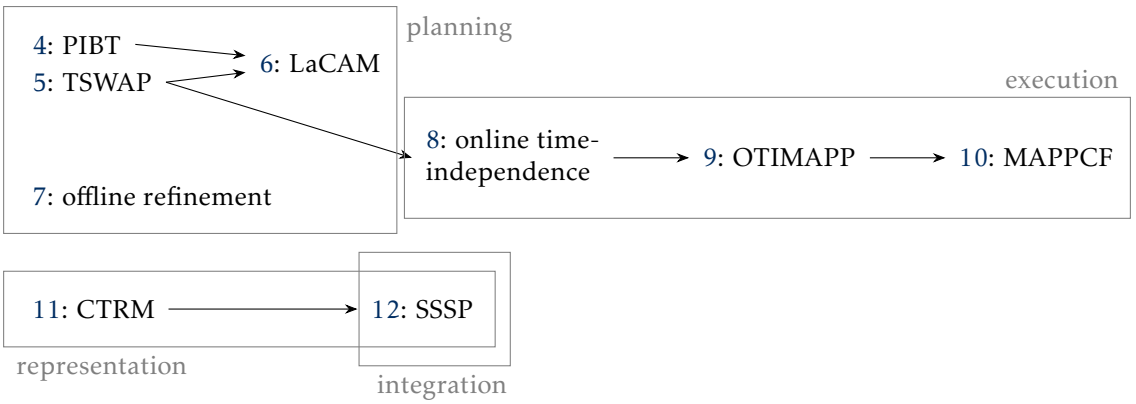


Figure 3.14: Structure of the dissertation.

Part I

Planning

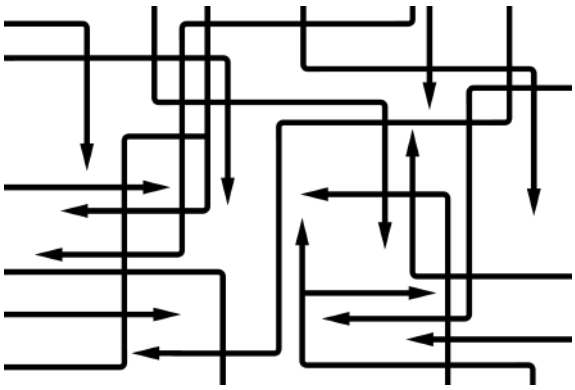


Image by Gerd Altmann / Pixabay License

Chapter 4

Short-Horizon Planning for MAPF

In this chapter, we see an example of short-horizon planning for MAPF called the *priority inheritance with backtracking (PIBT)* algorithm, originally aimed at solving MAPF *iteratively*. PIBT initially appeared in [Okumura *et al.*, 2019], later journalized in [Okumura *et al.*, 2022b] on which this chapter is based.[1] The chapter begins with a chapter overview.

4.1 Chapter Overview

The objective of the chapter is to see the power of short-horizon planning for MAPF, embodied as PIBT, despite lacking long-horizon deliberation.

4.1.1 What is PIBT

Recall that a *configuration* denotes a tuple of locations for all agents (see Chap. 3.2.1). *PIBT is understood as a scheme that generates a new configuration, given another configuration as input.* By continuously generating configurations, PIBT yields a sequence of configurations (i.e., solution for MAPF). Meanwhile, this configuration generator is applicable to other variants of conventional MAPF. As evidence, PIBT will apply to a lifelong version of MAPF called Multi-Agent Pickup and Delivery (MAPD) [Ma *et al.*, 2017b].

Mechanism

The mechanism of PIBT is simple and easy to implement. The algorithm focuses on the adjacent movements of multiple agents based on prioritized planning [Erdmann and Lozano-Perez, 1987] in a unit-length time window. Priority inheritance is a well-known approach for dealing effectively with priority inversion in real-time systems [Sha *et al.*, 1990] and is applied here to path adjustment. When a low-priority agent-X impedes the movement of a higher-priority agent-Y, agent-X temporarily inherits the higher-priority of agent-Y. To avoid agents getting stuck waiting, priority inheritance is executed in combination with a backtracking protocol. Because this mechanism only requires local interactions between agents, PIBT has a high potential for decentralized implementations.

4.1.2 Properties and Performance

Theoretical Properties

Combined with a dynamic priority assignment, PIBT ensures *reachability*, that is, every agent always reaches its destination within a finite time when the environment is a graph

[1] In the past, I also considered a "long-horizon" version of PIBT [Okumura *et al.*, 2020]; however, I concluded that it was not a good idea.

such that all pairs of adjacent vertices belong to a simple cycle (e.g., biconnected graphs). Reachability does not ensure that all agents are on their goals simultaneously, hence, *PIBT is incomplete for conventional MAPF*. However, agents often do not need to stay at their goals in real-life applications such as lifelong delivery tasks. Reachability is convenient in such scenarios. Indeed, the chapter introduces the application of PIBT to a lifelong variant of MAPF (i.e., MAPD) while ensuring completeness.

PIBT also has low-cost time complexity. It repeats one-timestep planning (i.e., configuration generation) until termination (e.g., until all agents have reached their destinations). For each timestep, PIBT requires the time complexity $O(A \cdot (\Delta(G) + F + \lg A))$, where A is a set of agents,[2] $\Delta(G)$ is a maximum degree of a graph G, and F is the time required to sort candidate locations for the next timestep, which can be $O(\Delta(G) \lg \Delta(G))$ with preprocessing. Consequently, PIBT is expected to work in a short runtime, even in the case of massive problems including thousands of agents.

Empirical Performance

For (one-shot) MAPF, PIBT immediately returns plausible solutions despite hundreds or more agents in running times that are orders of magnitudes faster than other established MAPF approaches, such as sub-optimal priority-based (PP) [Silver, 2005], rule-based [Luna and Bekris, 2011], and search-based [Li *et al.*, 2021c] algorithms, as well as state-of-the-art optimal two-level approaches of search-based [Sharon *et al.*, 2015] and compiling-based [Lam *et al.*, 2022] algorithms. For instance, using an ordinary laptop, we will observe that PIBT solves MAPF in a large grid map (530×481, number of vertices: 43,151) from [Stern *et al.*, 2019] with 1,000 agents in at most 5 s, while keeping sub-optimality below 1.5 on average. Furthermore, PIBT outperforms an existing approach for both runtime and solution quality on MAPD.

4.1.3 Original Motivation for PIBT

It is worth mentioning the original motivation to develop PIBT as a history.

Prior research on MAPF algorithms primarily focuses on solving a "one-shot" version of the problem, where agents reach their respective goals from their initial positions only once. In contrast, in practical applications such as conveying packages in a warehouse with hundreds or more robots [Wurman *et al.*, 2008], MAPF must be solved *online* and *iteratively* on a *lifelong* basis. That is, whenever an agent reaches a goal, it receives a new one, and the plan, a list of paths, is updated in real-time. Such scenarios rule out any simple adaptations of offline and compute-intensive optimal approaches because, despite state-of-the-art optimal algorithms [Lam *et al.*, 2022; Li *et al.*, 2021b], it is challenging to find solutions for a few hundred agents within a realistic timeframe. For this purpose, an attractive possibility is to design scalable sub-optimal algorithms that output plausible solutions within a predictable computational time, which also works in iterative situations.

The above motivation fruits as the PIBT algorithm. However, *in Chap. 6, PIBT will be treated as a configuration generator, completely moving on to another direction from the original motivation that focuses on online and lifelong scenarios.*

4.1.4 Chapter Organization

The rest of the chapter is organized as follows.

- Chapter 4.2 presents the PIBT algorithm along with theoretical analyses.

[2] $O(A)$ is used as a simplified notation instead of $O(|A|)$, following Chap. 2.1.

- Chapter 4.3 considers applying PIBT to specific problems such as MAPF and MAPD, as well as decentralization and an MAPF model without rotations.
- Chapter 4.4 empirically evaluates PIBT and presents a demonstration of real robots.
- Chapter 4.5 reviews existing MAPF algorithms focusing on those closely related to PIBT.
- Chapter 4.6 concludes the chapter.

The code and movie are available at `https://kei18.github.io/pibt2`.

4.1.5 Notations and Assumptions

Notations in this chapter are summarized below:

$\bot$	not found, undefined
$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \ldots, n\}$	a tuple of agents
$(s_1, s_2, \ldots, s_n)$	start configuration, where $s_i \in V$
$(g_1, g_2, \ldots, g_n)$	goal configuration, where $g_i \in V$
`dist`	$V \times V \mapsto \mathbb{N}$, function that returns shortest path length
Q	configuration, a tuple of locations for all agents

Caution

The chapter uses the representation by configurations.

4.2 The PIBT Algorithm

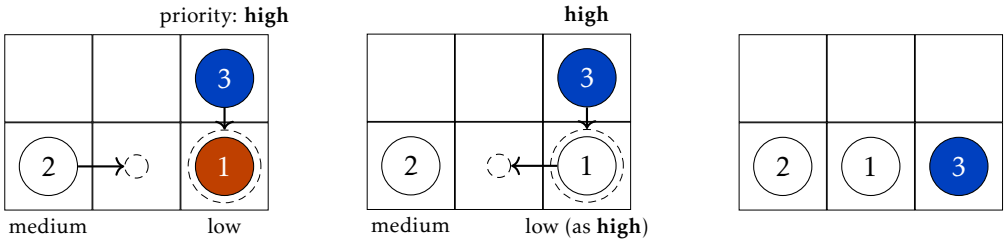
This section introduces the PIBT algorithm, including theoretical analyses such as reachability. The concept of PIBT is initially explained by intuitive examples. To avoid unnecessarily complex explanations, PIBT is introduced as a centralized algorithm, focusing on analyzing the prioritization scheme itself. Note that PIBT relies on a decoupled approach, rendering it readily amenable to decentralization, as briefly discussed later.

4.2.1 Concept

PIBT repeats one-timestep prioritized planning until it is terminated. At every timestep, each agent first updates its unique priority. Then, the agents sequentially determine their next location in decreasing order of priorities while avoiding vertices that have been requested by higher-priority agents. Prioritization alone, however, can still cause deadlocks. As shown in Fig. 4.1a, a stuck agent (agent-1) cannot go anywhere without collisions with other agents; hence, this situation can be regarded as a certain kind of deadlock.

Priority Inheritance

A situation with a stuck agent can also be regarded as a case of *priority inversion*, in the sense that a low-priority agent (agent-1) holding a resource claimed by a higher-priority agent (agent-3) fails to obtain a second resource held by an agent with medium priority agent-2. A classical way to deal with priority inversion is to rely on *priority inheritance* [Sha *et al.*, 1990] (Fig. 4.1b). The rationale is that a low-priority agent (agent-1) temporarily inherits the higher priority of the agent (agent-3) claiming the resources it holds, thus forcing medium-priority agents (agent-2) out of the way (Fig. 4.1c).

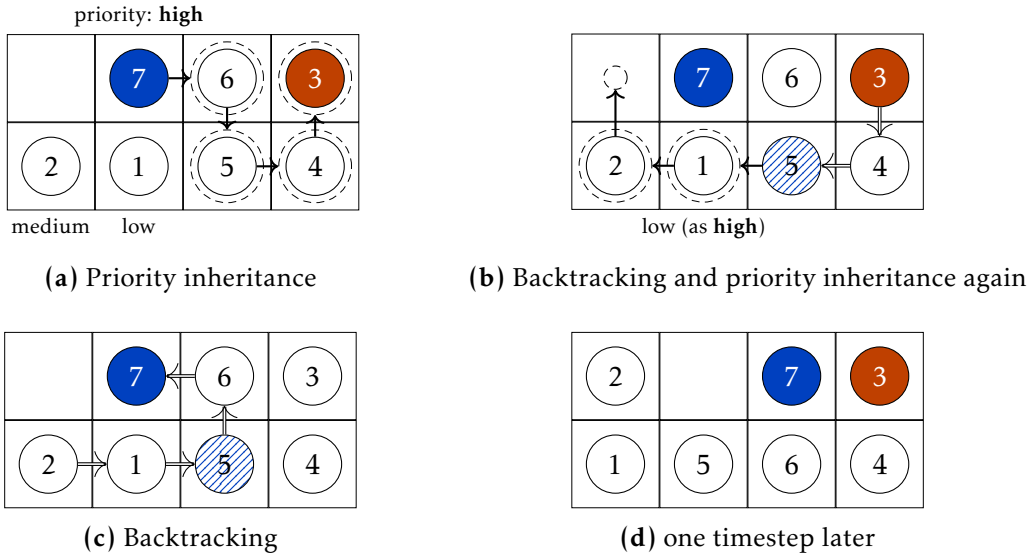


(a) Stuck agent **(b)** Priority inheritance **(c)** One timestep later

Figure 4.1: Examples of priority inheritance. Circles with solid lines represent agents. Requests for the next timestep are depicted by dashed circles, determined greedily according to agents' destinations (omitted here). Without inheritance (4.1a), a stuck agent (agent-1) cannot give way to a high-priority agent (agent-3) without risking collision into a third agent (agent-2). With priority inheritance (4.1b), agent-1 temporarily inherits the priority of agent-3 forcing agent-2 to solve the situation (4.1c).

Backtracking

Priority inheritance deals effectively with priority inversion; however, it does not completely ensure deadlock freedom. For instance, as shown in Fig. 4.2a, agent-3 cannot escape as a result of consecutive priority inheritances ($7 \to 6 \to 5 \to 4$).



(a) Priority inheritance **(b)** Backtracking and priority inheritance again

(c) Backtracking **(d)** one timestep later

Figure 4.2: Example of PIBT. The flow of backtracking is indicated by double-lined arrows. Because agent-3 is stuck (4.2a), backtracking returns as invalid to agent-4, and subsequently to agent-5. Agent-5 executes other priority inheritance to agent-1 (4.2b). Agents 1, 5, 6, and 7 wait for the results of backtracking (4.2c) and then start moving (4.2d).

The solution relies on *backtracking*; any agent i that executes priority inheritance must wait for an outcome, `VALID` or `INVALID`. If `VALID`, agent-i successfully moves to the desired vertex. Otherwise, it must request a different vertex, excluding: *(i)* vertices requested by a higher priority agent, and *(ii)* vertices having already returned an `INVALID` outcome. Upon finding no `VALID` or unoccupied vertices, agent-i sends back an `INVALID` outcome to the agent from which it inherited its priority.

For instance, in Fig. 4.2b, agent-3 sends `INVALID` back to agent-4 as the outcome of the request received previously from agent-4 (Fig. 4.2a). Agent-4 tries to replan its next

location; however, in the absence of other alternatives, in turn sends `INVALID` to agent-5. Because agent-1 has lower priority, agent-5 can let agent-1 inherit its priority as an alternative, which leads to the outcome of `VALID` (Fig. 4.2c), and agents can move (Fig. 4.2d).

By combining priority inheritance and backtracking, it is ensured that all agents are assigned to the next locations without collisions.

4.2.2 Minimum Implementation

The aforementioned concept is formalized by *recursive structure*, as presented in Alg. 4.1. The pseudocode presents a minimum version of PIBT. It takes one configuration ($\mathcal{Q}^{\text{from}}$) and then generates a new configuration ($\mathcal{Q}^{\text{to}}$).

Algorithm 4.1 Minimum PIBT

input: configuration $\mathcal{Q}^{\text{from}}$, agents A

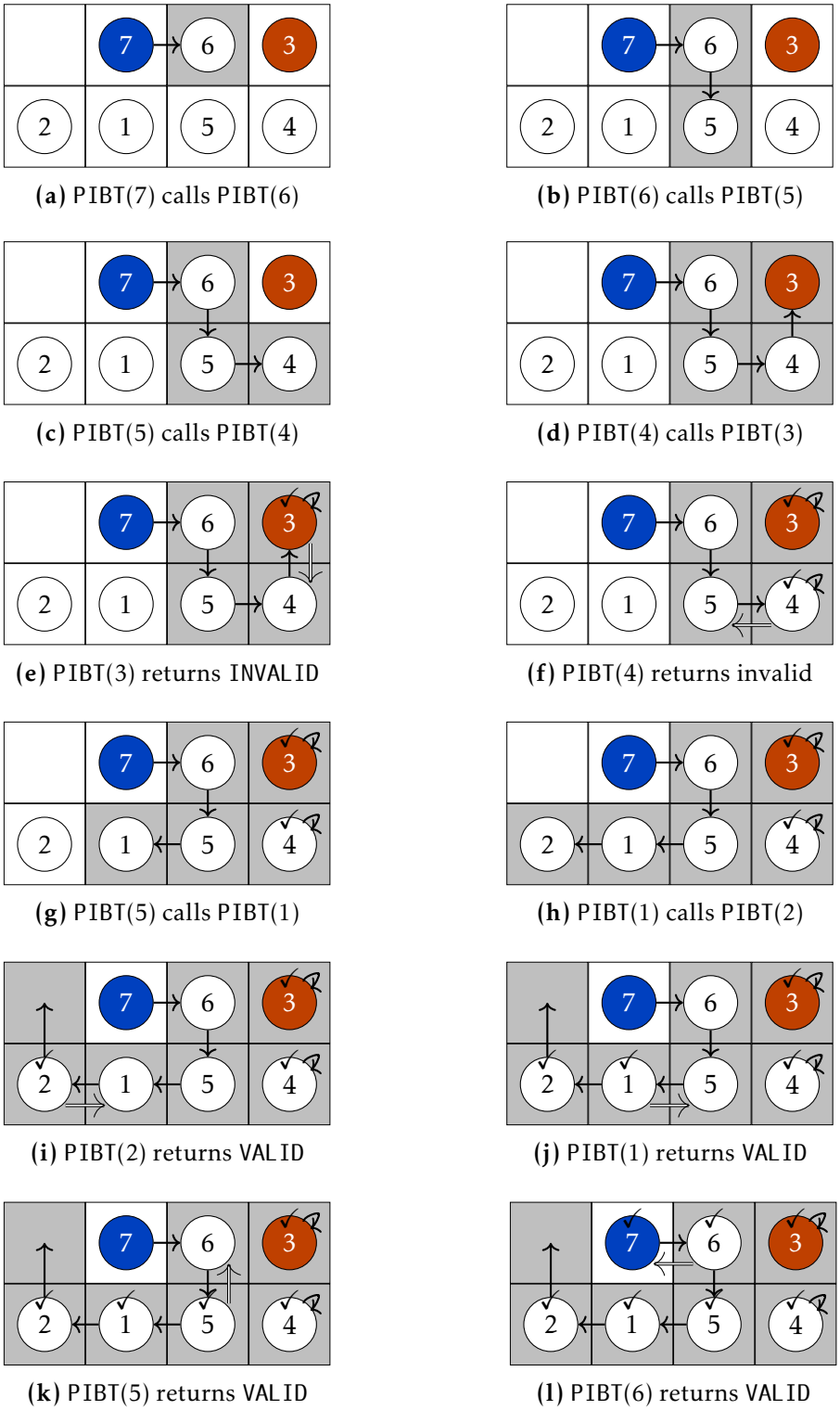
output: configuration $\mathcal{Q}^{\text{to}}$ (each element is initialized with $\perp$)

```
1:  for i ∈ A do
2:      if Q^to[i] = ⊥ then PIBT(i)
3:  return Q^to
4:  procedure PIBT(i)
5:      C ← neigh(Q^from[i]) ∪ { Q^from[i] }
6:      sort C following some criteria
7:      for v ∈ C do
8:          if collision occurs in Q^to supposing Q^to[i] = v then continue
9:          Q^to[i] ← v
10:         if ∃j ∈ A s.t. Q^from[j] = v ∧ Q^to[j] = ⊥ then
11:             if PIBT(j) = INVALID then continue
12:         return VALID
13:     Q^to[i] ← Q^from[i]
14:     return INVALID
```

The crux is understanding the `PIBT` procedure (Lines 4–14) that embodies the concept of priority inheritance and backtracking. It has one argument, representing the agent i making a decision. The procedure returns `INVALID` if i becomes a stuck agent like the red agent in Fig. 4.2 (agent-3), otherwise returns `VALID`. Once this procedure is called, i must determine the next location (i.e., $\mathcal{Q}^{\text{to}}[i]$) from the ordered set of candidate vertices C. Here, C consists of $\mathcal{Q}^{\text{from}}[i]$ and its neighbors (Line 5), sorted by user-specified criteria (Line 6). C is filtered so that excludes vertices that occur vertex/edge collisions in $\mathcal{Q}^{\text{to}}$ (Line 8). Formally, a vertex v is filtered when there is an agent k that satisfies either $\mathcal{Q}^{\text{to}}[k] = v$ or $\mathcal{Q}^{\text{to}}[k] = \mathcal{Q}^{\text{from}}[i] \wedge \mathcal{Q}^{\text{from}}[k] = v$. If no vertices remain in C, this is the case of a stuck agent; then, i must stay at the current location and the procedure returns `INVALID` (Lines 13–14).

For each vertex $v \in C$, the procedure checks whether i can move to v in the next timestep. After reserving v for i (Line 9), the priority inheritance occurs from i to another agent j when j, which has not yet determined the next location, occupies v (Lines 10–11). If j returns `INVALID`, the next location for i is replanned (Line 11), or else, i finishes the planning and returns `VALID`, as well as that v is unoccupied by others (Line 12).

A step-by-step example of Alg. 4.1 is illustrated in Fig. 4.3.



(a) PIBT(7) calls PIBT(6)

(b) PIBT(6) calls PIBT(5)

(c) PIBT(5) calls PIBT(4)

(d) PIBT(4) calls PIBT(3)

(e) PIBT(3) returns INVALID

(f) PIBT(4) returns invalid

(g) PIBT(5) calls PIBT(1)

(h) PIBT(1) calls PIBT(2)

(i) PIBT(2) returns VALID

(j) PIBT(1) returns VALID

(k) PIBT(5) returns VALID

(l) PIBT(6) returns VALID

Figure 4.3: Step-by-step example of Alg. 4.1 for Fig. 4.2. A black arrow corresponds to $\mathcal{Q}^{\text{to}}[i]$ updated at either Line 9 or Line 13. A checked mark means that $\mathcal{Q}^{\text{to}}[i]$ has been fixed. A gray-filled cell is a vertex requested as $\mathcal{Q}^{\text{to}}[j]$ from any agent j. A double-lined arrow shows backtracking, corresponding to either Line 12 (VALID) or Line 14 (INVALID).

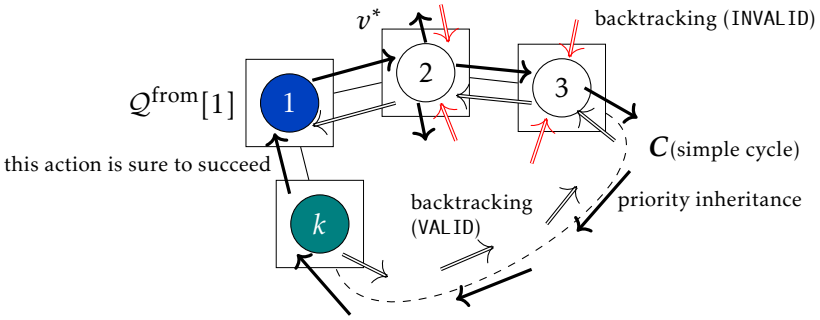


Figure 4.4: Proof sketch of Lemma 4.1. Agent-1 can relocate to v^* if there exists a simple cycle $\boldsymbol{C} = \left(Q^{\text{from}}[1], v^*, \ldots\right)$.

Analysis of Local Movements

Next, we see that an agent that initiates the planning with PIBT is always assigned to its desired vertex, provided that its location and the desired vertex belong to a simple cycle.

Lemma 4.1 (local movement of PIBT)**.** *Let agent-*1 *denote the first agent in* A*. Let* v^* *denote the first vertex in* C *for agent-1 at Line 6, i.e., one of neighboring vertices of* $Q^{from}[1]$*. If a simple cycle* $\boldsymbol{C} = \left(Q^{from}[1], v^*, \ldots\right)$ *exists, Alg. 4.1 assigns* v^* *to* $Q^{to}[1]$*.*

Proof. Figure 4.4 visualizes the following.

Assume that the top-level procedure calls PIBT(1) at Line 2. At this phase, $Q^{\text{to}}[i]$ is $\bot$ for any agent i, i.e., the next location for any agent has not been assigned yet. Consequently, agent-1 selects v^* as a target vertex v in the first iteration of Lines 7–12. Then, it remains to prove that Line 11, priority inheritance, never returns INVALID when another agent, say agent-2, occupies v^*. In the other cases, agent-1 evident gets v^*.

Next, assume that PIBT(2) is called from the original PIBT(1). The existence of the cycle $\boldsymbol{C} = (Q^{\text{from}}[1], v^*, \ldots)$ guarantees that agent-2 has a non-empty set of candidate vertices for $Q^{\text{to}}[2]$, say C_2, without colliding with agent-1. During the iterations of Lines 7–12, PIBT(2) returns VALID when an unoccupied vertex $v \in C_2$ from vertices in Q^{to} is selected. This forms the basis of the remaining proof by induction.

Following this, suppose that PIBT(i) is called before PIBT(2) returns any value. Similar to the case of agent-2, PIBT(i) must return VALID when one of the surrounding vertices of $Q^{\text{from}}[i]$ is unoccupied from vertices in Q^{to}. In addition, as distinct from agent-2, which must avoid an edge collision with agent-1, agent-i can select $Q^{\text{from}}[1]$ and return VALID when $Q^{\text{from}}[1]$ is neighbor to $Q^{\text{from}}[i]$. As a result, PIBT($i-1$), which calls PIBT(i), also returns VALID and eventually, PIBT(2) returns VALID.

Next, by contradiction, suppose that PIBT(2) returns INVALID. This assumption means that, for all agents $j(\neq 1)$ neighboring agent-2 in Q^{from}, PIBT(j) returns INVALID. This is the same for agent-j, and so forth. Meanwhile, the existence of $\boldsymbol{C}$ indicates that at least one agent $k(\neq 2)$ such that $Q^{\text{from}}[k] \in \boldsymbol{C}$ had initially at least one free neighbor vertex. Even though all vertices in $\boldsymbol{C}$ are occupied, the agent on the last vertex of $\boldsymbol{C}$ has a free neighbor vertex, i.e., $Q^{\text{from}}[1]$. This is a contradiction; PIBT(k) returns VALID. Consequently, PIBT(2) returns VALID. □

4.2.3 Dynamic Priority Assignment

Aiming to solve MAPF, we next incorporate a long-term perspective of planning into minimum PIBT (Alg. 4.1), that is, dynamic priority assignment. Algorithm 4.2 presents

the updated pseudocode. Given an MAPF instance, Alg. 4.2 generates a sequence of configurations, until reaching a user-specified termination condition. Note, this time, PIBT generates configurations beyond one timestep.

Algorithm 4.2 PIBT with dynamic priority assignment.

input: graph $G = (V, E)$, agents A, starts $(s_1, \dots s_n)$, goals $(g_1, \dots g_n)$

output: sequence of configurations $\mathbf{\Pi}$

preface: $\mathbf{\Pi}[t][i]$ is assumed to be initialized with $\perp$

1: $\mathbf{\Pi}[0][i] \leftarrow s_i$: for each agent $i \in A$

2: $p_i \leftarrow \epsilon_i$: for each agent $i \in A$ s.t. $\epsilon_i \in [0, 1)$ and $\epsilon_i \neq \epsilon_j$ for $i \neq j$ ▷ *setup priorities*

(for each timestep $t = 0, 1, \dots$ until terminates, repeat the following)

3: $p_i \leftarrow$ **if** $\mathbf{\Pi}[t][i] = g_i$ **then** ϵ_i **else** $p_i + 1$: for each agent $a_i \in A$ ▷ *update priorities*

4: sort A in decreasing order of priorities p_i

5: **for** $i \in A$ **do**

6: ⌊ **if** $\mathbf{\Pi}[t+1][i] = \perp$ **then** PIBT(i)

7: **procedure** PIBT(i) ▷ *t is used implicitly*

8: $C \leftarrow \mathtt{neigh}(\mathbf{\Pi}[t][i]) \cup \{\mathbf{\Pi}[t][i]\}$

9: sort C in increasing order of $\mathtt{dist}(u, g_i)$ where $u \in C$

10: **for** $v \in C$ **do**

11: **if** $\exists j \in A$ s.t. $\left(\mathbf{\Pi}[t+1][j] = v\right) \vee \left(\mathbf{\Pi}[t][j] = v \wedge \mathbf{\Pi}[t+1][j] = \perp\right)$ **then continue**

12: $\mathbf{\Pi}[t+1][i] \leftarrow v$

13: **if** $\exists k \in A$ s.t. $\mathbf{\Pi}[t][k] = v \wedge \mathbf{\Pi}[t+1][k] = \perp$ **then**

14: ⌊ **if** PIBT(k) is INVALID **then continue**

15: ⌊ **return** VALID

16: $\mathbf{\Pi}[t+1][i] \leftarrow \mathbf{\Pi}[t][i]$

17: ⌊ **return** INVALID

The major changes are as follows. The identical parts with Alg. 4.1 are grayed out in the pseudocode of Alg. 4.2.

Dynamic Priorities. Algorithm 4.2 first updates the priority $p_i \in \mathbb{R}_{\geq 0}$ for each agent i (Line 3). The rule is simple; increment p_i when i has not reached its goal g_i, otherwise, reset p_i to $\epsilon_i \in [0, 1)$. The tie-breaker ϵ_i, which is distinct for agents, has the role of keeping p_i unique among agents. Subsequently, according to priorities, the algorithm assigns locations to each agent i who has not been assigned with $\mathbf{\Pi}[t+1][i]$ via the procedure PIBT (Line 6). The termination of the algorithm is explained later in Chap. 4.3.1.

Vertex Scoring. The candidate vertices are sorted in increasing order of distance from the goal g_i (Line 9). Doing so makes agents progress towards their goal each timestep. In our implementation, to avoid unnecessary priority inheritance, the presence (or absence) of an agent is used as a tie-breaking rule.

4.2.4 Reachability

Here, a powerful property of Alg. 4.2 is proven, that is, all agents eventually reach their respective destination in graphs with adequate properties (e.g., biconnected graphs). The proof relies on Lemma 4.1, which states that the agent with the highest priority at each timestep, denoted as agent-1, is always assigned to its desired vertex.

Theorem 4.2 (reachability of PIBT)**.** *Given an MAPF instance such that G has a simple cycle for all pairs of adjacent vertices, Alg. 4.2 generates a sequence of connected configurations $\mathbf{\Pi}$ such that for any agent $i \in A$, $\mathbf{\Pi}[0][i] = s_i$ and there is a timestep $t \leq \mathtt{diam}(G) \cdot |A|$ such that $\mathbf{\Pi}[t][i] = g_i$.*

Proof. From Lemma 4.1, the agent with highest priority (i.e., the first agent calling PIBT) reaches its own goal within $\mathtt{diam}(G)$ timesteps. Based on the update rule of the priority p_i for an agent i, once some agent i has reached its goal, p_i is reset and is lower than the priority of all other agents who have not reached their goal yet. These agents see their priority increase by one. As long as such agents remain, exactly one of them must have the highest priority. In turn, this agent reaches its own goal after at most $\mathtt{diam}(G)$ timesteps. This repeats until all agents have reached their goal at least once, which takes at most $\mathtt{diam}(G) \cdot |A|$ timesteps in total. □

Remarks. Hereafter, this property is referred to as *reachability*, that is, all agents eventually reach their goal. Different from completeness for one-shot MAPF, the reachability never ensures that all agents reach their goal *simultaneously*; hence, PIBT is not ensured to solve conventional MAPF defined in Chap. 3.2.1. Indeed, we will see *livelock* situations of PIBT in Chap. 6. Alternatively, with the graph condition of Thrm. 4.2, PIBT is complete for the MAPF variant where agents need not necessarily stay at their goals.

Graph Condition. A typical example that satisfies the aforementioned graph condition is the biconnected undirected graph. The opposite is not true, however, and Thrm. 4.2 is expressed more generally to consider directed graphs. For instance, a directed ring satisfies the condition even though it is not biconnected.

Failure Case. Without the graph condition in Thrm. 4.2, several agents may remain in the same vertices permanently and may not reach their goals. Figure 4.5 shows such an example. In practice, PIBT needs a pre-defined maximum planning timestep to detect such failure cases. When PIBT reaches this timestep, it stops planning and reports a planning failure. For details, see also Chap. 4.3.1.

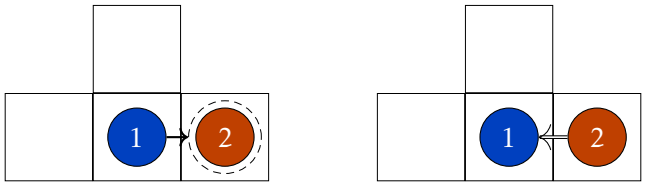
4.2.5 Time Complexity Analysis

Now we consider the time complexity of PIBT. Let F be the maximum time required for sorting candidate vertices (Line 6 in Alg. 4.1). F depends on both G and how to evaluate each vertex. Recall that $\Delta(G)$ denotes the maximum degree of G.

To begin with, the complexity of a minimum PIBT (Alg. 4.1) is analyzed as followed.

Proposition 4.3. *The time complexity of Alg. 4.1 is $O(A \cdot (\Delta(G) + F))$.*

Proof. The procedure PIBT is called exactly $|A|$ times, once for each agent. This is because PIBT(i) is called if and only if agent-i has not yet determined the next location (i.e., when $\mathcal{Q}^{\mathrm{to}}[i] = \bot$; Lines 2 and 11) but PIBT(i) assigns $\mathcal{Q}^{\mathrm{to}}[i]$ (Line 9) before calling another PIBT(k) for priority inheritance. The loop of Lines 7–12 iterates at most $\Delta(G) + 1$ times. Each operation in the loop is performed in constant time. From the assumption, Line 6 requires $O(F)$. As a result, Alg. 4.1 requires $O(A \cdot (\Delta(G) + F))$. □



(a) Priority inheritance **(b)** Backtracking

Figure 4.5: Failure case of PIBT on a graph without cycles. Assume that the goal of agent-1 is the location of another agent, say agent-2. The priority inheritance to agent-2 (4.5a) is `INVALID` (4.5b) because agent-2 has no escape vertex. Then, agent-1 does replanning, however, it selects the current location as the next location since it is the next nearest vertex to the goal. As a result, this situation will be held beyond the current timestep and there will be no progress. Later in Chap. 6.6, a countermeasure is provided.

The time complexity of Alg. 4.2 (PIBT for multiple timesteps) is assessed by simply adding the overhead of dynamic priorities.

Proposition 4.4. *The time complexity of Alg. 4.2 in one timestep is* $O(A \cdot (\Delta(G) + F + \lg A))$.

Proof. Line 3 requires $O(A)$. Line 4 requires $O(A \lg A)$. In total, together with Prop. 4.3, PIBT (Alg. 4.2) in one timestep requires $O(A \cdot (\Delta(G) + F + \lg A))$. □

The analysis is further continued for specific conditions. Assume that a distance table for the goal of each agent is computed by breadth-first search prior to executing PIBT, where the overhead is $O(A \cdot E)$. Then, F will be solely sorting candidate vertices following the table, resulting in $O(\Delta(G) \lg \Delta(G))$. Consequently, PIBT in one timestep is $O(A \cdot (\Delta(G) \lg \Delta(G) + \lg A))$. Assume further that G is a four-connected grid, as commonly used in MAPF studies as a benchmark. Here, instead of $\Delta(G)$, consider $\Delta(G) + 1$ for accurate analysis. Then, $(\Delta(G) + 1) \lg (\Delta(G) + 1) \simeq 11.6$. As a result, without a huge team of agents ($|A| < 3125$), the first term $|A| \cdot \Delta(G) \lg \Delta(G)$ is dominant; otherwise, the second term $|A| \lg |A|$ is dominant. In either case, *PIBT can be said to be computationally very inexpensive*.

The small time complexity provides many advantages:

- For instance, PIBT can address large instances for both A and G, which other MAPF algorithms cannot solve within a realistic timeframe.
- Another advantage is an application to *anytime planning*, a scheme that yields a feasible solution whenever interrupted but gradually improves solution quality as time goes by. Anytime planning is attractive, particularly in on-demand situations where the deliberation time is finite. We will see how to realize such a refinement method in Chap. 7. For such an approach, obtaining initial solutions as quickly as possible is key because we can leave much time for refinement. In this sense, PIBT is a good choice to obtain an initial solution. In contrast, the anytime approach cannot be realized by "slow" solvers because they may not provide solutions within a time limit.
- Lastly, given a certain timestep, the runtime of PIBT is *predictable*, which is a fundamental characteristic in real-time planning.

The power of PIBT is reflected in the experiment.

4.3 Application to Specific Problems

The previous section of Chap. 4.2 describes the fundamental mechanism of PIBT. This section complements how PIBT applies to specific problems, such as (one-shot) MAPF and MAPD, decentralization, and path planning without rotations.

4.3.1 One-shot MAPF

Termination

Thus far, when the PIBT run has been completed, has not been specified. In the conventional MAPF, PIBT is assumed to run until it reaches the goal configuration $(g_1, \ldots, g_n)$; otherwise, PIBT "fails" to solve the MAPF instance when it reaches the pre-defined sufficiently large timesteps.

Distance Evaluation

In Alg. 4.2, for each timestep, each agent has to evaluate distances from the surrounding vertices to its goal (Line 9). This operation could be implemented by calling A* on demand but it could also be a bottleneck. Instead, PIBT can save computation time by preparing distance tables from each agent's goal. This is computed by a breadth-first search from the goals with an overhead of $O(A \cdot E)$.[3]

Extension for One-shot MAPF

Even if G satisfies the condition for reachability, PIBT does not ensure that all agents reach their goal simultaneously, which is a requirement of conventional MAPF. In fact, with naive PIBT, a certain kind of livelock situation was confirmed in our experiment. This motivates the development of *PIBT*$^+$, a framework that enhances known MAPF solvers, presented in Alg. 4.3. The main idea behind PIBT$^+$ is to make problem instances easier for the subsequent MAPF solver by bringing agents near their destinations using PIBT. The target MAPF solver is called a *complement* solver.

Algorithm 4.3 PIBT$^+$

input: graph G, agents A, starts $(s_1, \ldots s_n)$, goals $(g_1, \ldots g_n)$

output: MAPF solution $\mathbf{\Pi}$

1: $t_{\min} \leftarrow \max_{j \in A} \texttt{dist}(\mathbf{\Pi}[0][j], g_j)$

2: Set initial priorities p_i to agents in descending order of $\texttt{dist}(s_i, g_i)$ by adjusting the tie-breaker ϵ_i at Line 2 in Alg. 4.2

3: $\mathbf{\Pi} \leftarrow$ PIBT (Alg. 4.2) until timestep $t_{\min}$

4: **if** $\mathbf{\Pi}[t_{\min}] = (g_1, \ldots, g_n)$ **then return** $\mathbf{\Pi}$

5: $\mathbf{\Pi}' \leftarrow$ other MAPF algorithm (*complement solver*), taking $\Pi[t_{\min}]$ as the start configuration

6: **return** a concatenation of $\mathbf{\Pi}$ and $\mathbf{\Pi}'$

The concept of Alg. 4.3 is as follows. $t_{\min}$ is the minimum number of timesteps needed for solutions from the start to the goal configurations. Because PIBT can be regarded as prioritized planning, the agent with the longest initial distance reaches its goal following

[3] Indeed, this is the main reason for the speedup of the implementation from [Okumura *et al.*, 2019].

the shortest path by providing priorities according to the initial distance. This implies that there is a chance that all agents reach their goals at $t_{\min}$. The remaining problem is relatively easy compared to the original because most agents are expected to have already reached their goals.

Theorem 4.5. *PIBT$^+$ is complete for MAPF on undirected graphs as long as the completeness conditions of the complement solver are satisfied.*

Proof. Let $\mathcal{Q}^{\text{init}}, Q^{\text{tmp}}, \mathcal{Q}^{\text{goal}}$ be the start configuration, the configuration at $t_{\min}$ planned by Line 3 in Alg. 4.3, and the goal configuration, respectively. There exists a solution from $\mathcal{Q}^{\text{tmp}}$ to $\mathcal{Q}^{\text{init}}$ because PIBT computes the plan from $\mathcal{Q}^{\text{init}}$ to $\mathcal{Q}^{\text{tmp}}$. If the original problem is solvable, there exists a solution from $\mathcal{Q}^{\text{init}}$ to $\mathcal{Q}^{\text{goal}}$, meaning that, at least one solution exists from $\mathcal{Q}^{\text{tmp}}$ to $\mathcal{Q}^{\text{goal}}$. According to this assumption, PIBT$^+$ finally uses the complete solver from $\mathcal{Q}^{\text{tmp}}$ to $\mathcal{Q}^{\text{goal}}$, and the complete solver must return a solution. The above satisfies the statement. □

PIBT$^+$ is essentially helpful for situations where it is difficult to obtain solutions through a direct adaptation of the complement solver but where plausible solutions are sought in a short time. As a complement solver, it is desirable to use complete algorithms such as rule-based, search-based, or compiling-based approaches. Note that PIBT$^+$ does not guarantee a better solution nor finish planning faster than using the complement solver from the beginning. On the other hand, since the transition from the initial configuration to the configuration at $t_{\min}$ is reversible, the additional burdens of sum-of-cost and makespan from optimal ones can be, respectively, at most $2|A| \cdot t_{\min}$ and $2t_{\min}$. Similarly, the additional burden of time complexity can be $O(t_{\min} \cdot A \cdot (\Delta(G) + F + \lg A))$ from the consequence of Prop. 4.4.

4.3.2 Multi-Agent Pickup and Delivery (MAPD)

The MAPD problem is a typical example of online and lifelong scenarios requiring task allocation and path planning. Here, how PIBT adapts to MAPD is described. To begin with, the MAPD problem is formulated.

Problem Formulation of MAPD

Similar to MAPF, a graph $G = (V, E)$ and a set of agents $A = \{1, 2, \ldots, n\}$ with $|A| \leq |V|$, and an injective initial state function $s : A \mapsto V$, are given. $s(i)$ is simply denoted as s_i.

Consider a stream of tasks $\Gamma = (\tau_1, \tau_2, \ldots)$. A task τ_j is defined as a tuple $(v_{\text{pick}}, v_{\text{deliv}})$, where $v_{\text{pick}}, v_{\text{deliv}} \in V$. MAPD includes situations in which tasks are added to Γ as time progresses. In other words, it is not assumed that all tasks are known initially. An agent is *free* when it has no assigned task. A task $\tau_j \in \Gamma$ can only be assigned to free agents and is assigned to at most one agent. When τ_j is assigned to agent-i, i has to visit v_{pick} and v_{deliv} in order. When the two vertices have been visited by i, τ_j is completed, and i becomes free again. A *service time* of τ_j is defined as the time interval from the generation of τ_j to its completion. Similar to MAPF, time is discretized. At each timestep, each agent can move to an adjacent vertex or stay at its current vertex, provided that no two agents collide, i.e., vertex and edge collisions are prohibited.

The objective of MAPD is to complete all tasks as quickly as possible. A *solution* to MAPD is a set of collision-free paths and task assignments, with each agent starting from s_i. When $|\Gamma|$ is finite, an MAPD algorithm is *complete* as it ensures that all tasks are finished within a finite number of timesteps.

Two kinds of objective functions are considered: *(i) service time* and *(ii) makespan*, that is, the timestep when all tasks are completed. Note that makespan is only defined when Γ is finite.

Applying PIBT to MAPD

To design a complete MAPD algorithm, we must guarantee two claims: *(i)* all non-assigned tasks are eventually assigned, and *(ii)* all assigned tasks are eventually completed, i.e., a task-assigned agent must visit pickup and delivery locations in order. For simplification, consider assigning a task to an agent only when the agent is at the pickup location. The remaining work for the agent is to visit the delivery location.

PIBT (Alg. 4.2) is convenient for the second claim due to the following reason. Recall that Lemma 4.1 states that, for each timestep, the agent with the highest priority can always move to an arbitrary neighbor vertex. Using this lemma, ensuring that a task-assigned agent completes the task in finite time is straightforward. This is achieved with a special prioritization scheme that satisfies the two conditions: *(i)* all task-assigned agents have higher priorities than free agents, and *(ii)* every task-assigned agent eventually gets the highest priority and holds it until the task is completed.

The first claim, all non-assigned tasks are eventually assigned, is achieved by various approaches. A simple approach is taken here, i.e., for each timestep, all free agents set their goals to the nearest pickup location of non-assigned tasks. Assume that every free agent eventually gets the highest priority among free agents and holds the highest until it reaches a goal (a pickup location). Then, this approach satisfies the first claim because all task-assigned agents eventually become free due to the second claim.

Algorithm 4.4 PIBT for MAPD

input: graph G, agents A, initial locations $(\mathbf{\Pi}[0][1],\dots,\mathbf{\Pi}[0][n])$, task stream Γ

output: task assignment and paths $\mathbf{\Pi}$

(repeat below for each timestep $t = 0, 1, \dots$ until all tasks are completed)

1: **for** $i \in A$ **do**
2: **if** i is assigned to a task $\tau = (v_{\text{pick}}, v_{\text{deliv}})$ **then** $g_i \leftarrow v_{\text{deliv}}$; **continue**
3: Let $\tau = (v_{\text{pick}}, v_{\text{deliv}}) \in \Gamma$ be an unassigned task that minimizes $\texttt{dist}\left(\mathbf{\Pi}[t][i], v_{\text{pick}}\right)$
4: **if** $\tau = \perp$ **then**
5: $g_i \leftarrow \mathbf{\Pi}[t][i]$
6: **else if** $v_{\text{pick}} = \mathbf{\Pi}[t][i]$ **then**
7: assign τ to i; remove τ from Γ; $g_i \leftarrow v_{\text{deliv}}$
8: **else**
9: $g_i \leftarrow v_{\text{pick}}$
10: $\mathcal{S} \leftarrow (\mathbf{\Pi}[t][1],\dots,\mathbf{\Pi}[t][n])$; $\mathcal{G} \leftarrow (g_1,\dots,g_n)$
11: $\mathbf{\Pi}[t+1][1],\dots,\mathbf{\Pi}[t+1][n] \leftarrow$ Alg. 4.2 for one-timestep with starts $\mathcal{S}$ and goals $\mathcal{G}$, while adding a prioritization rule such that all task-assigned agents have higher ones than those of free agents
12: **for** each agent $i \in A$ assigned to a task $\tau = (v_{\text{pick}}, v_{\text{deliv}})$ **do**
13: **if** $\mathbf{\Pi}[t+1][i] = v_{\text{deliv}}$ **then** make i free ▷ *τ is completed*

Algorithm 4.4 realizes the aforementioned concept to solve MAPD. For each timestep, the algorithm first assigns tasks while updating the agents' goals (Lines 1–9). Then, the algorithm uses PIBT to plan locations for the next timestep (Lines 10–11). As discussed earlier, for each timestep, Alg. 4.4 prioritizes the task-assigned agents than free agents.

As a secondary prioritization, it uses the original PIBT prioritization scheme (Line 3 of Alg. 4.2) as it is.

The following theorem is a consequence of the aforementioned discussion and its reachability.

Theorem 4.6. *Algorithm 4.4 is complete for an MAPD problem when G has a simple cycle for all pairs of adjacent vertices.*

Comparison of our Proposed Approach with an Existing Complete Approach

PIBT (Alg. 4.4) is later compared with the TP algorithm [Ma *et al.*, 2017b] experimentally. Before that, a qualitative discussion is provided as follows. TP is complete for MAPD in a limited situation. First, it requires locations that never be either pickup or delivery locations for any tasks, called non-task endpoints. The non-task endpoints are required at least for the number of agents. Then, for any two endpoints, a path must exist without traversing any other endpoints. Here, endpoints consist of non-task endpoints and candidates of pickup/delivery locations. Compared to TP, PIBT does not require such conditions and works in a wide range of situations as long as the graph condition is satisfied.

Other Task Allocation

Algorithm 4.4 takes a simple and greedy task allocation process. However, more aggressive approaches are applicable, e.g., incorporating linear cost-optimal assignment [Kuhn, 1955] or bottleneck assignment [Gross, 1959], as well as assignment algorithms discussed in the next chapter (Chap. 5). Doing so can improve solution qualities but potentially requires more computation time.

4.3.3 Decentralized Online Planning

Here, the decentralization of PIBT implementations is briefly discussed. In particular, we focus on the *online planning* of PIBT, that is, agents repeat a set of single-timestep planning and move actions. Since PIBT is based on prioritized planning, adopting a decentralized context is realistic. The part of priority inheritance and backtracking is performed by the propagation of information. Further, PIBT relies on local interactions between agents, implying that agents are not required to know other agents' information far away from their current location. To illustrate this, the concept of *interacting agents*, a set of agents that must negotiate their planning, is introduced.

Interacting Agents

When two agents are located within two hops of each other's move, they may collide in the next timestep; then, they are said to be *directly interacting* in that timestep. For an agent i, a group of *interacting agents* $\mathcal{A}_i(t) \subseteq A$ is then defined by transitivity over direct interactions in timestep t. Note that, given an agent i and $\mathcal{A}_i(t)$, for any other agent $j \in \mathcal{A}_i(t)$, we have $\mathcal{A}_j(t) = \mathcal{A}_i(t)$. Whenever the context is obvious, $\mathcal{A}$ is directly used.

Because agents belonging to different groups cannot affect each other, path planning, the negotiation process using priority inheritance and backtracking, can effectively occur in parallel. As a result, *theoretically speaking*,[4] PIBT can be decentralized, relying only on local interactions; it is sufficient that two agents in close proximity talk directly and

[4]Realizing truly decentralized systems is really difficult, which I learned through building the AFADA system [Kameyama *et al.*, 2021]. Therefore, I want to emphasize this term.

utilize multi-hop communication. However, note that the groups of interacting agents are fully identified prior to starting a PIBT process for one timestep.

Communication

Assume that interacting agents are fully detected at each timestep. Then, we analyze communication complexity, i.e., how many messages are required in PIBT. In a decentralized context, PIBT requires agents to know others' priorities before deciding on the next vertices. Usually, this requires $|\mathcal{A}|^2$ messages between agents. However, the update rule of the priority p_i relaxes this effort. For instance, storing other agents' priorities and communicating only when the priority is reset. Therefore, the communication cost of PIBT mainly depends on the information propagation phase.

Next, consider this information propagation phase. In PIBT, communication between agents corresponds to calling PIBT (priority inheritance), and when the procedure returns a value (corresponding to backtracking). PIBT(i) is never called twice within a timestep, as discussed in the analysis of time complexity. Moreover, each agent sends a backtracking message at most once in each timestep. Overall, the communication cost for PIBT at each timestep is linear for the number of agents $O(A)$. In reality, this can be even lower because the figures depend on the number of interacting agents $|\mathcal{A}|$, which can be much smaller than $|A|$.

4.3.4 Without Rotations

In physical environments such as mobile robots, movements corresponding to "rotations" might be difficult to realize due to synchronization problems. A *rotation* is a set of adjacent agents moving along a circle within one timestep. Formally, a rotation occurs for a subset of agents $\{i, j, k, \ldots, l\} \subseteq A$ during timestep t and $t+1$ when $\mathbf{\Pi}[t+1][i] = \mathbf{\Pi}[t][j] \wedge \mathbf{\Pi}[t+1][j] = \mathbf{\Pi}[t][k] \wedge \ldots \mathbf{\Pi}[t+1][l] = \mathbf{\Pi}[t][i]$ is satisfied. Let two connected configurations be *rotation-free* when there is no rotation. This part adapts PIBT to situations where rotations are prohibited.

PIBT (Alg. 4.2) does not require major changes even in a model without rotations. During the iteration for the candidate vertices of the next timestep (Lines 10–15), skip vertices resulting in rotations like Line 11 to prevent collisions. Let the corresponding Alg. 4.2 denote PIBT$^{\otimes}$. This variant outputs rotation-free paths.

Lemma 4.7 (Local movement in rotation-free model)**.** *Let agent-*1 *denote the agent with highest priority at timestep t and v^* the nearest vertex to g_1 neighboring $\mathbf{\Pi}[t][1]$. If $|A| < |V|$ and there exists a path between v^* to an arbitrary vertex without going through $\mathbf{\Pi}[t][1]$, PIBT$^{\otimes}$ assigns v^* to agent-*1 *as $\mathbf{\Pi}[t+1][1]$.*

Proof. Following $|A| < |V|$, we can derive $\exists v' \notin \{\mathbf{\Pi}[t][i] \mid i \in A\}$. There exists a path $\boldsymbol{D}$ between v^* to v' without going through $\mathbf{\Pi}[t][1]$. The subsequent proof is almost identical to that of Lemma 4.1 by using $\boldsymbol{D}$ instead of $\boldsymbol{C}$. □

Theorem 4.8 (Reachability in rotation-free model)**.** *Given an MAPF instance such that G is biconnected and $|A| < |V|$, PIBT$^{\otimes}$ generates a sequence of rotation-free and connected configurations $\mathbf{\Pi}$ such that, for any agent $i \in A$, $\mathbf{\Pi}[0][i] = s_i$ and there is a timestep $t \leq diam(G) \cdot |A|$ such that $\mathbf{\Pi}[t][i] = g_i$.*

Proof. The same proof procedure of the reachability (Thrm. 4.2) can be applied using Lemma 4.7 instead of Lemma 4.1. □

Note that this theorem is restricted to biconnected graphs compared to the original theorem on reachability (Thrm. 4.2).

4.4 Evaluation

This section evaluates PIBT thoroughly to empirically demonstrate that PIBT is a quick and scalable MAPF algorithm with acceptable solution quality. PIBT is tested in MAPF (Chap. 4.4.1) and MAPD (Chap. 4.4.2), followed by a stress test for the number of agents (Chap. 4.4.3) and robot demonstration (Chap. 4.4.4). The simulator was developed in C++, and the experiments were run on a laptop with Intel Core i9 2.3 GHz CPU and 16 GB RAM. The code is available at `https://kei18.github.io/pibt2`. The experimental result of PIBT for MAPF in extremely dense situations is complemented in Appendix B.1.

4.4.1 Multi-Agent Path Finding (MAPF)

Setup

Benchmark. The MAPF benchmark [Stern *et al.*, 2019], which includes a set of four-connected grids and start–goal pairs for agents, was used. Ten grids were first selected with different portfolios, e.g., size, sparseness, and complexity (see Fig. 4.6 and 4.7). For each grid, 25 "random scenarios" were used while increasing the number of agents by ten up to the maximum (1,000 agents in most cases). Therefore, identical instances were tested for the solvers in all settings. Note that almost all selected grids did not satisfy the graph condition of reachability. The problem setting follows conventional MAPF, i.e., all agents are necessary to be at their goals to be terminated.

Baselines. The following algorithms were carefully picked for comparisons with PIBT:

- **HCA*** [Silver, 2005] as standard prioritized planning [Erdmann and Lozano-Perez, 1987]. Aiming at improving solution quality, each agent was prioritized so that those with more considerable distances from their starts to goals have higher priorities, following heuristics from [Van Den Berg and Overmars, 2005].
- **EECBS** [Li *et al.*, 2021c] as a state-of-the-art bounded search-based sub-optimal solver. The sub-optimality was set to 1.2. Note that EECBS is bounded sub-optimal for the sum-of-costs metric.
- **EECBS-M**: The adapted version of EECBS for the makespan metric.
- **Push and Swap (PS)** [Luna and Bekris, 2011] as a standard rule-based approach. Although PS is incomplete as pointed out in [De Wilde *et al.*, 2014], it is the origin of many extended algorithms [Sajid *et al.*, 2012; De Wilde *et al.*, 2014; Wiktor *et al.*, 2014; Zhang *et al.*, 2016]. Hence, PS was used.
- **PS**$^+$: The post-processing solutions were obtained by PS. PS allows at most one agent to move, resulting in terrible outcomes. Thus, the solutions were compressed while preserving temporal dependencies of the solutions, influenced by the techniques in [Hönig *et al.*, 2016], which allow multiple agents to move simultaneously.
- **CBS** [Sharon *et al.*, 2015] with improvement techniques [Boyrasky *et al.*, 2015; Boyarski *et al.*, 2015; Felner *et al.*, 2018; Li *et al.*, 2019a; Li *et al.*, 2019b; Li *et al.*, 2021b; Zhang *et al.*, 2022a] as a state-of-the-art search-based two-level optimal solver. Note that CBS optimizes the sum-of-costs metric.
- **CBS-M**: The makespan optimal version of CBS instead of sum-of-costs.
- **BCP** [Lam *et al.*, 2022] as a state-of-the-art compiling-based two-level solver, which is optimal for the sum-of-costs metric.

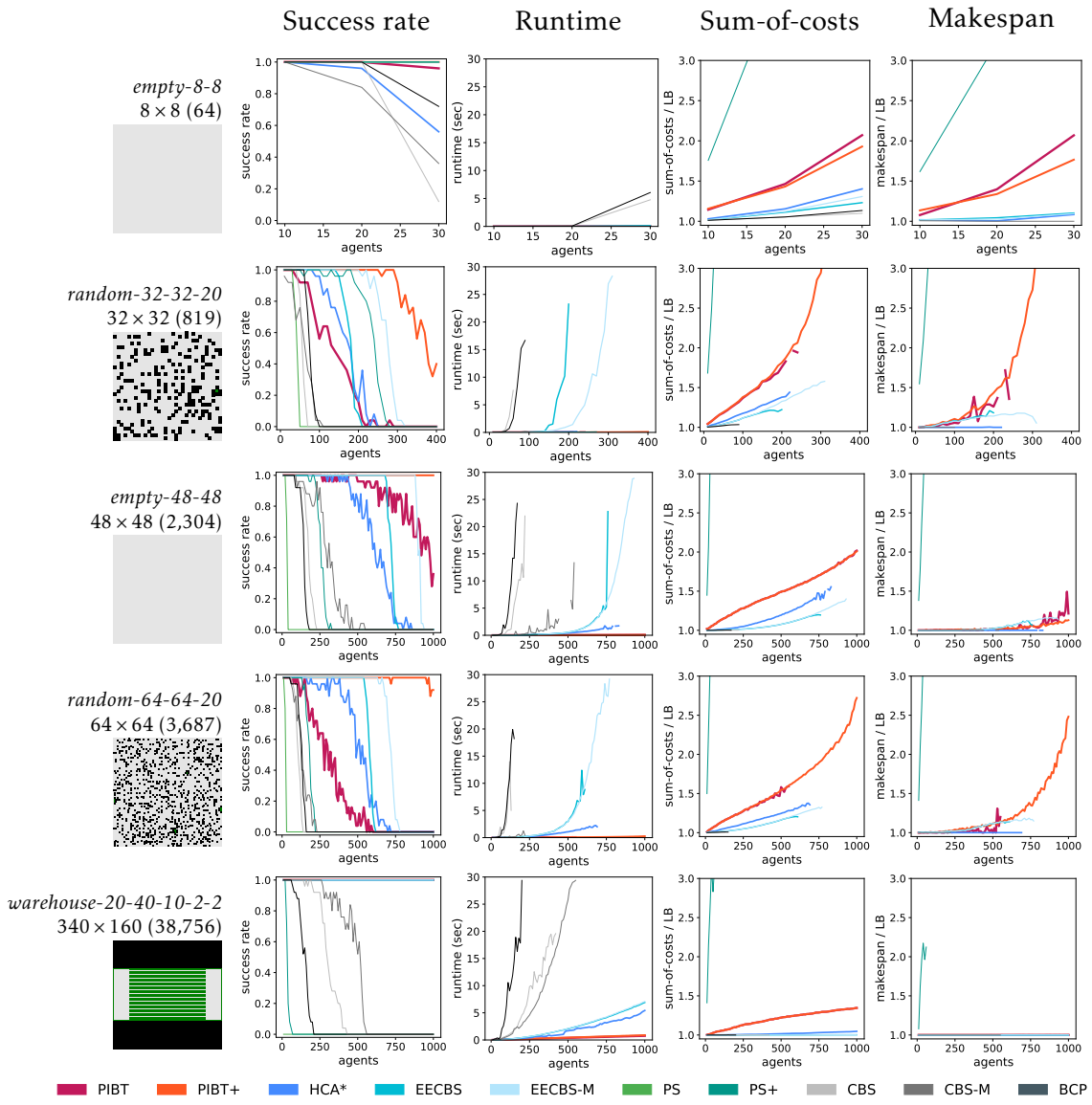


Figure 4.6: Summary of MAPF results (1/2). $|V|$ is shown in parentheses.

For EECBS, CBS, and BCP, the implementations coded by their respective authors were used.[5] When these implementations have parameter specifications, the default was used for each. The other solvers (HCA*, PS, and PS$^+$) were coded by the author, which is own-coded in C++.

Failure Case. An algorithm was regarded as having failed to solve an instance when either of the three conditions was met:

1. The algorithm reported failure.
2. The algorithm reached the runtime limit (30 s). This value was based on [Stern *et al.*, 2019].
3. The algorithm reached the makespan limit (2,000 in *brc202d*; otherwise 1,000).

[5]The codes are available on `https://github.com/Jiaoyang-Li/EECBS`, `https://github.com/Jiaoyang-Li/CBSH2-RTC`, and `https://github.com/ed-lam/bcp-mapf`, respectively. (EE)CBS-M was implemented in a best-first search manner that prioritizes makespan-better search nodes in the high-level tree, modifying the aforementioned codes.

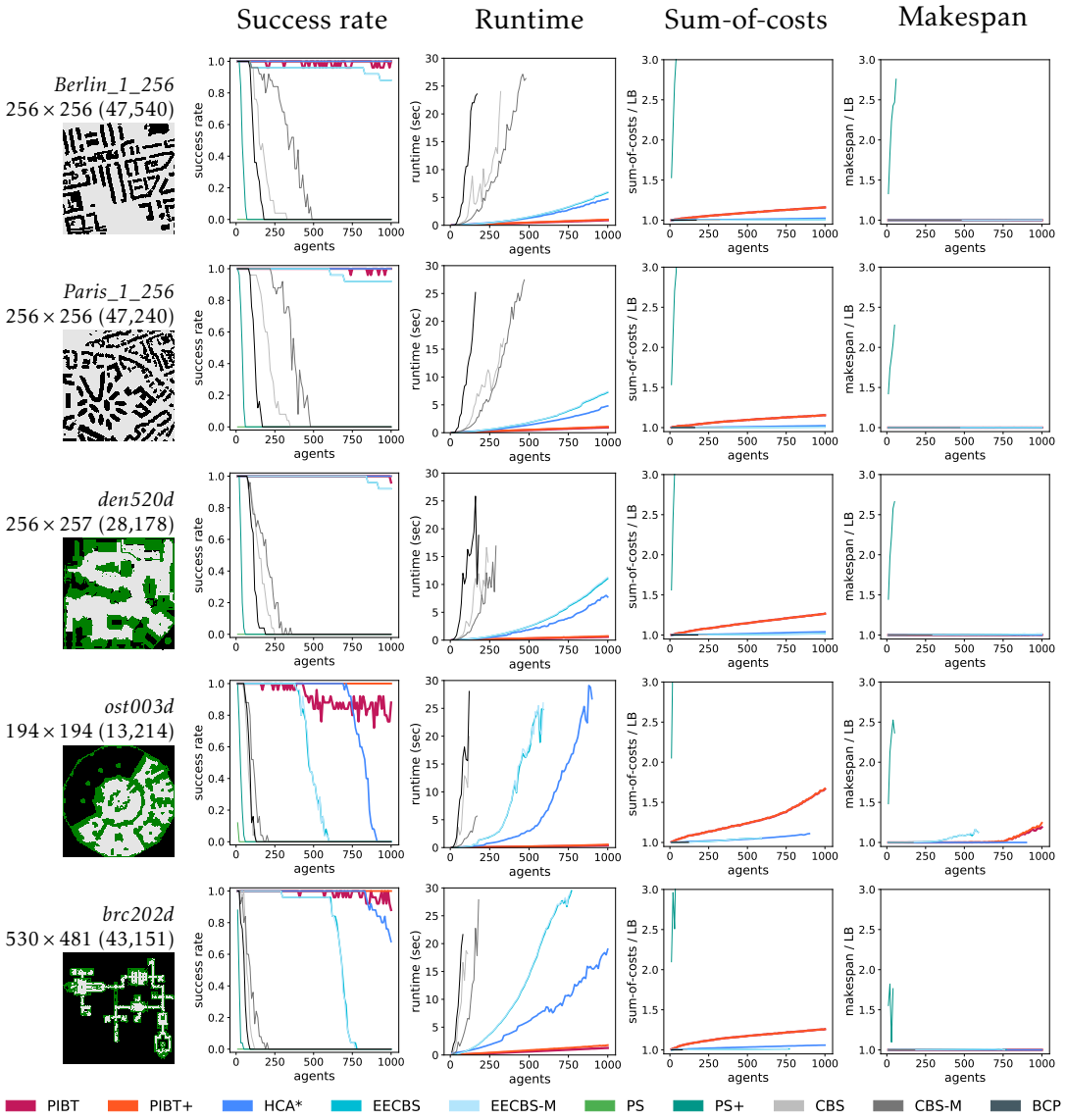


Figure 4.7: Summary of MAPF results (2/2).

This rules out impractical MAPF outcomes. The values were set according to preliminary results.

Metrics. The following four evaluation metrics were used, referring to other MAPF studies:

- **Success rate** over 25 instances.
- **Runtime**.
- **Sum-of-costs** divided by $\sum_{i \in A} \texttt{dist}(s_i, g_i)$.
- **Makespan** divided by $\max_{i \in A} \texttt{dist}(s_i, g_i)$.

The latter two assess the solution quality; smaller solutions are better and the minimum is one. Although optimal costs are challenging to obtain, these scores work as a lower bound of sub-optimality. Note that CBS and BCP are optimal for the sum-of-costs metric whereas CBS-M is optimal for makespan; hence makespan scores of CBS/BCP and sum-of-costs scores of CBS-M are omitted from the charts. Note further that sum-of-costs and

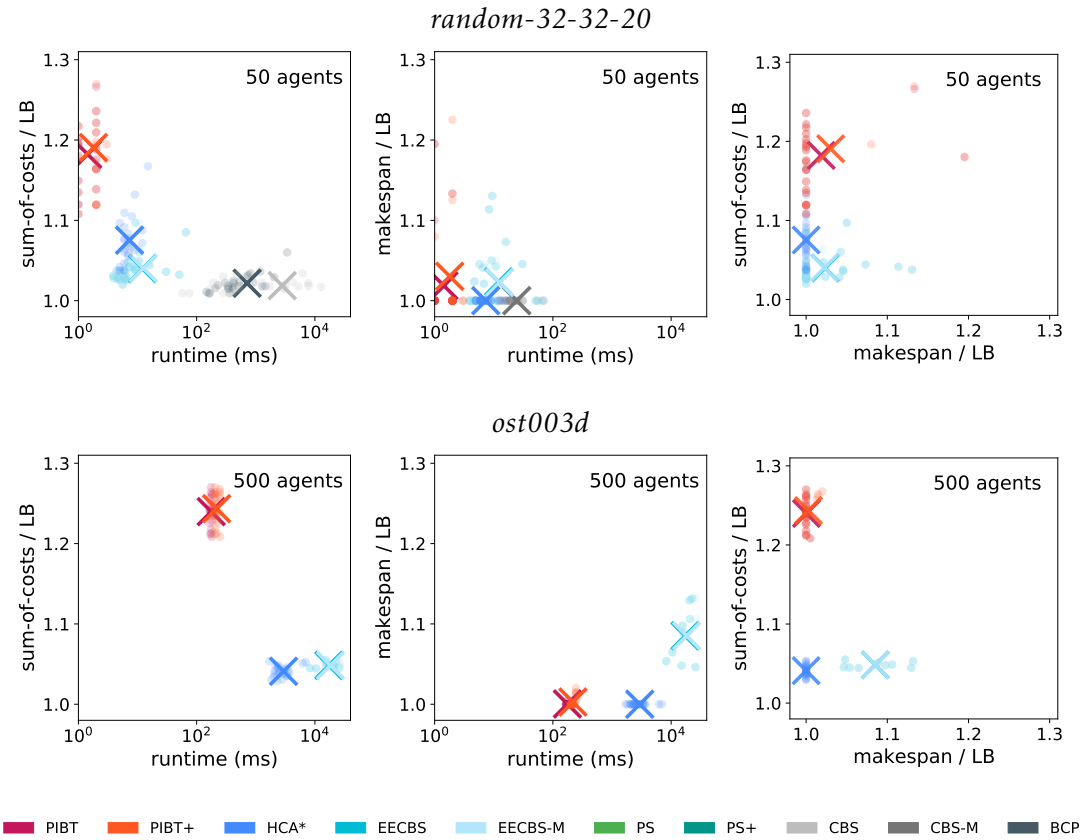


Figure 4.8: Details of MAPF results. As typical examples, *random-32-32-20* and *ost003d* were selected. Charts with a fixed number of agents are shown. The scores of sum-of-costs for CBS-M and those of makespan for CBS/BCP are omitted. In the rightmost chart of *ost003d*, the plots of EECBS and EECBS-M are overlapped.

makespan have Pareto optimal structure [Yu and LaValle, 2013b], i.e., they cannot be minimized simultaneously in general.

Other Remarks. PIBT$^+$ uses PS$^+$ as a complement solver. Own-coded solvers (PIBT$^{(+)}$, HCA*, PS$^{(+)}$) used a distance table for the start locations of each agent, computed by breadth-first search. The runtime included the procedure of creating the table.

Results

Figures 4.6 and 4.7 present the results. Figure 4.8 provides detailed perspectives of the results focusing on two maps. Excluding success rates, the scores are averaged over only the successful instances with that solver.[6] In addition, Table 4.1 shows the classification of failures of each solver.

Overall, PIBT outputs solutions immediately (≤ 5s in large maps) even with a thousand agents, with a slight decline in the solution quality (sum-of-costs). Specific findings are summarized as follows.

PIBT is extremely scalable. The scalability here is defined by map size and the number of agents. In both aspects, PIBT outperforms the other baselines. The runtime of PIBT is significantly faster by orders of magnitude compared to the other solvers. In other

[6] For a fair comparison, it is better to present the average scores of instances that all solvers solved. However, due to the high failure rates, the current style of data plots was selected.

random-32-32-20

	PIBT	PIBT$^+$	HCA*	EECBS	EECBS-M	PS	PS$^+$	CBS	CBS-M	BCP
total	674	102	590	568	333	908	426	872	862	823
1. failure report	0	3	590	0	0	0	0	0	0	0
2. makespan limit	674	99	0	0	0	908	426	0	0	0
3. time over	0	0	0	568	333	0	0	872	862	823

ost003d

	PIBT	PIBT$^+$	HCA*	EECBS	EECBS-M	PS	PS$^+$	CBS	CBS-M	BCP
total	204	0	473	1301	1296	2497	2439	2311	2237	2294
1. failure report	0	0	0	0	0	0	0	0	0	0
2. makespan limit	204	0	0	0	0	2497	2439	0	0	0
3. time over	0	0	473	1301	1296	0	0	2311	2237	2294

Table 4.1: Classification of failures. The numbers of failed instances, regardless of $|A|$, are reported.

words, PIBT can solve large instances that other solvers cannot solve. This is owing to the small-time complexity.

PIBT outputs solutions with acceptable quality in sparse situations. In general, PIBT does not guarantee solution quality. However, both sum-of-costs and makespan are adequate compared to others in sparse settings. This is in contrast to pure rule-based solvers (PS$^{(+)}$) that mostly fail due to the makespan limit even with fewer agents. PIBT is based on prioritized planning, enabling output solutions with acceptable quality.

Failure reasons of PIBT. Table 4.1 reveals that the failure reason for PIBT is due to the makespan limit. Two failure categories of PIBT are further explained as follows. The first case is due to the nature of reachability (Thrm. 4.2) that does not ensure that all agents reach their goals simultaneously, causing livelock situations (c.f. Chap. 6.6). The second case is due to graphs without a cycle for any two adjacent vertices. In such graphs, it is possible to reach situations where several agents stop moving, similar to Fig. 4.5. As those agents remain in their locations, PIBT eventually reaches the makespan limit. Therefore, PIBT often fails with many agents in small maps (for instance, *empty-48-48* and *random-64-64-20*).

PIBT$^+$ can increase the success rate of PIBT dramatically. The aforementioned shortcoming of PIBT for one-shot MAPF is overcome by adding a complement solver. Note that PIBT$^+$ sometimes failed because an incomplete solver PS$^+$ was used as a complement solver. Note further that it is possible to use other solvers for the complement solver.

Solution quality of PIBT$^+$ in dense situations. In Fig. 4.6, the upper bound of suboptimality of PIBT$^+$ in *random-32-32-20* dramatically increases with the number of agents. This is due to the complement solver. In the experiment, PIBT$^+$ used PS$^+$ as the complement solver to achieve high success rates within a small computation time. In general, PS$^+$ is quick even for large instances, however, its solution quality is not excellent, as seen in our results. In cluttered and dense situations such as *random-32-32-20*, it is difficult to obtain solutions by PIBT. Then, PIBT$^+$ often uses the complement solver, degrading the solution quality.

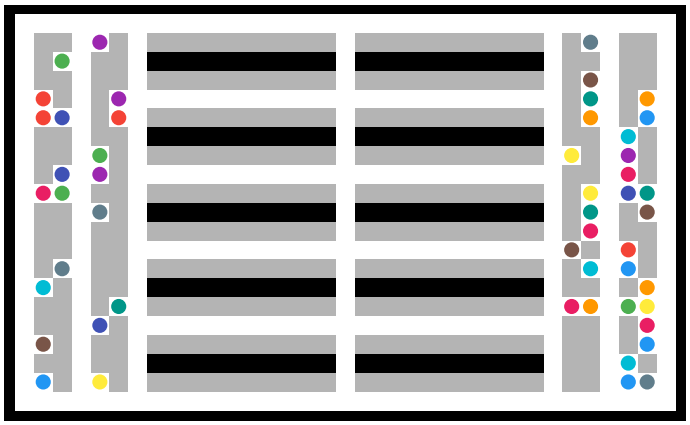


Figure 4.9: Map used in the MAPD experiment. The graph is 21×35 four-connected grid. Gray cells in the MAPD map are task endpoints, i.e., pickup and delivery locations.

Interpretations of comparisons' results. Optimal solvers (CBS and BCP) can address a few hundred agents in some cases, but usually, they cannot address several hundreds of agents, emphasizing the necessity of sub-optimal solvers. Pure rule-based solvers ($\text{PS}^{(+)}$) are quick, but their solution qualities are not acceptable compared to the other solvers. Indeed, failures of $\text{PS}^{(+)}$ are mainly due to the makespan limit. Prioritized planning (HCA^*) outputs plausible solutions for both sum-of-costs and makespan. It is scalable but not so much as PIBT. In addition, when scenarios are dense, it is significantly difficult to find solutions according to *static* priorities (see *random-32-32-20* in Table 4.1), making *dynamic* priorities attractive as used in PIBT. The overall result of the search-based sub-optimal solver (EECBS) is similar to HCA^*, i.e., it outputs plausible solutions but is not as quick and scalable as PIBT. In short, PIBT has a unique position compared to other established MAPF approaches, i.e., it is rapid and scalable with acceptable solution qualities.

4.4.2 Multi-Agent Pickup and Delivery (MAPD)

Setup. The experimental setup follows that of the original MAPD study [Ma *et al.*, 2017b], using the same undirected graph as a testbed, and setting the same locations as candidates for pickup and delivery (Fig. 4.9). A sequence of 500 tasks was generated by randomly choosing pickup and delivery locations from all task endpoints. The six different task frequencies where numbers of tasks are added to the task stream Γ were as follows: 0.2 (one task every five timesteps), 0.5, 1, 2, 5, and 10 with the number of agents increasing from 10 to 50. The token passing (TP) algorithm [Ma *et al.*, 2017b] was also tested as a baseline, which was own-programmed in C++. All experimental settings were performed over 100 instances in which the initial positions of agents were set randomly. The following three metrics were evaluated:

- **Runtime** per one timestep, because MAPD is assumed to be online.
- **Service time**, the duration from issue to completion of tasks.
- **Makespan**, the first timestep at which all tasks are completed.

Both PIBT (Alg. 4.4) and TP used an all-pairs distance matrix, pre-computed with the Floyd–Warshal algorithm [Floyd, 1962].

Result. Figure 4.10 summarizes the results. PIBT significantly outperforms TP in runtime and is comparable to or better than TP in solution quality, characterized by service

time and makespan. The runtime improvements are because of the low time complexity of PIBT. The improvements in solution quality are mainly due to how the algorithm deals with free agents. Assume that there is one unassigned task. In PIBT, all free agents move toward the pickup location, with only the earliest one actually getting the task and then having to start moving to the delivery location while *pushing the other free agents away*, thanks to the prioritization scheme. In contrast, TP evacuates all free agents to non-task endpoints to avoid deadlocks other than the one agent that is newly assigned. However, this assigned agent must still *"dodge" those free agents' locations* to prevent deadlocks. As a result, the path planned by PIBT for the task-assigned agent will be shorter than that of TP.

4.4.3 Stress Test for Scalability

Setup. The scalability of PIBT was evaluated through MAPF with a large map while varying the number of agents from 2,000 to 10,000. The map, shown in Fig. 4.11, was the largest in the MAPF benchmark [Stern *et al.*, 2019]. Here, only runtime per timestep was evaluated, averaged over the first 100 timesteps, regardless of whether the planning succeeded. Similar to the previous MAPF experiment, distance tables computed before executing PIBT were used. The starts and goals of agents were set randomly for each repetition.

Result. Figure 4.11 summarizes the result. The scores follow almost a linear trend in the number of agents. Surprisingly, even with 10,000 agents, PIBT scored around 10 ms per one timestep on an ordinary laptop.

4.4.4 Demonstrations with Real Robots

PIBT was also implemented with a team of small physical robots. Here, an online and lifelong scenario is presented where a new goal is immediately assigned to an agent who reaches its current goal. Note that, because of reachability, PIBT ensures that all assigned goals are eventually met, that is, visited by assigned robots.

Platform. *Toio* robots (`https://toio.io/`) were used to implement PIBT. The robots, connected to a master computer via the Bluetooth LE (low energy) protocol, advance on a specific playmat and are controllable by instructions using absolute coordinates.

Usage. The robots were controlled in a *centralized*, *synchronized*, and *online* mode, described as follows. A virtual grid (7×5 with obstacles) was created on the playmat. The robots followed the grid. A central server (a laptop) managed the locations of all robots. Periodically, the server executed PIBT for one single timestep and issued the instructions (i.e., where to go) to each robot. The code was written in Node.js.

Snapshot. Figure 4.12 shows a snapshot of the demo. The full video and code are also available at `https://kei18.github.io/pibt2`.

4.5 Related Work

Finally, how PIBT relates to other MAPF algorithms is discussed. The global picture of MAPF algorithms has already been presented in Chap. 3.2.4. Therefore, the below

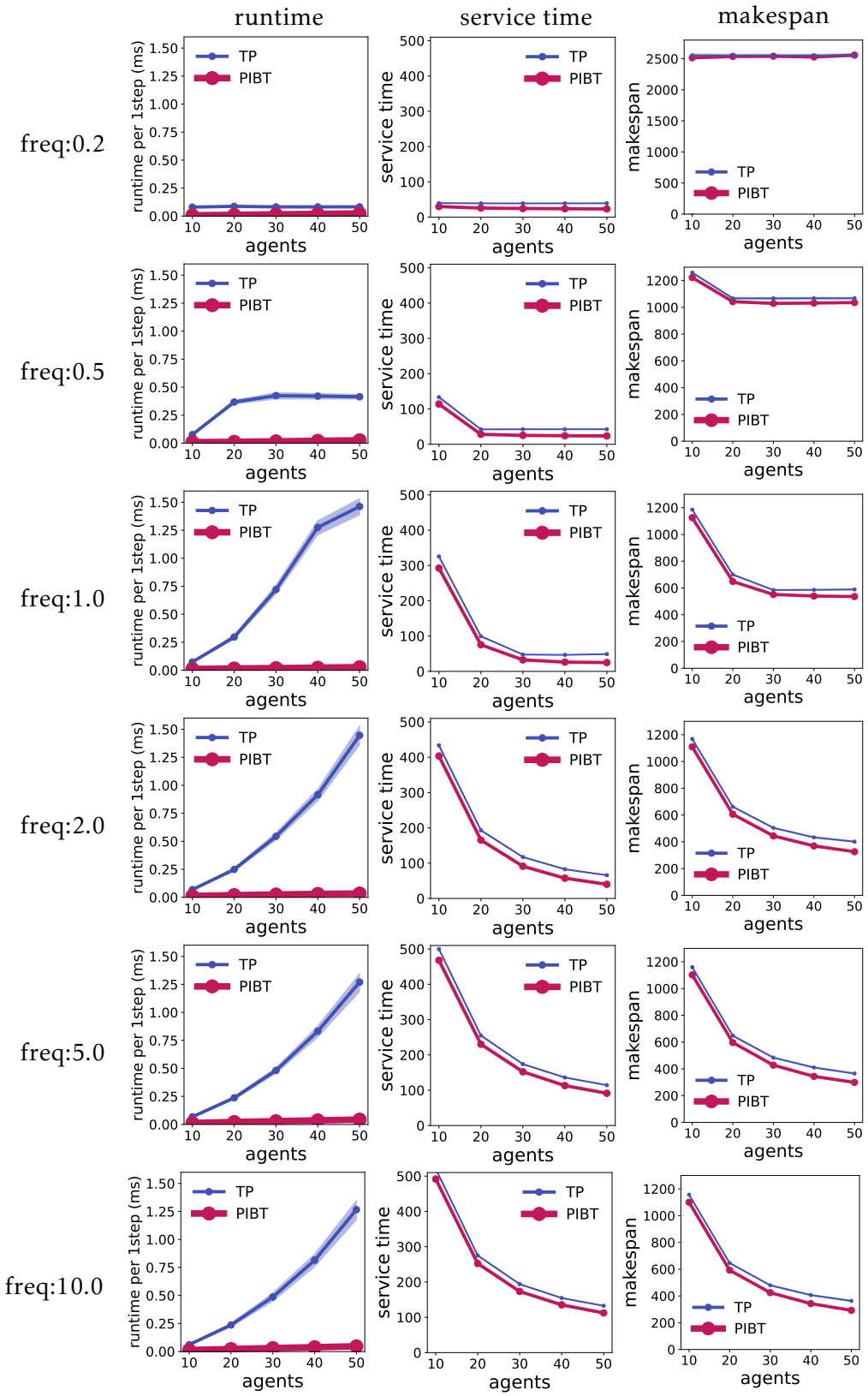
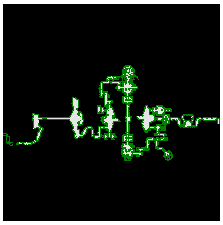


Figure 4.10: Summary of MAPD results. Each value is an average of 100 instances with 95% confidence intervals. For both service time and makespan, the intervals are hard to recognize because they are tiny.

orz900d
1491 × 656 (96, 603)



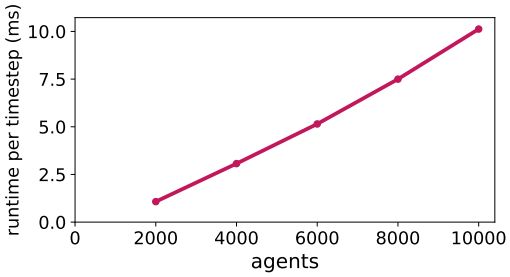


Figure 4.11: Results of the stress test. The average runtime per timestep is shown. The plots of the 95% confidence intervals are too small and difficult to recognize.

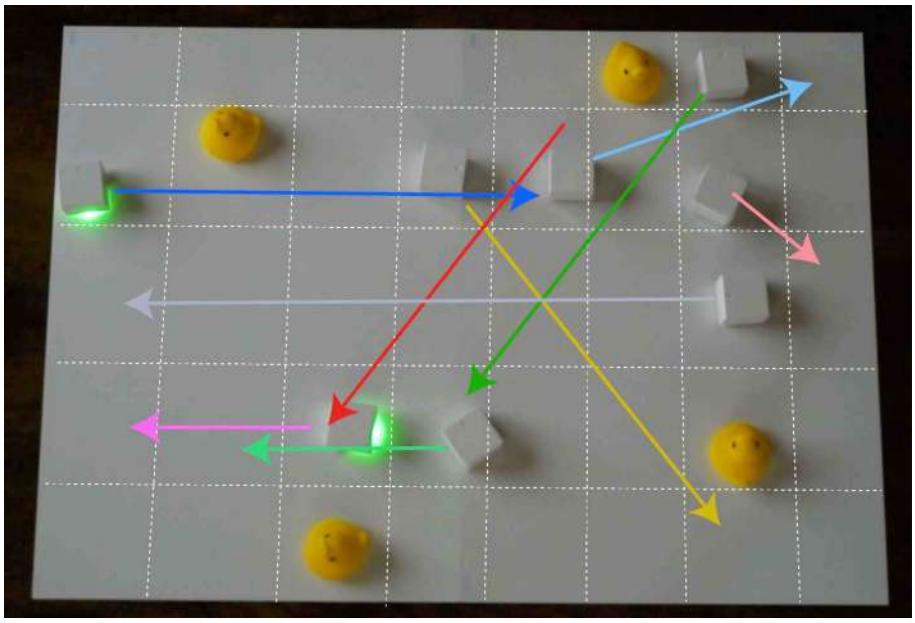


Figure 4.12: Snapshot of PIBT demonstration with real robots. The virtual gird is shown with white dotted lines. Each robot's goal is further annotated with colored arrows. Green-lighted robots have recently been allocated. Yellow ducks are obstacles.

discussion focuses on *prioritized planning (PP)* and *rule-based* approaches because PIBT has features of these two classes.[7]

4.5.1 Prioritized Planning (PP)

The concept of PP [Erdmann and Lozano-Perez, 1987] is that agents sequentially plan paths according to their unique priorities while avoiding conflicts with previously planned paths (see Alg. 3.3). PP is neither complete nor optimal, however, it is a computationally inexpensive approach to MAPF, hence it has been incorporated into many MAPF algorithms. For instance, the MAPP algorithm [Wang and Botea, 2011] combines techniques of PP for the way the blanks moved around in sliding tile puzzles and proposed a complete algorithm for a specific class of instances. In [Velagapudi *et al.*, 2010], decentralized implementations of PP that divide planning into several iterations. Variants of PP have also been studied, e.g., multi-robot decoupled planning that first computes individual plans neglecting collisions, and then replans individual paths while incorporating collision costs gradually [Jiang *et al.*, 2019]. This scheme has similarities to PIBT,

[7] In my frank opinion, PIBT is PP rather than rule-based approaches. It just sequentially assign resources (vertices) – without graph topology analysis.

albeit the applied problem is different and the approach has no theoretical guarantee. The well-known algorithm of PP for MAPF is hierarchical cooperative A* (HCA*) [Silver, 2005]. The "windowed" version of HCA* is known as windowed HCA* (WHCA*) [Silver, 2005], which repeatedly plans fixed-length (windowed) paths for all agents. PIBT can be considered as using WHCA* with unit-length window size.

Because priority ordering is crucial for PP, many studies addressed this issue. The negotiation process between agents regarding the ordering was studied in [Azarm and Schmidt, 1997]; this process solves conflicts by having involved agents try all priority orderings and deals with congestion by limiting negotiation to at most three agents while letting others wait. A similar approach is taken in [Bnaya and Felner, 2014] where the winner determines the prioritization scheme. There are some heuristic approaches for offline planning to find a reasonable ordering, for instance, adjusting the ordering by simple hill-climbing [Bennewitz *et al.*, 2002], or using distances between initial locations and destinations [Van Den Berg and Overmars, 2005]. There is a sufficient condition that the sequential collision-free solution can always be constructed regardless of priority orders, called well-formed instances [Čáp *et al.*, 2015], such that for each pair of start and goal, a path exists that traverses no other starts and goals. However, well-formed instances are difficult to realize in dense situations. A recent theoretical analysis for PP [Ma *et al.*, 2019a] identifies instances that fail for any order of *static* priorities, which provides a strong case for planning based on *dynamic* priorities, such as the approach taken with PIBT. The paper also presents a variant of PP that searches for good static priorities using techniques from CBS. However, the method has no theoretical guarantee.

4.5.2 Rule-based Approaches

Rule-based approaches make agents move step-by-step following ad-hoc rules. The representatives are push and swap/rotate (PS/PR) [Luna and Bekris, 2011; De Wilde *et al.*, 2014], which partly influenced the proposed PIBT. PS/PR are a sub-optimal rule-based approach for arbitrary graphs. They rely on two primitives: the "push" operation to move an agent toward its goal and the "swap" operation to allow two agents to swap locations without altering the configuration of other agents. These approaches, however, only allow a single agent or a pair of agents to move at a time.

Some studies enhanced PS/PR, for instance, parallel push and swap [Sajid *et al.*, 2012] by enabling all agents to move simultaneously, or push–swap–wait [Wiktor *et al.*, 2014] by taking a decentralized approach in narrow passages. In [Wei *et al.*, 2014], a decentralized approach that partly uses the PS technique to avoid deadlocks is presented. DisCoF [Zhang *et al.*, 2016], a decentralized method for MAPF, also uses swap operations to ensure completeness. There is an algorithm similar to PS, called TASS [Khorshid *et al.*, 2011], targeting trees.

PIBT can be regarded as a combination of safe "push" operations because of the backtracking protocol and dynamic priorities. Note that PIBT does not require the operational equivalent of "swap."

4.6 Concluding Remarks

This chapter introduced PIBT, a quick and scalable algorithm for solving MAPF iteratively. PIBT focuses on the adjacent movements of multiple agents, relying on a simple prioritization scheme; hence, it can be applied to many domains, including online and lifelong situations. The four empirical results support this aspect:

- PIBT$^{(+)}$ can promptly solve large MAPF instances, whereas other solvers take the time or require unrealistic computational times.

- PIBT outperforms current solutions for pickup and delivery (MAPD).
- Despite thousands of agents, PIBT can yield planning in a real-time manner.
- A real-robot execution of PIBT was further presented.

PIBT has good theoretical properties as well as powerful empirical results, meanwhile, several drawbacks had already been observed in the chapter. The critical critique is that *PIBT is incomplete for MAPF.* Although PIBT$^+$ can overcome this issue to some extent, however, in the experiment, a rule-based algorithm PS was used. This is problematic. As discussed in Chap. 3.2.4, rule-based approaches are very sensitive to problem specifications, shutting a door for *domain-independent* planning for multiple agents like MRMP. Later in Chap. 6, we will see the incompleteness of PIBT is overcome by LaCAM. But before that, let us see another example of short-horizon planning for unlabeled-MAPF, in the next chapter.

Chapter 5

Short-Horizon Planning for Unlabeled-MAPF

This chapter provides another example of short-horizon planning. This time, it is for *unlabeled-MAPF*, a joint problem of target assignment and collision-free pathfinding (see the definition in Chap. 3.4). The proposed algorithm is called the *TSWAP* algorithm. TSWAP initially appeared in [Okumura and Défago, 2022b].[1] The chapter begins with its overview.

5.1 Chapter Overview

The objective of the chapter is to see the power of short-horizon planning for unlabeled-MAPF, embodied as TSWAP, despite lacking long-horizon deliberation.

5.1.1 What is TSWAP

Similar to PIBT, *TSWAP is understood as a configuration generator for unlabeled-MAPF*. That is, given one configuration, TSWAP produces another configuration. By continuously generating configurations, TSWAP yields a sequence of configurations, constituting a solution. Different from PIBT, it is possible for this successive generation to converge to the goal configuration. Therefore, *TSWAP is complete* for unlabeled-MAPF. Note, in general, TSWAP is sub-optimal.

Mechanism

The structure of TSWAP is simple and easy to be implemented. Specifically, it first assigns each agent to one of the target locations, using arbitrary target assignment algorithms (e.g., greedy, bottleneck, or linear optimal assignments). Then, it repeatedly generates a sequence of configurations until the termination (i.e., all agents have reached targets) *while swapping targets between agents as necessary*.

5.1.2 Properties and Performance

Theoretical Properties

TSWAP is complete. Furthermore, it has low-cost time complexity like PIBT. The time complexity for one timestep is $O(A(\alpha + \beta))$, where A is agents, α is the time required for vertex evaluation, which can be constant with preprocessing. β denotes the time required

[1] To align with the dissertation structure, some parts of the paper will appear in Chap. 8. Moreover, this chapter is based on the journal submission currently under review.

to detect deadlocks. In the worst case, the detection is linear to $|A|$, however, this value is usually so small and can be negligible in practice. Consequently, TSWAP can quickly solve unlabeled-MAPF, even with thousands of agents.

TSWAP is also applicable to online planning assuming timing uncertainties of execution. This is another topic discussed in the execution part (Chap. 8).

Empirical Performance

TSWAP is scalable and can yield near makespan-optimal solutions while reducing runtime by orders of magnitude in most cases, compared to the polynomial-time optimal algorithm [Yu and LaValle, 2013a]. For instance, with an appropriate target assignment algorithm, TSWAP solved unlabeled-MAPF near-optimally in around a second on average, even with 2,000 agents in a large grid map (418×530; $|V| = 43,151$). Furthermore, TSWAP can generate good solutions with respect to *sum-of-costs*, another commonly used metric in MAPF studies.

5.1.3 Target Assignment with Lazy Evaluation

Assuming to use with TSWAP, the chapter also presents efficient assignment algorithms with the *lazy evaluation* of distances. In unlabeled-MAPF, costs for assignments are unknown initially and require distance evaluation, possibly being bottlenecks for quick planning. The lazy evaluation eliminates this overhead by evaluating the distance on demand. Specifically, two algorithms are presented. The first is bottleneck assignment [Gross, 1959] combined with the lazy evaluation, which minimizes the largest distance in the assignment and can generate makespan near-optimal solutions with TSWAP. The second is a greedy assignment followed by iterative refinement, aimed at addressing large instances in a very short time.

5.1.4 Original Motivation for TSWAP

Similar to PIBT, the development of TSWAP had another motivation, noted below.

Finding makespan-optimal solutions for unlabeled-MAPF is easier than for MAPF which is known to be NP-hard [Yu and LaValle, 2013b]. Indeed, unlabeled-MAPF has a polynomial-time optimal algorithm based on a reduction to maximum flow [Yu and LaValle, 2013a]. However, the size of the flow network is quadratic to the size of the original graph, making practical problems in large graphs (e.g., 500×500 grid) still challenging. Despite its importance, unlabeled-MAPF has received little attention compared to conventional MAPF, for which many scalable sub-optimal solvers have been developed, such as [Surynek, 2009; Wang and Botea, 2011; De Wilde *et al.*, 2014] as well as PIBT. Quick sub-optimal algorithms for unlabeled-MAPF are also attractive because it is possible to efficiently refine the solution quality for known MAPF solutions, as presented in Chap. 7.

To this end, we originally aimed at developing a centralized approach to solving large unlabeled-MAPF instances with sufficiently good quality in a small computation time. This motivation fruits as the TSWAP algorithm. However again, in Chap. 6, TSWAP will be regarded as a configuration generator.

5.1.5 Chapter Organization

- Chapter 5.2 presents the TSWAP algorithm and its theoretical analysis.
- Chapter 5.3 presents assignment algorithms with lazy evaluation.
- Chapter 5.4 presents empirical results.

- Chapter 5.5 reviews the studies closely related to TSWAP.
- Chapter 5.6 concludes the chapter.

The code and movie are available at https://kei18.github.io/tswap.

5.1.6 Notations and Assumptions

Notations in this chapter are summarized below:

$\bot$	not found, undefined
$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \ldots, n\}$	a tuple of agents
$\mathcal{S} = (s_1, s_2, \ldots, s_n)$	start configuration, where $s_i \in V$
$\mathcal{T} = \{g_1, g_2, \ldots, g_m\}$	target vertices, where $g_i \in V$
$\texttt{dist}$	$V \times V \mapsto \mathbb{N}$, function that returns shortest path length
$\texttt{h}$	$V \times V \mapsto \mathbb{R}$, admissible heuristic function
$\mathcal{Q}$	configuration
$\mathcal{M}$	$A \mapsto \mathcal{T}$, bijective function, assignment (matching)
$\texttt{diam}(G)$	diameter of G

Unless explicitly mentioned, the chapter assumes the number of targets is equal to the size of agents (i.e., $|\mathcal{T}| = |A|$). Furthermore, the chapter assumes admissible heuristics $\texttt{h}(u, v)$ for computing the shortest path length in constant time, i.e., $\texttt{h}(u, v) \leq \texttt{dist}(u, v)$, such as the Manhattan distance.

Caution

The chapter uses the representation by configurations.

5.2 The TSWAP Algorithm

This section presents the *TSWAP* algorithm. Essentially, TSWAP generates a pair of a configuration and an assignment, given another pair. Here, an *assignment* is a bijective function $\mathcal{M} : A \mapsto \mathcal{T}$. For convenience, the chapter uses a list representation $\mathcal{M}[i]$, instead of $\mathcal{M}(i)$. The section begins with descriptions of the concept behind the algorithm.

5.2.1 Concept

A vanilla approach to unlabeled-MAPF is to disjoint target assignment and path planning parts. That is, such approaches first obtain an assignment by off-the-shelf assignment algorithms. Then, taking the assignment as a goal configuration, they solve the MAPF problem with off-the-shelf MAPF algorithms. However, this type of approach potentially fails because the initial assignment might generate an unsolvable MAPF instance.[2] This is a critical pitfall since unlabeled-MAPF is always solvable (see Chap. 3.4.2). Therefore, we cannot apply MAPF solvers directly to design a complete unlabeled-MAPF algorithm.

The above issue is resolved by *swapping assigned targets as necessary* during path planning. TSWAP realizes this concept, with five simple rules as described below. The rules are also illustrated in Fig. 5.1.

- **rule-1, stay in target**: An agent remains in the current location if the agent is at the assigned target.

[2] For instance, see an assignment of Fig. 5.2 at $t = 0$.

- **rule-2, move toward target**: If the nearest neighboring vertex towards the assigned target is unoccupied, the agent moves there.
- **rule-3, swap targets**: If the nearest vertex is occupied by another agent that is at its assigned target, two agents swap assigned targets.
- **rule-4, resolve deadlocks**: If two or more agents form a deadlock about the next locations, rotate targets between agents and resolve the deadlock. The formulation of deadlock appears later.
- **rule-5, do nothing**: Otherwise, remain there.

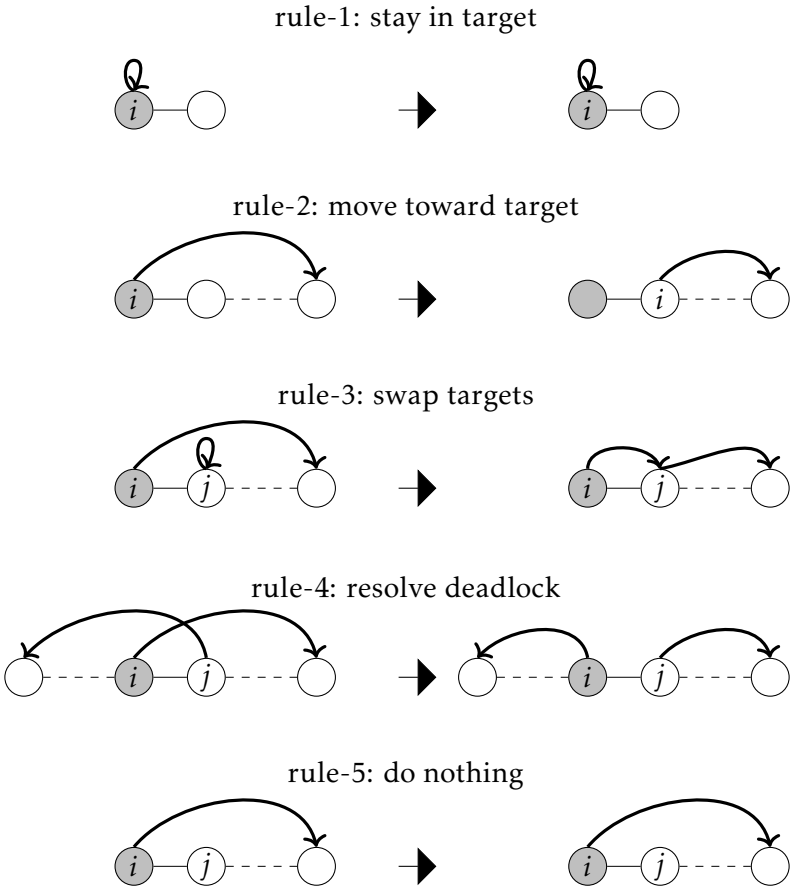


Figure 5.1: Rules constituting TSWAP. "Before" and "after" configurations are depicted on the left and right, respectively. Bold arrows represent assigned targets. Dashed lines correspond to the shortest paths for assigned targets of each agent. One vertex is filled with gray for visualization.

5.2.2 Minimum Implementation

Algorithm 5.1 presents the pseudocode of minimum TSWAP, akin to the minimum implementation of PIBT (Alg. 4.1). Using the five rules in Fig. 5.1, Alg. 5.1 generates a pair of a configuration and an assignment, given another pair.

Several details of Alg. 5.1 are complemented below.

Vertex Evaluation (Line 4). The `next_vertex` function is defined as follows.

$$\texttt{next\_vertex}(u,v) := \underset{w \in \texttt{neigh}(u)}{\text{argmin}}\ \texttt{dist}(w,v) \tag{5.1}$$

Algorithm 5.1 Minimum TSWAP (function TSWAP)

input: configuration $\mathcal{Q}^{\text{from}}$, assignment $\mathcal{M}^{\text{from}}$, agents A

output: configuration $\mathcal{Q}^{\text{to}}$, assignment $\mathcal{M}^{\text{to}}$ (each element is initialized with $\perp$)

1: $\mathcal{Q}^{\text{to}} \leftarrow \texttt{copy}\left(\mathcal{Q}^{\text{from}}\right)$; $\mathcal{M}^{\text{to}} \leftarrow \texttt{copy}\left(\mathcal{M}^{\text{from}}\right)$
2: **for** $i \in A$ **do**
3: **if** $\mathcal{Q}^{\text{to}}[i] = \mathcal{M}^{\text{to}}[i]$ **then continue** ▷ *rule-1*
4: $u \leftarrow \texttt{next\_vertex}\left(\mathcal{Q}^{\text{to}}[i], \mathcal{M}^{\text{to}}[i]\right)$
5: let j be an agent in A s.t. $\mathcal{Q}^{\text{to}}[j] = u$
6: **if** $j = \perp$ **then** $\mathcal{Q}^{\text{to}}[i] \leftarrow u$; **continue** ▷ *rule-2*
7: **if** $\mathcal{M}^{\text{to}}[j] = u$ **then** $\mathcal{M}^{\text{to}}[i], \mathcal{M}^{\text{to}}[j] \leftarrow \mathcal{M}^{\text{to}}[j], \mathcal{M}^{\text{to}}[i]$; **continue** ▷ *rule-3*
8: **if** $\exists$ deadlock for $A' \subseteq A \wedge i \in A'$ **then** rotate $\mathcal{M}^{\text{to}}$ of A'; **continue** ▷ *rule-4*
(otherwise; do nothing) ▷ *rule-5*
9: **return** $\mathcal{Q}^{\text{to}}$, $\mathcal{M}^{\text{to}}$

This function is assumed to be deterministic, i.e., tie-break between vertices having the same scores is done deterministically. The function can be implemented, e.g., via the shortest pathfinding algorithms.

Deadlock (Line 8). A deadlock is defined as follows. Let denote u_i a shorthand notation of $\texttt{next\_vertex}\left(\mathcal{Q}^{\text{to}}[i], \mathcal{M}^{\text{to}}[i]\right)$. A set of agents $A' = (i, j, k, \ldots, l)$ is in a *deadlock* when $u_i = \mathcal{Q}^{\text{to}}[j] \wedge u_j = \mathcal{Q}^{\text{to}}[k] \wedge \ldots \wedge u_l = \mathcal{Q}^{\text{to}}[i]$. When detecting a deadlock for A', the algorithm "rotates" targets. That is, $\mathcal{M}^{\text{to}}[i] \leftarrow \mathcal{M}^{\text{to}}[l], \mathcal{M}^{\text{to}}[j] \leftarrow \mathcal{M}^{\text{to}}[i], \mathcal{M}^{\text{to}}[k] \leftarrow \mathcal{M}^{\text{to}}[j], \ldots$. The detection is done by incrementally checking whether the next location of each agent is occupied by another agent and concurrently checking the existence of a loop.

Analysis of Local Movements

To analyze Alg. 5.1, consider the following function, defined by a configuration $\mathcal{Q}$ and an assignment $\mathcal{M}$.

$$\phi(\mathcal{Q}, \mathcal{M}) := \sum_{i \in A} \left[\texttt{dist}\left(\mathcal{Q}[i], \mathcal{M}[i]\right) + \left| \left\{ j \in A \mid \mathcal{M}[j] \in \mathcal{P}(\mathcal{Q}[i], \mathcal{M}[i]) \right\} \right| \right] \tag{5.2}$$

Here, $\mathcal{P}(u, v) \subset V$ is a set of vertices in the shortest path from $u \in V$ to $v \in V$, identified by next_vertex, except for endpoints u and v. Observe that $\phi(\mathcal{Q}, \mathcal{M}) \geq 0$ and ϕ becomes zero only when $\mathcal{Q}$ is equal to $\mathcal{M}$. Since $\mathcal{M}$ is a bijective function with reference to targets, ϕ is seen as a *potential function* that evaluates how close $\mathcal{Q}$ is to the terminal configuration of unlabeled-MAPF.

Lemma 5.1 (local movement of TSWAP)**.** *With Alg. 5.1, $\phi\left(\mathcal{Q}^{to}, \mathcal{M}^{to}\right)$ is smaller than $\phi\left(\mathcal{Q}^{from}, \mathcal{M}^{from}\right)$. Otherwise, $\mathcal{Q}^{from}$ is equal to $\mathcal{M}^{from}$, i.e., $\phi\left(\mathcal{Q}^{from}, \mathcal{M}^{from}\right) = 0$.*

Proof. It is trivial to see ϕ is non-increasing by each operation of Alg. 5.1. Therefore, the remaining part proves that ϕ decreases when $\phi\left(\mathcal{Q}^{\text{from}}, \mathcal{M}^{\text{from}}\right) > 0$ by contradiction.

Suppose that ϕ does not differ between $\langle \mathcal{Q}^{\text{from}}, \mathcal{M}^{\text{from}} \rangle$ and $\langle \mathcal{Q}^{\text{to}}, \mathcal{M}^{\text{to}} \rangle$. Then, let $B := \left\{ i \in A \mid \mathcal{Q}^{\text{from}}[i] \neq \mathcal{M}^{\text{from}}[i] \right\}$. This is non-empty because $\phi\left(\mathcal{Q}^{\text{from}}, \mathcal{M}^{\text{from}}\right) \neq 0$, i.e., there are agents not on their targets.

We now confirm each operation. First, there is no swap operation by Line 7; otherwise, the second term of ϕ must decrease. Second, all agents in B do not move, i.e., $\mathcal{Q}^{\text{from}}[i] = \mathcal{Q}^{\text{to}}[i]$ for all $i \in B$; otherwise, the first term of ϕ must decrease. Furthermore, for an agent $i \in B$, $\texttt{next\_vertex}\left(\mathcal{Q}^{\text{from}}[i], \mathcal{M}^{\text{to}}[i]\right)$, say u_i, must be occupied by another agent $j \in B$ in $\mathcal{Q}^{\text{to}}$; otherwise, i moves to u_i. This is the same for j, i.e., there is an agent $k \in B$ such that $\mathcal{Q}^{\text{to}}[k] = u_j$. By induction, this sequence of agents must form a deadlock somewhere; however, by deadlock detection and resolution in Line 8, the first term of ϕ must decrease. Hence, this is a contradiction. □

Lemma 5.1 is interpreted as, by each applying of Alg. 5.1, the resulting configuraiton approaches the terminal configuraiton.

5.2.3 Complete Algorithm

We now see a complete algorithm for unlabeled-MAPF, build upon Alg. 5.1. Algorithm 5.2 is it. After getting an initial assignment by external algorithms (Line 1), for each timestep until all targets are occupied, either of the five rules is applied to each agent and determines the location for the next timestep (Lines 3–5). These operations bring the agents to the targets. Figure 5.2 shows a running example.

Algorithm 5.2 TSWAP.

input: unlabeled-MAPF instance $(A, G, \mathcal{S}, \mathcal{T})$

output: solution $\boldsymbol{\Pi}$

1: get an initial assignment $\mathcal{M}$
2: $\boldsymbol{\Pi}[0] \leftarrow \mathcal{S}$
3: **for** $t = 0, 1, 2, \ldots$ **do**
4: **if** $\boldsymbol{\Pi}[t] = \mathcal{M}$ **then return** $\boldsymbol{\Pi}$
5: $\boldsymbol{\Pi}[t+1], \mathcal{M} \leftarrow \texttt{TSWAP}(\boldsymbol{\Pi}[t], \mathcal{M})$ ▷ *Alg. 5.1*

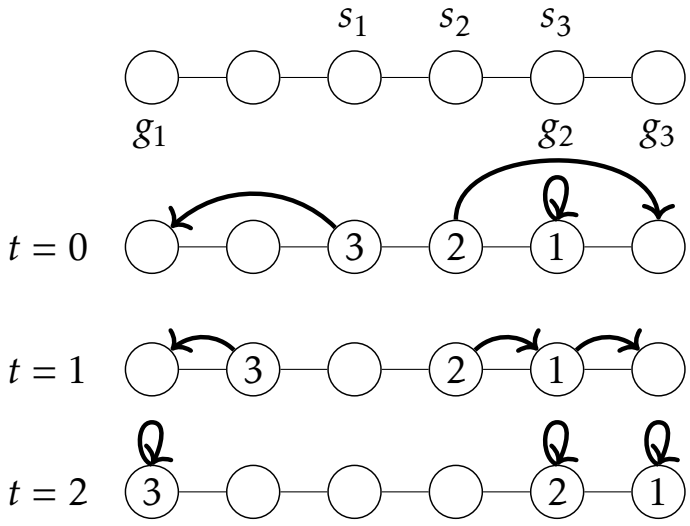


Figure 5.2: Running example of TSWAP. An unlabeled-MAPF instance is shown at the top. Current locations of agents, i.e., $\Pi[t][i]$, are shown within vertices. The assigned target is illustrated with bold arrows. A target swapping happens between agent-1 and agent-2 at $t = 0$ (Line 7 in Alg. 5.1).

5.2.4 Theoretical Analyses

Completeness

Theorem 5.2. *TSWAP (Alg. 5.2) is complete for unlabeled-MAPF (Def. 3.7).*

Proof. This is trivial from Lemma 5.1. □

Any assignment algorithms can be applied to TSWAP because Thrm. 5.2 does not rely on initial assignments. Furthermore, TSWAP can easily adapt to unlabeled-MAPF instances such that the number of targets is less than that of agents (i.e., $|A| > |\mathcal{T}|$) by assigning agents without targets to any non-target locations.

Solution Quality

Proposition 5.3. *TSWAP has upper bounds of;*

- *makespan:* $O(A \cdot \texttt{diam}(G))$
- *sum-of-costs (aka. flowtime):* $O(A^2 \cdot \texttt{diam}(G))$
- *maximum-moves:* $O(A \cdot \texttt{diam}(G))$
- *sum-of-moves (aka. sum-of-fuels):* $O(A \cdot \texttt{diam}(G))$

Proof. The potential function ϕ of Eq. (5.2) is $O(A \cdot \texttt{diam}(G))$. This is the makespan upper bound. Note that the second term of ϕ is bounded by $\texttt{diam}(G)$ because $\mathcal{P}$ assumes the shortest paths on G. That of sum-of-costs is trivially obtained by multiplying $|A|$.

To derive the upper bound of sum-of-moves, consider another potential function $\psi(\mathcal{Q}, \mathcal{M}) := \sum_{i \in A} \texttt{dist}(\mathcal{Q}[i], \mathcal{M}[i])$. Similarly to ϕ, ψ is non-increasing. ψ becomes zero when the problem is solved, i.e., $\mathcal{Q} = \mathcal{M}$. Furthermore, with the call of `TSWAP` (Alg. 5.1), ψ is decremented in a new configuration by each "move" action at Line 6. Consequently, ψ eventually reaches zero. Observe that $\psi = O(A \cdot \texttt{diam}(G))$; this is the upper bound of sum-of-moves. This bound works also for maximum-moves. □

Compared to sum-of-costs, the upper bound of sum-of-moves is significantly reduced because it ignores all "wait" actions. The bound on sum-of-moves is tight in some scenarios, such as a line graph with all agents starting on one end and all targets on the opposite end. Furthermore, TSWAP is optimal for sum-of-moves depending on the initial assignment.

Proposition 5.4. *TSWAP is optimal for sum-of-moves when the initial assignment $\mathcal{M}$ minimizes $\sum_{i \in A} \texttt{dist}(s_i, g_i)$.*

Proof. Because ψ of Eq. (5.2) never increases during Alg. 5.2. □

In general, the upper bound on makespan is greatly overestimated. Later, we will see that TSWAP yields near-optimal solutions for makespan depending on initial assignments through the experiments. On the other hand, there are adversarial instances of TSWAP.

Observation 5.5. *There is an instance such that TSWAP requires $\Omega(A)$ makespan regardless of initial assignments, while the optimal makespan is constant for A.*

Proof. Figure 5.3 shows an adversarial instance for TSWAP. The makespan-optimal solution is to make all agents just move down (3 steps). However, with TSWAP, regardless of initial assignments, all agents use the rightmost vertex because it makes agents move along the shortest paths. As a result, its makespan is $|A| + 1$. □

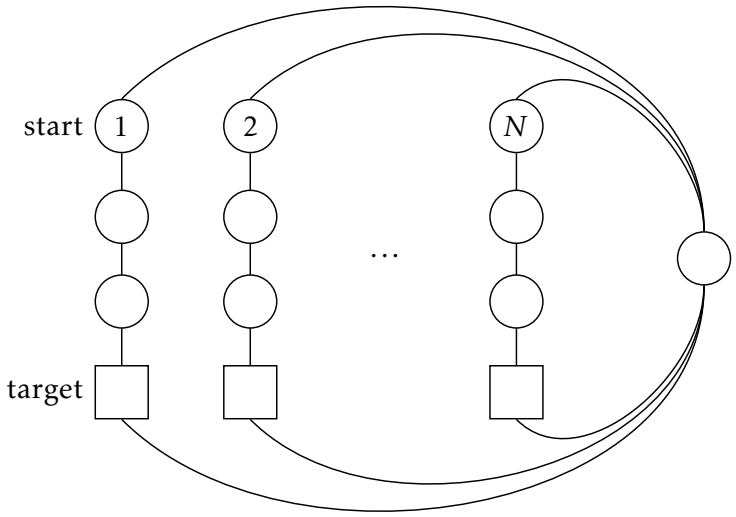


Figure 5.3: Adversarial instance for TSWAP.

In Fig. 5.3, the sum-of-costs of TSWAP is $2+3+\ldots+(N+1) = N(N+3)/2 = \Theta(A^2)$ while the optimal case is $3|A|-1 = \Theta(A)$. We will later see a similar adversarial instance in the experiment of Chap. 5.4.5.

Time Complexity

Proposition 5.6. *Assume that the time complexity of* `next_vertex` *and the deadlock resolution (Line 8) in Alg. 5.1 are α and β,*[3] *respectively. Then, the time complexity of the function* `TSWAP` *(Alg. 5.1) is $O(A(\alpha+\beta))$.*

Proof. Each operation in Lines 2–8 is constant except for Line 4, $O(\alpha)$, and Line 8, $O(\beta)$. These operations repeat exactly $|A|$ times for each timestep, thus deriving the statement. □

Proposition 5.7. *The time complexity of TSWAP (Alg. 5.2) excluding Line 1 is $O(A^2 \cdot \mathtt{diam}(G) \cdot (\alpha+\beta))$.*

Proof. According to Prop. 5.3, the makespan is $O(A \cdot \mathtt{diam}(G))$, i.e., the repetition number of Lines 3–5. The statement is then derived by multiplying the complexity obtained from Prop. 5.6. □

From Prop. 5.7, TSWAP has an advantage in large fields compared to the time complexity $O(AV^2)$ of the makespan-optimal algorithm [Yu and LaValle, 2013a] with a natural assumption that $E = O(V)$. Note that, depending on implementations, α can be constant once the shortest paths between starts and the initial targets are obtained because agents follow these paths even with target swapping.

[3] β is treated as blackbox, but it can be obviously implemented by $O(A \cdot \alpha)$. Note that this is too conservative in practice, i.e., deadlocks with many agents are rare.

TSWAP on General Graphs

Not limited to connected undirected graphs, TSWAP is complete for general graphs such as digraphs or non-connected graphs, when the initial assignment is feasible. The assignment is *feasible* when, for each agent, there is a path on the graph from its start to the assigned target. If so, TSWAP works without modifications.

5.2.5 Effective Implementation

Next, an "efficient" implementation of TSWAP is presented to obtain better solutions than those produced by a vanilla TSWAP (Alg. 5.2). Recall that TSWAP applies the five rules to each agent sequentially. This order affects the solution quality, illustrated as follows.

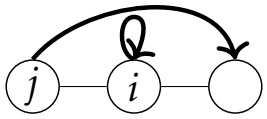


Figure 5.4: Example that a planning order affects makespan.

Consider Fig. 5.4 as a motivating example:

- **case-1**: Assume that j plans its next location prior to i. The makespan is two; in the first timestep, the rule-3 "swap targets" is applied to j, then i moves to the updated target (i.e., one step right). In the next timestep, j reaches the swapped target.
- **case-2**: Assume that an agent i plans its next location prior to j. The makespan is again two; in the first timestep, since i is on its assigned target, the rule-1 "stay in target" is applied to i. Then the rule-3 "swap targets" is applied to j. In the next timestep, both agents move to their targets.

However, the optimal makespan is clearly one; in the first timestep, make both agents move one step to the right. In what follows, methodologies to overcome this pitfall are provided.

Algorithm 5.3 is an extended version of TSWAP. The main difference from Alg. 5.2 is the use of queue $\mathcal{U}$ (Line 5; taken from "undecided"), instead of naive sequential planning (i.e., for-loop). This trick realizes a flexible planning order of TSWAP. We explain it as follows.

For each timestep, as long as the queue $\mathcal{U}$ is not empty, the algorithm selects one agent i (Line 7) and applies the condition-matched rules to i:

1. If the rule-1 "stay in target" is *not* applied (Line 9–), i tries to move to the desired vertex u.
2. When u will be occupied by another agent j in the next timestep $t+1$, then i stays in its current location (rule-5 "do nothing"; Line 10).
3. Otherwise, u is unoccupied in the next timestep. i can move to u only when *(i)* there is no agent at u in the current timestep, or, *(ii)* there is another agent j at u but j will move to another vertex. If either condition holds, the rule-2 "move toward target" is applied (Lines 11–12).
4. If the above rules are not applied, i *skips* determining the next location by inserting i to the queue $\mathcal{U}$. Before that, rule-3 "swap targets" and rule-4 "resolve deadlock" are sequentially applied if the conditions are matched.

Algorithm 5.3 TSWAP with flexible planning order.

input: unlabeled-MAPF instance $(A, G, \mathcal{S}, \mathcal{T})$

output: solution $\boldsymbol{\Pi}$

preface: $\Pi[t][i]$ is initialized with $\perp$

1: get an initial assignment $\mathcal{M}$
2: $\boldsymbol{\Pi}[0] \leftarrow \mathcal{S}$
3: **for** $t = 0, 1, 2, \ldots$ **do**
4: **if** $\boldsymbol{\Pi}[t] = \mathcal{M}$ **then return** $\boldsymbol{\Pi}$
5: initialize $\mathcal{U}$ with A ▷ *queue*
6: **while** $\mathcal{U} \neq \emptyset$ **do**
7: $i \leftarrow \mathcal{U}$.pop()
8: **if** $\Pi[t][i] = \mathcal{M}[i]$ **then** $\Pi[t+1][i] \leftarrow \Pi[t][i]$; **continue** ▷ *rule-1*
9: $u \leftarrow \texttt{next\_vertex}(\Pi[t][i], \mathcal{M}[i])$
10: **if** $\exists j, \Pi[t+1][j] = u$ **then** $\Pi[t+1][i] \leftarrow \Pi[t][i]$; **continue** ▷ *rule-5*
(u *is unoccupied at* $t+1$)
11: **if** $\left(\nexists j, \Pi[t][j] = u\right) \vee \left(\exists j, \Pi[t][j] = u \wedge \Pi[t+1][j] \neq \perp\right)$ **then**
12: $\Pi[t+1][i] \leftarrow u$; **continue** ▷ *rule-2*
13: **if** $\exists j$ s.t. $\Pi[t][j] = u \wedge \Pi[t][j] = \mathcal{M}[j]$ **then** $\mathcal{M}[i], \mathcal{M}[j] \leftarrow \mathcal{M}[j], \mathcal{M}[i]$ ▷ *rule-3*
14: **if** $\exists$deadlock for $A' \subseteq A \wedge i \in A'$ **then** rotate $\mathcal{M}$ of A' ▷ *rule-4*
15: $\mathcal{U}$.push(i)

With Alg. 5.3, the case-1 (j plans first) of Fig. 5.4 is resolved. The running example is described as follows.

1. Assume that $\mathcal{U}$ is initialized with $[\![j, i]\!]$. In the first iteration of Lines 6–15, j is popped from $\mathcal{U}$ and the rule-3 "swap goals" is applied (Line 13). Then j is re-inserted to $\mathcal{U}$. $\mathcal{U}$ is now $[\![i, j]\!]$.
2. In the second iteration, i is popped. The rule-2 "move toward target" (Line 12) is applied. $\mathcal{U}$ is now $[\![j]\!]$.
3. Lastly, j is popped again then the rule-2 "move toward target" (Line 12) is applied.

These operations are the planning within one timestep. Consequently, the makespan becomes one, rather than two.

Proposition 5.8. *Algorithm 5.3 terminates.*

Proof. It is proven that $\mathcal{U}$ eventually becomes empty with operations in Lines 6–15. This is done by using the same potential function ϕ as the proof of Lemma 5.1. Assume, by contradiction, that $\mathcal{U}$ does not change for $|A|$ iterations. Meanwhile, ϕ must decrease by the same logic as the proof of Lemma 5.1; hence this is a contradiction. □

The case-2 (i plans first) of Fig. 5.4 can be avoided by using a priority queue rather than just a queue. The modification counts how many times the agent is called in the iterations and uses this score in the ascending order in this priority queue. This prevents

an eternal loop. The tie-break is done by giving the priority to agents that are not on their target, i.e., j in Fig. 5.4. Consequently, the planning by TSWAP for Fig. 5.4 results in a solution with the makespan one.

5.3 Target Assignment with Lazy Distance Evaluation

Although TSWAP works with an arbitrary initial assignment, path planning is significantly affected by the assignment. Ideal assignment algorithms are quick, scalable, and with reasonable quality for solution metrics (e.g., makespan). It is possible to apply conventional assignment algorithms such as the Hungarian algorithm [Kuhn, 1955]. However, costs (i.e., distances) for each start-target pair are unknown initially, which is typically computed via breadth-first search with time complexity $O(A(V+E))$. This would be a non-negligible overhead. This section thus presents two examples that efficiently solve target assignment with *lazy distance evaluation*, which avoids exhaustive distance evaluation.

5.3.1 Bottleneck Assignment

Algorithm 5.4 aims to minimize makespan by solving the bottleneck assignment problem [Gross, 1959], i.e., assign each agent to one target while minimizing the maximum cost, regarding distances between initial locations and targets as costs. A running example is shown in Fig. 5.5.

Algorithm 5.4 Bottleneck assignment.

input: unlabeled-MAPF instance $(A, G, \mathcal{S}, \mathcal{T})$

output: assignment $\mathcal{M}$

1: initialize $\mathcal{M}$; Let $\mathcal{B}$ be a bipartite graph $(A, \mathcal{T}, \emptyset)$
2: *Open*: priority queue of $\langle i \in A, g \in \mathcal{T}, \text{real distance}, \text{estimated distance}\rangle$
   in increasing order of distance (use real one if exists, otherwise use estimated one)
3: **for** $i \in A, g \in \mathcal{T}$ **do**
4: $\quad$ *Open*.push$\Big(\langle s_i, g, \bot, \mathrm{h}(s_i, g)\rangle\Big)$
5: **while** *Open* $\neq \emptyset$ **do**
6: $\quad \langle i, g, d_{\text{real}}, d_{\text{est}}\rangle \leftarrow$ *Open*.pop()
7: $\quad$ **if** $d_{\text{real}} = \bot$ **then** *Open*.push$\Big(\langle i, g, \mathtt{dist}(s_i, g), d_{\text{est}}\rangle\Big)$; **continue**
8: $\quad$ add a new edge (i, g) to $\mathcal{B}$
9: $\quad$ update $\mathcal{M}$ by finding an augmenting path on $\mathcal{B}$
10: $\quad$ **if** $|\mathcal{M}| = |\mathcal{T}|$ **then**
11: $\quad\quad$ †optional: add all $\langle \cdot, \cdot, d, \cdot\rangle \in$ *Open* to $\mathcal{B}$ s.t. $d = d_{\text{real}}$
12: $\quad\quad$ **break**
13: †optional: $\mathcal{M} \leftarrow$ minimum cost maximum matching on $\mathcal{B}$
14: **return** $\mathcal{M}$

The algorithm incrementally adds pairs of an agent and a target to a bipartite graph $\mathcal{B}$ (Line 8), in increasing order of their distances using a priority queue *Open*. $\mathcal{B}$ is initialized as $(A, \mathcal{T}, \emptyset)$ (Line 1). This iteration continues until all targets are matched to agents

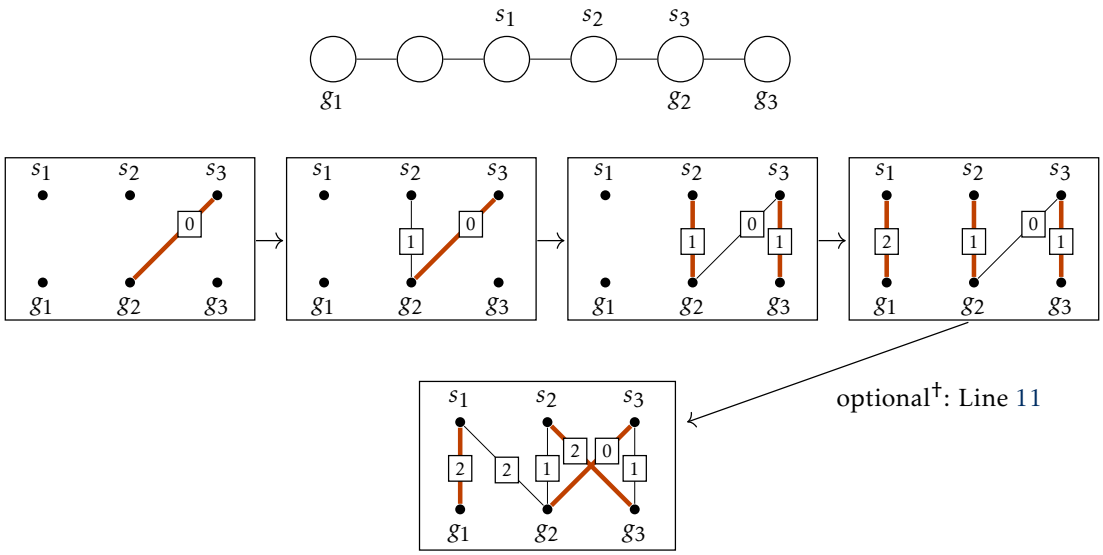


Figure 5.5: Running example of the bottleneck assignment (Alg. 5.4†). An unlabeled-MAPF instance is shown at the top. The target assignment is illustrated using a bipartite graph $\mathcal{B}$. The assignment $\mathcal{M}$ is denoted by red lines. The middle part, with annotation of costs, corresponds to finding the bottleneck cost (Lines 4–12), i.e., incrementally adding pairs of an initial location and a target and then updating the matching. The bottom part corresponds to solving the minimum cost maximum matching problem (Line 13). Two edges are added from the last situation due to Line 11. The final outcome is equivalent to the assignment of Fig. 5.2.

(Line 10). At each iteration, the maximum bipartite matching problem on $\mathcal{B}$ is solved (Line 9). In general, the Hopcroft-Karp algorithm [Hopcroft and Karp, 1973] efficiently solves this problem in $O(\sqrt{V'}E')$ runtime for any bipartite graph (V', E'), but Alg. 5.4 uses the reduction to the maximum flow problem and the Ford-Fulkerson algorithm [Ford and Fulkerson, 1956]. The basic concept of this algorithm is finding repeatedly an *augmenting path*, i.e., a path from source to sink with available capacity on all edges in the path, then making the flow along that path. Such paths are found, e.g., via depth-first or breadth-first search. Here, finding a single augmenting path in $O(E')$ runtime is sufficient to update the matching because the number of matched pairs increases at most once for each adding.

The algorithm uses *lazy evaluation* of real distance (Line 7). This is realized by the use of priority queue *Open* (Line 2) and admissible heuristics h (Line 4), then evaluating the real distance as needed. $\perp$ denotes that the corresponding real distance has not been evaluated yet. The lazy evaluation contributes to the speedup of the target assignment, as we will see later.

The algorithm *optionally* solves the minimum cost maximum matching problem (Line 13), aiming at improving the sum-of-costs metric. The problem can be solved by reducing to the minimum cost maximum flow problem and then using the successive shortest path algorithm [Ahuja *et al.*, 1993], a generalization of the Ford-Fulkerson algorithm that uses Dijkstra's shortest path algorithm [Dijkstra, 1959], to find an augmenting path with minimum cost. Its time complexity is $O(f(E' + V' \lg V'))$ where f is the maximum flow size and V' and E' represent the network; hence $O\left(A^3\right)$. Note that when finding the bottleneck cost, all edges in *Open* with their costs equal to the bottleneck cost are added to $\mathcal{B}$ to improve the sum-of-costs metric of the assignment (Line 11). This operation includes lazy evaluation similar to the main loop (Lines 5–12). The corresponding algorithm is denoted as Alg. 5.4†.

Proposition 5.9. *The time complexity of Alg. 5.4(†) is* $O\left(\max\left(A(V+E), A^4\right)\right)$.

Proof. Consider the worst case, i.e., all agent-target pairs are evaluated and contained in $\mathcal{B}$. The number of vertices and edges of $\mathcal{B}$ are $2|A|$ and $|A|^2$, respectively. Then, Line 13 is $O\left(A^3\right)$ by the successive shortest path algorithm because its time complexity is $O(f(E'+V'\lg V'))$ where f is the maximum flow size and V' and E' represent the network. The total operations of `dist` become running the breadth-first search $|A|$ times, therefore, $O(A(V+E))$. Operations for a priority queue *Open* are both $O(\lg n)$ for extracting and inserting, where n is the length of the queue. Thus, the runtime of Line 4 is $O\left(A^2 \lg A\right)$. The queue operations in Lines 5–12 require $O\left(A^2 \lg A\right)$. Line 9 finds a single augmenting path and this is linear for the number of edges in $\mathcal{B}$, thus, its complexity is $1+2+\cdots+\left|A^2\right| = O\left(A^4\right)$. As the result, the complexity of Alg. 5.4 is;

$$
\begin{aligned}
& O(A(V+E)) && \text{finding shortest path} \\
+ & O\left(A^3\right) && \text{min-cost maximum matching} \\
+ & O\left(A^2 \lg A\right) && \text{queue operations} \\
+ & O\left(A^4\right) && \text{update matching}
\end{aligned}
$$

□

5.3.2 Greedy Assignment with Refinement

Since TSWAP is sub-optimal in general, it makes sense to use sub-optimal assignment algorithms. Algorithm 5.5 aims at finding a sub-optimal but reasonable assignment for makespan as quickly as possible, which uses;

- *Greedy assignment* (Lines 1–9) assigns one target to one agent step by step, while allowing reassignment if a better assignment will be expected (Lines 8–9).
- *Iterative refinement* (Lines 10–15) swaps targets of two agents until no improvements are detected.
- *Lazy evaluation* of distances for start-target pairs, implemented by pausing the breadth-first search as soon as the query start-target pair is in the search tree, similar to reverse resumable A* [Silver, 2005].

A running example is shown in Fig. 5.6. The correctness of Alg. 5.5 is proven as follows.

Proposition 5.10. *Algorithm 5.5 is correct; returns a distinct target for each agent.*

Proof. We focus on the initial assignment phase (Lines 1–9) because the refinement phase (Lines 10–15) only swaps targets for the existing assignment $\mathcal{M}$. Moreover, it terminates within finite iterations because each target for each agent is evaluated at most once. Trivially, each assignment operation never assigns one target to more than one agent. Observe that $|\mathcal{U}|+|\mathcal{M}| = |A|$ is invariant. Thus, the output of the algorithm is correct.

The termination of the algorithm is derived by contradiction. Assume the invalid state of the algorithm, namely, Line 5 does not have any corresponding values, meaning that all targets have already been evaluated. This violates the invariance of $|\mathcal{U}|+|\mathcal{M}| = |A|$; hence such states never are realized. For each agent, no target is evaluated more than once. Therefore, the algorithm eventually terminates. □

This algorithm is expected to run in a very short time;

Algorithm 5.5 Greedy assignment with refinement for makespan.

input: unlabeled-MAPF instance $(A, G, \mathcal{S}, \mathcal{T})$

output: assignment $\mathcal{M}$

1: initialize $\mathcal{M}$; initialize queue $\mathcal{U}$ by A
2: **while** $\mathcal{U} \neq \emptyset$ **do**
3: $\quad i \leftarrow \mathcal{U}.\mathrm{pop}()$
4: $\quad$ **while** TRUE **do**
5: $\quad\quad g \leftarrow$ the non-evaluated nearest target from s_i
6: $\quad\quad$ **if** $\nexists (j, g) \in \mathcal{M}$ **then**
7: $\quad\quad\quad$ add (i, g) to $\mathcal{M}$; **break**
8: $\quad\quad$ **else if** $\exists (j, g) \in \mathcal{M} \wedge \mathtt{dist}(s_i, g) < \mathtt{dist}(s_j, g)$ **then**
9: $\quad\quad\quad$ replace $(j, g) \in \mathcal{M}$ by (i, g); $\mathcal{U}.\mathrm{push}(j)$; **break**
10: **while** $\mathcal{M}$ is updated in the last iteration **do**
11: $\quad (i, g_i) \leftarrow \underset{(k,g)\in\mathcal{M}}{\mathrm{argmax}}\ \mathtt{dist}(s_k, g)$; $c_{\mathrm{now}} \leftarrow \mathtt{dist}(s_i, g_i)$
12: $\quad$ **for** $(j, g_j) \in \mathcal{M}$ **do**
13: $\quad\quad$ **if** $\mathrm{h}(s_j, g_i) \geq c_{\mathrm{now}}$ **then continue** ▷ *for lazy evaluation*
14: $\quad\quad c_{\mathrm{swap}} \leftarrow \max\big(\mathtt{dist}(s_j, g_i), \mathtt{dist}(s_i, g_j)\big)$
15: $\quad\quad$ **if** $c_{\mathrm{swap}} < c_{\mathrm{now}}$ **then** swap g_i and g_j of $\mathcal{M}$; **break**
16: **return** $\mathcal{M}$

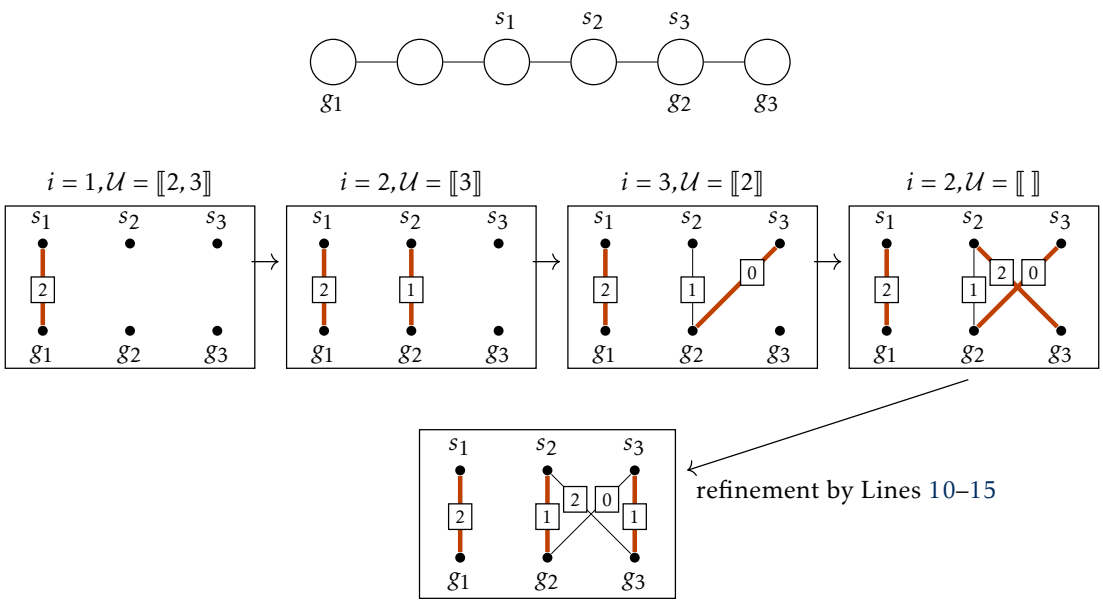


Figure 5.6: Running example of the greedy assignment (Alg. 5.5). An unlabeled-MAPF instance is shown at the top. The assignment $\mathcal{M}$ is denoted by red lines. The middle part, with annotation of costs, corresponds to finding a greedy assignment (Lines 1–9). The bottom part corresponds to the refinement (Lines 10–15).

Proposition 5.11. *The time complexity of Alg. 5.5 is* $O(A(V+E))$.

Proof. In the worst case, the algorithm requires $O(A(V+E))$ in total to evaluate all distances of agent-target pairs by running the breadth-first search $|A|$ times. The operations in Lines 5–9 repeat at most $|A|^2$ times because each target for each agent is evaluated at most once. Lines 10–15 repeat at most $\texttt{diam}(G)$ times because each iteration must reduce the maximum cost of the assignment $\mathcal{M}$. Both Line 11 and Lines 12–15 are $O(A)$. As a result, the algorithm is $O\Big(A(V+E)+A^2+A\cdot\texttt{diam}(G)\Big)$, which equals to $O(A(V+E))$. □

Algorithm 5.5 presents the refinement for makespan. Its sum-of-costs version is straightforward, as presented in Alg. 5.6.

Algorithm 5.6 Refinement for sum-of-costs.

input: unlabeled-MAPF instance $(A, G, \mathcal{S}, \mathcal{T})$

output: assignment $\mathcal{M}$

1: execute Lines 1–9 of Alg. 5.5
2: **while** $\mathcal{M}$ is updated in the last iteration **do**
3: **for** $(i, g_i), (j, g_j) \in \mathcal{M}, i \neq j$ **do**
4: $c_{\text{now}} \leftarrow \texttt{dist}(s_i, g_i) + \texttt{dist}(s_j, g_j)$
5: **if** $\mathrm{h}(s_j, g_i) + \mathrm{h}(s_i, g_j) \geq c_{\text{now}}$ **then continue**
6: $c_{\text{swap}} \leftarrow \texttt{dist}(s_j, g_i) + \texttt{dist}(s_i, g_j)$
7: **if** $c_{\text{swap}} < c_{\text{now}}$ **then** swap g_i and g_j of $\mathcal{M}$; **break**
8: **return** $\mathcal{M}$

Proposition 5.12. *The time complexity of Alg. 5.6 is* $O\Big(A(V+E)+A^3\cdot \mathit{diam}(G)\Big)$.

Proof. Lines 2–7 repeat at most $|A|\cdot\texttt{diam}(G)$. Each iteration requires $O\left(A^2\right)$. The statement is derived together with the proof of Prop. 5.11. □

5.4 Evaluation

The experiments aim at demonstrating that TSWAP is efficient, i.e., it returns near-optimal solutions within a short time and scales well, depending on initial assignments. In particular, this section constitutes four parts:

- Chapter 5.4.1 assesses the effect of planning order in TSWAP. In other words, we compare a vanilla TSWAP (Alg. 5.2) and the extended version with flexible planning order (Alg. 5.3).
- Chapter 5.4.2 illustrates the effect of initial assignments including the proposed assignment algorithms (Alg. $5.4^{(\dagger)}$, Alg. 5.5, and Alg. 5.6).
- Chapter 5.4.3 compares TSWAP with the makespan-optimal polynomial-time algorithm [Yu and LaValle, 2013a].
- Chapter 5.4.4 assesses another metric, sum-of-costs.

Several four-connected grids were carefully picked up from the MAPF benchmark [Stern *et al.*, 2019] as a graph G, shown in Fig. 5.7. The simulator was developed in C++ and the experiments were run on a laptop with Intel Core i9 2.3 GHz CPU and 16 GB RAM.

For each setting, 50 instances were prepared while randomly generating starts and targets. Note that, as the number of agents increases, the optimal makespan decreases because initial locations and targets were set randomly. Implementation details of [Yu and LaValle, 2013a] are described in Appendix C.1. Throughout this section, the runtime evaluation of TSWAP includes both target assignment and path planning.

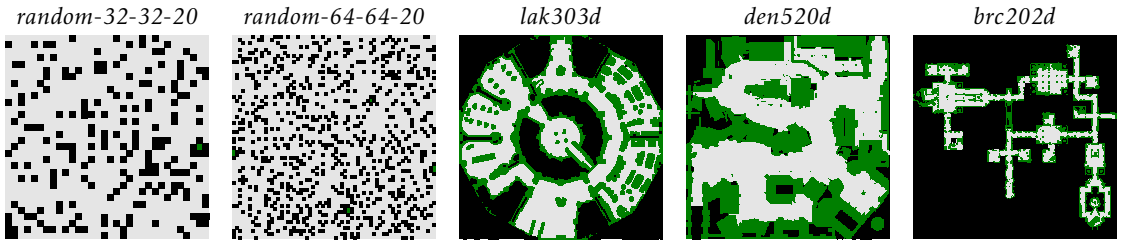


Figure 5.7: Used maps. $|V|$ is shown with parentheses.

5.4.1 Effect of Planning Order

At first, the effect of planning order in TSWAP was investigated. In particular, Alg. 5.2 (vanilla TSWAP) and Alg. 5.3 (TSWAP with flexible planning order) were compared, regarding the solution quality of makespan and sum-of-costs. To obtain initial assignments, both implementations used Alg. 5.4† that performs the bottleneck assignment minimizing maximum distance. Therefore, both implementations started with the exactly same initial assignments. The used map was *random-64-64-20* and *lak303d*.

Table 5.1 presents the result. Overall, Alg. 5.3 outperforms Alg. 5.2 especially when the instances become denser (i.e., with more agents); the solution quality of TSWAP is affected by the planning order. In what follows, the experiment used Alg. 5.3 as the implementation of TSWAP since it outputs better solutions.

5.4.2 Effect of Initial Target Assignment

Next, the effect of initial assignments on TSWAP was evaluated, while varying the number of agents $|A|$. Several assignment algorithms were tested: Alg. 5.4 (bottleneck; minimizing maximum distance), Alg. 5.4† (with min-cost maximum matching), Alg. 5.5 (greedy with refinement for makespan), Alg. 5.6 (for sum-of-costs), naive greedy assignment [Avis, 1983], and optimal linear assignment (minimizing total distances) solved by the successive shortest path algorithm [Ahuja *et al.*, 1993]. These assignment algorithms do not consider inter-agent collisions. The last two used distances for start-target pairs obtained by the breadth-first search as costs. To assess the effect of lazy evaluation, the adapted version of Alg. 5.4† and Alg. 5.5 without lazy evaluation were also tested. They are denoted as Alg. 5.4$^{\dagger *}$ and Alg. 5.5*.

Table 5.2 summarizes the results on *random-64-64-20* and *lak303d*. In summary;

- Algorithm 5.4 contributes to finding good solutions for makespan.
- Algorithm 5.4† significantly improves sum-of-costs.
- Algorithm 5.5 and Alg. 5.6 are blazing fast while solution qualities outperform those of the naive greedy assignment.
- The lazy evaluation speedups each assignment algorithm.
- The optimal linear assignment requires time because its time complexity is $O\left(A^3\right)$.

random-64-64-20

$\|A\|$	metric	vanilla (Alg. 5.2)	practical (Alg. 5.3)	improvement
110	makespan	18 (17,18)	17 (17,18)	5.6%
	sum-of-costs	951 (914,988)	936 (900,971)	1.6%
500	makespan	13 (12,13)	**11** (10,11)	15.4%
	sum-of-costs	2372 (2276,2463)	**2169** (2084,2252)	8.6%
1000	makespan	13 (12,13)	**9** (9,9)	30.8%
	sum-of-costs	3585 (3449,3723)	**2922** (2811,3029)	18.5%
2000	makespan	15 (14,16)	**7** (7,8)	53.3%
	sum-of-costs	5612 (5337,5878)	**3469** (3313,3615)	38.2%

lak303d

$\|A\|$	metric	vanilla (Alg. 5.2)	practical (Alg. 5.3)	improvement
100	makespan	89 (83,95)	89 (83,94)	0%
	sum-of-costs	3274 (3041,3495)	3264 (3038,3482)	0.3%
500	makespan	63 (58,68)	61 (56,66)	3.2%
	sum-of-costs	9406 (8558,10155)	9226 (8398,9958)	1.9%
1000	makespan	52 (47,55)	47 (44,50)	9.6%
	sum-of-costs	12494 (11559,13315)	11891 (11039,12654)	4.8%
2000	makespan	60 (55,66)	**49** (44,53)	18.3%
	sum-of-costs	20598 (18940,22120)	18183 (16819,19520)	11.7%

Table 5.1: Effect of planning order of TSWAP. 95% confidence intervals of the mean are also displayed, on which bold characters are based. The improvements of Alg. 5.3 over Alg. 5.2 are also shown, which are calculated based on the mean values.

5.4.3 Makespan-optimal Algorithm vs. TSWAP

This part is further divided into two: *(i)* assessing scalability for both G and A, and *(ii)* testing the solvers in large graphs.

Scalability

Figure 5.8 displays the average runtime and makespan of "quadrupling" the size of G, i.e., those of *random-64-64-20*, regarding the results of *random-32-32-20* as a baseline. The main observations are:

- TSWAP quickly yields near-optimal solutions in non-dense situations, with the initial assignments of either Alg. 5.4 or Alg. 5.5.
- The runtime of TSWAP remains small when enlarging G while the optimal algorithm increases dramatically. This empirical result is consistent with the time com-

random-64-64-20

$\|A\|$	metric	Alg. 5.4	Alg. 5.4†	Alg. 5.4$^{\dagger *}$	Alg. 5.5	Alg. 5.5*	Alg. 5.6	greedy	linear
	runtime(ms)	5 (4, 5)	9 (8,9)	15 (15,16)	**2** (2,2)	12 (11,12)	3 (3,3)	12 (12,12)	23 (23,24)
110	makespan	**17** (17,18)	**17** (17,18)	**17** (17,18)	21 (20,22)	21 (20,22)	37 (35,39)	84 (79,88)	36 (33,39)
	sum-of-costs	1078 (1037,1119)	**936** (900,972)	**936** (899,971)	1130 (1086,1173)	1133 (1090,1175)	**957** (921,991)	1391 (1313,1469)	**941** (901,979)
	runtime(ms)	66 (58,73)	172 (156,185)	203 (193,211)	**11** (10,12)	83 (83,84)	18 (18,19)	80 (79,80)	582 (572,590)
500	makespan	**10** (10,11)	**11** (10,11)	**11** (10,11)	13 (12,13)	13 (12,13)	33 (31,35)	79 (75,82)	32 (29,35)
	sum-of-costs	2595 (2490,2690)	**2169** (2081,2252)	**2169** (2082,2251)	2889 (2780,2996)	2888 (2778,2996)	2521 (2414,2627)	4620 (4366,4873)	2423 (2304,2541)
	runtime(ms)	315 (261,362)	758 (670,830)	849 (787,904)	**22** (21,23)	265 (264,266)	53 (52,55)	251 (250,252)	3783 (3725,3829)
1000	makespan	**9** (8,9)	**9** (9,9)	**9** (9,9)	11 (10,11)	11 (10,11)	28 (26,30)	71 (68,74)	26 (23,28)
	sum-of-costs	3593 (3451,3731)	**2922** (2810,3029)	**2922** (2812,3030)	4041 (3870,4205)	4059 (3893,4220)	3659 (3496,3822)	7457 (7008,7881)	3477 (3298,3652)
	runtime(ms)	1395 (1205,1564)	2975 (2754,3170)	3354 (3140,3541)	**58** (54,61)	947 (943,949)	207 (202,213)	915 (913,917)	31713 (31604,31817)
2000	makespan	**8** (7,8)	**7** (7,8)	**7** (7,8)	10 (9,11)	10 (10,11)	23 (21,25)	57 (53,60)	23 (21,24)
	sum-of-costs	4666 (4479,4852)	3469 (3312,3617)	3469 (3312,3617)	5213 (4961,5450)	5266 (5032,5492)	5434 (5072,5780)	12203 (11327,13045)	5097 (4725,5441)

lak303d

$\|A\|$	metric	Alg. 5.4	Alg. 5.4†	Alg. 5.4$^{\dagger *}$	Alg. 5.5	Alg. 5.5*	Alg. 5.6	greedy	linear
	runtime(ms)	31 (30,32)	39 (37,41)	39 (38,39)	**18** (17,19)	34 (33,35)	21 (20,22)	36 (35,36)	42 (42,43)
100	makespan	**89** (83,94)	**89** (83,94)	**89** (83,95)	**89** (83,95)	**89** (83,95)	229 (206,252)	387 (363,412)	242 (217,267)
	sum-of-costs	3831 (3537,4095)	**3264** (3029,3483)	**3264** (3024,3482)	3930 (3636,4197)	3939 (3648,4213)	**3450** (3176,3702)	4503 (4222,4779)	**3496** (3231,3755)
	runtime(ms)	570 (473,657)	794 (684,894)	817 (718,904)	**77** (73,81)	213 (211,215)	119 (110,127)	212 (210,214)	707 (704,710)
500	makespan	**60** (56,65)	**61** (56,66)	**61** (56,66)	**61** (56,66)	**61** (56,66)	260 (232,286)	425 (403,448)	287 (257,316)
	sum-of-costs	11137 (10164,12054)	**9226** (8411,9953)	**9226** (8429,9961)	12508 (11416,13549)	12465 (11349,13482)	11211 (10106,12240)	16956 (15857,17979)	11721 (10552,12816)
	runtime(ms)	3386 (2879,3885)	4405 (3824,4945)	4515 (3970,5035)	**139** (131,146)	557 (552,562)	250 (235,266))	536 (531,540)	3998 (3983,4012)
1000	makespan	**46** (43,49)	**47** (43,50)	**47** (43,50)	**47** (44,50)	**47** (44,50)	218 (194,242)	394 (374,415)	239 (215,263)
	sum-of-costs	14353 (13300,15325)	**11891** (11013,12668)	**11891** (11044,12661)	17085 (15727,18250)	17100 (15786,18304)	15070 (13791,16256)	26547 (24954,28027)	15563 (14173,16843)
	runtime(ms)	25237 (20298,29658)	30884 (25682,35651)	31309 (26076,35964)	**336** (310,361)	1584 (1566,1602)	637 (602,671)	1506 (1500,1512)	34098 (34007,34195)
2000	makespan	**47** (42,51)	**49** (44,53)	**49** (44,53)	**48** (43,52)	**48** (43,52)	185 (165,204)	372 (354,390)	212 (188,234)
	sum-of-costs	21239 (19556,22790)	**18183** (16834,19479)	**18183** (16828,19477)	26831 (24736,28758)	26736 (24676,28674)	23336 (21222,25279)	45769 (43364,48105)	24554 (22386,26656)

Table 5.2: Effect of initial assignments on TSWAP. 95% confidence intervals of the mean are also displayed, on which bold characters are based.

plexities, that is, the makespan-optimal algorithm is quadratic to $|V|$ while TSWAP excluding the target assignment is linear to $\texttt{diam}(G)$.

Figure 5.9 shows dense situations such that $|A| \geq |V|/8$. As the size of agents increases, the runtime of TSWAP with Alg. 5.4 (bottleneck) quickly increases compared to the optimal algorithm, because it is quartic on $|A|$ (see Prop. 5.9). Meanwhile, TSWAP with Alg. 5.5 (greedy) immediately yields solutions even with a few thousand agents while a bit compromising the solution quality of makespan.

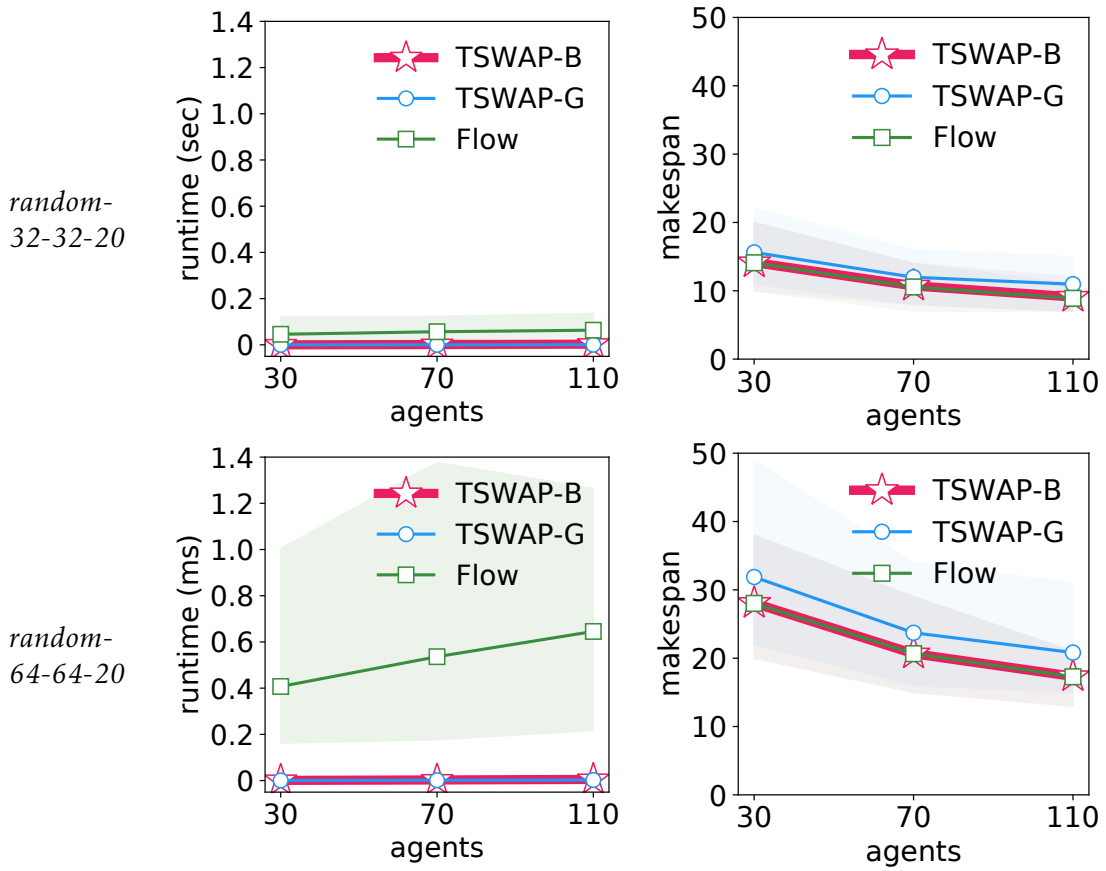


Figure 5.8: Results on instances with "quadrupling" the size of G. The average runtime and makespan are shown with minimum and maximum values displayed by transparent regions. Algorithm 5.4 (TSWAP-B; bottleneck) and Alg. 5.5 (TSWAP-G; greedy) were used in the initial target assignment of TSWAP. "Flow" is the makespan-optimal algorithm.

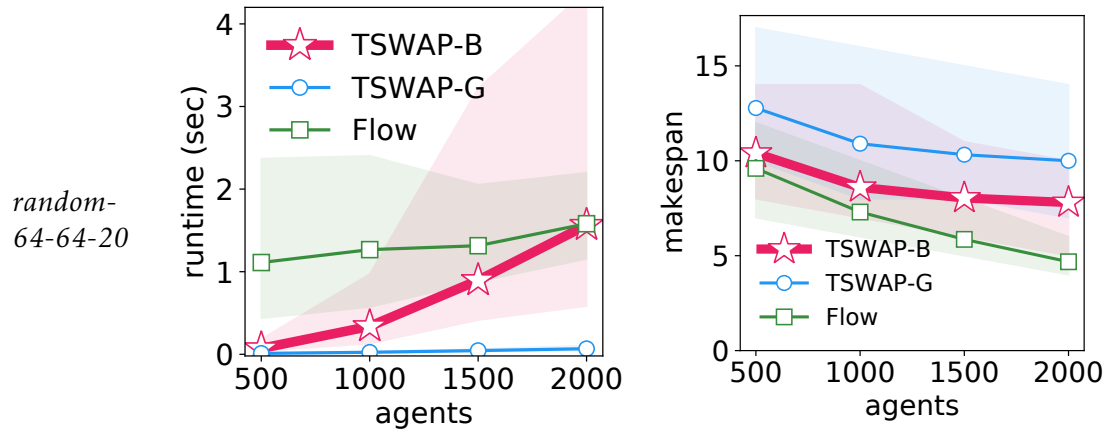


Figure 5.9: Results in dense situations. See also the caption of Fig. 5.8.

Large Graphs

Table 5.3 shows the results on large graphs with a timeout of 5 min. The makespan-optimal algorithm took time to return solutions or sometimes failed due to the timeout, whereas TSWAP succeeded in all cases in a comparatively very short time. This result highlights the need for sub-optimal algorithms of unlabeled-MAPF. In addition, TSWAP yields high-quality solutions for the makespan.

		runtime (sec)			success rate(%)			sub-optimality	
map	$\|A\|$	Flow	Alg. 5.4	Alg. 5.5	Flow	Alg. 5.4	Alg. 5.5	Alg. 5.4	Alg. 5.5
lak303d	100	26.2 (21.3,30.6)	0.0 (0.0,0.0)	**0.0** (0.0,0.0)	100	100	100	1.001	1.003
	500	60.6 (47.8,71.3)	0.6 (0.5,0.7)	**0.1** (0.1,0.1)	100	100	100	1.008	1.019
	1,000	54.7 (47.7,61.2)	3.6 (3.1,4.2)	**0.2** (0.2,0.2)	100	100	100	1.069	1.090
	2,000	56.3 (49.1,63.0)	31.0 (24.9,36.6)	**0.5** (0.4,0.5)	100	100	100	1.334	1.372
den520d	100	46.0 (38.6, 52.7)	0 (0,0)	**0** (0,0)	100	100	100	1.000	1.052
	500	67.5 (56.9,77.3)	0.3 (0.2,0.3)	**0.1** (0.1,0.1)	100	100	100	1.003	1.113
	1,000	82.3 (72.4,91.7)	1.6 (1.3,1.9)	**0.2** (0.2,0.2)	98	100	100	1.015	1.129
	2,000	89.8 (81.7,97.7)	9.4 (7.7,11.0)	**0.4** (0.4,0.4)	100	100	100	1.041	1.177
brc202d	100	141.9 (118.5,164.7)	0.1 (0.1,0.1)	**0.1** (0.1,0.1)	60	100	100	1.000	1.000
	500	214.5 (194.0,236.3)	0.7 (0.6,0.7)	**0.3** (0.3,0.3)	48	100	100	1.001	1.001
	1,000	238.5 (219.4,259.1)	2.7 (2.4,3.1)	**0.5** (0.5,0.5)	42	100	100	1.002	1.008
	2,000	230.2 (196.1,273.9)	14.8 (10.7,19.3)	**1.0** (1.,1.1)	16	100	100	1.019	1.023

Table 5.3: Results in large graphs. "Flow" is the optimal algorithm. The scores are averages over instances that were solved by all solvers. The sub-optimality is for makespan, dividing the makespan of TSWAP by the optimal scores. 95% confidence intervals of the mean are also displayed.

5.4.4 Sum-of-costs Metric

Next, the sum-of-costs metric of TSWAP was assessed. As a baseline, ECBS-TA [Hönig *et al.*, 2018a] was used, which yields bounded sub-optimal solutions with respect to the sum-of-costs. The implementation of ECBS-TA was obtained from the authors.[4] The *random-32-32-20* map was used with 30, 70, and 110 agents. The sub-optimality of ECBS-TA was set to 1.3, which was adjusted to solve problems within the acceptable time (5 min). TSWAP used the bottleneck assignment with min-cost maximum matching (Alg. 5.4†) as the initial target assignment. Note that we preliminary confirmed that ECBS-TA in denser situations failed to return solutions within a reasonable time.

Figure 5.10 shows that TSWAP yields solutions with acceptable quality while reducing computation time (lower, vertical axis) by orders of magnitude compared to the others; the quality of sum-of-costs (horizontal axis) is competitive with ECBS-TA, with makespan quality close to optimal (upper, vertical axis). TSWAP is significantly faster than ECBS-TA because, unlike ECBS-TA, TSWAP uses a one-shot target assignment and a simple path planning process.

5.4.5 Limitation; Adversarial Instance

Lastly, a planning demo of TSWAP in an adversarial instance is presented, similar to Fig. 5.3. The used instance is shown in Fig. 5.11, together with planning results at an intermediate timestep. For comparison, the makespan-optimal algorithm was also tested.

With TSWAP, all agents tried to follow the shortest paths toward assigned targets. Consequently, all agents tried to pass through a narrow corridor located at the center,

[4]The code is available on `https://github.com/whoenig/libMultiRobotPlanning`.

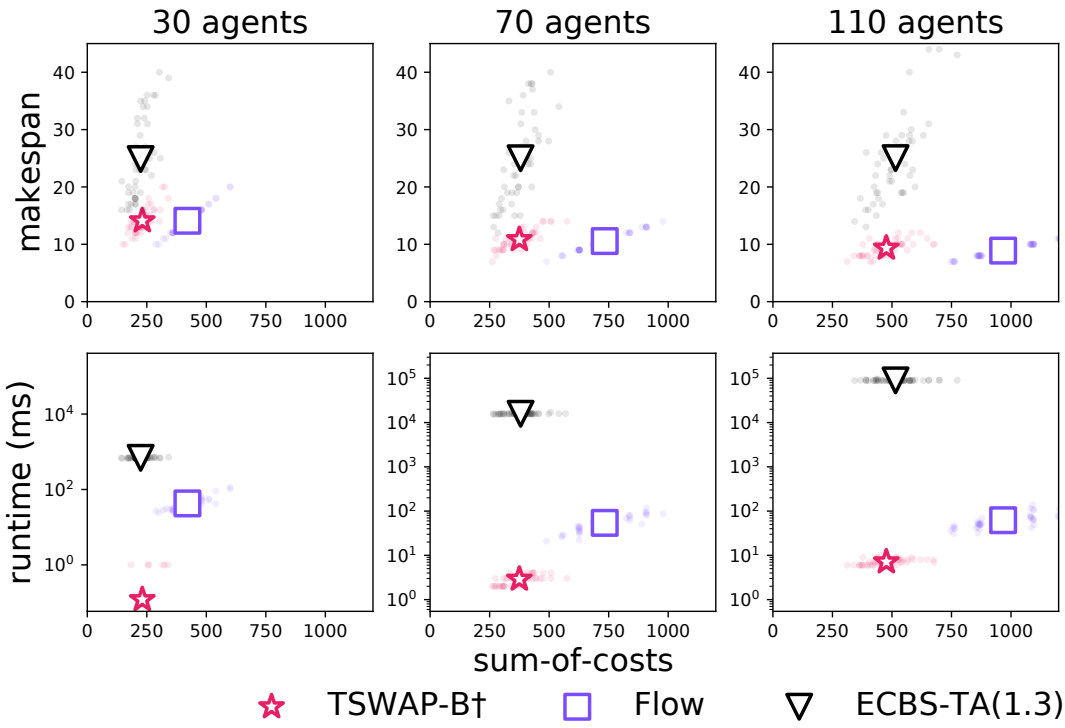
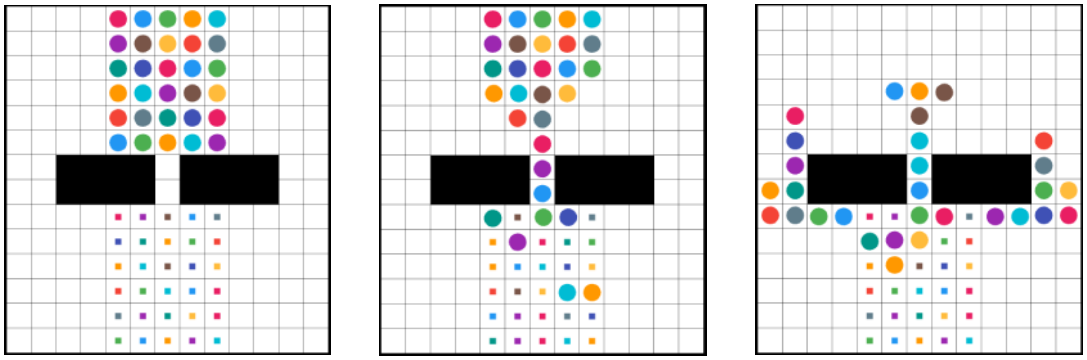


Figure 5.10: Results for the sum-of-costs metric. The data was obtained on *random-32-32-20*. The average scores are plotted. Scatter plots of 50 instances are also plotted by transparent points.



(a) instance **(b)** TSWAP ($t = 8$; $T = 32$) **(c)** Flow ($t = 8$; $T = 15$)

Figure 5.11: Evaluation of an adversarial instance. *(a)* The initial locations of agents and target locations are represented by colored circles and boxes, respectively. *(b)* A configuration of timestep 8 from the planning result by TSWAP. The makespan is 32. *(c)* That of the makespan-optimal algorithm. The makespan is 15.

compromising the solution quality of the makespan. In contrast, the makespan-optimal algorithm uses side corridors located on the left and right, resulting in a much better outcome than TSWAP. This pitfall may be overcome by improving `next_vertex` so that taking the non-shortest paths.

5.5 Related Work

The general review of unlabeled-MAPF has already appeared in Chap. 3.4. Therefore, the below review focuses on studies closely related to TSWAP.

Unlike conventional MAPF, unlabeled-MAPF is always solvable [Kornhauser *et al.*, 1984; Yu and LaValle, 2013a; Adler *et al.*, 2015; Ma *et al.*, 2016]. Among them, TSWAP relates to the analysis presented in [Yu and LaValle, 2013a] because both approaches use target swapping. Their analysis relies on optimal linear assignment whereas TSWAP works for any assignment.

The multi-agent pickup and delivery (MAPD) problem [Ma *et al.*, 2017b] is a popular variant of MAPF, which includes target assignment. Although MAPD is a problem different from unlabeled-MAPF, TSWAP is similar to an MAPD algorithm TPTS [Ma *et al.*, 2017b] in the sense that both algorithms swap assigned targets adaptively. One primary difference though is that, unlike TSWAP, TPTS sets additional conditions about start and target locations.

The unlabeled version of pebble motion has also been studied [Kornhauser *et al.*, 1984; Călinescu *et al.*, 2008; Goraly and Hassin, 2010]. However, in unlabeled-MAPF, agents can move simultaneously. Different from those studies, TSWAP explicitly assumes this fact, resulting in practical outcomes.

Pattern formation of multiple agents [Oh *et al.*, 2015] is one of the motivating examples of unlabeled-MAPF; various approaches have been proposed. Among them, several studies are highlighted below. SCRAM [MacAlpine *et al.*, 2015] is a target assignment algorithm considering collisions and works only in open space without obstacles; hence its applications are limited. The bottleneck assignment algorithm (Alg. 5.4) uses a scheme similar to SCRAM but differs in its use of lazy evaluation. Incorporating lazy evaluation for optimal linear target assignment is studied in [Aakash and Saha, 2022], but missing the path planning aspect. In [Turpin *et al.*, 2014], a method that first solves the lexicographic bottleneck assignment [Burkard and Rendl, 1991] and then plans trajectories on graphs is proposed. To avoid collisions, the method uses the delay offset about when agents start moving, resulting in a longer makespan. TSWAP avoids using such offsets by swapping targets on demand.

5.6 Concluding Remarks

The chapter presented the TSWAP algorithm to solve unlabeled-MAPF. Similar to PIBT, TSWAP consecutively generates a sequence of configurations but it also swaps targets assigned to agents as necessary. Theoretically, TSWAP is complete regardless of initial assignments. Empirically, depending on assignment algorithms, TSWAP scored excellent performance; it can solve large instances near-optimally in a very short time, which is order-of-magnitude faster than those of the optimal algorithm. This is the power of short-horizon planning.

As a drawback, *TSWAP is sub-optimal*. Indeed, with adversarial instances, TSWAP can output solutions far from optimal ones. The next chapter includes how to overcome this limitation, by incorporating *long-horizon planning*.

Chapter 6

Short-Horizon Planning Guides Long-Horizon Planning

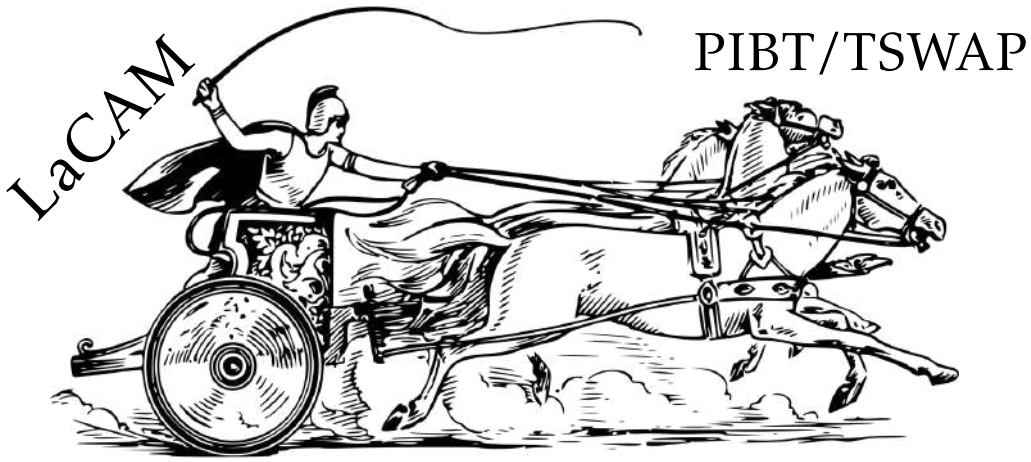


Figure 6.1: Concept of LaCAM. When I came up with the algorithm, I actually thought like this. The original image is by OpenClipart-Vectors / Pixabay License.

In the previous two chapters (Chap. 4 and 5), we have seen the powers of short-horizon planning such as excellent speed and scalability. They are "horses," enabling the search to progress speedily and bravely. Meanwhile, they lack a global planning perspective due to their short planning horizons. For instance, PIBT lacks completeness for MAPF. TSWAP lacks optimality for unlabeled-MAPF. To overcome such shortcomings, incorporating long-horizon planning is mandatory. The question here is whether it is possible to incorporate the long-horizon nature without compromising the benefits of short-horizon planning.

To this end, this chapter presents a good "horse-rider" for short-horizon planning. The proposed algorithm is called *lazy constraint addition search for MAPF (LaCAM)*. The chapter is a peak of Part I. LaCAM originally appeared in [Okumura, 2023], but the chapter includes non-trivial and remarkable extensions.[1]

6.1 Chapter Overview

The objective of the chapter is to understand planning style such that short-horizon planning guides long-horizon planning, embodied as LaCAM, and see its power.

[1] The extended parts are currently under review.

6.1.1 What is LaCAM

A lazy constraint addition search is a graph pathfinding algorithm. This is a newly developed concept to overcome planning problems with huge branching factors. LaCAM is an implementation of this search for MAPF.

Mechanism

LaCAM comprises a two-level search. At the high-level, it searches a sequence of configurations (i.e., a tuple of locations for agents). At the low-level, it searches constraints that specify which agents go where in the next configuration. Successors at the high-level (i.e., configurations) are generated in a lazy manner while following constraints from the low-level, leading to a dramatic reduction of the search effort. Moreover, each successor is generated by short-horizon planning (such as PIBT), therefore, LaCAM can inherit the benefits of the small computational effort of such algorithms.

This chapter also presents *LaCAM**, an anytime version of LaCAM that gradually improves solution quality once found. This is achieved by rewriting the tree structure when finding better connections between search nodes.

Throughout the chapter, we focus on solving MAPF. However, LaCAM is expected to be applicable to unlabeled-MAPF with the transition from PIBT to TSWAP.

6.1.2 Properties and Performance

Theoretical Properties

LaCAM is complete for MAPF; it returns a solution for solvable instances, otherwise reports the non-existence. Moreover, LaCAM* eventually returns optimal solutions, provided that a solution cost takes the form of accumulative transition costs.

Empirical Performance

LaCAM with PIBT solved a variety of MAPF instances in a very short time, including complicated puzzle-like instances, instances with large maps (e.g., $1,491 \times 656$ four-connected grid), instances with massive agents (e.g., $10,000$), or dense situations. For instance, it solved *all* instances with 400 agents on a 32×32 grid with 20% obstacles from the MAPF benchmark [Stern *et al.*, 2019], with a median runtime of 1 s. In contrast, baseline sub-optimal MAPF algorithms, such as [Silver, 2005; Standley, 2010; Okumura *et al.*, 2022b; Li *et al.*, 2021c; Li *et al.*, 2022], mostly failed to solve the instances with the timeout of 30 s. Moreover, combined with an extended version of PIBT specially customized to overcome livelock situations, LaCAM solved 99% instances of the MAPF benchmark sub-optimally in 10 s, with the guarantee of eventual optimality. From the empirical evidence, *beyond dispute*, *LaCAM has developed a new frontier in MAPF*.

6.1.3 Chapter Organization

- Chapter 6.2 explains the concept of lazy constraints addition search.
- Chapter 6.3 describes the sub-optimal LaCAM algorithm.
- Chapter 6.4 assesses the performance of a vanilla LaCAM.
- Chapter 6.5 attaches eventual optimality to LaCAM and presents LaCAM*.
- Chapter 6.6 presents an improved successor generator, specifically, an enhanced version of PIBT.

- Chapter 6.7 evaluates the performance of the enhancements presented in Chap. 6.5 and 6.6.
- Chapter 6.8 reviews algorithms closely related to LaCAM.
- Chapter 6.9 concludes the chapter.

The code and movie are available at `https://kei18.github.io/lacam`.

6.1.4 Notations and Assumptions

$\bot$	undefined, not found
$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \dots, n\}$	a set of agents
$\mathcal{S} = (s_1, \dots, s_n)$	start configuration, where $s_i \in V$
$\mathcal{G} = (g_1, \dots, g_n)$	goal configuration, where $g_i \in V$
Δ	maximum degree of G (G is omitted for simplicity)

Caution

The chapter uses the representation by configurations.

6.2 Concept of Lazy Constraints Addition Search

To begin with, we see the concept behind LaCAM, called *lazy constraints addition search*, shortly denoted *LaCA search*. This is illustrated with an example of single-agent pathfinding.

6.2.1 Classical Search

See first Fig. 6.2, illustrating how the usual search scheme solves grid pathfinding. In particular, the figure visualizes the greedy best-first search with a heuristic of the Manhattan distance. Here, the location of the agent corresponds to the "state" of the search. Let's use the term "configuration" instead of "state" for consistency of the remaining part. From the initial configuration (i.e., start location), the search generates three successor nodes (left, up, right), each corresponding to one configuration. It then takes one of the generated nodes according to specific criteria (i.e., heuristic), and expands successors. This procedure continues until finding the goal configuration.

Consider now how many search nodes are generated. Even though the solution length is eight, 22 nodes are generated. This number is related to the number of neighboring configurations (i.e., branching factor). It is four in single-agent grid pathfinding, therefore, the number of node generations is acceptable. However, in MAPF, the number of successors is exponential for the number of agents. Consequently, the generation itself becomes intractable. This is why the vanilla A* is hopeless to solve large MAPF instances, as we have already observed in Fig. 3.3.

6.2.2 Configuration Generator and Constraints

The LaCA search tries to relieve this huge-branching-factor issue when good *configuration generator* is available. A configuration generator takes one configuration and *constraints*. Then, it returns one neighboring (i.e., connected) configuration (i.e., successor) from the given configuration. *Constraints should be embodied by domains.* In this example, consider a constraint as a prohibition of direction, such as not moving up, left, right, or down.

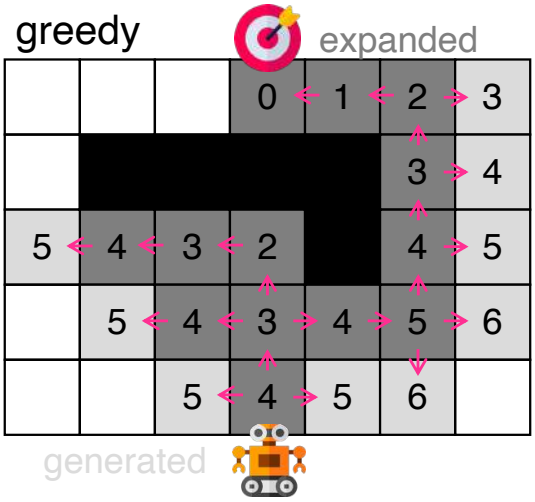


Figure 6.2: Greedy search with the heuristic of the Manhattan distance.

6.2.3 Constraint Tree

In the LaCA search, each search node contains not only a configuration but also constraints, typically taking the form of a tree structure. Constraints are added in a lazy manner as follows. For each node invoke, the LaCA search gradually develops the constraint tree by *low-level search*, implemented by e.g., breadth-first search (BFS). A node on the tree has a constraint and represents several constraints by tracing a path to the root.

6.2.4 Algorithm Flow

The LaCA search is now explained using Fig. 6.3, with a depth-first search (DFS) style. The attempt to find a sequence of configurations is called *high-level search*.

At the beginning of the search, a (high-level) search node of the start configuration is examined (Fig. 6.3a). The node has no constraints for the first invoke, that is, the configuration generator can generate any connected configuration. Here, it is assumed that the generator generates an "up" configuration, following the Manhattan distance guide, as illustrated by the pink arrow. Preparing for the second invoke of the node, the node proceeds the low-level search and expands a constraint tree with new constraints (e.g., "not go up"). The high-level search does not discard the examined node immediately, rather, it discards when all connected configurations have been generated. This corresponds to when all nodes in its constraint tree have been examined.

Next, Fig. 6.3b and 6.3c show an example of the second invoke of high-level nodes. In Fig. 6.3b, the generator generates an already known configuration. Since this example assumes DFS, the LaCA search examines the blue-colored node again in Fig. 6.3c. This time, the generator must follow a constraint "not go right" and its parent "no constraint." Consequently, the example assumes that a "left" configuration is generated.

The search continues until finding the goal configuration (Fig. 6.3d). It then eventually outputs a solution by backtracking.

With appropriate designs of constraint trees, the LaCA search can be an exhaustive search (i.e., guaranteeing completeness). LaCAM exemplifies the LaCA search for MAPF, described immediately after. The LaCA search can greatly decrease the number of node generations if the configuration generator is promising in outputting configurations that are close to the goal. *This is a silver bullet for quick planning, especially in planning problems where the branching factor is huge like MAPF.*

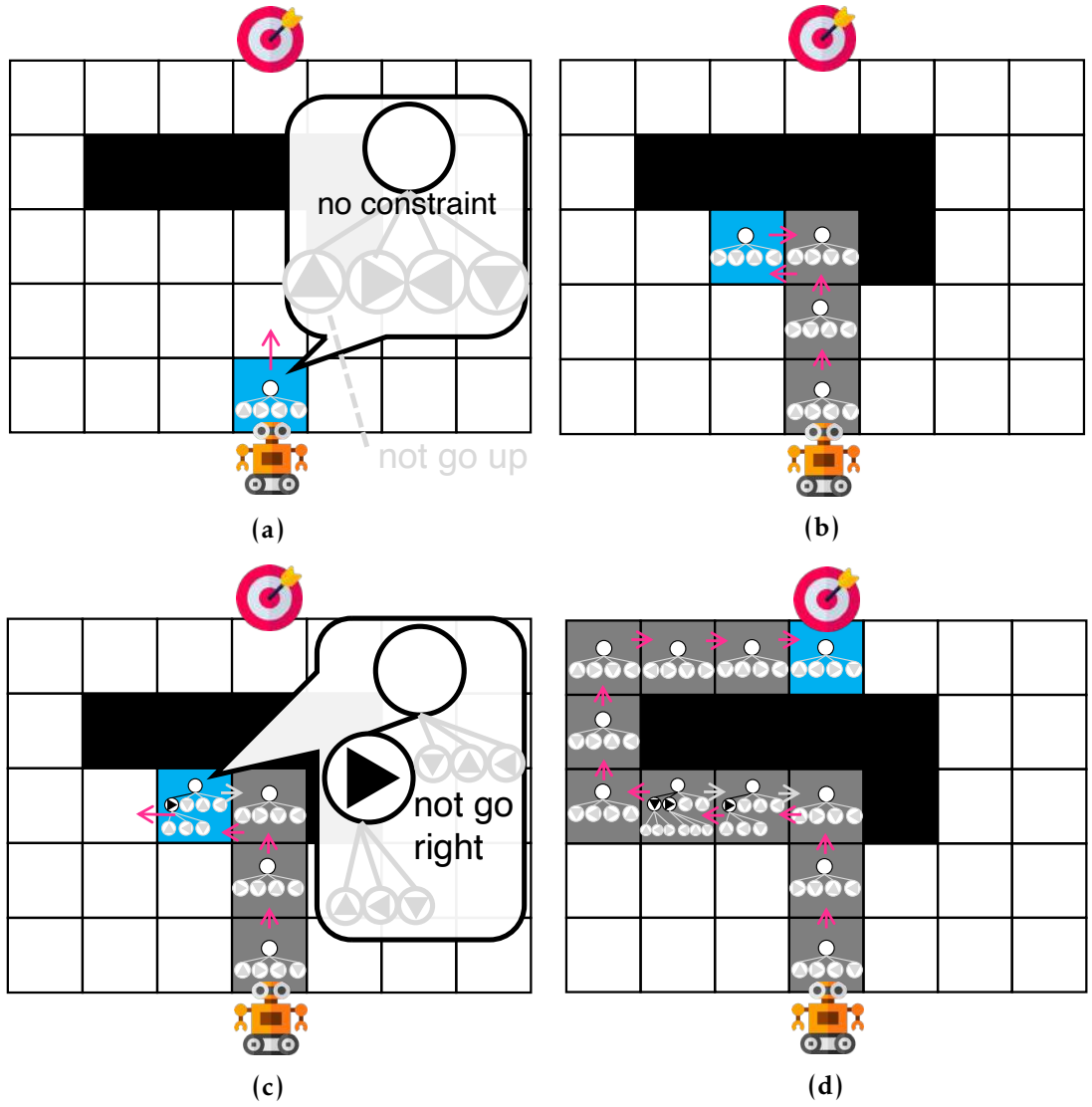


Figure 6.3: Illustration of the LaCA search using single-agent pathfinding.

6.3 Algorithm Description

6.3.1 High-Level Description

Next, some artifacts of LaCAM to solve MAPF are specified with a concrete example, shown in Fig. 6.4. The following part explains the figure step by step. Herein, the term "configuration" refers to a tuple of locations for all agents.

Overview. LaCAM is a two-level search. At the high-level, it explores a sequence of configurations; each search node corresponds to one configuration. For each high-level node, it also performs a low-level search that creates *constraints*. A constraint specifies which agent is where in the next configuration. The low-level search proceeds lazily, creating a minimal successor each time the corresponding high-level node is invoked.

High-Level Search. As in general search schema like Alg. 2.1, LaCAM progresses by updating an *Open* list that stores the high-level nodes. *Open* is implemented by data structures of stack, queue, or priority queue. Throughout the chapter, we assume using the stack. Thus, LaCAM is explained as a DFS style. The first row of Fig. 6.4 illustrates

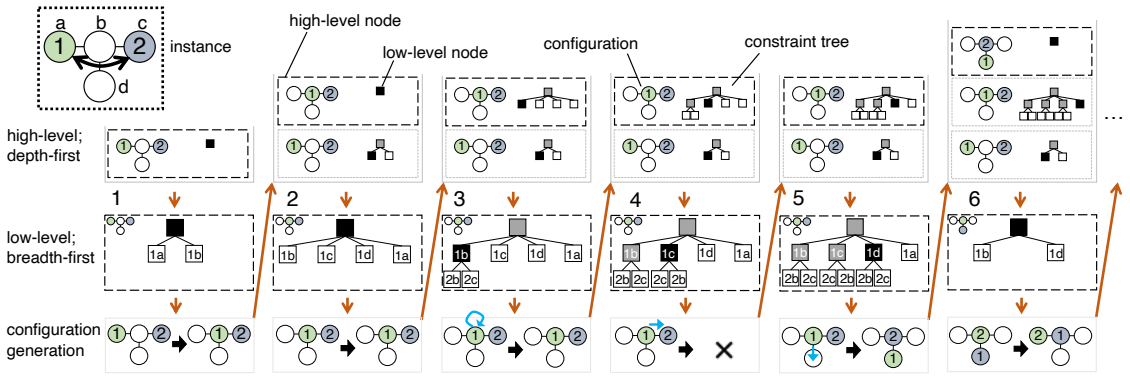


Figure 6.4: Running example of LaCAM. Orange arrows represent the search progress order. Selected and searched low-level nodes are filled with black and gray, respectively. Constraints are shown by blue-colored arrows.

Open. For each search iteration, LaCAM selects one node from *Open*. Different from the general search schema, LaCAM does not immediately discard the selected node, as explained after three paragraphs.

Low-Level Search. Each high-level node comprises a configuration and a *constraint tree*. The constraint tree gradually grows at each time invoking the high-level node; this is the low-level search of LaCAM. Throughout the chapter, BFS is used for the low-level. The middle row of Fig. 6.4 visualizes this step. Each node of the constraint tree has a constraint, except for the root node. For instance, in the first column, the root node has two successors: '1a' and '1b.' This means that an agent-1 must go to the vertex-a or vertex-b in the next configuration. Successors of the low-level search are created by two steps:

1. Select an agent i. Let v be the vertex of i in the configuration.
2. Create successors that specifies i is on $u \in \mathrm{neigh}(v)$ or v.

The agent is selected so that each path from each low-level node to the root does not contain duplicated agents. Therefore, those paths specify constraints for several agents. In addition, no successors are created when the depth of the node is beyond $|A|$ because constraints have been assigned for all agents.

Configuration Generation. Once both the high- and low-level nodes are specified, a new configuration is generated. The new one must satisfy the constraints of the low-level node, which are specified by a path to the low-level root node. Excluding that, any connected configuration from the original configuration can be generated. *We will see how to generate new configurations following constraints later in Chap. 6.3.4, but for now, regard this as a blackbox function.* The generation step is visualized in the third row of Fig. 6.4. According to the new configuration, a new high-level node is created. For instance, at the end of the first column of Fig. 6.4, a new configuration (b, c) is generated and inserted to *Open*.

Discarding High-Level Nodes. When finished searching all low-level nodes, the corresponding high-level node has been generating all configurations connected to its configuration. Therefore, this high-level node is discarded and removed from *Open*.

6.3.2 Example

LaCAM continues the above search operations until finding the goal configuration $\mathcal{G}$. Once $\mathcal{G}$ is found, it is trivial to obtain a solution by backtracking high-level nodes. Next, a running example of Fig. 6.4 is explained in detail step by step in columns.

1. Initially, *Open* contains only a high-level node for the start configuration $\mathcal{S}$. At the low-level, two successors are created. In this step, any agent can be selected; the example chooses agent-1. Next, LaCAM generates a configuration connected to the original one (a, c). Since the target low-level node is the root, there is no constraint. Assume that a new configuration (b, c) is generated. Then, the corresponding new high-level node is inserted into *Open*.
2. The high-level node generated in the previous iteration is selected. At the low-level, the example again chooses agent-1. This time, LaCAM generates four successors: '1a,' '1b,' '1c,' and '1d.' The example adds them to the low-level tree in order of '1b,' '1c,' '1d,' and '1a' to make it interesting. Assume that the same configuration (b, c) is generated. Then, a high-level node is not created since the generated configuration has already appeared in the search.
3. The same high-level node is selected as the previous iteration. The low-level search generates two nodes for agent-2: '2b' and '2c.' This time, the configuration generation must follow the constraint of '1b.' Consequently, the same configuration (b, c) is generated and no new high-level node is created.
4. According to the selected low-level node, it is impossible to generate a connected configuration due to a collision between agent-1 and agent-2; this iteration skips the creation of a high-level node.
5. The constraint makes agent-1 move to vertex-d. Then, a new configuration (d, b) is generated. The corresponding high-level node is created and inserted into *Open*.
6. The high-level node for (d, b) is selected, and then, a new configuration (b, a) is generated. The search can find the goal configuration in the next iteration.

6.3.3 Pseudocode

Algorithm 6.1 shows an example implementation of LaCAM. In the pseudocode, $\mathcal{N}$ and $\mathcal{C}$ correspond to high- and low-level nodes, respectively. The low-level search uses queue (*tree*) because it is breadth-first. Several details are below.

Configuration Generation. This is performed by a blackbox function `configuration_generator` (Line 14). The function returns a configuration connected to a configuration of a high-level node, following constraints specified by a low-level node. It returns $\perp$ when failing to generate such configurations (e.g., the fourth column of Fig. 6.4). Note that, at the bottom of the low-level tree, all agents have constraints. Therefore, exactly one configuration is specified without freedom.

High-Level Node Management. To manage already known configurations, Alg. 6.1 uses an *Explored* table that takes a configuration as a key and stores a high-level node.

Low-Level Agent Selection. To generate low-level search trees, a high-level node includes *order*, an enumeration of all agents sorted by specific criteria, specified by two

Algorithm 6.1 LaCAM

input: MAPF instance ($\mathcal{S}$: starts, $\mathcal{G}$: goals)

output: solution or `NO_SOLUTION`

notation: $\mathcal{C}^{\text{init}} := \langle\, parent : \bot, who : \bot, where : \bot \,\rangle$ ▷ *initial constraint*

1: initialize *Open*, *Explored* ▷ *Open: stack*
2: $\mathcal{N}^{\text{init}} \leftarrow \left\langle\, config : \mathcal{S}, tree : [\![\, \mathcal{C}^{\text{init}} \,]\!], order : \texttt{get\_init\_order}(), parent : \bot \,\right\rangle$
3: $Open.\texttt{push}\left(\mathcal{N}^{\text{init}}\right)$; $Explored[\mathcal{S}] = \mathcal{N}^{\text{init}}$
4: **while** $Open \neq \emptyset$ **do**
5: $\quad \mathcal{N} \leftarrow Open.\texttt{top}()$
6: $\quad$ **if** $\mathcal{N}.config = \mathcal{G}$ **then return** $\texttt{backtrack}(\mathcal{N})$
7: $\quad$ **if** $\mathcal{N}.tree = \emptyset$ **then** $Open.\texttt{pop}()$; **continue**
8: $\quad \mathcal{C} \leftarrow \mathcal{N}.tree.\texttt{pop}()$ ▷ *low-level search begins*
9: $\quad$ **if** $\texttt{depth}(\mathcal{C}) \leq |A|$ **then**
10: $\quad\quad i \leftarrow \mathcal{N}.order[\texttt{depth}(\mathcal{C})]$; $v \leftarrow \mathcal{N}.config[i]$
11: $\quad\quad$ **for** $u \in \texttt{neigh}(v) \cup \{v\}$ **do**
12: $\quad\quad\quad \mathcal{C}^{\text{new}} \leftarrow \langle\, parent : \mathcal{C}, who : i, where : u \,\rangle$
13: $\quad\quad\quad \mathcal{N}.tree.\texttt{push}(\mathcal{C}^{\text{new}})$ ▷ *low-level search ends*
14: $\quad \mathcal{Q}^{\text{new}} \leftarrow \texttt{configuration\_generator}(\mathcal{N}, \mathcal{C})$
15: $\quad$ **if** $\mathcal{Q}^{\text{new}} = \bot$ **then continue**
16: $\quad$ **if** $Explored[\mathcal{Q}^{\text{new}}] \neq \bot$ **then continue**
17: $\quad \mathcal{N}^{\text{new}} \leftarrow \langle\, config : \mathcal{Q}^{\text{new}}, tree : [\![\, \mathcal{C}^{\text{init}} \,]\!], order : \texttt{get\_order}(\mathcal{Q}^{\text{new}}, \mathcal{N}), parent : \mathcal{N} \,\rangle$
18: $\quad Open.\texttt{push}(\mathcal{N}^{\text{new}})$; $Explored[\mathcal{Q}^{\text{new}}] = \mathcal{N}^{\text{new}}$
19: **return** `NO_SOLUTION`

functions `get_init_order` (Line 2) and `get_order` (Line 17). The agent is selected following *order* and depth of the low-level search tree (starting at one; obtained by a function `depth`) (Line 10). This scheme ensures that each path of the constraint tree has no duplicate agents.

Theorem 6.1 (completeness)**.** *LaCAM (Alg. 6.1) returns a solution for solvable MAPF instances; otherwise, it reports* `NO_SOLUTION`*.*

Proof. A search space is finite:

- For the high-level, the number of configurations is $O\left(V^A\right)$.
- For the low-level, the number of search iterations is upper bounded by $1 + \Delta + \Delta^2 + \ldots + \Delta^A = O\left(\Delta^{A+1}\right)$.

When the low-level search is finished, the corresponding high-level node has been generating all configurations connected to its configuration. Consequently, all *reachable* configurations from the start configuration, defined by transitivity over connections of two configurations, are examined, deriving the theorem. □

6.3.4 Implementation Details

Configuration Generation. The heart of LaCAM is how to generate configurations following constraints (Line 14). Ideally, this sub-procedure should be sufficiently quick and generate a promising configuration to reach the goal configuration. This can be realized by adapting existing MAPF algorithms that can compute a partial solution, i.e., a list of paths until a certain timestep. Regarding the target configuration as a start configuration and then producing a partial solution by these algorithms, we can extract a configuration at timestep one in the partial solution. For instance, a naive approach is to adapt prioritized planning with a limited planning horizon, such as windowed HCA* [Silver, 2005] with single-step window size. A rolling horizon approach for lifelong MAPF [Li *et al.*, 2021d] is also available to generate configurations. An aggressive approach is to adapt PIBT (Alg. 4.1), a scalable MAPF algorithm that repeats planning for one-timestep; the experiments used PIBT. *Those algorithms with short planning horizons, originally developed to solve MAPF, are available to create promising successors in the high-level search of LaCAM.*

Order of Agents. In our implementation, the agent order of the initial high-level node was in descending order of the distance between the start and goal (Line 2), influenced by commonly used heuristics of PP [Van Den Berg and Overmars, 2005]. In other high-level nodes (Line 17), the implementation prioritized agents who are not on their goal, aiming to create constraints for those agents earlier in the low-level search. A tie-break used the last arrival timestep of agents in the high-level search so that agents who have been apart from goals for a long time are prioritized, akin to PIBT.

Order of Low-Level Nodes. As seen in Fig. 6.4, the order of inserting low-level nodes affects search progress. The implementation provisionally made the order random. This part requires further investigation in the future.

Reinsert High-Level Node. Algorithm 6.1 takes a naive DFS style. Instead, when finding an already known configuration, reinserting the corresponding high-level node to *Open* (Line 16) can improve solution quality. The reason is that repeatedly appearing configurations in the search can be seen as a bottleneck; it makes sense to advance the low-level search of the high-level node, which is performed by the reinsert operation. This modification is empirically tested in Chap. 6.4.6. Note that LaCAM does not lose completeness with this modification.

6.4 Evaluation of LaCAM

This section evaluates LaCAM using PIBT as a configuration generator. Specifically, the section presents five empirical results:

- Chap. 6.4.2 evaluates LaCAM with small complicated MAPF instances.
- Chap. 6.4.3 evaluates LaCAM with the MAPF benchmark.
- Chap. 6.4.4 reveals the limitation of LaCAM using an adversarial instance.
- Chap. 6.4.5 assesses scalability of LaCAM with up to $10,000$ agents.
- Chap. 6.4.6 investigates other implementation designs.

6.4.1 Experimental Setups

Baselines. The following six sub-optimal MAPF algorithms were carefully selected as baselines.

- **Prioritized Planning (PP)** [Erdmann and Lozano-Perez, 1987; Silver, 2005] as a basic approach for MAPF. PP used distance heuristics [Van Den Berg and Overmars, 2005] for the planning order and A* [Hart *et al.*, 1968] for single-agent pathfinding.
- **A* with operator decomposition (OD)** [Standley, 2010] as an adaptation of the general search scheme to MAPF. OD used a greedy search fashion to obtain solutions as much as possible (i.e., neglecting g-value of A*). The heuristic (i.e., h-value) was the sum of distance towards goals.
- **PIBT** [Okumura *et al.*, 2022b], which repeats one-timestep planning to solve MAPF. A vanilla PIBT was tested because the LaCAM implementation used PIBT as a subprocedure (Chap. 6.3.4). To detect planning failure, PIBT was regarded as a failure to solve an instance when it reached pre-defined sufficiently large timesteps.
- **PIBT+** [Okumura *et al.*, 2022b] as a state-of-the-art scalable MAPF solver,[2] which uses PIBT until a certain timestep. The rest of the planning is performed by another MAPF algorithm. The complement phase used a rule-based solver, push and swap [Luna and Bekris, 2011].
- **EECBS** [Li *et al.*, 2021c] as a state-of-the-art search-based solver that bases on a celebrated MAPF algorithm, CBS [Sharon *et al.*, 2015]. The sub-optimality was set to five to find solutions as much as possible. In Chap. 6.4.5, it was set to the default value (1.2) of the authors' implementation due to better performance.
- **MAPF-LNS2 (LNS2)** [Li *et al.*, 2022] as another excellent MAPF solver based on large neighborhood search.

It is worth mentioning that PP, PIBT(+), EECBS, and LNS2 are incomplete; they cannot detect unsolvable instances, unlike LaCAM. For PIBT(+), EECBS, and LNS2, the implementations coded by their respective authors were used. The codes are available on `https://github.com/Jiaoyang-Li/EECBS`, `https://github.com/Jiaoyang-Li/MAPF-LNS2`, and `https://github.com/Kei18/pibt2`. For PP, an implementation was used that included in [Okumura *et al.*, 2022b]. OD was own-coded in C++.

Evaluation Environment. LaCAM was also coded in C++. The experiments were run on a desktop PC with Intel Core i9-7960X 2.8 GHz CPU and 64 GB RAM. A maximum of 32 different instances was run in parallel using multi-threading.

Solution Metric. This section uses sum-of-costs (aka. flowtime) to rate solution cost.

6.4.2 Small Complicated Instances

Setup. First, LaCAM was tested on instances used in [Luna and Bekris, 2011], shown in Table 6.1. The runtime limit was 10 s. Since the LaCAM implementation used nondeterminism (see Chap. 6.3.4), it was run five times for the same instance while changing random seeds.

Result. Table 6.1 summarizes the results. As reference records, sum-of-costs optimal solutions are presented, obtained by a vanilla A* ignoring the runtime limit. Although most baseline methods failed several instances, LaCAM solved all the instances regardless of random seeds, within reasonable timeframes. Regarding solution quality (i.e., sum-of-costs), LaCAM compromises the quality compared to PP, EECBS, and LNS2. This is due to the nature of LaCAM, which progresses the search with a short planning horizon.

[2] In 2022. Because we have now LaCAM*.

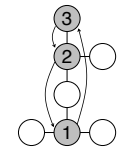
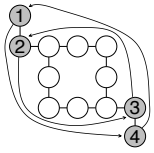
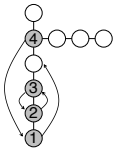
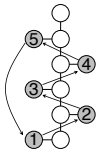
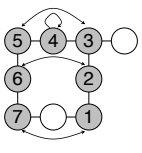
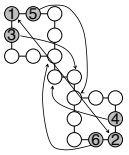
	tree		*corners*		*tunnel*		*string*		*loop-chain*		*connector*		
													
	time(ms)	SOC	time(ms)	SOC	time(ms)	SOC	time(ms)	SOC	time(ms)	SOC	time(ms)	SOC	Solved
LaCAM(med)	0	19	17	47	173	190	0	66	34	1,752	0	168	**6/6**
LaCAM(worst)	0	41	31	70	208	254	0	68	124	2,593	3	281	
PP	N/A	N/A	0	32	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1/6
OD	0	31	0	47	30	191	0	22	5,882	2,269	27	129	**6/6**
PIBT	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0/6
PIBT$^+$	0	55	0	54	0	227	0	52	N/A	N/A	0	130	5/6
EECBS	1	16	1	32	N/A	N/A	0	20	N/A	N/A	N/A	N/A	3/6
LNS2	N/A	N/A	0	32	N/A	N/A	0	20	N/A	N/A	160	0	3/6
A*(SOC-opt.)	0	16	16	32	24	53	1	20	17,752	121	391,138	80	**6/6**

Table 6.1: Results of the small complicated instances.

6.4.3 MAPF Benchmark

Setup. Next, LaCAM was tested with the MAPF benchmark [Stern *et al.*, 2019], which includes a list of four-connected grids and start–goal pairs for agents. Twelve grids with different portfolios (e.g., size, sparseness, and complexity) were selected. For each grid, 25 "random scenarios" were used while increasing the number of agents by 50 up to the maximum. Therefore, identical instances were tried for the solvers in all settings. The runtime limit was set to 30 s following [Stern *et al.*, 2019]. LaCAM was run five times for each setting. For reference, A* used in Table 6.1 failed to solve an instance with ten agents in *random-32-32-20*.

Result. Figures 6.5 and 6.6 summarize the results. In most scenarios, LaCAM outperforms PP, OD, EECBS, and LNS2 in both success rate and runtime, while compromising the solution quality. The runtime of LaCAM is comparable with PIBT$^{(+)}$, furthermore, LaCAM outperforms a vanilla PIBT in the success rate. The most competitive results with LaCAM were scored by PIBT$^+$. However, overall, the sum-of-costs scores of LaCAM are better than those of PIBT$^+$, especially in dense situations. Furthermore, LaCAM solved challenging scenarios, such as *random-32-32-20* with 400 agents, where the baseline methods almost failed to solve. In summary, LaCAM can solve various instances within short timeframes, with acceptable solution quality. Meanwhile, we can observe that LaCAM scored poor performance in several grids, such as *random-64-64-20* and *warehouse-20-40-10-2-1*. This reason is investigated in the next.

6.4.4 Adversarial Instance

Observation from MAPF Benchmark. In the previous experiment, LaCAM quickly solved various scenarios but scored poor performance in several grids. Specifically, LaCAM solved all instances of *warehouse-20-40-10-2-2* while it failed frequently in *warehouse-20-40-10-2-1*. The two maps differ in the width of corridors: the former is two while the latter is one. Therefore, it is natural to consider as the existence of narrow corridors such that two agents cannot pass through could be a bottleneck for LaCAM.

Setup and Result. For investigation, an adversarial instance for LaCAM was prepared (see Table 6.2), where two pairwise agents need to swap their locations in narrow corridors. The number of search iterations of the high-level search was counted while changing the number of agents (two: only agents-$\{1, 2\}$ appear, four, and six). LaCAM solved

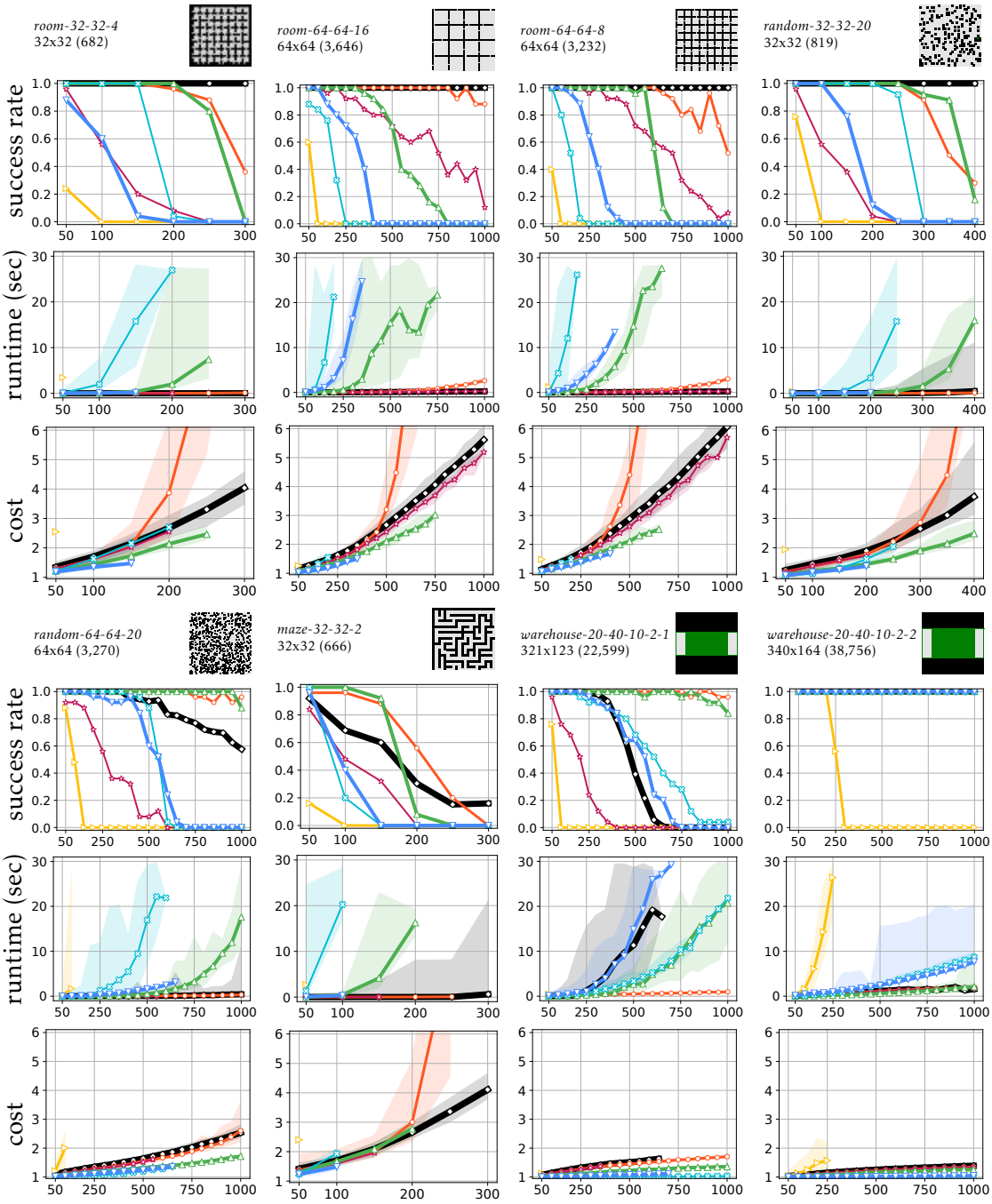


Figure 6.5: Results of the MAPF benchmark (1/2). The number of vertices for each grid is shown with parentheses. "cost" represents sum-of-costs divided by the total distance of start-goal pairs, $\sum_{i \in A} \mathtt{dist}(s_i, g_i)$. This score works as the upper bound of sub-optimality, where the minimum is one. For "runtime" and "cost," median scores of solved instances within each solver are displayed. The figure further shows the minimum and maximum scores using semi-transparent regions. The success rate of LaCAM was based on the number of successful trials over total trials.

all instances, however, the search effort dramatically increases with more agents. This is because, with more agents, the high-level search can contain a huge number of *slightly different configurations*. Those configurations differ only one or two agents differ in their locations and disturb the progression of the search.

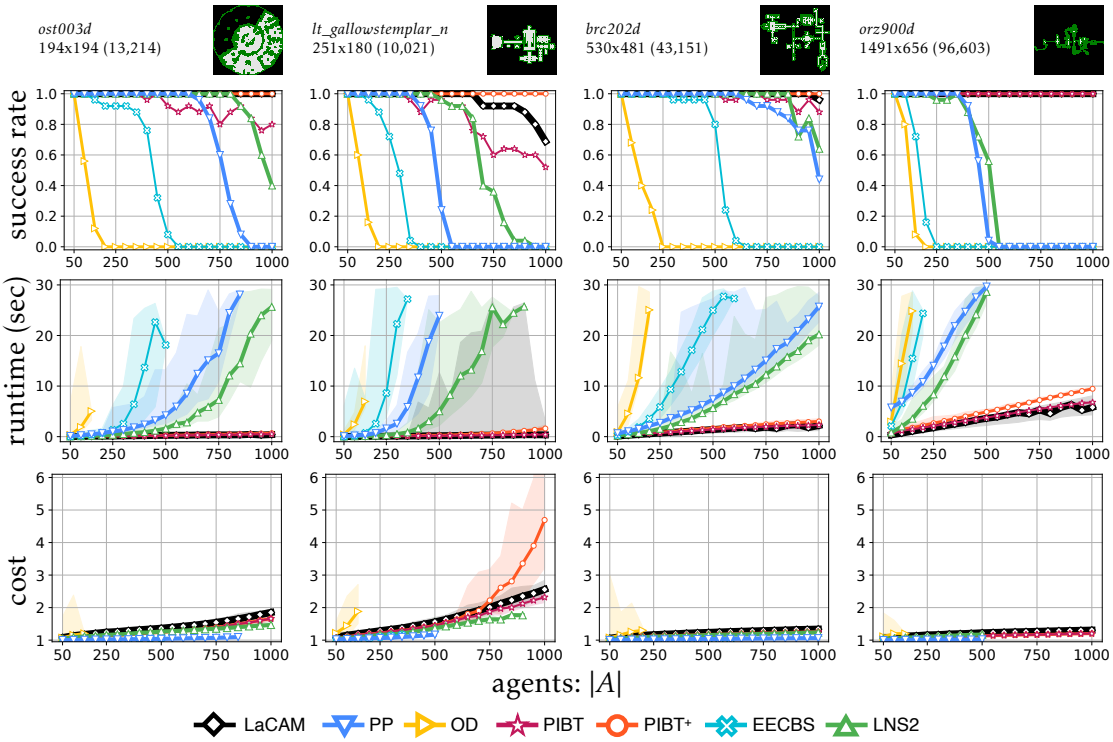
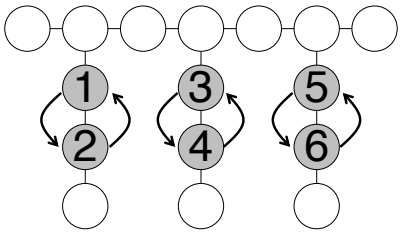


Figure 6.6: Results of the MAPF benchmark (2/2). See the caption of Fig. 6.5 for details.



$\|A\|$	search iteration
2	128
4	23,907
6	287,440

Table 6.2: LaCAM performance in an adversarial instance.

Discussion. From this observation, the poor performance in several grids of Fig. 6.5 and 6.6 is considered as follows. The LaCAM implementation used a DFS style, therefore, once a configuration similar to Table 6.2 appears during the search, resolving this configuration towards the goal configuration requires significant search effort. Consequently, LaCAM reaches the timeout. Overcoming this limitation is one promising direction. One resolution might be developing a better configuration generator other than vanilla PIBT, which will be further investigated in Chap. 6.6.

6.4.5 Scalability Test

Setup. Next, the scalability of the number of agents was evaluated, using instances with up to 10,000 agents in *warehouse-20-40-10-2-2*. The runtime limit was set to 1000 s. OD was excluded since it run out of memory.

Result. Figure 6.7 summarizes the result. Only LaCAM solved all instances. Furthermore, the runtime was in at most 30 s even with 10,000 agents, demonstrating the excellent scalability of LaCAM. Note that LaCAM can be faster depending on computational environments. As a pilot study, the same setting was tested with a single-thread run

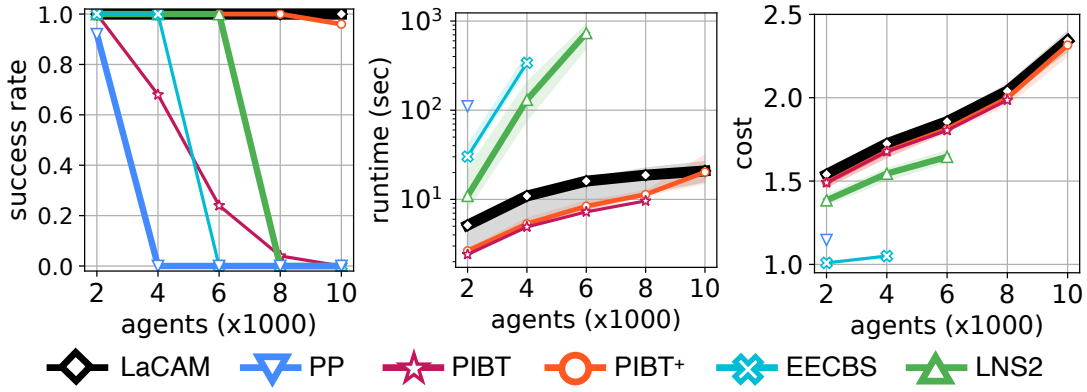


Figure 6.7: Results with massive agents. The used map was *warehouse-20-40-10-2-2*.

in a laptop with Intel Core i9 2.3 GHz CPU and 16 GB RAM. Even with 10,000 agents, LaCAM solved all instances at most in 10 s in the worst case.

6.4.6 Design Choice of LaCAM

Setup. Finally, the design choice of LaCAM implementation was investigated. Specifically, two variants were tested:

- **DFS** does not use the reinsert operation at the high-level. See Chap. 6.3.4.
- **GREEDY** uses another configuration generator instead of PIBT, such that agents greedily determines the location according to the priority order. This is equivalent to PP with a single-step planning horizon.

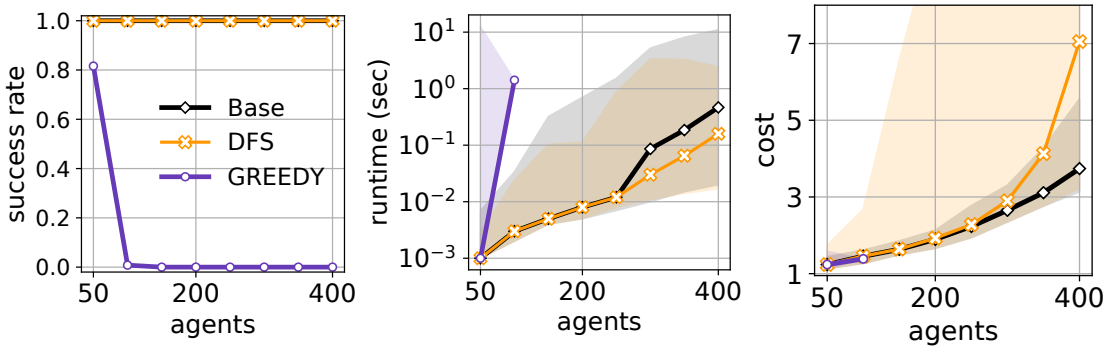


Figure 6.8: Results with different LaCAM designs. The used map was *random-32-32-20*.

Result. Figure 6.8 shows the result in *random-32-32-20*. There are two observations:

- The reinsert operation improves the sum-of-costs metric.
- The choice of configuration generator significantly affects the search as seen in that GREEDY failed in most settings. Consequently, the excellent performance of LaCAM in Fig. 6.5 and 6.6 relies on promising successor generation at the high-level, which is done by PIBT.

6.4.7 Discussion

Overall, the empirical results in this section demonstrate that LaCAM with PIBT is very promising, i.e., solving various types of MAPF in a very short time, and having excellent scalability.

Why is LaCAM quick? In general, the average branching factor largely determines the search effort. A vanilla A* for MAPF generates $O\left(\Delta^A\right)$ configurations from one search node, which is intractable especially when $|A|$ is large. In contrast, each high-level node of LaCAM initially generates only one successor (i.e., configuration), and if necessary, gradually generates other successors in a lazy manner. This scheme virtually suppresses the branching factor of LaCAM. If the generated successor is promising (i.e., closer toward the goal configuration from the original), LaCAM can dramatically reduce the number of node generations. Promising successors can be quickly obtained by techniques of recent MAPF studies such as PIBT. This is a trick of quickness in LaCAM. Although virtually reducing the branching factor in A* (aka. *partial expansion*) for MAPF has been proposed [Standley, 2010; Goldenberg *et al.*, 2014; Wagner and Choset, 2015], those studies never achieve the dramatic reductions as LaCAM.

Shortcomings. LaCAM with PIBT achieved excellent performance, meanwhile, several shortcomings have already been observed, that is, *(i)* solution quality and *(ii)* solvability in instances with narrow corridors. Therefore, the following two sections try to resolve these shortcomings. The first section (Chap. 6.5) is for *(i)* solution quality, while the second section (Chap. 6.6) is for *(ii)* improving solvability by improving the configuration generator (i.e., PIBT).

6.5 LaCAM*: Eventually Optimal Algorithm

Thus far, LaCAM is presented as a sub-optimal algorithm; it neglects solution quality. This section improves this aspect by presenting an anytime version of LaCAM called *LaCAM**. The main difference from LaCAM is that it rewrites the search tree as needed, while updating the accumulated transition costs of nodes from the start node. Such rewriting schemes are partially seen in asymptotically optimal motion planning studies, e.g., [Karaman and Frazzoli, 2011; Shome *et al.*, 2020].

Optimization Problem. As a solution cost for an MAPF solution $\mathbf{\Pi}$, this section assumes accumulative transition costs, taking the form of:

$$\mathtt{cost}(\mathbf{\Pi}) := \sum_{t=0}^{\mathtt{length}(\mathbf{\Pi})-1} \mathtt{cost}_e\Big(\mathbf{\Pi}[t], \mathbf{\Pi}[t+1]\Big) \tag{6.1}$$

where $\mathtt{cost}_e : V^{|A|} \times V^{|A|} \mapsto \mathbb{R}_{\geq 0}$ returns a transition cost between two configurations. In other words, we are interested in minimizing Eq. (6.1). See also Chap. 3.2.2 that describes solution costs for MAPF.

Admissible Heuristic. This section assumes an *admissible heuristic* $\mathsf{h} : V^{|A|} \mapsto \mathbb{R}_{\geq 0}$, such that $\mathsf{h}(Q)$ never overestimates the optimal cost from $\mathcal{Q}$ to $\mathcal{G}$. For instance, $\mathsf{h}(\mathcal{Q}) := \sum_{i \in A} \mathtt{dist}(\mathcal{Q}[i], g_i)$ is available for sum-of-{loss, fuels}, while $\mathsf{h}(\mathcal{Q}) := \max_{i \in A} \mathtt{dist}(\mathcal{Q}[i], g_i)$ exists for makespan. Since $\mathtt{cost}_e(\cdot) \geq 0$, a constant function $\mathsf{h}(\mathcal{Q}) := 0$ is always admissible, meaning that, there are admissible heuristics for any form of Eq. (6.1).

6.5.1 Pseudocode

Algorithm 6.2 presents LaCAM*. The same parts as LaCAM (Alg. 6.1) are gray-out. The added parts from Alg. 6.1 are displayed in black. The blue-colored lines are not necessary from the theoretical side but are effective in speeding up the search. The low-level part of LaCAM is abstracted in Line 10. For the convenience of notations, the transition cost function $\mathtt{cost}_e$ and admissible heuristic function $\mathtt{h}$ can take search nodes (corresponding to configurations) as arguments.

The main differences from LaCAM are twofold: *(i)* it continues the search when finding the goal configuration $\mathcal{G}$, and, *(ii)* it rewrites parent relationships between search nodes as necessary. In what follows, the updated parts and their intentions are explained.

Keeping Goal Node. LaCAM* keeps the goal node $\mathcal{N}^{\mathrm{goal}}$, rather than immediately returning solutions when first finding the goal configuration (Line 6). The search terminates when there is no remaining search node otherwise there is an interruption from users such as timeout (Line 4). A solution is then constructed by backtracking from $\mathcal{N}^{\mathrm{goal}}$ (Lines 27–28). Doing so makes LaCAM* an anytime algorithm. In other words, *after finding the goal node, it is interruptible whenever a solution is required, while gradually refining solution quality as time allows.*

Search Node Ingredients. Each search node contains a set *neigh* that stores connected configurations (i.e., nodes) and g-value that represents cost-to-come from the start configuration (Line 2), equivalent to g-value in A*. They are initialized and updated appropriately when finding a new configuration (Lines 24–26).

Updating Parents and Costs. The crux of LaCAM* lies in operations done when finding an already known configuration. It first updates *neigh* (Line 14) and then updates the node cost and *parent* to maintain the optimality of the search tree, explained later in the proof. This is done by an adaptation of Dijkstra's algorithm (Lines 15–21). The visual illustration is available in Fig. 6.9.

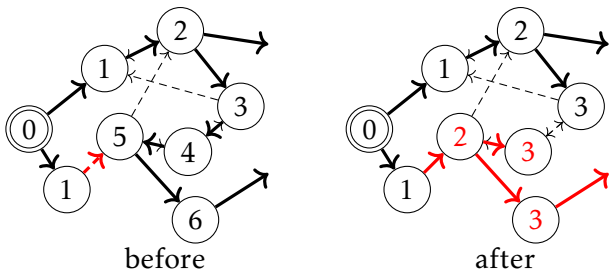


Figure 6.9: Updating parents and costs. Each circle is a search node (i.e., configuration), including its cost (i.e., g) of the makespan metric. Arrows represent known neighboring relationships. Among them, solid lines correspond to *parent*. The updated parts are red-colored.

Discarding Redundant Nodes. Once the goal node is found, LaCAM* can discard nodes that do not contribute to improving solution quality (Line 7). To maintain the optimality, LaCAM* revives nodes as necessary when their cost is updated (Line 22).

Algorithm 6.2 LaCAM*.

input: MAPF instance, transition cost $\texttt{cost}_e$, admissible heuristic $\texttt{h}$

output: solution, `NO_SOLUTION`, or `FAILURE`

notation: $\texttt{f}(\mathcal{N}) := \mathcal{N}.g + \texttt{h}(\mathcal{N})$; $\spadesuit := \left(\mathcal{N}^{\text{goal}} \neq \bot\right)$

1: initialize *Open*, *Explored*; $\mathcal{N}^{\text{goal}} \leftarrow \bot$
2: $\mathcal{N}^{\text{init}} \leftarrow \left\langle \mathit{config} : \mathcal{S}, \mathit{tree} : [\![\, \mathcal{C}^{\text{init}} \,]\!], \mathit{parent} : \bot, \mathit{neigh} : \emptyset, g : 0 \right\rangle$
3: $\mathit{Open}.\texttt{push}\left(\mathcal{N}^{\text{init}}\right)$; $\mathit{Explored}[\mathcal{S}] = \mathcal{N}^{\text{init}}$
4: **while** $\mathit{Open} \neq \emptyset \ \wedge \ \neg\texttt{interrupt}()$ **do**
5: $\quad \mathcal{N} \leftarrow \mathit{Open}.\texttt{top}()$
6: $\quad$ **if** $\mathcal{N}.\mathit{config} = \mathcal{G}$ **then** $\mathcal{N}^{\text{goal}} \leftarrow \mathcal{N}$
7: $\quad$ **if** $\spadesuit \wedge \texttt{f}(\mathcal{N}^{\text{goal}}) \leq \texttt{f}(\mathcal{N})$ **then** $\mathit{Open}.\text{pop}()$; **continue**
8: $\quad$ **if** $\mathcal{N}.\mathit{tree} = \emptyset$ **then** $\mathit{Open}.\text{pop}()$; **continue**
9: $\quad \mathcal{C} \leftarrow \mathcal{N}.\mathit{tree}.\text{pop}()$
10: $\quad \texttt{low\_level\_expansion}(\mathcal{N}, \mathcal{C})$ ▷ *Lines 9–13 of Alg. 6.1*
11: $\quad \mathcal{Q}^{\text{new}} \leftarrow \texttt{configuration\_generator}(\mathcal{N}, \mathcal{C})$
12: $\quad$ **if** $\mathcal{Q}^{\text{new}} = \bot$ **then continue**
13: $\quad$ **if** $\mathit{Explored}[\mathcal{Q}^{\text{new}}] \neq \bot$ **then**
14: $\quad\quad \mathcal{N}.\mathit{neigh}.\text{append}(\mathit{Explored}[\mathcal{Q}^{\text{new}}])$
15: $\quad\quad D \leftarrow [\![\, \mathcal{N} \,]\!]$ ▷ *Dijkstra, priority queue of ascending order of g*
16: $\quad\quad$ **while** $D \neq \emptyset$ **do**
17: $\quad\quad\quad \mathcal{N}^{\text{from}} \leftarrow D.\text{pop}()$
18: $\quad\quad\quad$ **for** $\mathcal{N}^{\text{to}} \in \mathcal{N}^{\text{from}}.\mathit{neigh}$ **do**
19: $\quad\quad\quad\quad g \leftarrow \mathcal{N}^{\text{from}}.g + \texttt{cost}_e\left(\mathcal{N}^{\text{from}}, \mathcal{N}^{\text{to}}\right)$
20: $\quad\quad\quad\quad$ **if** $g < \mathcal{N}^{\text{to}}.g$ **then**
21: $\quad\quad\quad\quad\quad \mathcal{N}^{\text{to}}.g \leftarrow g$; $\mathcal{N}^{\text{to}}.\mathit{parent} = \mathcal{N}^{\text{from}}$; $D.\text{push}(\mathcal{N}^{\text{to}})$
22: $\quad\quad\quad\quad\quad$ **if** $\spadesuit \wedge \texttt{f}\left(\mathcal{N}^{\text{to}}\right) < \texttt{f}\left(\mathcal{N}^{\text{goal}}\right)$ **then** $\mathit{Open}.\texttt{push}\left(\mathcal{N}^{\text{to}}\right)$
23: $\quad$ **else**
24: $\quad\quad g \leftarrow \mathcal{N}.g + \texttt{cost}_e(\mathcal{N}, \mathcal{Q}^{\text{new}})$
25: $\quad\quad \mathcal{N}^{\text{new}} \leftarrow \left\langle \mathit{config} : \mathcal{Q}^{\text{new}}, \mathit{tree} : [\![\, \mathcal{C}^{\text{init}} \,]\!], \mathit{parent} : \mathcal{N}, \mathit{neigh} : \emptyset, g : g \right\rangle$
26: $\quad\quad \mathit{Open}.\text{push}(\mathcal{N}^{\text{new}})$; $\mathit{Explored}[\mathcal{Q}^{\text{new}}] = \mathcal{N}^{\text{new}}$; $\mathcal{N}.\mathit{neigh}.\text{append}(\mathcal{N}^{\text{new}})$
27: **if** $\spadesuit \wedge \mathit{Open} = \emptyset$ **then return** $\texttt{backtrack}\left(\mathcal{N}^{\text{goal}}\right)$ ▷ *optimal solution*
28: **else if** $\spadesuit$ **then return** $\texttt{backtrack}\left(\mathcal{N}^{\text{goal}}\right)$ ▷ *sub-optimal solution*
29: **else if** $\mathit{Open} = \emptyset$ **then return** `NO_SOLUTION` ▷ *unsolvable instance*
30: **else return** `FAILURE`

6.5.2 Theoretical Analysis

Theorem 6.2. *Without interruption, LaCAM* (Alg. 6.2) returns an optimal solution with respect to Eq. (6.1) for solvable instances, otherwise reports* `NO_SOLUTION`.

Proof. Consider first Alg. 6.2 without the blue lines (Line 7, Line 22); they just speed up the search. In this proof, the term "path" refers to a sequence of connected configurations.

The proof introduces a directed graph H, where its vertex corresponds to one configuration. Initially, H consists of a single vertex of the start configuration $\mathcal{S}$. Then, H is built gradually according to the search iteration of LaCAM*. When a new search node is created (Line 25), its configuration is added to H. Arcs from a vertex in H follow *neigh* of the corresponding search node, i.e., an arc occurs when the search finds a connection from the vertex.

The proof now states that: *(♣) for any configuration $\mathcal{Q}$ in H, a path from $\mathcal{S}$ to $\mathcal{Q}$ constructed by backtracking (i.e., by following $\mathcal{N}$.parent) is the shortest path in H, defined with Eq. (6.1), at any phase of the search*. In other words, there are no paths with a smaller cost than the path from $\mathcal{S}$ to $\mathcal{Q}$ in H.

The statement ♣ is derived by induction. Initially, H comprises a single vertex $\mathcal{S}$, clearly satisfying ♣. Assume now that ♣ is satisfied in the previous iteration of Lines 4–26. In the next iteration, H is updated by:

- generating a new configuration (Lines 25–26), or
- finding a known configuration (Lines 14–21).

The former case holds ♣ since only a new vertex and an arc toward the vertex are added to H. The latter case also holds ♣ due to the following reason. Observe that only one arc is added to H from the previous iteration, regarding the search node $\mathcal{N}$ (Line 14); it is sufficient to consider the effect of this addition. Lines 15–21 essentially performs the Dijkstra algorithm starting from $\mathcal{N}$. Therefore, the shortest paths from $\mathcal{N}$ to other configurations are constructed, or, the remaining configurations have paths with smaller costs from the start $\mathcal{S}$, without passing through $\mathcal{N}$. Consequently, the statement ♣ holds.

LaCAM* is ensured to terminate; the search space of LaCAM* is also finite, as already presented in the proof of Thrm. 6.1. Moreover, it examines all *reachable* configurations from $\mathcal{S}$, defined by transitivity over connections of two configurations. Consequently, when LaCAM* terminates, H includes all possible paths from $\mathcal{S}$ to $\mathcal{G}$ for solvable instances. Together with ♣, LaCAM* returns optimal solutions, otherwise, reports the non-existence.

Finally, the blue lines are complemented as follows. Line 7 prunes search nodes with larger costs than that of the known best solution because they do not contribute to finding better solutions. However, depending on the manner of constructing H, discarded nodes due to overestimated costs potentially lay in optimal paths. LaCAM* thus re-inserts such nodes in Line 22 when costs are updated. By Line 22, Line 7 does not break the optimal guarantee. □

6.5.3 Implementation Tips

Following Chap. 6.3.4, when finding an already known configuration at Line 13, our implementation reinserts the corresponding node to *Open*. Moreover, with a small probability (e.g., 0.1%), our implementation reinserts a node of the start configuration instead of the found one. Doing so enables the search to "escape" from configurations being bottlenecks. Such techniques relying on non-determinism have already appeared in other search problems [Kautz *et al.*, 2002] as well as MAPF studies [Andreychuk and Yakovlev, 2018; Cohen *et al.*, 2018b]. Indeed, it was informally observed that this random replacement slightly improved the success rate. Note that the optimality (Thrm. 6.2) still holds with these modification.

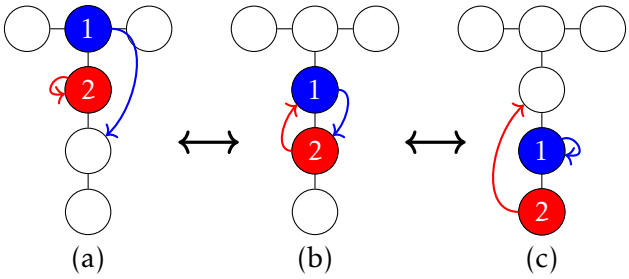


Figure 6.10: Failure example of PIBT. Agents are shown with colored circles. Their goals are represented by arrows.

6.6 Improving Configuration Generator

The performance of LaCAM heavily relies on a configuration generator, therefore, the development of a good generator is the heart of LaCAM. The implementation in Chap. 6.4 uses a vanilla PIBT, resulting in poor performances in several scenarios, especially in instances with narrow corridors. In short, this is because PIBT itself often fails such scenarios, hence being ineffective guides for long-horizon planning. This section elaborates on this phenomenon and presents its countermeasure.

6.6.1 Failure Analysis of PIBT

As explained in Chap. 4, PIBT is priority-based. For each timestep, it tries to make the agent with highest priority move one step ahead toward its goal. Once an agent reaches its goal, the agent drops its priority, while the others increase their priorities. Consequently, PIBT assigns high priorities to agents being not on their goals. This scheme is sufficient for scenarios wherein all agents are not necessarily being goals simultaneously, such as MAPD [Ma *et al.*, 2017b]. However, in MAPF, livelock situations can be triggered. See Fig. 6.10. In the figure, two agents reach their destinations periodically but PIBT never reaches the goal configuration.

LaCAM can break such livelocks by adding constraints. However, it may require significant effort because appropriate combinations of constraints should be explored. Even worse, with more agents, the search effort dramatically increases, as explicitly shown in Table 6.2 of the prior experiment.

6.6.2 Enhancing PIBT by Swap

Livelocks in PIBT can be handled by "swap" operation, originally developed in rule-based approaches [Luna and Bekris, 2011; De Wilde *et al.*, 2014]. Roughly describing, the swap operation swaps the locations of two agents using a vertex with a degree of three or more. Figure 6.11 shows its example. The last two steps were omitted since they are trivial, i.e., just moving two agents toward their goals. The rule-based algorithms themselves have several critiques as discussed so far. However, incorporating flavors of such rules into PIBT is possible by adjusting vertex scoring. More precisely, this is done by adjusting Line 6 in Alg. 4.1.

Algorithm 6.3 extends the PIBT function from Alg. 4.2. The same parts as Alg. 4.2 are gray-out. The modification is simple; if the agent i and neighboring agent j are judged to require swap (Line 4), i reverses the order of candidate vertices (Line 5). In other words, i tries to be apart from goals. Then, if i successfully moves to the first vertex in the candidate set C, i "pulls" j to the current occupying vertex (Line 11). With appropriate implementations of the function `swap_required_and_possible`, PIBT does not fall into

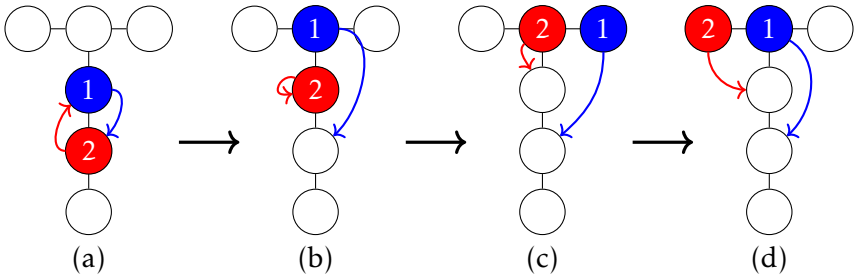


Figure 6.11: Swap operation.

Algorithm 6.3 PIBT **with swap.**

```
1: procedure PIBT_swap(i)
2:     C ← neigh(Q^from[i]) ∪ {Q^from[i]}
3:     sort C in increasing order of dist(u, g_i) where u ∈ C
4:     j ← swap_required_and_possible(i, C[1], Q^from)      ▷ index of C starts from one
5:     if j ≠ ⊥ then reverse C
6:     for v ∈ C do
7:         if collision occurs in Q^to supposing Q^to[i] = v then continue
8:         Q^to[i] ← v
9:         if ∃j ∈ A s.t. Q^from[j] = v ∧ Q^to[j] = ⊥ then
10:            if PIBT_swap(j) = INVALID then continue
11:        if v = C[1] ∧ j ≠ ⊥ ∧ Q^to[j] = ⊥ then Q^to[j] ← Q^from[i]
12:        return VALID
13:    Q^to[i] ← Q^from[i]
14:    return INVALID
```

the livelock of Fig. 6.10, rather, it generates a sequence of configurations depicted in Fig. 6.11.

The `swap_required_and_possible` function is a pattern detector that should be manually designed, i.e., it is *implementation dependent*. Note, *the section does not aim at designing complete detectors because pitfalls of the detector can be complemented by LaCAM*. However, a well-tuned implementation can relax the search effort of LaCAM. Below, our example implementation is illustrated., while omitting tiny fine-tunings.

Pattern Detector Implementation

Assume that `swap_required_and_possible` is called. Assume further that another agent j is on v such that $\deg(v) \leq 2$. The implementation uses two emulations.

The first emulation asks about the necessity of the swap. This is done by continuously moving i to j's location while moving j to another vertex not equal to i's location, ignoring the other agents. The emulation stops in two cases:

- The swap is *not required* when j's location has a degree of more than two.
- The swap is *required* when j's location has a degree of one, or, when i reaches its goal while j's nearest neighboring vertex towards its goal is i's goal.

If the swap is required, the second emulation asks about the possibility of the swap. This is done by continuously moving j to i's location while moving i to another vertex, i.e., reversing the emulation direction of the first emulation. It stops in two cases:

- The swap is *possible* when i's location has a degree of more than two.
- The swap is *impossible* when i is on a vertex with degree of one.

The function returns j when the swap is required and possible. For instance, in configurations of Fig. 6.11(a,b), the swap is required for both agent-$(1, 2)$. In contrast, the swap is possible only for agent-1, then, the candidate set of the agent is reversed (Line 5). Consequently, Alg. 6.3 generates configurations of Fig. 6.11(b,c). An exception is the case of Fig. 6.11c, where agent-2 needs to reverse the candidate sets to generate a configuration of Fig. 6.11d. It is noted that the detection is possible by slightly changing the above two emulations.

6.7 Evaluation of Improvements

This section evaluates two techniques presented in Chap. 6.5 and 6.6. Specifically, the section presents the following empirical results:

- Chapter 6.7.2 sees how the improved configuration generator reduces planning effort of LaCAM.
- Chapter 6.7.3 sees how LaCAM* refines solution.
- Chapter 6.7.4 evaluates the effect of discarding redundant nodes.
- Chapter 6.7.5 evaluates LaCAM* with small complicated instances.
- Chapter 6.7.6 evaluates LaCAM* with the MAPF benchmark.
- Chapter 6.7.7 compares LaCAM* with another anytime MAPF algorithm.
- Chapter 6.7.8 evaluates LaCAM* with extremely congested scenarios.

6.7.1 Experimental Setup

Baselines. A variety of representative or state-of-the-art MAPF algorithms that diverse properties for solvability and optimality were tested as follows. See also Fig. 6.13.

- **A*** [Hart *et al.*, 1968] as a vanilla search algorithm. It is complete and optimal. The used objective is makespan (A*-m) and sum-of-loss (A*-l).
- **A* with operator decomposition (OD)** [Standley, 2010] as an adaptation of the general search to MAPF. It was implemented as a greedy search to obtain solutions as much as possible. The heuristic was the sum of distance towards goals.
- **ODrM*** [Wagner and Choset, 2015] as a state-of-the-art optimal and complete algorithm, based on graph pathfinding. The used objective is sum-of-loss (ODrM*-l) and makespan (ODrM*-m). The implementation was retrieved from `https://github.com/gswagner/mstar_public`. The original implementation uses sum-of-loss as an objective function. The makespan version was adapted from it.
- **Inflated ODrM* (I-ODrM*)** [Wagner and Choset, 2015] as a state-of-the-art bounded sub-optimal and complete algorithm, which is a variant of ODrM*. The used objectives were makespan (I-ODrM*-m) and sum-of-loss (I-ODrM*-l). The sub-optimality was set to five to find solutions as much as possible.

- **BCP** [Lam *et al.*, 2022] as a state-of-the-art optimal solver, a representative of two-level approaches using reduction to a mathematical optimization problem. BCP is solution complete, namely, it cannot distinguish unsolvable instances. It used CPLEX [Cplex, 2009] for mathematical optimization. The implementation was retrieved from `https://github.com/ed-lam/bcp-mapf`. The used objective was flowtime (aka. sum-of-costs), following the original implementation.
- **CBS** [Sharon *et al.*, 2015] with many improvement techniques as appeared in [Li *et al.*, 2021b], as a state-of-the-art optimal solver. This is representative of two-level approaches using combinatorial search. CBS is solution complete. The implementation was retrieved from `https://github.com/Jiaoyang-Li/CBSH2-RTC`. The used objectives were flowtime (CBS-l), makespan (CBS-m), and sum-of-loss (CBS-l). The original implementation uses flowtime as an objective function. The makespan and sum-of-loss versions were adapted from it.
- **EECBS** [Li *et al.*, 2021c] as a state-of-the-art bounded sub-optimal but solution complete algorithm, which is a variant of CBS. The used objectives were flowtime (EECBS-f), makespan (EECBS-m), and sum-of-loss (EECBS-l). The sub-optimality was set to five. The implementation was retrieved from `https://github.com/Jiaoyang-Li/EECBS`. The original implementation uses flowtime as an objective function. The makespan and sum-of-loss versions were adapted from it.
- **Prioritized planning (PP)** [Erdmann and Lozano-Perez, 1987; Silver, 2005] as a basic approach for MAPF. The implementation first uses distance heuristics [Van Den Berg and Overmars, 2005] for the planning order. Furthermore, it involves the repetition of PP with random priorities until the problem is solved. The implementation was adapted from `https://github.com/Kei18/pibt2`.
- **MAPF-LNS2 (LNS2)** [Li *et al.*, 2022] as a state-of-the-art sub-optimal and incomplete solver, based on a large neighborhood search. The implementation was retrieved from `https://github.com/Jiaoyang-Li/MAPF-LNS2`.
- **PIBT** [Okumura *et al.*, 2022b], which is an incomplete and sub-optimal algorithm. A vanilla PIBT was tested because LaCAM used PIBT. To detect planning failure, PIBT was regarded as a failure when it reached pre-defined sufficiently large timesteps. The implementation was retrieved from `https://github.com/Kei18/pibt2`.
- **PIBT$^+$** [Okumura *et al.*, 2022b], as a state-of-the-art scalable MAPF solver, which is incomplete and sub-optimal. It used push and swap [Luna and Bekris, 2011] to complement solutions.
- **LaCAM** [Okumura, 2023], used in Chap. 6.4, which is a complete sub-optimal algorithm. It used a vanilla PIBT as a configuration generator. The implementation was from `https://github.com/Kei18/lacam`.

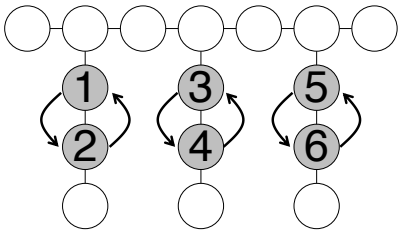
The section additionally rates the following anytime MAPF algorithm.

- **AFS** [Cohen *et al.*, 2018a], an anytime version of CBS that eventually converges to optima. its implementation was obtained from the authors of the paper. The original implementation uses flowtime as an objective function. The sum-of-loss versions were adapted from it.

Experimental Environment. The experiments were run on a desktop PC with Intel Core i7-7820X 3.6 GHz CPU and 32 GB RAM. A maximum of 16 different instances was run in parallel using multi-threading. LaCAM* was coded in C++. Unless mentioned, this section uses a timeout of 30 s for solving MAPF.

6.7.2 Effect of Improved Configuration Generator

Table 6.3 presents the number of search iterations of LaCAM on an instance that requires "swap," using a vanilla PIBT (Alg. 4.2) and the improved one (Alg. 6.3) as a configuration generator. Table 6.4 further compares the generators with larger instances. The results show that Alg. 6.3 dramatically reduced the search iterations of LaCAM, contributing to smaller computation time in large instances. Note however that the pattern detector has runtime overhead, as seen in $|A| = 100$ of Table 6.4.



$\|A\|$	w/Alg. 4.2	w/Alg. 6.3
2	128	**6**
4	23,907	**8**
6	287,440	**8**

Table 6.3: The number of search iterations of LaCAM to solve instances. The table corresponds to Table 6.2. A vanilla PIBT (Alg. 4.2) and PIBT with swap (Alg. 6.3) are compared. When $|A| = 2$, only agents-1, 2 appear, and so forth.

	search iterations				runtime (ms)			
$\|A\|$	w/Alg. 4.2		w/Alg. 6.3		w/Alg. 4.2		w/Alg. 6.3	
100	374	(344,54468)	366	(338,401)	65	(31,1218)	112	(34,216)
300	54802	(388,369131)	392	(357,482)	3049	(291,18858)	301	(187,409)
500	181459	(44534,268724)	410	(391,432)	18063	(4598,29820)	500	(347,574)

Table 6.4: Effect of configuration generators. For each $|A|$, median, min, and max scores are presented for instances solved by both algorithms among 25 instances on *warehouse-20-40-10-2-1*.

6.7.3 Refinement of LaCAM*

Figure 6.12 shows how LaCAM* refines solutions. As baselines, the experiment used scores of a complete and optimal algorithm , (I-)ODrM*. In the small instances, LaCAM* quickly found initial solutions and converged to optimal ones. Meanwhile, the convergence speed was slow in the large instances with many agents. This is due to finding new connections between known configurations becoming rare, hence reducing the chance of rewriting the search tree.

6.7.4 Effect of Discarding Redundant Nodes

Table 6.5 shows how discarding redundant search nodes (blue lines of Alg. 6.2) affects the search to identify optimal solutions. Regardless of the generators, the discarding dramatically reduced the search effort. The reduction was larger with Alg. 6.3 because initial solutions can be found with smaller search iterations than Alg. 4.2. Note that without the discarding, the numbers of search iterations are equivalent between Alg. 4.2 and Alg. 6.3 because the search spaces are identical.

In the remaining parts, LaCAM* uses Alg. 6.3 while LaCAM denotes the original implementation that uses Alg. 4.2.

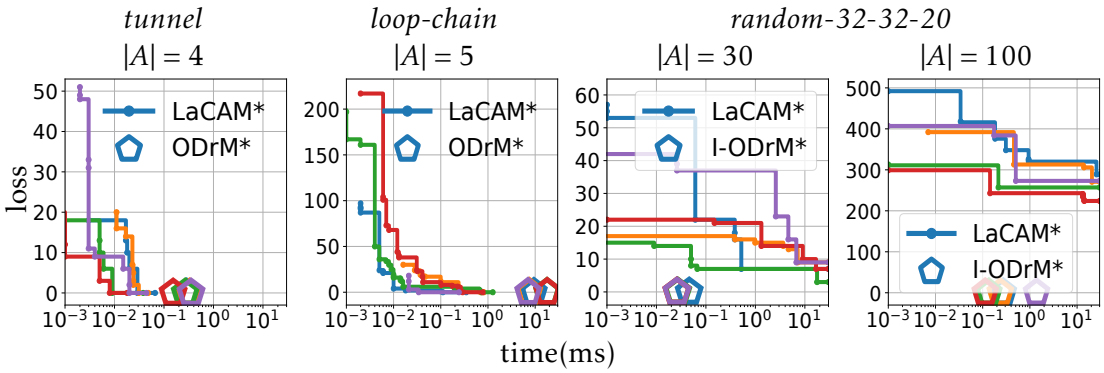


Figure 6.12: Refinement of LaCAM*. Three maps were used, shown in Table 6.1 and Fig. 6.5. For each chart, five identical instances were used where starts and goals were set randomly. The optimization was for sum-of-loss. "loss" shows the gaps from scores of (I-)ODrM*. In *random-32-32-20*, the bounded sub-optimal version with sub-optimality of 1.5 was used because ODrM* failed to solve the instances. LaCAM* used Alg. 4.2 as a configuration generator.

	tree	*corners*	*tunnel*	*string*	*loop-chain*	*connector*
no discard	10K	2M	410M	19M	N/A	N/A
w/Alg. 4.2	1K	28K	287K	103K	N/A	N/A
w/Alg. 6.3	**1K**	**1K**	**199K**	**103K**	N/A	N/A

Table 6.5: The number of search iterations for termination. "no discard" is blue-lines omitted version of LaCAM*. The metric was for makespan. The instances are displayed in Table 6.1. In the last two instances, LaCAM* did not terminate before the timeout.

6.7.5 Small Complicated Instances

Table 6.6 summarizes the result of small complicated instances, which are the same setting used in Chap. 6.4.2. Both two types of LaCAM* were evaluated, one for makespan (LaCAM*-m) and another for sum-of-loss (LaCAM*-l). Note that flowtime (aka. sum-of-costs) is difficult to represent by Eq. (6.1); see Chap. 3.2.2. Overall, LaCAM* immediately found not only initial solutions but also (near-)optimal ones. In contrast, the baselines failed some instances or returned low-quality solutions.

6.7.6 MAPF Benchmark

Next, LaCAM* was tested on the MAPF benchmark [Stern *et al.*, 2019]. This time, as already introduced in Chap. 3.6.2, 13,900 instances on 33 four-connected grid maps were retrieved from the benchmark. Note, when increasing the number of agents by 50, we stopped testing some algorithms if they failed to solve all instances in the previous round to avoid unnecessary computation.

Results

The percentage of solved instances is summarized in Fig. 6.13. Figure 6.14 presents partial results for each map. The full results are available in Appendix D.1. Overall, LaCAM* solved a variety of problem instances that have diverse sizes of graphs or agents, sparseness, and complexity, within several seconds. Indeed, LaCAM* only failed in the instances of *maze-128-128-1* and sub-optimally solved all the other instances within 10 s, outperforming the other algorithms. The failures might be reduced by improving the

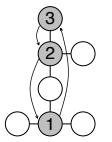
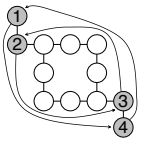
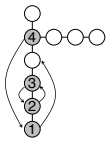
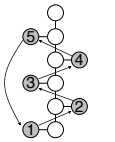
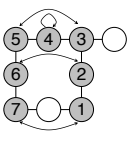
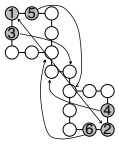
		tree		*corners*		*tunnel*		*string*		*loop-chain*		*connector*		
														
		time	s-opt	time	s-opt	time	s-opt	time	s-opt	time	s-opt	time	s-opt	solved
makespan	LaCAM*-m	0	1.20	0	1.23	0	1.41	0	1.81	2	6.58	0	1.62	**6/6**
	after 1 s	0	1.00	2	1.00	6	1.00	7	1.00	578	1.35	226	1.00	
	A*-m	0	1.00	0	1.00	30	1.00	27	1.00	11125	1.00	N/A	N/A	5/6
	ODrM*-m	5	1.00	2	1.00	396	1.00	402	1.00	N/A	N/A	N/A	N/A	4/6
	I-ODrM*-m	1	1.00	0	1.50	70	1.07	2	1.25	N/A	N/A	N/A	N/A	4/6
	CBS-m	71	1.00	0	1.00	N/A	N/A	149	1.00	N/A	N/A	N/A	N/A	3/6
	EECBS-m	2	1.00	1	1.00	N/A	N/A	0	1.00	N/A	N/A	N/A	N/A	3/6
	OD	0	1.00	0	1.88	14	2.73	0	1.25	2133	31.22	5	1.50	**6/6**
	LaCAM	0	1.17	1	2.12	92	2.00	0	2.25	55	17.83	0	1.56	**6/6**
	PP	N/A	N/A	0	1.00	N/A	N/A	0	1.00	N/A	N/A	N/A	N/A	2/6
	LNS2	N/A	N/A	0	1.00	N/A	N/A	0	1.00	N/A	N/A	29	1.00	3/6
	PIBT	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0/6
	PIBT+	0	3.50	0	1.25	0	4.07	0	2.12	N/A	N/A	0	1.81	5/6
sum-of-loss	LaCAM*-l	0	1.25	0	1.12	0	1.46	0	2.36	2	7.12	0	(1.48)	**6/6**
	after 1 s	0	1.00	13	1.00	77	1.00	8	1.00	700	1.44	408	(1.08)	
	A*-l	0	1.00	149	1.00	35	1.00	25	1.00	15124	1.00	N/A	N/A	5/6
	ODrM*-l	3	1.00	40	1.00	675	1.00	0	1.00	N/A	N/A	N/A	N/A	4/6
	I-ODrM*-l	0	1.00	0	1.31	257	1.08	0	1.00	N/A	N/A	124	(1.39)	5/6
	CBS-l	70	1.00	0	1.00	N/A	N/A	180	1.00	N/A	N/A	N/A	N/A	3/6
	EECBS-l	2	1.00	0	1.00	N/A	N/A	0	1.00	N/A	N/A	86	(1.61)	4/6
	OD	0	1.00	0	1.50	14	2.57	0	1.20	2133	30.62	5	(1.38)	**6/6**
	LaCAM	0	1.23	1	1.69	92	1.91	0	3.30	55	19.15	0	(1.45)	**6/6**
	PP	N/A	N/A	0	1.00	N/A	N/A	0	1.00	N/A	N/A	N/A	N/A	2/6
	LNS2	N/A	N/A	0	1.00	N/A	N/A	0	1.00	N/A	N/A	29	(1.00)	3/6
	PIBT	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0/6
	PIBT+	0	4.38	0	1.12	0	3.91	0	2.20	N/A	N/A	0	(1.68)	5/6
	BCP	194	-	150	-	N/A	-	117	-	N/A	-	N/A	-	3/6

Table 6.6: Results of the small complicated instances. The unit of "time" is milliseconds. "s-opt" scores are solution costs divided by optimal ones; hence the minimum is one. Two rows show results of LaCAM*: *(i)* scores for initial solutions and *(ii)* solution quality at 1 s and the runtime when that solution was obtained; they are an average of 10 trials with different random seeds. Regarding sum-of-loss in *connector* (those decorated by parentheses), the scores normalized by known best values (80) are presented since the optimal score had not been obtained. Algorithms are categorized into LaCAM*, those optimizing makespan, sub-optimal ones, and BCP optimizing another metric (i.e., flowtime).

pattern detector; however, the author considers such implementations are too optimized for the benchmark. As shown in Fig. 6.15, the refinement was steady but not dramatic due to the same reason of Fig. 6.12.

Discussion of Other Algorithms

Although recent remarkable progress in MAPF studies, the scalability of optimal algorithms (e.g., ODrM*, BCP, CBS) is limited; they failed to handle a few hundred agents in most cases. Bounded sub-optimal algorithms such as I-ODrM* and EECBS can solve a variety of instances but struggle to solve challenging instances (e.g., with 1000 agents). Sub-optimal algorithms such as PP, LNS2, or PIBT$^{(+)}$ sometimes can handle such challenging instances but still failed many instances. From another perspective, it is ideal for an algorithm to be complete, however, the completeness can be the bottleneck for achieving speed. This is observed in A*, OD, and ODrM*.

In general, makespan-optimal solutions are easier to obtain than sum-of-loss-optimal ones. This is because, given an instance, the number of makespan-optimal solutions is larger than that of sum-of-loss. Empirically, this is validated with ODrM* and CBS in Fig. 6.13. For instance, CBS-m solved much more instances than CBS-l. Meanwhile, we can see a reverse trend in I-ODrM*, which is a bounded sub-optimal complete algorithm; I-ODrM*-l solved more instances than I-ODrM*-m. This is considered as an effect of admissible heuristics. The sum-of-loss optimization uses $\sum_{i \in A} \texttt{dist}(s_i, g_i)$ while the makespan optimization uses $\max_{i \in A} \texttt{dist}(s_i, g_i)$; the former provides more beneficial guidance than the latter.

6.7.7 Comparison with Anytime MAPF Solver

LaCAM* was compared with AFS [Cohen *et al.*, 2018a], a CBS-based anytime MAPF solver that guarantees to converge optima. Table 6.7 summarizes the results. Contrary to LaCAM*, AFS can obtain plausible solutions from the beginning, however, it compromises scalability. This quality gap may be overcome by developing better generators other than PIBT.

	solved(%)		time-init(ms)		loss-init		loss-30 s	
$\|A\|$	AFS	LaCAM*	AFS	LaCAM*	AFS	LaCAM*	AFS	LaCAM*
50	100	100	88	**1**	**25**	159	**24**	118
100	56	**100**	7223	**2**	**140**	609	**139**	545
150	0	**100**	N/A	**4**	N/A	1463	N/A	1368

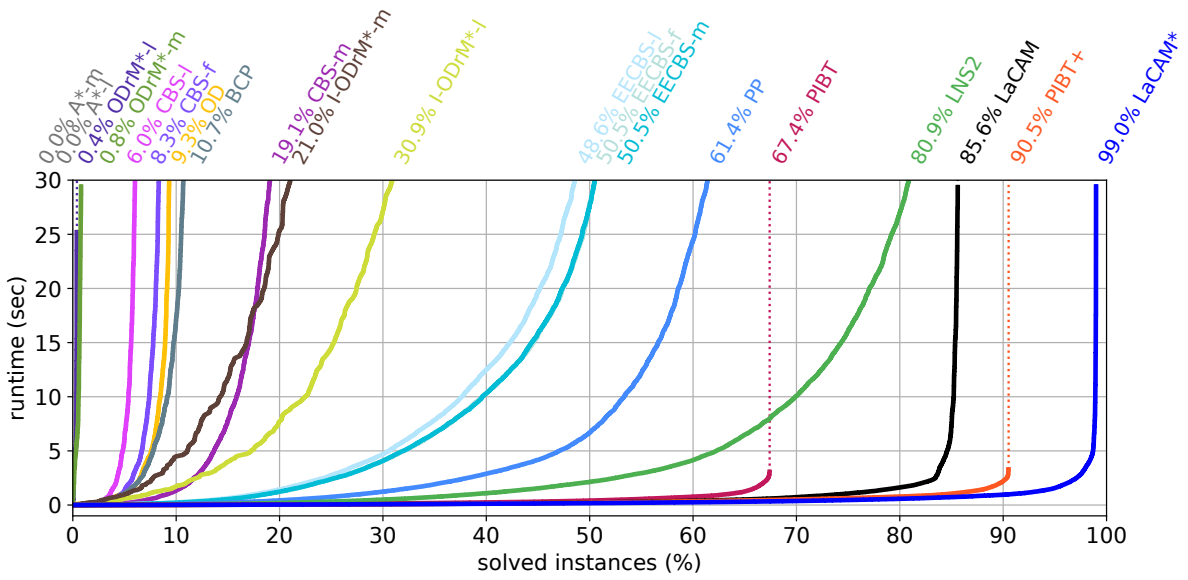
Table 6.7: Comparison of anytime MAPF algorithms. The experiment used sum-of-loss and 25 "random" scenarios of *random-32-32-20*. "init" shows scores related to initial solutions. "loss" is the gap scores from $\sum_{i \in A} \texttt{dist}(s_i, g_i)$. The scores are averaged for instances solved by both solvers, except for $|A| = 150$ because AFS failed all.

6.7.8 Extremely Dense Scenarios

Table 6.8 reports LaCAM* in very congested scenarios that existing solvers mostly fail. Even with such challenging cases, LaCAM* solved many instances, demonstrating its excellent scalability.

6.7.9 Discussion

Overall, LaCAM* resulted in remarkable results. That is, it empirically broke the tradeoff between good theoretical properties (i.e., completeness and optimality) and the small planning effort (i.e., quickness and scalability). This is achieved under the concept of integrating short- and long-horizon planning.



algorithm	reference	solvability	optimality	metrics
A*	[Hart *et al.*, 1968]	complete	optimal	makespan, sum-of-loss
OD	[Standley, 2010]	complete	sub-optimal	(using greedy search)
ODrM*	[Wagner and Choset, 2015]	complete	optimal	makespan, sum-of-loss
I-ODrM*	[Wagner and Choset, 2015]	complete	bounded sub-optimal	makespan, sum-of-loss
BCP	[Lam *et al.*, 2022]	solution complete	optimal	flowtime
CBS	[Sharon *et al.*, 2015]	solution complete	optimal	flowtime, makespan, sum-of-loss
EECBS	[Li *et al.*, 2021c]	solution complete	bounded sub-optimal	flowtime, makespan, sum-of-loss
PP	[Silver, 2005]	incomplete	sub-optimal	
LNS2	[Li *et al.*, 2022]	incomplete	sub-optimal	
PIBT(+)	[Okumura *et al.*, 2022b]	incomplete	sub-optimal	
LaCAM	[Okumura, 2023]	complete	sub-optimal	
LaCAM*		**complete**	**eventually optimal**	**makespan, sum-of-loss**

Figure 6.13: Performance on the MAPF benchmark. *upper*: The number of solved instances among 13,900 instances on 33 four-connected grid maps, retrieved from [Stern *et al.*, 2019]. The size of agents varies up to 1,000. '-f,' '-m,' and '-l' respectively mean that an algorithm tries to minimize flowtime, makespan, or sum-of-loss. The scores of LaCAM* are for initial solutions. *lower*: Representative or state-of-the-art MAPF algorithms. "Solution complete" means that an algorithm ensures returning solutions for solvable instances, however, it cannot identify unsolvable instances.

map	$\|A\|$	success(%)	time(s)	other algorithms
empty-8-8	58	100	0.00	PIBT&LaCAM (100%; 0.00 s)
random-32-32-20	737	100	0.63	LaCAM (4%; 14.81 s)
random-64-64-20	2943	68	11.64	N/A
maze-128-128-10	9772	100	55.54	N/A

Table 6.8: Results on extremely dense scenarios. $|A|$ was adjusted so that $|A|/|V| = 0.9$. For each scenario, 25 instances were prepared while randomly placing starts and goals. "success(%)" is the success percentage by LaCAM* with timeout of 60 s. "time" is the median runtime to obtain initial solutions. The other solvers in Fig. 6.13 were also tested. The table reports solvers that solved in at least one instance.

From another perspective, the implementation can be understood as combination of four types of MAPF approaches discussed in Chap. 3.2.4: graph pathfinding, two-level, prioritized planning, and rule-based approaches. LaCAM* itself is categorized into graph

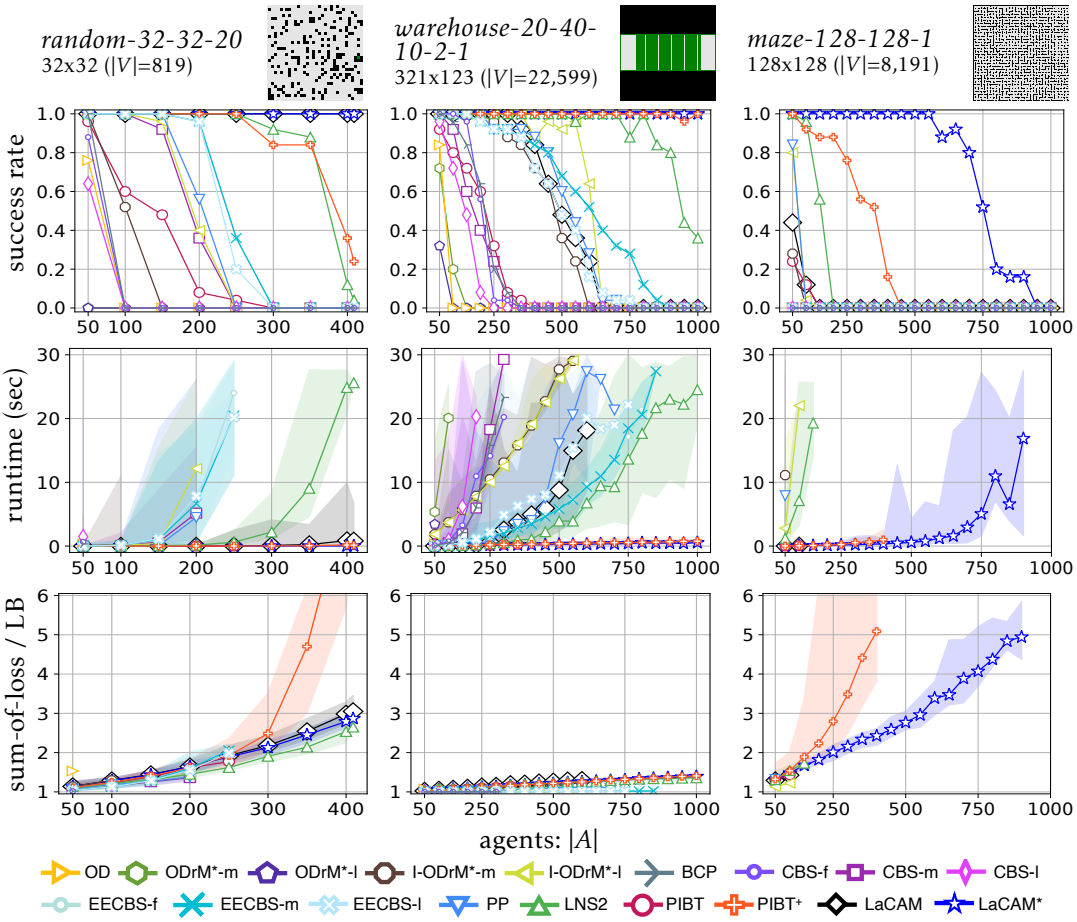


Figure 6.14: Results of the MAPF benchmark. Scores of sum-of-loss are normalized by $\sum_{i \in A} \texttt{dist}(s_i, g_i)$. For runtime and sum-of-loss, median, min, and max scores of solved instances within each solver are displayed. Scores of LaCAM* are from initial solutions. Full results appear in Appendix D.1.

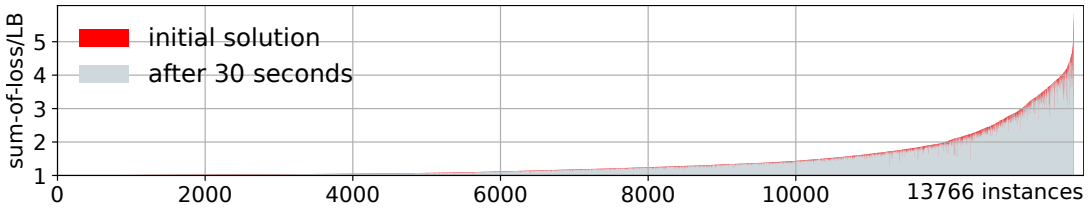


Figure 6.15: Refinement by LaCAM* for the MAPF benchmark. The effect is visualized by red zones.

pathfinding, however, it uses the two-level scheme to be the exaustive search. Meanwhile, the used configuration generator (extended PIBT; Alg. 6.3) relied on both characteristics of prioritized planning and rule-based approaches. This implies that, *for the sake of breaking the tradeoff, it was necessary to reflect perspectives of various MAPF algorithms*. These relationships are organized in the next section.

6.8 Related Work

LaCAM(*) borrows several ideas from the prior art of MAPF. This section summarizes the relationships between LaCAM and existing algorithms.

Graph Pathfinding. LaCAM is categorized into graph pathfinding approaches, such as [Standley, 2010; Goldenberg *et al.*, 2014; Wagner and Choset, 2015]. In those studies, successors of a search node are generated without *aggressive* coordination between agents beyond collision checking. In contrast, LaCAM considers aggressive coordination when generating successors at the high-level, which is incorporated by the use of other MAPF algorithms (i.e., short-horizon planning) as a configuration generator. Moreover, LaCAM dramatically reduces the number of successor generations compared to existing graph pathfinding-based approaches to MAPF, achieving both completeness and scalability.

Two-Level Approaches. Modern MAPF algorithms such as [Sharon *et al.*, 2013; Sharon *et al.*, 2015; Surynek, 2019; Lam *et al.*, 2022] take the form of two-level approaches. At the high-level, they usually search *negative* constraints that specify which agents cannot use where and when, while at the low-level, they perform single-agent pathfinding following constraints. LaCAM also relies on a two-level search, however, constraints are addressed at the low-level. In other words, the roles of high-level and low-level searches are reversed from the known two-level approaches to MAPF. Furthermore, the implementation uses *positive* constraints that specify who to be where. Positive constraints are not new in the literature; we can see an example in CBS [Li *et al.*, 2019b]. Note that LaCAM with negative constraints is possible to implement, however, the search space for the low-level can be dramatically larger than using positive ones.

Constraint Tree. The manner of adding constraints is partially influenced by A* with operator decomposition [Standley, 2010]. This algorithm creates successor nodes that correspond to one action of one agent, instead of actions for the entire agents. LaCAM adds constraints in a similar scheme.

Anytime MAPF. Anytime MAPF algorithms that converge to optimal solutions have been studied [Standley and Korf, 2011; Cohen *et al.*, 2018a; Vedder and Biswas, 2021]. However, their scalability is limited; they often fail to derive initial solutions, as we empirically saw. Techniques to refine arbitrary MAPF solutions have also been studied [Surynek, 2013; De Wilde *et al.*, 2014; Okumura *et al.*, 2021a; Li *et al.*, 2021a] but they do not ensure optimality.

Rewriting Edge Connection. LaCAM* is partially influenced by RRT* [Karaman and Frazzoli, 2011], which is an *asymptotically optimal* version of RRT [LaValle, 1998] for sampling-based motion planning (SBMP). Roughly, SBMP algorithms are called asymptotically optimal when, for solvable instances, the probability to find optimal solutions approach one as the number of samples (i.e., vertices of roadmaps) increases. The crux of RRT* is rewriting edge connection on demand. LaCAM* uses a similar trick to ensure eventual optimality.

PIBT Extension. As mentioned, incorporating swap into PIBT (Alg. 6.3) is directly inspired from rule-based algorithms [Luna and Bekris, 2011; De Wilde *et al.*, 2014]. This is accomplished by almost only manipulating the vertex scoring in PIBT. Since the implementation of `swap_required_and_possible` uses a perspective of graph topology analysis, the current implementation of LaCAM* also incoporates a flavors of rule-based approaches. It would be nice to remove this dependency by, for example, using machine learning to learn good vertex scoring in PIBT.

6.9 Concluding Remarks

This chapter introduced LaCAM$^{(*)}$, a novel search-based MAPF algorithm, designed following the concept that *short-horizon planning guides long-horizon planning*. LaCAM is complete. Not only that, it can be an eventually optimal anytime algorithm, provided that solution costs are accumulated transition costs. The exhaustive experiments reveal that LaCAM can solve various instances in a very short time, even with complicated, dense, and challenging scenarios that other state-of-the-art MAPF algorithms cannot handle. Furthermore, LaCAM is scalable; even with $10,000$ agents, it solved instances in tens of seconds. In summary, *LaCAM had developed a new horizon of MAPF algorithms*.

6.9.1 Interesting Directions

Some interesting directions for the development of LaCAM are listed below.

- **Developing more effective configuration generators** than PIBT variants which can output near-optimal initial solutions.
- **Improving the convergence speed** of LaCAM*.
- **Applying the LaCA search to other planning problems** because it is just a graph pathfinding algorithm. I am especially interested in applying the same concept to path planning in continuous spaces (aka. motion planning) because it is understood as a planning problem with an infinite branching factor.

Chapter 7

Improving Solution Quality by Iterative Refinement

Thus far, we have seen powerful sub-optimal solvers, which do not care about solution quality (excluding LaCAM*). In contrast, this chapter presents *how to iteratively refine MAPF solutions*. The iterative refinement framework originally appeared in [Okumura *et al.*, 2021a].[1]

7.1 Chapter Overview

The objective of the chapter is to see a method to refine an arbitrary solution to MAPF.

7.1.1 What is Iterative Refinement

This chapter presents a framework that, given an MAPF solution, returns a solution with better quality, which is generated based on the original solution. By repeatedly applying this procedure, it is possible to iteratively refine solutions. Despite its significance, such a framework had received little attention in the literature.

7.1.2 Why Iterative Refinement is Attractive

Applications of MAPF are inherently real-time systems with a limited time for planning as seen in the operations of automated warehouses [Wurman *et al.*, 2008]. Therefore, the primary challenge is to obtain feasible solutions before deadlines. Then, the aspect of solution quality is considered, that is, we want to obtain a better solution as much as possible.

The solution quality is important, however, recall that optimization of MAPF is a computationally demanding problem, as discussed in Chap. 3.2.3. Indeed, it is known to be an NP-hard problem for various optimization criteria [Yu and LaValle, 2013b]. Furthermore, it is still intractable when restricting fields in grid structures [Banfi *et al.*, 2017; Geft and Halperin, 2022], or approximating solution quality in near-optimal constant values [Ma *et al.*, 2016]. Even with state-of-the-art optimal algorithms [Li *et al.*, 2021b; Lam *et al.*, 2022], as seen in previous chapters, planning with hundreds of agents is still challenging. Such planners have no feasible solution when optimal solutions are not obtained. After all, there is no choice but to use sub-optimal algorithms.

On the other hand, we can create sub-optimal solutions in a very short time, as seen in Chap. 4 to 6. With feasible solutions obtained quickly, we can use the remaining

[1] Most figures and tables presented here were re-created for this dissertation but some of them were reused from the seminal paper. See Copyrights.

time until the deadlines for *iterative refinement*. This is the basic motivation of *anytime algorithms* [Zilberstein, 1996], which can yield a feasible solution whenever interrupted, the quality of which improving as time passes, well-match with practical demands.

Iterative refinement is also fruitful for online situations where goals are allocated dynamically [Ma *et al.*, 2017b; Švancara *et al.*, 2019]. The challenging part here is replanning. Two intuitive approaches exist: *(i)* replan all paths, or *(ii)* replan a single path for one agent with a new goal while keeping the others unchanged. The first approach may return efficient solutions but it is costly and typically inappropriate for online use. The second approach may return inefficient solutions but is nearly costless. We can apply the second approach for an initial solution that we can gradually refine within the time constraints.

7.1.3 How to Refine Solutions

Iterative refinement is promising though under-explored in MAPF because, so far, it was unclear how to incrementally improve a known solution. In the context of local search, this corresponds to finding a good neighborhood solution.

This chapter provides one answer based on an effective combination of existing solvers. The proposed framework first uses a sub-optimal MAPF solver to quickly obtain an initial feasible solution, then, it uses an optimal MAPF solver to find good neighborhood solutions. Precisely, the framework refines the solution iteratively by selecting a subset of agents and using an optimal solver to refine their paths while keeping other paths fixed. Although the refinement process uses an optimal solver, each refinement are completed quickly because it solves a sub-problem whose size depends on the number of selected agents, typically much smaller than the original. This chapter also presents reasonable candidates on how to select a subset of agents.

The effectiveness of the approach is evaluated in various benchmarks. The main observation is that the framework converges almost optimally within a short time in small instances, and remains responsive even for very large instances (i.e., large environments and/or many agents). In other words, it brings many practical advantages based on prior art.

7.1.4 Chapter Organization

- Chapter 7.2 reviews related work.
- Chapter 7.3 describes the framework including basic theoretical analysis.
- Chapter 7.4 presents construction rules of a subset of agents.
- Chapter 7.5 evaluates the proposal from a variety of aspects.
- Chapter 7.6 concludes the chapter.

The code and movie are available at `https://kei18.github.io/mapf-IR`.

7.1.5 Notations and Assumptions

$\bot$	undefined, not found
$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \dots, n\}$	a set of agents
$s_i \in V, g_i \in V$	start and goal for agent-i

As a solution cost of MAPF, this chapter uses sum-of-costs (aka. flowtime), however, the iterative refinement framework can easily adapt to other metrics.

Caution

Unlike previous chapters, this chapter is based on the representation by paths.

7.2 Related Work

Some anytime MAPF algorithms have been developed so far. For instance, in [Standley and Korf, 2011], an anytime version of the optimal MAPF algorithm called A* with operator decomposition [Standley, 2010] has been developed. In [Cohen *et al.*, 2018a], an anytime algorithm based on a variant of A* was applied to the high-level search of CBS. X* [Vedder and Biswas, 2021] is an anytime MAPF solver assuming sparse scenarios, i.e., agent distributed sparsely in fields and where the potential for collisions is rare. The BCP algorithm [Lam *et al.*, 2022] also has an anytime property due to the nature of branch-cut-price optimization. Moreover, LaCAM* in Chap. 6.5 is an anytime version of LaCAM [Okumura, 2023]. These methods search for non-optimal solutions by relaxing some constraints, then eventually converge to optimal solutions by iteratively tightening the constraints. A drawback is that they are each tied to a specific solver, and that they may fail to obtain initial solutions in a reasonable time thus returning nothing.

Several ad-hoc rules to repair MAPF solutions have also been seen in the literature [Surynek, 2013; De Wilde *et al.*, 2014]. However, since improvements are done by ad-hoc local changes, redundancies of *a priori* unknown patterns remain in the solution.

In a wider view, the study can also be seen as solving a very large-scale neighborhood search [Ahuja *et al.*, 2002]. Closer to our concept, in [Balyo *et al.*, 2012], local replanning for domain-independent planning problems was studied to optimize makespan. It repeats the following; create a sub-problem, obtain an optimal sub-solution by SAT-based techniques, and replace the part of the original solution with a new one.

Finally, [Li *et al.*, 2021a] was concurrent to this study, which has the same concept that uses MAPF algorithms to refine solutions. This chapter further provides limitations of this concept, practical techniques such as early stop, as well as various strategies to choose a subset of agents.

7.3 Framework

Algorithm 7.1 presents the proposed framework. In short, it first obtains an initial solution by a sub-optimal MAPF solver, and then iteratively refines selected parts of the solutions, the paths of a selected subset of the agents, using an optimal MAPF solver.

Algorithm 7.1 Framework of iterative refinement.

input: MAPF instance

output: solution $\boldsymbol{\Pi}$ or `FAILURE`

1: $\boldsymbol{\Pi} \leftarrow$ initial solution obtained by an MAPF solver
2: **if** $\boldsymbol{\Pi} = \bot$ **then return** `FAILURE`
3: **while** $\neg$`interrupt()` **do**
4: create a modification set $M \subseteq A$ using $\boldsymbol{\Pi}$
5: $\boldsymbol{\Pi} \leftarrow$ refined MAPF solution for M while fixing the others' paths in $\boldsymbol{\Pi}$
6: **return** $\boldsymbol{\Pi}$

7.3.1 Framework Details

At first, an initial feasible solution is quickly obtained by a sub-optimal MAPF solver (Line 1), referred to as an *initial solver*. If the initial solver failed to obtain solutions, the framework ends with FAILURE (Line 2); otherwise, the refinement starts (Lines 3–5). The refinement iteratively conducts two procedures:

1. Create a modification set $M \subseteq A$ using the current solution $\mathbf{\Pi}$ (Line 4).
2. Refine the current solution $\mathbf{\Pi}$ by changing paths for agents in M (Line 5), using an optimal MAPF solver.

The solver that does refinement is called a *refinement solver*. The refinement solver only changes the paths for agents in M; paths for agents not in M are unchanged. The refinement continues until interrupted, e.g., timeout, reaching the predetermined iteration number, when no improvement is expected, interruption by users, etc. After the interruption, the framework eventually returns the final solution (Line 6).

Used MAPF Solvers. The initial solver can be any sub-optimal MAPF solver, as long as it provides feasible solutions. As the refinement solver, it is desirable to use versions adapted from an optimal solver. The adaptation is simple; let it plan paths for agents in M regarding the others as dynamic obstacles. For instance, with CBS, it solves MAPF only for agents in M while prohibiting the low-level search to use all space-time pairs used by agents outside of M. In a precise sense, the refinement solver is not limited to optimal MAPF solvers. The requirement is that the refined solution never worsens from the original. Considering that cost of paths for agents outside of M does not change, the requirement is that cost of paths for agents in M is non-increasing before and after refinement.

The next property is obvious from the requirements set on the refinement solver.

Corollary 7.1 (Monotonicity)**.** *For each iteration in Alg. 7.1, the solution cost is non-increasing.*

A key point is that the refinement solver recomputes the paths for a selected subset M of agents, rather than for the entire set A of all agents. Compared to solving the original problem directly using optimal solvers, the problem solved at each iteration by the refinement solver is significantly smaller, ensuring that the framework is scalable even to a large number of agents.

7.3.2 Early Stop

Even though sub-problems solved by the refinement solver are small compared to the original problem, the refinement may still take too long if $|M|$ is too big. In such cases, it is preferable to abort the current refinement by returning the current solution, and then start a new iteration with a new set M. The criteria can be a timeout or using a threshold value for the size of a search tree in the refinement solver.

7.3.3 Limitations

As a limitation, the framework may have the local minimum with no sub-optimality bounds from the optimal.

Proposition 7.2 (no sub-optimality bounds)**.** *Consider the optimal cost c^*. With Alg. 7.1, there is no $w \geq 1$ such that always $c \leq wc^*$ unless selecting A itself as a modification set M, where c is the solution cost in each iteration.*

Proof. Consider an example in Fig. 7.1. Assume that an initial solution assigns agent i to a clockwise path (cost: k) and agent j to a counterclockwise path (cost: 1). With $k \geq 6$, this is not optimal because i can take a counterclockwise path if j temporally moves over from its goal (sum-of-costs: 6). Unless $M \neq A$, the solution of the refinement is unchanged. Assume $w \geq 1$ such that $c \leq wc^*$. We can take an arbitrary k, contradicting the existence of w. □

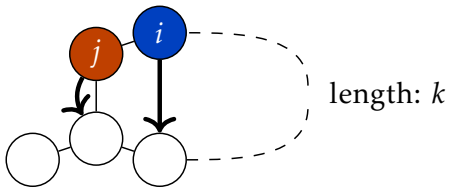


Figure 7.1: Example of local minimum. Goals are depicted by arrows.

Corollary 7.3 (Existence of local minimum)**.** *Depending on initial solutions, it may be impossible to reach the optimal solution unless selecting A itself as M.*

Note that when $M = A$, the refinement solver has to solve the original MAPF problem.

7.4 Design of Modification Set

The modification set is an important component of the framework, and its design will affect the performance such as computation time and solution quality. This section defines several *selection rules* to provide reasonable candidates.

7.4.1 Random

One naive approach is to pick a subset of agents randomly. The size of a modification set M is then a user-specified parameter. Not to mention, there is a tradeoff in the size of M; large $|M|$ has a chance to reduce costs largely in one iteration but takes time for refinement because sub-problems become challenging, and vice versa.

7.4.2 Single Agent

This rule always picks a single agent as M. This can be regarded as a special case of the previous rule (*random*). Even with a single agent, the cost might be reduced by the refinement. In this case, the refinement becomes just a single-agent pathfinding problem and can be computed efficiently without MAPF solvers, e.g., by A*.

7.4.3 Focusing at Goals

Consider an example in Fig. 7.2. Assume that the current solution is $\Pi_1 = (v_2, v_3, \underline{v_6}, v_3, v_3)$ and $\Pi_2 = (v_1, v_2, \underline{v_3}, v_4, v_5)$. An agent-1 cannot achieve a shorter path because agent-2 uses a goal of agent-1 (i.e., v_3) at a timestep 2. In general, for agent i, one reason for a gap between ideal cost $\mathtt{dist}(s_i, g_i)$ and cost contribution of agent i in a solution $\mathbf{\Pi}$, denoted as $\mathtt{cost}(i, \mathbf{\Pi})$, might be that another agent j uses a goal for i (i.e., g_i) at a timestep $t \geq \mathtt{dist}(s_i, g_i)$. At least before t, i cannot arrive at g_i and remain there. In this case, it is required to jointly refine paths of i and j.

This observation motivates us to create a following simple rule, taking a current solution $\mathbf{\Pi}$ and one agent i as input.

$$M \leftarrow \left\{ j \in A \mid \Pi_j[t] = g_i, \mathtt{dist}(s_i, g_i) \leq t \leq \mathtt{cost}(i, \mathbf{\Pi}) \right\} \tag{7.1}$$

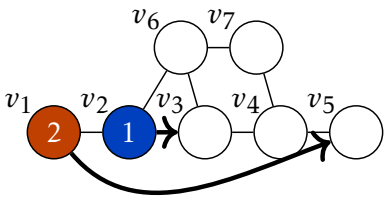


Figure 7.2: Example of local repair around goals.

The selecting rule of agent-i is arbitrary.

7.4.4 Local Repair around Goals

This is a special case of the previous rule (*focusing-at-goals*). Assume again the example in Fig. 7.2; $\Pi_1 = (v_2, v_3, \underline{v_6}, v_3, v_3)$ and $\Pi_2 = (v_1, v_2, v_3, v_4, v_5)$. In *focusing-at-goals*, M for agent-1 is $\{1, 2\}$, therefore, the refinement solver has to solve a sub-problem with two agents; however, this effort can be reduced. Consider obtaining a better path for agent-1 ignoring Π_2. In this example, a new path is obtained without searching by *local repair around the goal*; $(v_2, v_3, \underline{v_3}, v_3, v_3)$. Next, compute a single path for agent-2 while avoiding collisions with this new path and the other agents' paths. If the sum-of-costs of two new paths is smaller than the original, replace Π_1 and Π_2 with the new paths. Since the search effort is reduced, the refinement is expected to finish faster.

In general, when $\Pi_i = (\ldots, g_i, v, g_i, \ldots, g_i)$ where $v \neq g_i$ and another agent j uses g_i at that timestep, this rule can be applied.

7.4.5 Using MDD

Given a single path cost c, a set of paths from s_i to g_i can be compactly represented as a *multi-valued decision diagram (MDD)* [Srinivasan *et al.*, 1990]. MDD is a directed acyclic graph where a vertex is a pair of a location $v \in V$ and a timestep $t \in \mathbb{N}$. Each vertex in an MDD satisfies two conditions:

- a reachable location at that timestep from a start, and
- a reachable location to a goal from that timestep.

Let MDD_i^c be an MDD for agent i with a cost c. Figure 7.3 shows two examples: MDD_1^2 and MDD_1^3. MDDs are used often in MAPF solvers, e.g., [Sharon *et al.*, 2013; Boyarski *et al.*, 2015].

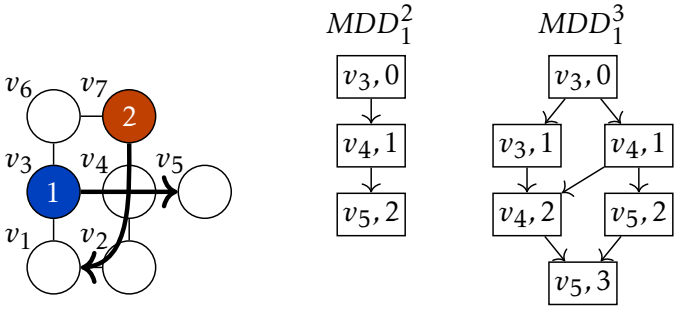


Figure 7.3: Examples of MDD.

Using MDD_i^c where $\texttt{dist}(s_i, g_i) \leq c < \texttt{cost}(i, \boldsymbol{\Pi})$, a set of agents interfering with Π_i can be detected. See an example in Fig. 7.3. Assume that the current solution is $\Pi_1 =$

(v_3, v_3, v_4, v_5) and $\Pi_2 = (v_7, v_4, v_2, v_1)$. Consider to update MDD_1^2 by Π_2; remove vertices of MDD_1^2 that collides of Π_2, i.e., $(v_4, 1)$. Then, remove all redundant vertices that do not satisfy the two conditions due to the first operation: $(v_3, 0)$ and $(v_5, 2)$. Now it turns out to be impossible that agent-1 reaches its goal with a cost of two because there is no remaining vertex. In other words, Π_2 is preventing that agent-1 achieves a smaller cost; hence Π_1 and Π_2 should be jointly refined. The general procedure is described in Alg. 7.2.

Algorithm 7.2 ***using-MDD*.**

input: solution $\mathbf{\Pi}$, agent $i \in A$

output: modification set $M \subseteq A$

1: $M \leftarrow \{i\}$
2: **for** $\mathtt{dist}(s_i, g_i) \leq c < \mathtt{cost}(i, \mathbf{\Pi})$ **do**
3: create MDD_i^c
4: **for** $j \in A \setminus \{i\}$ **do**
5: update MDD_i^c by Π_j
6: **if** MDD_i^c is updated by Π_j **then** $M \leftarrow M \cup \{ j \}$
7: **return** M

7.4.6 Using Bottleneck Agent

Consider the example of Fig. 7.3 again; $\Pi_1 = (v_3, v_3, v_4, v_5)$ and $\Pi_2 = (v_7, v_4, v_2, v_1)$. If removing Π_2, agent-1 can take a shorter path, meaning that, agent-2 is a *bottleneck* for agent-1. There is a chance to reduce cost by refining jointly with such a bottleneck agent and agents that can take shorter paths without the agent. This concept is described in Alg. 7.3.

Algorithm 7.3 ***using-bottleneck-agent*.**

input: solution $\mathbf{\Pi}$, agent $i \in A$

output: modification set $M \subseteq A$

1: $M \leftarrow \{i\}$
2: **for** $j \in A \setminus \{i\}$ **do**
3: $c \leftarrow$ cost of the best path for j while avoiding collisions with $\mathbf{\Pi} \setminus \{\Pi_i, \Pi_j\}$
4: **if** $c < \mathtt{cost}(j, \mathbf{\Pi})$ **then** $M \leftarrow M \cup \{ j \}$
5: **return** M

7.4.7 Composition

Each rule might have suitable situations, e.g., the rule *focusing-at-goals* (Chap. 7.4.3) is costless to create modification sets, but it might be weak to detect effective sets when solutions are already efficient to some extent. On the other hand, the rule *using-MDD* (Chap. 7.4.5) takes time but they are highly expected to detect effective sets. Therefore, one promising direction is to *composite* these rules, namely, execute the first rule until no improvement is expected, and then switch to the second rule; same as above.

7.5 Evaluation

To assess the performance of the iterative refinement framework, a series of experiments were conducted, comprising the following six parts:

- Chapter 7.5.2 compares the selection rules of agents with *inefficient* initial solutions.
- Chapter 7.5.3 compares the rules with *efficient* initial solutions.
- Chapter 7.5.4 evaluates dependencies on *different initial solvers*.
- Chapter 7.5.5 assesses solution costs obtained by the framework compared to the optimal ones.
- Chapter 7.5.6 compares to another anytime MAPF solver.
- Chapter 7.5.7 tests the framework in challenging scenarios, i.e., huge maps with many agents.

7.5.1 Experimental Setup

Instance Generation. Several four-connected grids were carefully selected from [Sturtevant, 2012; Stern *et al.*, 2019] as a graph G, shown in Fig. 7.4. In all settings, identical instances between solvers were tested. All instances were created by choosing randomly initial locations and destinations.

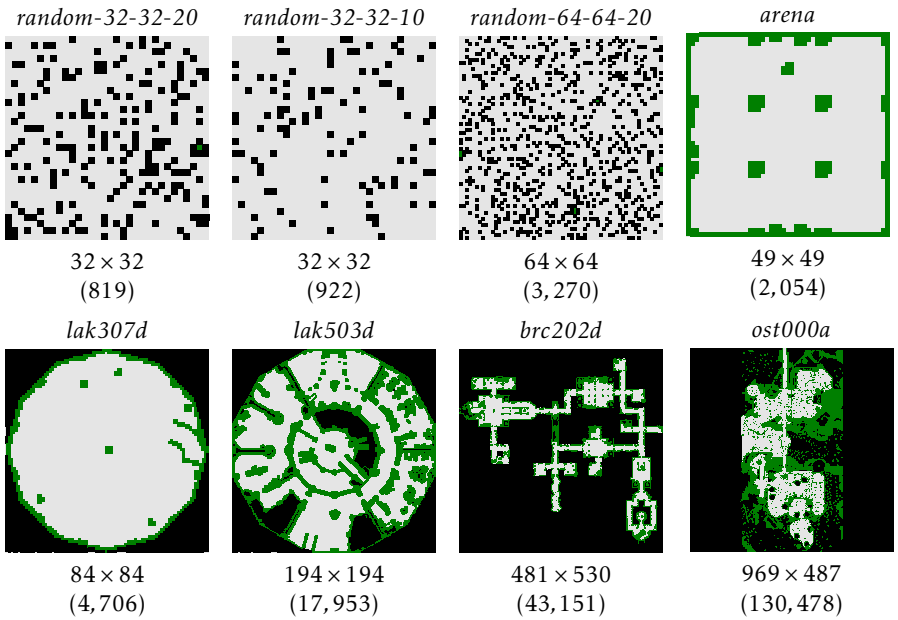


Figure 7.4: Used maps. $|V|$ is shown with parentheses.

Metrics. The objective here is to minimize the sum-of-costs metric. The below often presents sum-of-costs normalized by $\sum_{i \in A} \mathtt{dist}(s_i, g_i)$ as the solution quality; smaller is better and the minimum is one. Even though optimal costs are hard to obtain, these values work as upper bounds of sub-optimality.

Experimental Environment. All experiments were run on a laptop with Intel Core i9 2.3 GHz CPU and 16 GB RAM. The framework was coded in C++.

Implementation of Iterative Refinement

Refinement Solver. As a refinement solver, an adapted version of ICBS [Boyarski *et al.*, 2015] was used for the following reasons. First, CBS [Sharon *et al.*, 2015] is a promising and actively-studied optimal solver; it is however sensitive to tie-break of choosing high-level nodes, resulting in pure CBS being poorly scalable. ICBS, an extension of CBS, improves this aspect. Though not state-of-the-art, ICBS is stable and has been used in many studies, e.g., [Felner *et al.*, 2018; Li *et al.*, 2019a; Stern *et al.*, 2019]. Therefore, ICBS was considered sufficient as a baseline for our experiments. Note that results heavily depend on the refinement solver, meaning that, the refine speed might become much faster with a faster refinement solver.

Early Stop. In each setting, the early stop by the timeout was introduced (see Chap. 7.3.2). They were adjusted to appropriate values before experiments.

Agent Selection. In the rules *single-agent*, *focusing-at-goal*, *using-MDD*, *using-bottleneck-agent*, one agent needs to be selected. In the implementation, that agent is selected sequentially from A.

Composition Rule. In the refinement rule *composition* (Chap. 7.4.7), the following rules were sequentially applied:

1. *local-repair-around-goals* (Chap. 7.4.4)
2. *focusing-at-goals* (Chap. 7.4.3)
3. *using-MDD* (Chap. 7.4.5)
4. *random* (Chap. 7.4.1) with 30 agents.

These rules were chosen according to preliminary results. The switching was when no improvement is achieved for all agents.

7.5.2 with Inefficient Initial Solutions

Setup. The first experiment aims at assessing how each rule refines inefficient initial solutions. The initial solver was PIBT$^+$ (Chap. 4.3.1). The refinements were stopped after 90 s in the small maps (*random-32-32-20* and *arena*) and 10 min in *lak503d*. The numbers of agents were fixed at 110, 300, and 500, respectively. This duration includes the time required for the initial solver. The refinement timeout was 1 s for *lak503d*, otherwise 500 ms.

Result. Figure 7.5 shows the average progress of the refinement over 25 instances. The rules *single-agent* and *local-repair-around-goals* reduce costs immediately but soon reach their limits, i.e., no improvement even with room for refinement. The rule *focusing-at-goals* dramatically improves solution quality in each case while the rule *use-bottleneck-agent* does not work well as expected. Note that PIBT$^+$ returned solutions within 500 ms even for the worst case (in *lak503d* with 500 agents; see also Table 7.1).

7.5.3 with Efficient Initial Solutions

Setup. Next, the refinement was applied to already efficient solutions to some extent, obtained by ECBS [Barer *et al.*, 2014] or RPP [Čáp *et al.*, 2015]. The former is a bounded sub-optimal version of CBS while the latter is a variant of prioritized planning. Both algorithms were own-coded in C++. The used instances were the same as the previous ones.

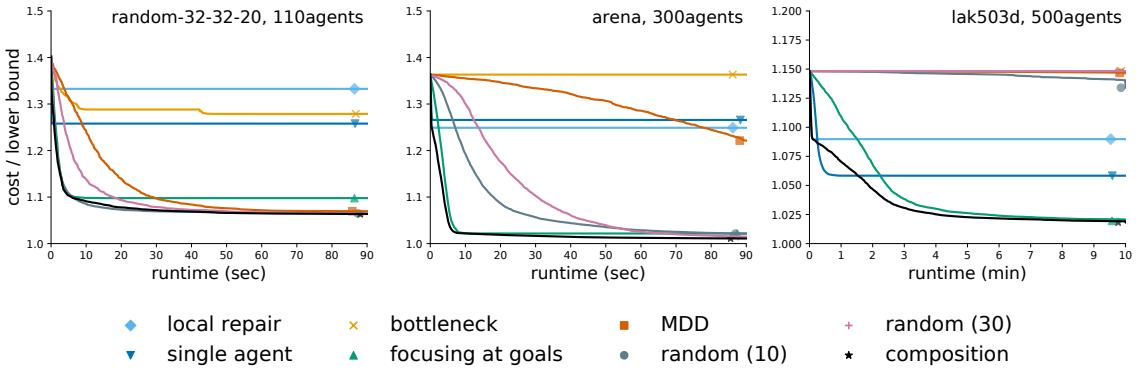


Figure 7.5: Average progress of the refinement with *inefficient* initial solutions. The initial solver was PIBT$^+$.

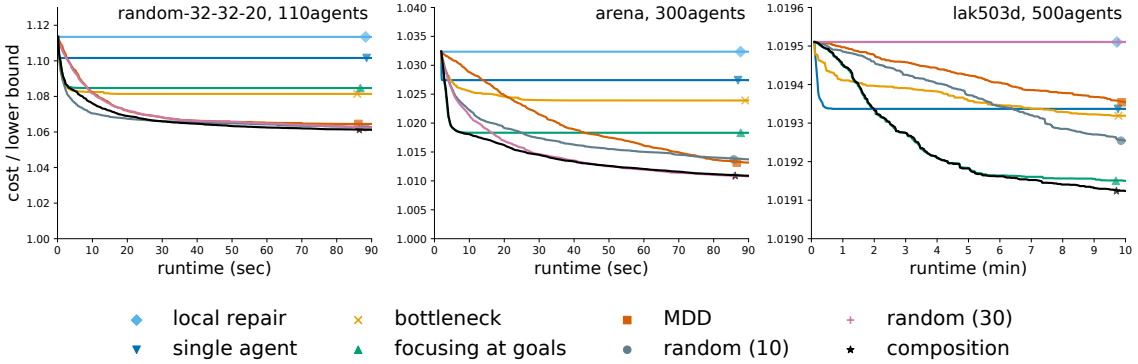


Figure 7.6: Average progress of the refinement with *efficient* initial solutions. In *random-32-32-20* and *arena*, we used ECBS to obtain the initial solutions. The suboptimality was 1.2 and 1.1 respectively, which were adjusted to balance runtime and solution quality. For *lak503d*, we prepared well-formed instances and used RPP. Note that it is difficult to get such instances with other settings because they are too dense. In *lak503d*, we do not start the y-axis from one to see differences between rules; the improvements were tiny.

Result. Figure 7.6 shows the results, revealing a limitation of the rule *focusing-at-goals* in *arena* and *random-32-32-20*; it is difficult to refine efficient enough solutions by this rule. Rather, the rules *using-MDD* and *random* achieved smaller final costs. In *lak503d*, initial solutions with little room for refinement were often obtained, therefore, the effect of refinement was subtle (see y-axis). Even so, several rules still improved the solution quality.

Throughout two experiments so far (Chap. 7.5.2 and 7.5.3), the rule *composition* successfully reduced costs with reasonable speeds; therefore, hereinafter, the experiments used this rule.

7.5.4 with Different Initial Solvers

Setup. The third experiment evaluated dependencies on initial solvers. Five initial solvers were used: PIBT$^+$, HCA* [Silver, 2005], WHCA* [Silver, 2005], ECBS, and RPP. HCA* and WHCA* are variants of prioritized planning, own-coded by C++. The refinement timeout was set 1 s for *lak503d*, otherwise 500 ms.

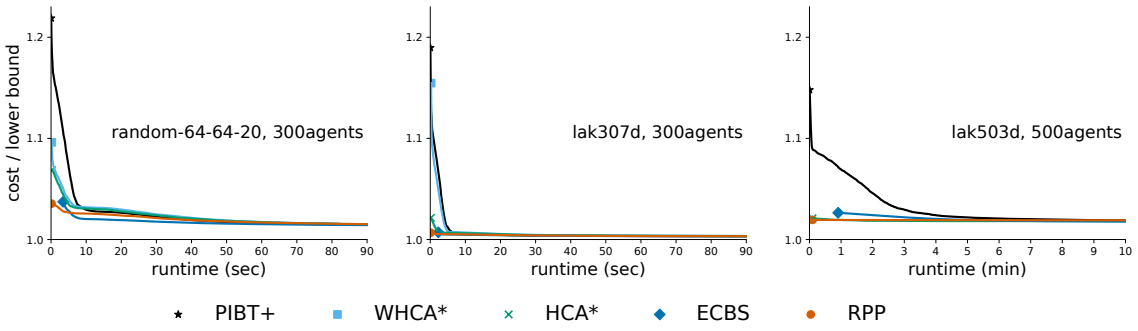


Figure 7.7: Average progress of the refinement with *different initial solvers*. All instances were well-formed. The averages are for success instances in all initial solvers. In *lak503d*, the scores were calculated removing those of WHCA* because WHCA* failed most cases. From the left, the suboptimality of ECBS was 1.05, 1.05, and 1.1. The window size of WHCA* was 30, 10, 30. Those parameters were adjusted to balance success rate, cost, and runtime.

		PIBT$^+$	HCA*	WHCA*	ECBS	RPP
	cost (init)	1.219	1.069	1.096	1.037	1.035
random-64-64-20	cost (last)	1.015	1.015	1.015	1.014	1.016
300 agents	#success	25	23	15	25	25
	runtime (ms)	43	201	228	3436	153
	cost (init)	1.190	1.021	1.155	1.007	1.007
lak307d	cost (last)	1.003	1.003	1.003	1.003	1.003
300 agents	#success	25	25	23	25	25
	runtime (ms)	43	171	247	2306	119
	cost (init)	1.148	1.021	-	1.026	1.019
lak503d	cost (last)	1.019	1.018	-	1.018	1.019
500 agents	#success	25	25	1	24	25
	runtime (ms)	376	5716	-	54433	5640

Table 7.1: Detailed results with different initial solvers. "cost" is the sum-of-costs divided by the lower bound. Both initial and last scores are shown. "#success" is the number of successful instances in 25 instances. Some solvers failed in some instances because they returned failure due to incompleteness, or, failed to obtain solutions before the deadlines (90 s in *random-64-64* and *lak307d*; 10 min in *lak503d*). "runtime" is when initial solvers return solutions. All scores were averaged over success instances in all initial solvers except for *lak503d* where WHCA* failed most cases; for *lak503d*, the average scores without WHCA* are shown. ©2021 IEEE.

Result. Figure 7.7 shows the average progress. Table 7.1 summarizes the details. The main observation is that, although the initial costs were widely different between the solvers, the final costs ended up not so. This implies that any initial solvers can be used if you have enough time for refinement.

In the following, PIBT$^+$ was used because it instantly returns a feasible solution and meets well with anytime property.

7.5.5 vs. Optimal Solutions

Setup. According to Prop. 7.2, the approximation ratio of refined solutions is unbounded from the optimal. In practice, however, the estimation from empirical data is useful hence this was evaluated. Two small settings were used to test this aspect (30 agents in *random-32-32-20*; 50 agents in *random-32-32-10*) because optimal solvers often fail to obtain solutions within a reasonable time in large maps or with many agents. Optimal solutions were obtained by CBSH [Li *et al.*, 2019a], an extension of CBS. Its implementation was obtained from the authors of that paper. The refinements continued for 1 s including the time for the initial solver.

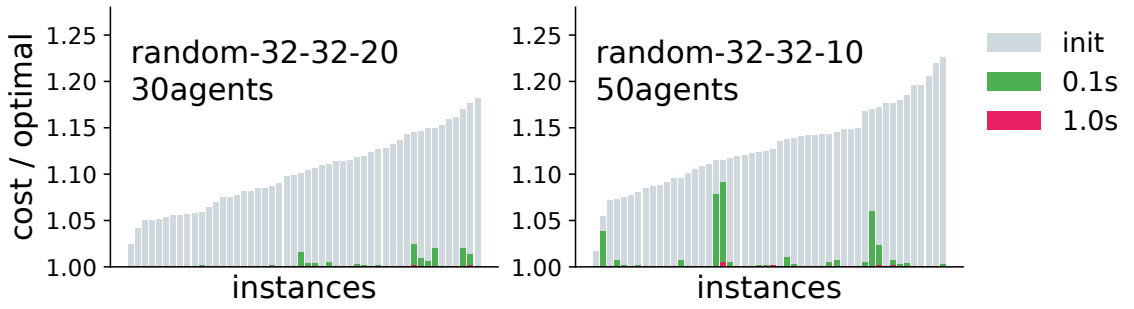


Figure 7.8: Results vs. *optimal* solutions. Three types of the suboptimality of 50 instances are shown: initial scores, 0.1 s later, and at 1 s. The scores at 1 s are hard to recognize because most of them reach the optimal. ©2021 IEEE.

Result. Figure 7.8 summarizes the results of 50 instances, showing the sum-of-costs divided by the optimal costs. The average runtime of CBSH was 710 ms with 30 agents and 1743 ms with 50 agents while the refinement (PIBT$^+$) got initial solutions less than 3 ms in all instances. Despite large gaps between the initial and optimal costs, the refinement dramatically reduced the gaps within a short time. Furthermore, most solutions reached the optimal within 1 s.

7.5.6 vs. Other Anytime MAPF Solver

Setup. Next, the proposed framework was compared with another anytime MAPF solver, AFS [Cohen *et al.*, 2018a], using *random-32-32-20* while changing the number of agents. AFS applies anytime planning to the high-level search of CBS. Its implementation was retrieved from the authors of the paper. Note that AFS theoretically converges the optimal someday but the proposal may not. The refinement timeout was 100 ms. The runtime limit was 30 s.

Result. Figure 7.9 shows the results of 25 instances. AFS failed to obtain solutions within the time for 2 instances with 90 agents. Clearly, the proposal has an advantage; it obtained initial solutions immediately while AFS did not, and the convergence was fast enough with better costs.

7.5.7 Challenging Scenarios

Setup. Finally, the refinement with many agents in huge grids was tested, namely, 1,500 agents in *brc202d* and 3,000 agents in *ost000a*. The refinement timeout was 3 s for *brc202d* and 10 s for *ost000a*. In such scenarios, it is unrealistic to obtain the optimal solutions; hence, iterative refinement is attractive to obtain good enough solutions.

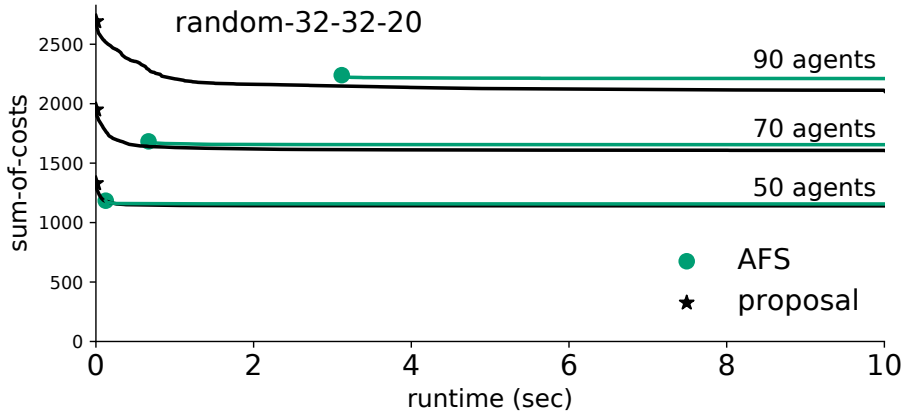


Figure 7.9: Results vs. another *anytime* MAPF solver. We omit scores after 10 s because they are almost flat. The improvements of AFS were subtle and hard to recognize. ©2021 IEEE.

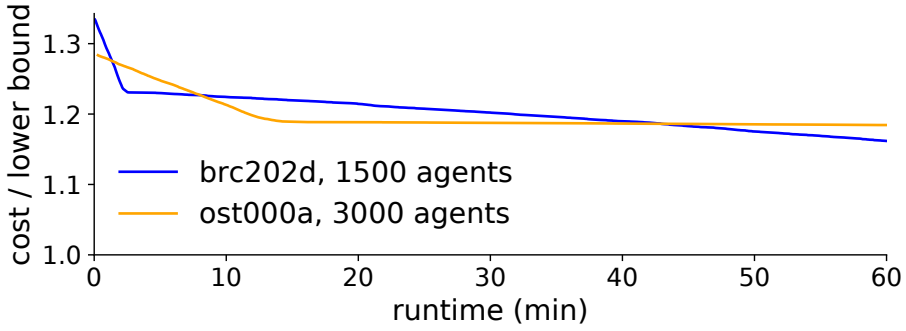


Figure 7.10: Results of challenging scenarios. ©2021 IEEE.

Result. Figure 7.10 shows the progress of 10 instances with one-hour refinement. The initial solutions were obtained 4 s for *brc202d* and 17 s for *ost000a* on average.[2] The refinement gradually reduced costs; however, the speed of the refinement was not so fast.

7.6 Concluding Remarks

This chapter presented the iterative refinement framework for MAPF. The framework uses two MAPF solvers as sub-procedures: a sub-optimal solver to obtain an initial solution and an optimal solver to refine the solution. Although the framework does not guarantee finding the optimal solution, the empirical results demonstrate its usefulness; the framework finds a solution with acceptable costs in a small computation time with high scalability. Furthermore, it is anytime planning, which is a desired property for real-time systems with severe deadlines. MAPF studies are very active and the framework can be better with their developments, as we have seen through Part I. With their effective use, it is expected that the framework becomes better than empirically presented here.

Practical Anytime MAPF. According to the experiments, the costs can be reduced near-optimal regardless of the initial solutions; however, it is better to start with efficient

[2]Preliminary, other solvers were tried including CBSH and AFS but most of them could not solve any instances. As a result, it is unlikely to obtain efficient enough solutions from the beginning in such huge scenarios.

enough solutions if available, because we can get better solutions at an early stage. Thus, a practical anytime MAPF scheme will be the following. First, in parallel, start several initial solvers with different portfolios between runtime and solution quality (e.g., PIBT$^+$, LaCAM, RPP, EECBS). Then, apply refinement to the first solution you get. If another initial solver gets a better solution compared to the refined solution at that time, replace the current one with the new one. This scheme complements a time lag of an efficient initial solver with a fast inefficient initial solver.

Part II

Execution

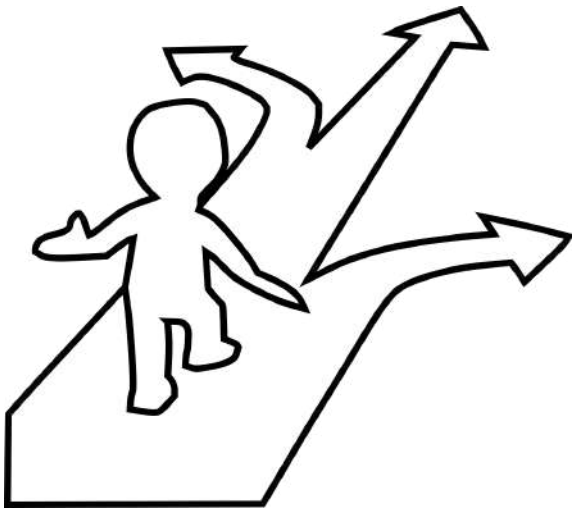


Image by Clker-Free-Vector-Images / Pixabay License

Chapter 8

Online Planning to Overcome Timing Uncertainties in Execution

From this chapter, the dissertation starts considering the execution perspective, aiming at overcoming uncertainties entailed in multi-agent navigation.

To begin with, this chapter shortly revisits the TSWAP algorithm for unlabeled-MAPF (Chap. 5) and presents its online version. The resulting algorithm, called *online TSWAP*, solves the execution of unlabeled-MAPF allowing *timing uncertainties*. The algorithm is based on the notion of *(online) time independence*, playing a crucial role in the next two chapters.

Time independence initially appeared in [Okumura *et al.*, 2021b], which studied online time-independent planning for MAPF by adapting PIBT. Later in [Okumura and Défago, 2022b], the notion became sophisticated; this chapter is based on the second paper.[1]

8.1 Chapter Overview

The objective of the chapter is to understand the notion of time-independent planning for multiple moving agents, developed to overcome timing uncertainties during execution. Specifically, this chapter presents *online* time-independent planning for *unlabeled-MAPF*, embodied as online TSWAP. Time-independent planning poses a significantly different planning style from previous chapters.

8.1.1 What is Online Time-Independent Planning

Since timing uncertainties (e.g., delays of robot motions) are inevitable in execution, many studies have studied the uncertainties (see 3.5.2). Among them, the limitation of robust offline planning is the necessity of pre-defined timing uncertainty models. Such models are vulnerable in practice because the system behavior must fit into the model. Otherwise, the behavior is totally uncontrollable and unpredictable. If so, it might be better not to put such uncertainty models from the beginning.

To this end, this chapter abandons all timing assumptions (e.g., synchronization, traveling time, rotation time, delay probabilities) and *regards the whole system as a transition system that changes its configuration according to atomic actions of agents*. The primary challenges are the following two: *(i)* how to formulate the notion of time independence, and, *(ii)* how to design an algorithm to ensure liveness (i.e., all agents reach their goals),

[1] Actually, the concept got clear when studying [Okumura *et al.*, 2022a] presented in the next chapter, which was before [Okumura and Défago, 2022b]. The history of research was a bit complicated!

regardless of when agents start and finish each action. The former is approached by introducing a notion of *scheduler*, originally developed in studies of theoretical distributed algorithms. The latter is approached by developing an extension of TSWAP.

Note that online TSWAP assumes centralized execution (see Chap. 3.1.1); agents require instructions repeatedly issued by a central controller at runtime. This is not the case in the following two chapters that allow decentralized execution.

8.1.2 Properties and Performance

Online TSWAP is *complete* in the sense that, as long as agents eventually take action, all targets are guaranteed to be occupied by agents. The formal definition will be provided in the chapter. We will see the power of this property in demonstrations by real robots that solve unlabeled-MAPF despite disturbance at runtime.

8.1.3 Chapter Organization

- Chapter 8.2 formulates the online time-independent problem and highlights differences from conventional MAPF.
- Chapter 8.3 presents online TSWAP.
- Chapter 8.4 presents robot demos.
- Chapter 8.5 concludes the chapter.

The code and movie are available at `https://kei18.github.io/tswap`.

8.1.4 Notations and Assumptions

$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \dots, n\}$	a tuple of agents
$\mathcal{S} = (s_1, s_2, \dots, s_n)$	start configuration, where $s_i \in V$
$\mathcal{T} = \{g_1, g_2, \dots, g_m\}$	target vertices, where $g_i \in V$
$\mathcal{M}$	$A \mapsto \mathcal{T}$, bijective function, assignment (matching)
$\mathcal{E} = (i, j, k, \dots)$	execution schedule, infinite sequence of agents

Caution

The chapter uses the representation by configurations.

8.2 Online Time-Independent Problem

8.2.1 Problem Formulation

An *execution schedule* is defined by an infinite sequence of agents, $\mathcal{E} = (i, j, k, \dots)$, defining the order in which each agent is *activated* and can move one step. We also call i's turn in $\mathcal{E}$ an *activation* for i. $\mathcal{E}$ is called *fair* when all agents appear infinitely often in $\mathcal{E}$.

Given an unlabeled-MAPF instance (agents A, graph G, starts $\mathcal{S}$, targets $\mathcal{T}$), a situation where all agents are at their initial locations (i.e., $\mathcal{S}$), and an execution schedule $\mathcal{E}$, an agent i can move to an adjacent vertex if *(i)* it is i's turn in $\mathcal{E}$ (i.e., i is activated) and *(ii)* the vertex is unoccupied by others. *Termination* is a situation where all targets are occupied by a subset of agents simultaneously. An algorithm is called *complete* when termination is achieved within a finite number of activations for any fair execution schedules.

It is worth mentioning that the above definition can be applied to *online time-independent MAPF* by changing the terminal condition. That is, termination is when all agents reach their assigned goals.

Metrics. Given an execution schedule, we rate the efficiency of agents' behaviors according to two metrics: *maximum-moves* and *sum-of-moves*. Their definitions are the same as for the offline problem (see Chap. 3.4.1). Note that sum-of-costs and makespan for the online problem are not formally defined because they should be measured according to actual time.

8.2.2 Differences from Conventional (unlabeled-)MAPF

The primary differences from conventional MAPF problems are as follows.

Collision. In MAPF, collisions are prevented by timings. In other words, collisions are prevented by planning such that two agents never share the same spatiotemporal point. Meanwhile, time-independent planning assumes collision avoidance is done on the fly according to online situations. Therefore, algorithm design needs to care about liveness (targets are eventually occupied), rather than safety (collision-free).

Scheduling. From the view of time-independent planning, though several minor differences exist, MAPF is seen as a planning problem assuming a special execution schedule like $\mathcal{E} = (1, 2, \ldots, n, 1, 2, \ldots, n, \ldots)$. Meanwhile, time-independent planning aims at overcoming planning under timing uncertainties. Therefore, "completeness" of time-independent planning is not only defined by problem instances but also considers schedules. Since any complete algorithms must deal with any fair schedules, they inherently assume timing uncertainties.

8.2.3 Other Remarks

For simplicity, the above definition assumes that at most one agent is activated at any time, therefore, the execution is determined by a sequence over the agents. However, there is no loss of generality as long as an agent can atomically reserve its next vertex before each move. Indeed, several robots acted simultaneously in our demonstrations. A similar concept regarding activation is known as a *locally central daemon* in theoretical distributed algorithms [Altisen *et al.*, 2019].

8.3 Online TSWAP

TSWAP is not limited to offline planning and can also apply to online planning with timing uncertainties. Algorithm 8.1 presents *online TSWAP* to solve the online time-independent problem. Before execution, TSWAP assigns targets to each agent (Line 1) and initiates the virtual configuration $\mathcal{Q}$ with $\mathcal{S}$ (Line 2). During execution, online TSWAP runs the procedures of the offline version for one agent (Lines 3–8). The primary difference is Line 6; if the virtual location (i.e., $\mathcal{Q}[i]$) is updated, then let the agent *actually* moves there.

Theorem 8.1. *Online TSWAP (Alg. 8.1) is* complete *for the online time-independent problem (Chap. 8.2).*

Proof. Most part is the same as for the proof of the completeness of the offline version (Thrm. 5.2). The problem assumes a fair execution schedule, therefore, the potential

Algorithm 8.1 Online TSWAP. The same parts with offline TSWAP is gray-out.

input: unlabeled-MAPF instance $(A, G, \mathcal{S}, \mathcal{T})$

offline phase

1: get an initial assignment $\mathcal{M}$

2: $\mathcal{Q} \leftarrow \mathcal{S}$

online phase when agent $i \in A$ is activated

3: **if** $\mathcal{Q}[i] = \mathcal{M}[i]$ **then continue** ▷ *rule-1*

4: $u \leftarrow \texttt{next\_vertex}(\mathcal{Q}[i], \mathcal{M}[i])$

5: let j be an agent in A s.t. $\mathcal{Q}[j] = u$

6: **if** $j = \perp$ **then** $\mathcal{Q}[i] \leftarrow u$; move i to u; **continue** ▷ *rule-2*

7: **if** $\mathcal{M}[j] = u$ **then** $\mathcal{M}[i], \mathcal{M}[j] \leftarrow \mathcal{M}[j], \mathcal{M}[i]$; **continue** ▷ *rule-3*

8: **if** $\exists$ deadlock for $A' \subseteq A \wedge i \in A'$ **then** rotate $\mathcal{M}$ of A'; **continue** ▷ *rule-4*

(otherwise; do nothing) ▷ *rule-5*

function ϕ (Eq. (5.2)) must decrease within the sufficiently long period during which all agents are activated at least once; otherwise, $\phi(\mathcal{Q}, \mathcal{M}) = 0$. □

Proposition 8.2. *Regardless of execution schedules, Online TSWAP has upper bounds of;*

- *maximum-moves:* $O(A \cdot \texttt{diam}(G))$
- *sum-of-moves:* $O(A \cdot \texttt{diam}(G))$

Proof. The same proof of Prop. 5.3 is applied. □

8.4 Demonstrations of Online Planning

Next, we see online TSWAP with real robots, using the bottleneck assignment of Alg. 5.4†.

8.4.1 Setup

Platform. *Toio* robots (`https://toio.io/`) were used to implement online TSWAP. The robots, connected to a computer via Bluetooth, evolve on a specific playmat and are controllable by instructions of absolute coordinates.

Usage. The robots were controlled in a *centralized* style, described as follows. A virtual grid was created on the playmat; the robots followed the grid. A central server (a laptop) managed the locations of all robots and issued the instructions (i.e., where to go) to each robot step-by-step. The instructions were periodically issued to each robot (per 50 ms) but they were issued *asynchronously* between robots. The code was written in Node.js.

8.4.2 Demonstration of Time Independence

Figure 8.1 shows a three-robot demonstration highlighting the time independence of TSWAP. In this demonstration, the experimenter disturbed robots' progression during the execution. No matter when the robots moved and no matter what order the robots moved, the online problem was certainly solved.

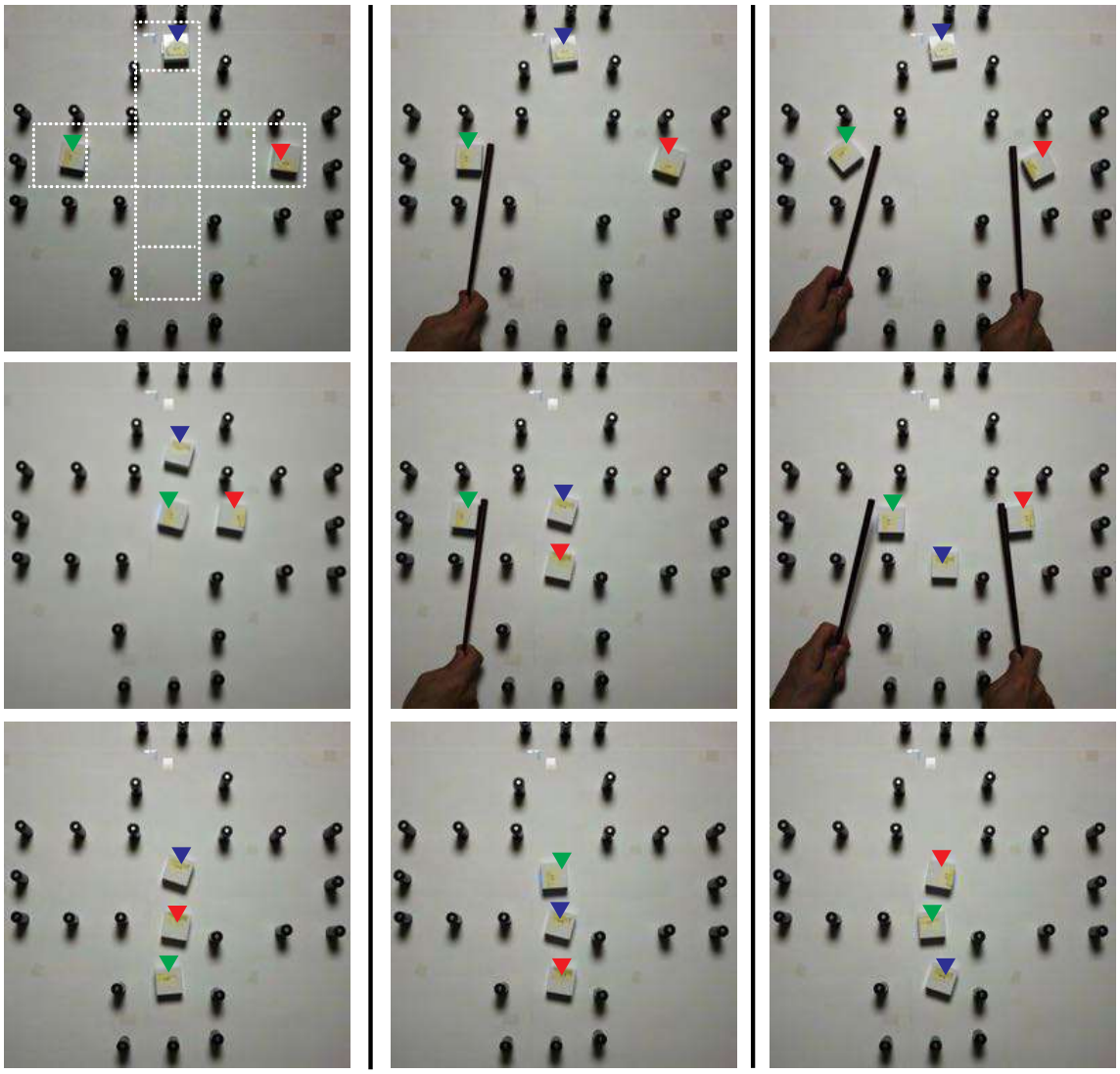


Figure 8.1: Demonstrations of time-independence of online TSWAP. Three scenarios with identical initial (top for each) and terminal (bottom for each) configurations were prepared. A graph is illustrated in the top-left image. The pictures are annotated by colored triangles to distinguish each robot. Although the experimenter disturbed robots' progression with chopsticks during the execution (middle and right settings), all robots reached the targets while flexibly swapping their assigned targets.

8.4.3 Demonstration of Delay Tolerance

The next demonstration presents a simple setting that highlights the delay tolerance of TSWAP. The comparison baseline was MCPs [Ma *et al.*, 2017a], an execution policy that makes agents move while preserving temporal dependencies of offline MAPF solution. Two scenarios with two robots were prepared while manipulating the robot's speed. In one scenario, robots moved at the usual speeds. In another scenario, one of them halved its speed, i.e., with delays, assuming an accident happened.

Figure 8.2 shows the configuration and both makespan and sum-of-costs results over ten repetitions measured in actual time. Without delays, TSWAP and MCPs did not differ in the results; however, with delays, the results of TSWAP were clearly better for both metrics. In MCPs, disturbing/delaying one robot may critically affect the entire performance because it cannot change the temporal orders during the execution, while TSWAP can adaptively address such effects.

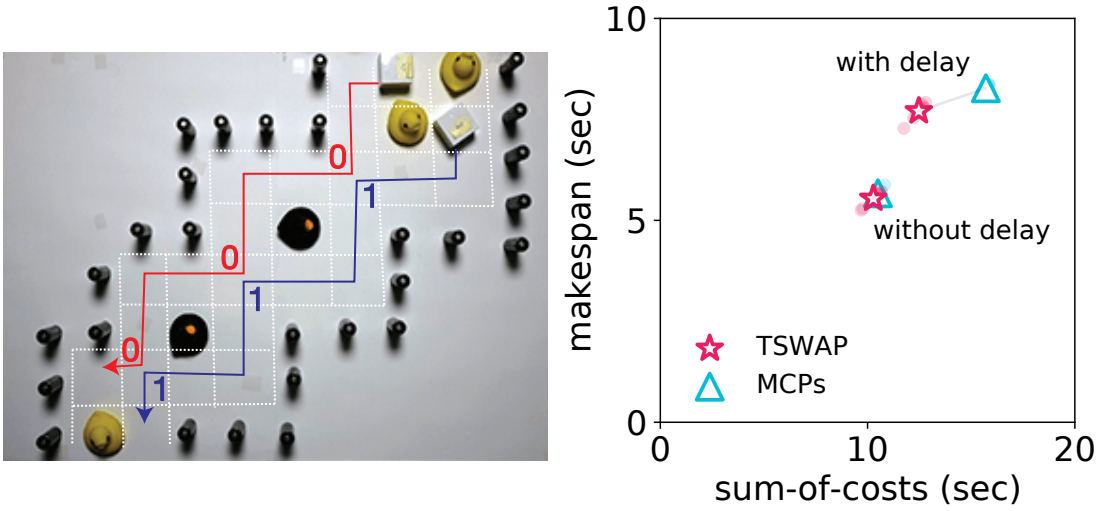


Figure 8.2: Demonstrations of delay tolerance. *left*: The initial configuration and the offline plan, accompanied by temporal orders, used in MCPs are illustrated. Targets are shown by the tips of arrows. The red one's speed was made half in one scenario (with delay). *right*: Average sum-of-costs. Scatter plots of ten repetitions are plotted by transparent points.

8.4.4 Demonstration with Eight Robots

Finally, Fig. 8.3 presents eight-robot demonstrations. Two scenarios were carefully designed to clarify the characteristics of TSWAP. MCPs' offline plans were obtained by the makespan-optimal algorithm [Yu and LaValle, 2013a] and ECBS-TA [Hönig *et al.*, 2018a]. In the first scenario, TSWAP performed better than the other two, but not so in the second scenario. This is because the latter has bottleneck vertices that all shortest paths from starts to targets must use (two middle vertices in the second column). Since TSWAP makes robots move following the shortest paths, all robots must use the bottleneck vertices, causing unavoidable congestion. In contrast, the former does not have such bottleneck vertices, resulting in a small execution time compared to the others.

8.5 Concluding Remarks

This chapter shortly presented an online time-independent problem and its complete algorithm called online TSWAP. Its benefits were demonstrated through real-robot demonstrations, such as time independence and delay tolerance.

Though the online time-independent problem can cope with timing uncertainties, it requires online intervention at runtime, inherently suffering from the weakness of centralized execution such as scalability. In the next chapter, we will see how to overcome this limitation by considering its *offline* version. Such a planning-execution style does not need online intervention by any central unit at runtime.

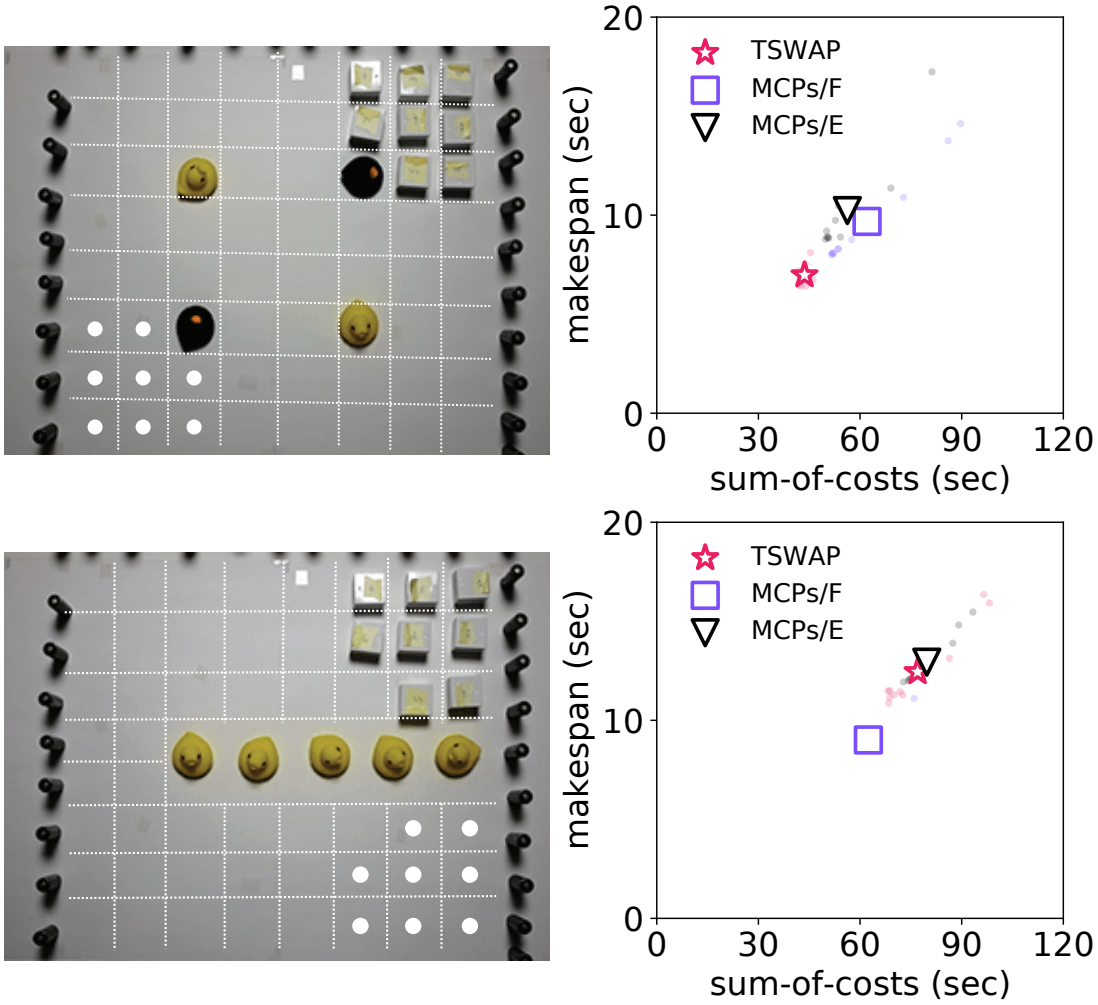


Figure 8.3: Demonstrations with eight robots execution. *left*: The initial configurations and virtual grids are shown. Targets are marked by white-filled circles. *right*: The results for makespan and sum-of-costs in two scenarios. Scatter plots of ten repetitions are shown with transparent points. The points "X" are average scores. "MCPs/F" represents MCPs with an offline plan obtained by the makespan optimal algorithm. "MCPs/E" represents those with ECBS-TA.

Chapter 9

Offline Planning to Overcome Timing Uncertainties in Execution

This chapter extends the concept of time-independent planning described in the previous chapter (Chap. 8). Here, we consider *offline* version of the problem, called *offline time-independent multi-agent path planning (OTIMAPP)*. The chapter is a "peak" of Part II because OTIMAPP embodies the execution strategy described in Chap. 3.6.5; solving OTIMAPP exemplifies an execution style in which deliberative planning guides reactive execution.

The chapter presents the *labeled* problem, unlike the previous chapter for unlabeled-MAPF. OTIMAPP was originally presented in [Okumura *et al.*, 2022a]; this chapter contains non-trivial extensions of the previous publication.[1]

9.1 Chapter Overview

The objective of the chapter is to establish an offline planning style that overcomes timing uncertainties in execution, embodied as OTIMAPP. The chapter provides both theoretical foundations and practical methods, as well as robot demonstrations.

9.1.1 What is OTIMAPP

OTIMAPP is a novel planning problem for multiple agents that cannot share holding resources. Given a graph and a set of start-goal pairs, the problem to be addressed is assigning a path to each agent, such that every agent eventually reaches its destination without blocking others, regardless of when each agent starts and finishes each own action. In contrast to conventional solution concepts of MAPF that rely on timings, once OTIMAPP solutions are obtained, they can be executed without any synchronization between agent actions. Moreover, there is a theoretical guarantee that all agents eventually reach their destinations, provided that agents avoid collisions reactively.

Example

Consider a situation presented in Fig. 9.1(left). This plan (a combination of two paths) runs the risk of an execution failure. If agent j is delayed for any reason while agent i moves two steps to the right, then each agent blocks the other and neither agent can progress on its respective path. In contrast, in Fig. 9.1(right), regardless of how the two agents are scheduled, both agents eventually reach their destinations unless they permanently stop the progression. OTIMAPP asks about the existence of such paths.

[1] This chapter is based on the journal submission currently under reviewed.

Figure 9.1: Example of OTIMAPP. A graph is depicted with black lines. Two agents (i, j) and their paths are colored. *left*: Both agents stop progression permanently due to mutual exclusion (i.e., no collision) if i moved two steps before j moves. *right*: As long as each agent follows its respective path, both agents eventually reach their last vertex; these paths constitute an OTIMAPP solution.

9.1.2 Why OTIMAPPP is Attractive

As discussed in the strategy of Chap. 3.6.5, unlike the previous chapter, we here focus on centralized planning followed by decentralized execution. Conventionally, robust offline planning methods without runtime intervention rely on their uncertainty models of timings. We can see such examples in [Wagner and Choset, 2017; Mansouri *et al.*, 2019; Peltzer *et al.*, 2020; Atzmon *et al.*, 2020a], to name just a few. Such approaches model uncertainties based on the probabilities of traveling time or assuming maximum delays. However, as discussed repeatedly throughout the dissertation, once failing to maintain the system status in pre-defined uncertainty models owing to some happening, we lose the outlook of system outcomes.

Instead of computing "timed" paths vulnerable to timing uncertainties, OTIMAPP allows agents to spontaneously act without any timing assumptions, as long as agents locally avoid collisions by some means. Once OTIMAPP solutions are obtained, *asynchronous execution* of the solutions can be performed, with a theoretical guarantee that all agents eventually reach their destinations. Here, asynchronous execution refers to those without cumbersome action synchronization between multiple agents, as assumed in conventional MAPF execution methods. In addition, since OTIMAPP execution only requires collision avoidance at runtime, if the avoidance scheme is implemented by each robot with local interactions (i.e., observation and communication), multi-agent navigation can be performed only with local interactions without any central control, again with the theoretical guarantee as above.

Potential Applications

Applications of OTIMAPP include robotic navigation, but they are not limited to robotics. Rather, OTIMAPP can be applied to situations wherein each *agent* attempts to transit to its *goal status* while always using certain *shared resources* in mutual exclusion, thus blocking other agents from accessing them until release. For instance, consider software agents in packet-switched networks with limited buffer spaces [Tel, 2000], where an agent is a packet, the goal is a packet destination, and shared resources are buffer spaces. Another example is the lock operations of transactions on distributed databases [Knapp, 1987], where an agent makes operation requests to the database, the goal is the completion of the operations, and shared resources are entries in the database. These are just a few examples of OTIMAPP applications.

9.1.3 What will be Presented

The chapter aims at the establishment of the foundation of OTIMAPP both theoretically and practically. Specifically, the contents below are categorized into theoretical and practical parts as follows.

Theoretical Part

We first see the formalization of OTIMAPP. The chapter then derives a necessary and sufficient condition for a *solution*, i.e., a tuple of paths that makes all agents reach their goals without timing assumptions. For this purpose, four types of *deadlocks* are introduced; *cyclic* or *terminal*; *potential* or *reachable*. Using this condition, computational complexity analyses are performed. The results reveal that *(i)* finding a solution is NP-hard, and *(ii)* verifying a solution is co-NP-complete. Both proofs are by reductions of the 3-SAT problem. Further analyses are provided regarding the cost of time independence, in particular, solvability and optimality, compared to two well-known problems for multiple moving agents on graphs: the pebble motion (PM) problem [Kornhauser *et al.*, 1984] (i.e., a generalization of a sliding puzzle) and MAPF. Against the others, OTIMAPP is restricted in solvability and results in higher optimal costs.

Practical Part

Two approaches for deriving solutions are presented: prioritized planning (PP) and deadlock-based search (DBS). Both algorithms are derived from basic MAPF algorithms [Erdmann and Lozano-Perez, 1987; Sharon *et al.*, 2015] and relied on a newly developed procedure to detect deadlocks within a set of paths. The chapter further presents a relaxed solution concept, called m-tolerant solutions, which ensures no deadlocks with m agents or fewer, aiming at solving large OTIMAPP instances. The rationale is that deadlocks involving many agents are rare. Unfortunately, the complexity class does not change, and it is still in NP-hard. Through experiments, including robot demonstrations, the chapter evaluates these algorithms and demonstrates the following four:

- Either PP or DBS can compute large OTIMAPP instances to a certain extent.
- OTIMAPP solutions cause robots to move efficiently in an adverse environment for timing assumptions compared to existing approaches with runtime support [Ma *et al.*, 2017a; Okumura *et al.*, 2021b].
- A relaxation of the solution concept (m-tolerant solutions) moderately offloads the burden of computation but with a risk of execution failure.
- Solutions are executable with physical robots in both a centralized style and a decentralized style with only local interactions, without cumbersome procedures of online interventions.

9.1.4 Chapter Organization

- Chapter 9.2 formulates the OTIMAPP problem.
- Chapter 9.3 identifies a necessary and sufficient condition for solutions.
- Chapter 9.4 conducts computational complexity analyses.
- Chapter 9.5 analyzes the cost of time independence, compared to PM and MAPF.
- Chapter 9.6 presents how to solve OTIMAPP.
- Chapter 9.7 presents a relaxed solution concept.
- Chapter 9.8 presents empirical results including robot demonstrations.
- Chapter 9.9 reviews related work to OTIMAPP.
- Chapter 9.10 concludes the chapter with discussions of interesting directions to extend OTIMAPP.

The code and movie are available at `https://kei18.github.io/otimapp`.

9.1.5 Notations and Assumptions

$\bot$	not found, undefined
$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \ldots, n\}$	a tuple of agents
s_i, g_i	start and goal vertices of agent i
$\mathcal{E} = (i, j, k, \ldots)$	execution schedule, infinite sequence of agents

Caution

In this chapter, *all indexes start from one*. This is because OTIMAPP no more includes time factors. Furthermore, the chapter uses the representation by paths.

9.2 Offline Time-Independent Problem

9.2.1 Problem Formulation

Problem Instance. An *OTIMAPP instance* is given by a graph $G = (V, E)$, a set of agents $A = \{1, 2, \ldots, n\}$, injective initial-state function $s : A \mapsto V$, and injective goal-state function $g : A \mapsto V$. Let s_i and g_i denote $s(i)$ and $g(i)$, respectively. An OTIMAPP instance on a digraph is similar to the undirected case.

Execution. An *execution schedule* is an infinite sequence of agents (e.g., $\mathcal{E} = (i, j, k, \ldots)$). An *OTIMAPP execution* is defined by an OTIMAPP instance, execution schedule $\mathcal{E}$, and tuple of paths $\mathbf{\Pi} = (\Pi_1, \ldots, \Pi_n)$ as follows. The agents are *activated* in turn according to $\mathcal{E}$. Upon activation and until it reaches the end of its path Π_i, agent i takes a single step along Π_i if the vertex is vacant or stays at its current location otherwise. After reaching the end of the path, the agent remains at the last vertex of the path. $\mathcal{E}$ is called *fair* when every agent appears infinitely-many times in $\mathcal{E}$.

Decision Problem. An *OTIMAPP problem* determines whether there exists a tuple of paths $(\Pi_1, \ldots, \Pi_n)$, such that

- each path for agent i, Π_i, begins from s_i and ends at g_i,
- for any fair execution schedule, all agents reach the end of their paths (i.e., goals) with a finite number of activations.

A *solution* is a tuple of paths that satisfies these two conditions.

Progres Index. For convenience, the location of agent i is associated with a *progress index*, denoted as $clock_i \in \{1, \cdots, |\Pi_i|\}$, and is represented as $\Pi_i[clock_i]$, where $\Pi_i[k]$ is the k-th vertex in Π_i. Every progress index starts at one and is incremented each time the agent moves a step along its path. The progress index is non-decreasing and no longer increases after reaching the end of the path.

9.2.2 Difference from Online Problem

In the online problem, it is assumed to perform planning and execution based on the current configuration; hence, both a planner and agents are *reactive*. In contrast, in the offline problem, we only know the initial configuration and cannot access what configurations are realized at runtime; therefore, the planner is *deliberative*. Meanwhile, agents are *reactive* because they are assumed to react to online situations. In short, OTIMAPP disjoints styles of the planner and the agents.

9.2.3 Other Remarks

Similar to online time-independent planning, any solution must address all timing uncertainties because the execution schedules are unknown during offline planning. Moreover, the OTIMAPP definition assumes that agents are activated sequentially, i.e., each activation is atomic. However, there is no loss of generality provided that an agent can atomically reserve its next location before each move. In other words, unless two agents never enter the same vertex simultaneously, OTIMAPP solutions are executable. Indeed, several robots acted simultaneously in our demonstrations. A similar concept regarding activation is known as a *locally central daemon* in theoretical distributed algorithms [Altisen *et al.*, 2019]. Note that "wait" actions in conventional MAPF, such that an agent i chooses staying in the current location as the next action, is meaningless in OTIMAPP. This is because there are execution schedules such that i consecutively takes actions, i.e., $(\ldots, i, i, \ldots)$. In the rest of the chapter, each path Π_i is assumed to start from s_i and end at g_i to focus on analyses related to schedules.

9.3 Solution Analysis

Given a solution candidate (i.e., a tuple of paths), the first question is whether it is a (feasible) solution. This section derives the necessary and sufficient condition for the solutions. For this purpose, four types of *deadlocks* are introduced, categorized as; *cyclic* or *terminal*; *potential* or *reachable*. Informally, a cyclic deadlock is a situation wherein agent i wants to move to the current vertex of j, who wants to move to the current vertex of k, and who wants to move to … of i. A terminal deadlock is a situation wherein agent i reaches its destination and blocks the progress of agent j. A potential deadlock is called reachable when an execution schedule exists and leads to the deadlock.

Definition 9.1 (potential cyclic deadlock)**.** *Given an OTIMAPP instance and a tuple of paths* $(\Pi_1, \ldots \Pi_n)$*, a* potential cyclic deadlock *is a pair of tuples* $\big((i, j, k, \ldots, l), (t_i, t_j, t_k, \ldots, t_l)\big)$ *such that* $\Pi_i[t_i + 1] = \Pi_j[t_j] \wedge \Pi_j[t_j + 1] = \Pi_k[t_k] \wedge \ldots \wedge \Pi_l[t_l + 1] = \Pi_i[t_i]$*. The elements of the first tuple are not duplicated.*

Definition 9.2 (potential terminal deadlock)**.** *Given an OTIMAPP instance and a tuple of paths* $(\Pi_1, \ldots \Pi_n)$*, a* potential terminal deadlock *is a tuple* (i, j, t_j) *such that* $\Pi_i[-1] = \Pi_j[t_j]$ *and* $i \neq j$*.*

Definition 9.3 (reachable cyclic deadlock)**.** *A potential cyclic deadlock* $\big((i, j, \ldots, l), (t_i, t_j, \ldots, t_l)\big)$ *is* reachable *when there is an execution schedule that leads to a situation where* $clock_i = t_i \wedge clock_j = t_j \wedge \ldots \wedge clock_l = t_l$*. This deadlock is called a* reachable cyclic deadlock*.*

Definition 9.4 (reachable terminal deadlock)**.** *A potential terminal deadlock* $\big(i, j, t_j\big)$ *is* reachable *when there is an execution schedule that leads to a situation where* $clock_i = |\Pi_i| \wedge clock_j = t_j - 1$*. This deadlock is called a* reachable terminal deadlock*.*

Both reachable (or potential) cyclic/terminal deadlocks are called reachable (resp. potential) deadlocks and simply called "deadlock" whenever the context is obvious. At least one execution schedule is required to verify whether a potential deadlock is reachable. For instance, in Fig. 9.1(left), a schedule $(i, i, \ldots)$ is evidence. A potential deadlock is not always reachable as illustrated in Fig. 9.2.

Theorem 9.5 (necessary and sufficient condition)**.** *Given an OTIMAPP instance, a tuple of path* $(\Pi_1, \ldots, \Pi_n)$ *is a solution if and only if there are:*

- *No reachable terminal deadlocks.*

Figure 9.2: Examples of unreachable potential deadlocks. *left*: cyclic; $\big((i,j,k),(3,1,2)\big)$. *right*: terminal; $(i,j,2)$.

- *No reachable cyclic deadlocks.*

Proof. Without "no reachable terminal deadlocks," there is an execution that one agent arrives at its goal and remains there, disturbing the progression of another agent. Without "no reachable cyclic deadlocks," a cyclic deadlock might occur, and these agents stop the progression. Therefore, these two conditions are necessary.

Now, consider the proof for these two conditions being sufficient. Given a solution candidate $(\Pi_1,\dots,\Pi_n)$ with no reachable deadlocks, consider the potential function $\phi := \sum_{i\in A}(|\Pi_i| - clock_i)$, defined over a configuration $(clock_1,\dots,clock_n)$.[2] As the progress indexes $clock_i$ are non-decreasing, ϕ is non-increasing, and $\phi = 0$ means that all agents have reached their goals. Furthermore, when $\phi > 0$, ϕ is guaranteed to decrease if each agent is activated at least once. This is proven by contradiction as follows.

Assume that there exists a configuration such that the value of ϕ is unchanged after the activation of all agents. Since $\phi \neq 0$, there are agents with progress indices lower than the maximum values. Let them be $B \subseteq A$. For an agent $i \in B$, $\Pi_i[clock_i + 1]$ is occupied by another agent j; otherwise, i moves there. The agent j must be in B due to "no reachable terminal deadlocks." This is the same for j; i.e., there exists an agent $k \in B$, such that $\Pi_j[clock_j + 1] = \Pi_k[clock_k]$. By induction, since the number of agents is finite, this sequence of agents must form a cycle somewhere, i.e., a cyclic deadlock. However, this contradicts "no reachable cyclic deadlocks."

Each agent is activated at least once in a sufficiently long period because of the fair assumption, deriving the statement. □

When reachable deadlocks are inevitable, OTIMAPP instances have no solution. Figure 9.3 shows such examples.

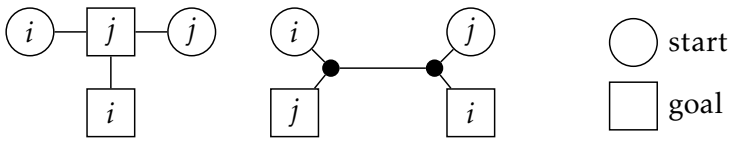


Figure 9.3: Unsolvable OTIMAPP instances. To visualize instances, the figures distinguish three types of vertices: start (circle), goal (square), and others (small dots). *left*: There is an inevitable reachable terminal deadlock, i.e., j reaches its goal before i. *right*: There is an inevitable reachable cyclic deadlock, i.e., when both agents enter the middle two vertices. Note that these instances are solvable in MAPF or the pebble motion (PM) problem.

[2] Precisely, this is not a "configuration" used before that specifies a tuple of locations, however, it is equivalent because each progress index $clock_i$ is associated with a path Π_i.

9.4 Computational Complexity

Next, this section discusses the complexity of OTIMAPP. In particular, the section answers two questions: the difficulty of finding solutions (Chapter 9.4.1) and that of verifying solutions (Chapter 9.4.2). The primary result is that both problems are computationally intractable; the former is NP-hard and the latter is co-NP-complete. Both proofs are based on reductions of the 3-SAT problem, determining the satisfiability of a formula in conjunctive normal form with three literals in each clause.

9.4.1 Finding Solutions

The NP-hardness for directed graphs is first derived. Then, it is extended to the case of undirected graphs. The following proof is partially inspired by the NP-hardness of MAPF in digraphs [Nebel, 2020].

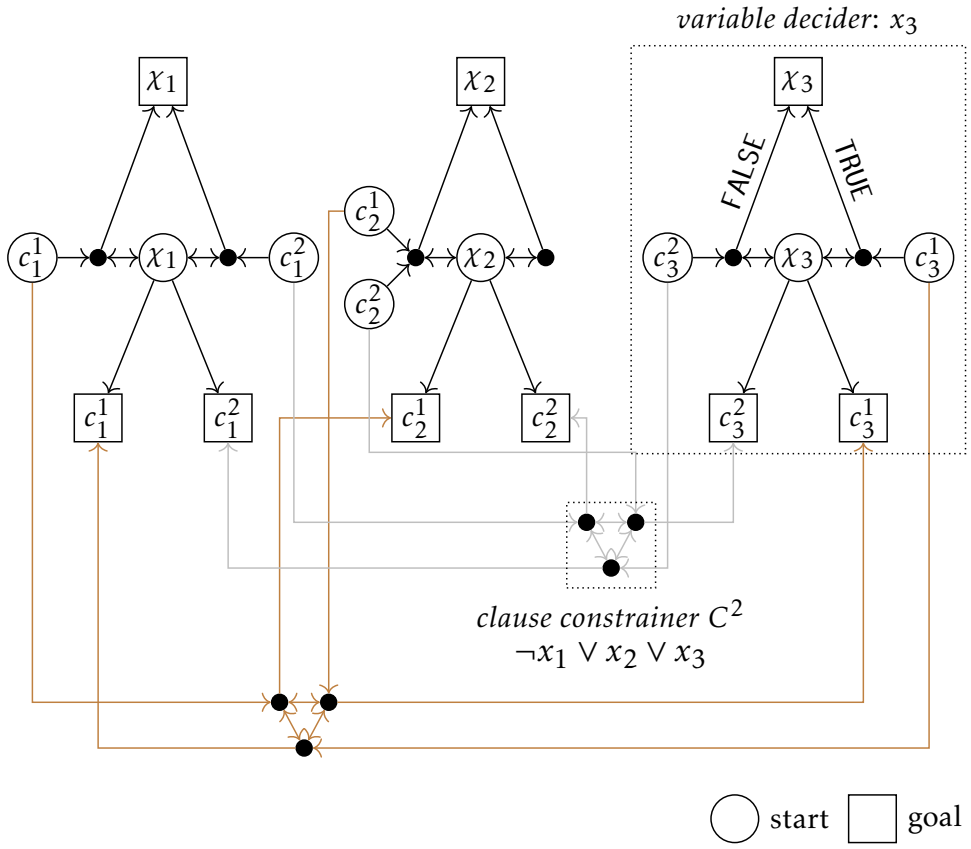


Figure 9.4: OTIMAPP instance reduced from 3-SAT. The corresponding formula is $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$.

Theorem 9.6 (complexity on digraphs)**.** *OTIMAPP on* directed *graphs is NP-hard.*

Proof. The proof is a reduction of the 3-SAT problem. Figure 9.4 is an example of the reduction from a formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$.

A. Construction of an OTIMAPP instance

The proof introduces two gadgets, called *variable decider* and *clause constrainer*. The OTIMAPP instance contains one variable decider for each variable and one clause constrainer for each clause.

The variable decider for variable x_i assigns x_i as TRUE or FALSE. This gadget contains one agent χ_i with two paths to reach its goal: *left* or *right*. Taking a left path corresponds to assigning x_i as FALSE, and vice versa. For the j-th clause C^j in the formula, when its k-th literal is either x_i or $\neg x_i$, the gadget further includes one agent c_k^j. Its start and goal are positioned on the *right* side from χ_i when the literal is a negation; otherwise, they are positioned on the *left* side. c_k^j has two alternate paths for reaching its goal: a path within the variable decider or a path via a clause constrainer. The former is only available when χ_i takes a path in the opposite direction to avoid a reachable cyclic deadlock.

The clause constrainer for clause C^j connects the start and the goal of c_k^j. The gadget contains a triangle. Each literal c_k^j enters the triangle from a distinct vertex and exits it from another vertex. As a result, this gadget prevents three literals in C^j from being FALSE simultaneously; otherwise, three agents enter the gadget, and there is a reachable cyclic deadlock.

The number of agents, vertices, and edges are all polynomials with respect to the size of the formula.

B. The formula is satisfiable if OTIMAPP has a solution

The use of one clause constrainer by three literal agents results in a reachable cyclic deadlock. Thus, in every OTIMAPP solution, at least one literal agent for each clause avoids using a clause constrainer. Then, the corresponding variable agent follows the opposite path to that clause agent, thus satisfying every clause. For instance, in Fig. 9.4, if c_3^2 avoids using the clause constrainer, χ_3 must take the right path. This sets x_3 to TRUE, thus satisfying clause C^2.

C. OTIMAPP has a solution if the formula is satisfiable

If satisfiable, let the variable agent χ_i follow a path that follows the assignment. Let c_k^j take a path within the variable decider when χ_i follows the opposite direction. Otherwise, let c_k^j use the clause constrainer. For instance, if x_3 is set to TRUE, χ_3 takes the right path. c_3^2 uses the path within the variable decider. In contrast, c_3^1 uses a clause constrainer. Observe that three agents never enter one clause constrainer due to satisfiability; otherwise, the corresponding clause is not satisfied. Consequently, these paths constitute a solution. □

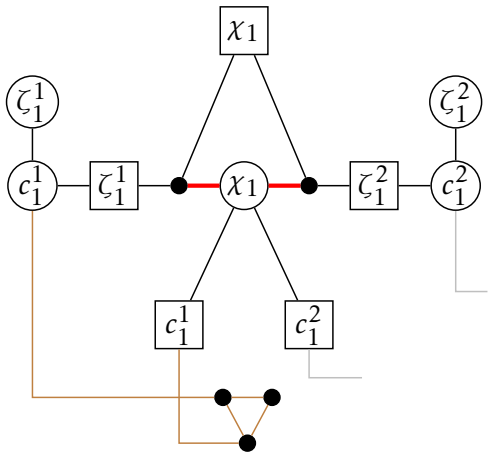


Figure 9.5: Variable decider of x_1 with undirected edges.

Theorem 9.7 (complexity on undirected graphs)**.** *OTIMAPP on* undirected *graphs is NP-hard.*

Proof. The result is derived by extending the proof of NP-hardness on the digraphs. Recall that each variable decider for variable x_i contains an agent c_k^j corresponding to k-th literal in clause C^j when the literal is either x_i or $\neg x_i$. The proof for undirected graphs revises the variable decider by adding a new agent ζ_k^j for each c_k^j and converting all directed edges of the original variable decider into undirected edges. Figure 9.5 shows an example of the revised one for x_1. The start and goal of ζ_k^j are positioned next to the start of c_k^j.

Using this revised gadget, the proof claims that an OTIMAPP instance has a solution if and only if the formula is satisfiable. This claim holds when solution paths for an agent χ_i in a variable decider for x_i is virtually the two shortest paths (both two steps; 'left – up' or 'right – up'). If so, to avoid deadlocks, c_k^j positioned on the left side must use a clause constrainer if χ_i takes the left path, and vice versa. The remainder of the proof is performed by applying the same arguments as in directed graphs. The reduction is still in polynomial time.

It is now proven that χ_i takes the shortest paths in the OTIMAPP solutions, as follows:

A. ζ_k^j must take the shortest path (two steps)

Observe that ζ_k^j must pass through at least one goal of another agent if it does not follow the shortest path. For instance, ζ_1^1 must use either goal of c_1^1, c_1^2, or ζ_1^2. Now, consider an execution schedule such that ζ_k^j is not activated for a sufficiently long time and remains at its start. In such execution schedules, the other agents reach their goal because the start of ζ_k^j never blocks any other paths. ζ_k^j has no choice other than to take the shortest path; otherwise, terminal deadlocks exist.

B. χ_i must take one of the shortest paths (two steps)

Assume, by contradiction, that χ_i does not take the shortest paths. Specifically, we are interested in paths without the use of the left/right next edge at the start of χ_i (red-colored in Fig. 9.5). However, such paths must use one of the goals of ζ agents. As these ζ agents take their shortest paths, there are reachable terminal deadlocks.

Strictly speaking, neither χ_i nor ζ_k^j must take their shortest paths. For instance, χ_1 can first visit the left vertex next to its start, then move back to its start, again visit the left vertex, and finally, visit its goal. However, such trivial variant paths do not affect the proof structure. □

9.4.2 Verification

The co-NP completeness of the verification relies on a lemma stating that finding cyclic deadlocks is computationally intractable. Subsequently, the complexity result is derived because the solution has no reachable deadlocks, according to the necessary condition (Thrm. 9.5).

Lemma 9.8 (complexity of detecting cyclic deadlocks)**.** *Determining whether a set of paths contains reachable or potential cyclic deadlocks is NP-complete.*

Proof. The proof is a reduction of the 3-SAT problem, i.e., constructing a combination of an OTIMAPP instance and a set of paths, such that potential cyclic deadlocks exist if and

only if the corresponding formula is satisfiable. The below shows the case of directed graphs. The proof procedure applies to the undirected case without modifications. In addition, all potential cyclic deadlocks are reachable in the translated problem. The reduction is performed in polynomial time, deriving the NP-hardness for detecting both reachable and potential cyclic deadlocks. Since a potential cyclic deadlock can be verified in polynomial time, and since a reachable cyclic deadlock can be verified in polynomial time with an execution schedule, they are NP-complete.

It is now explained how to translate the 3-SAT formula to the OTIMAPP instance and the corresponding set of paths. Without loss of generality, the proof assumes that all variables appear positively and negatively in the formula. Throughout the proof, the following example is used:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \wedge \neg x_3) \tag{9.1}$$

Its outcome is partially depicted in Fig. 9.6. The complete version is presented in Fig. 9.7.

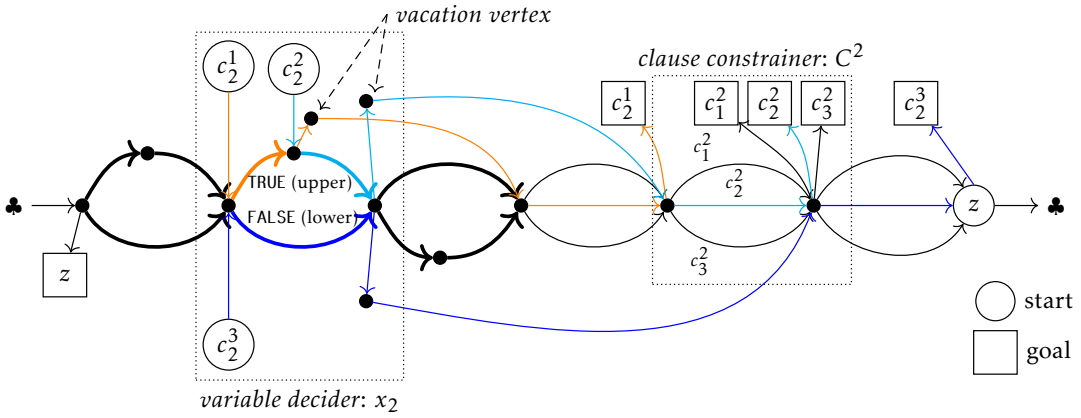


Figure 9.6: OTIMAPP instance and solution reduced from 3-SAT. The corresponding instance is $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \wedge \neg x3)$. For visualization, we break a large circle; regard two ♣ marks as connected. Omitted vertices and edges are complemented in Fig. 9.7.

A. Construction of an OTIMAPP instance and a set of paths

For each literal in each clause, a *literal agent* is introduced. Here, c_k^j denotes a literal agent for the k-th literal in j-th clause C^j in the formula. In addition, a special agent, z, is used.

Next, consider two gadgets: *variable decider* and *clause constrainer*. Note that they are different from those used in the proof of Thrm. 9.6; however, their intuitions are similar, thus, the same names are used.

The variable decider determines whether the variable x_i occurs positively or negatively. One gadget is introduced for each variable. All the literal agents for x_i (i.e., either x_i or $\neg x_i$) begin from the vertices in this gadget. The gadget contains two paths: an *upper* path, corresponding to assigning TRUE to x_i, and a *lower* path, corresponding to FALSE assignment to x_i. Positive literals are connected to the upper path, whereas negative literals are connected to the lower path. For instance, x_2 has three literal agents: c_2^1 (x_2), c_2^2 (x_2), and c_2^3 $(\neg\, x_2)$. In Fig. 9.6, the upper and lower paths are highlighted by bold lines. c_2^1 and c_2^2 are connected to the upper path while c_2^3 is connected to the lower path. Each literal agent uses one edge in the upper/lower path and moves to a clause constrainer via one *vacation* vertex.

The clause constrainer contains all goals of the literal agents in the clause. Three edges are used to reach the goals. Each edge is for each literal agent. For instance, the clause constrainer of C^2 contains the goals of c_1^2, c_2^2, and c_3^2. In Fig. 9.6, three edges are annotated with the agent's name. c_2^2 is supposed to use the colored middle one. Note that multiple edges are used for simplicity. The gadget can be easily converted into a simple graph, as shown immediately after the proof.

As a result, all literal agents take six edges to reach their goals. This is visualized by colored edges in Fig. 9.6 and 9.7. The special agent z uses two edges to reach its goal, through ♣ marks in the figure. Now, the description has finished about how to construct the OTIMAPP instance and the corresponding set of paths. The remaining part indicates that these paths contain potential/reachable cyclic deadlocks if and only if the formula is satisfiable. This translation from the formula into an OTIMAPP instance and paths is clearly realized in polynomial time.

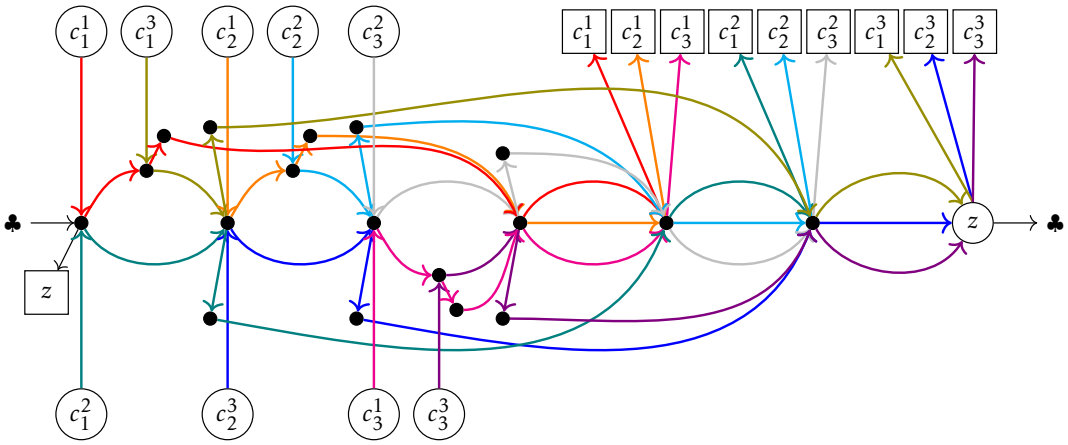


Figure 9.7: Complete version of Fig. 9.6. The formula is $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \wedge \neg x3)$. Each color corresponds to a path for each agent.

B. A potential cyclic deadlock exists if the formula is satisfiable

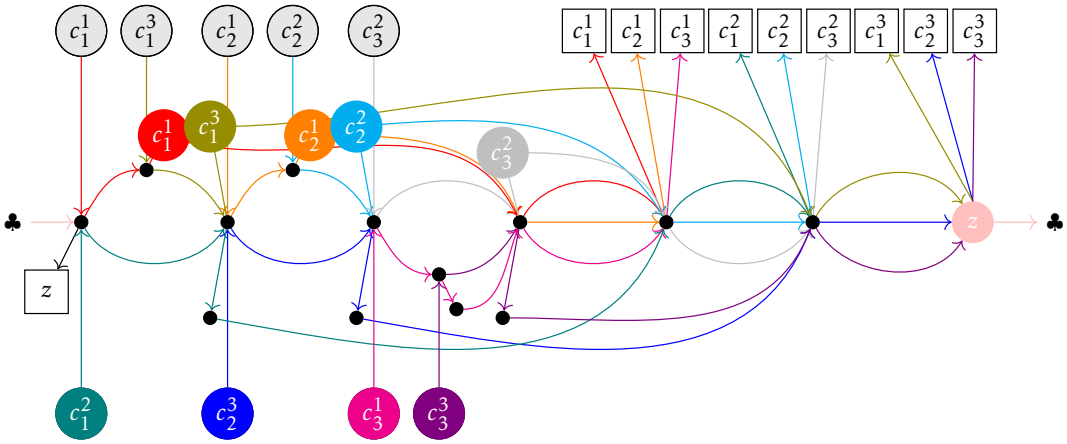
To observe this, if a potential cyclic deadlock exists, the agents must attempt to use: a) either an upper or a lower path for each variable decider, b) one edge for each clause constrainer, and c) edge for z (i.e., ♣).

When the formula is satisfiable for one assignment, consider the following execution.

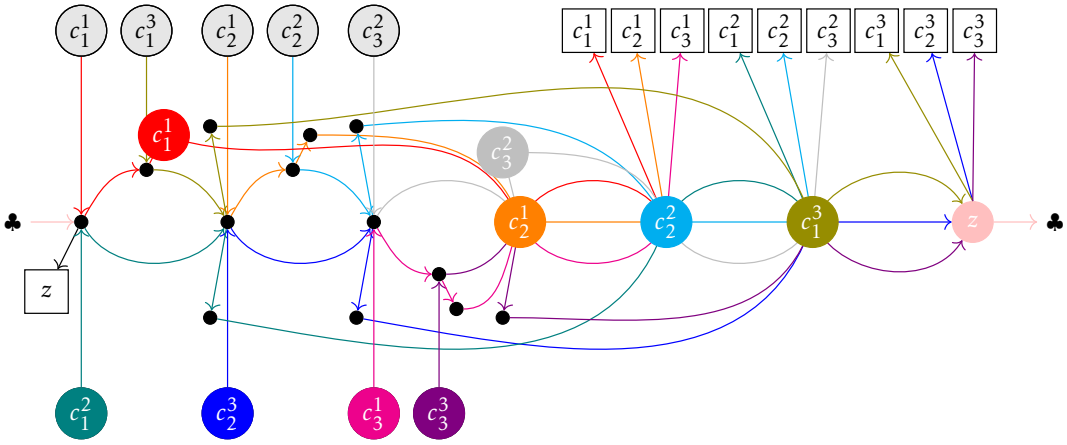
1. For each assigned value, move the corresponding clause agents to vacation vertices in each variable decider, i.e., one step before clause constrainers.
2. Among the above agents, for each clause constrainer, there is at least one agent capable of entering the clause constrainer owing to satisfiability. Move them one step further. As a result, all clause constrainers have one agent at the first vertices. Vertices in upper/lower paths in the variable deciders must be vacant now.
3. Move all unassigned clause agents one step. As a result, all vertices in the unassigned paths are filled by the unassigned clause agents.

We now have a cyclic deadlock. This deadlock is reachable and thus potential.

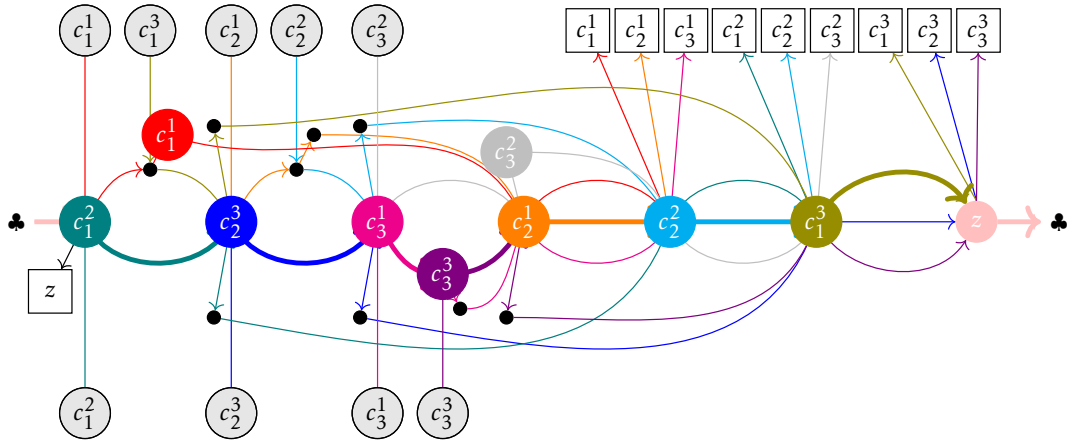
For example, consider a satisfiable assignment x_1 = TRUE, x_2 = TRUE, x_3 = TRUE. Initially, move assigned agents, c_1^1, c_2^1, c_2^2, c_3^2, and c_1^3 to vacation vertices in each variable decider (Fig. 9.8; Step 1). Next, move c_2^1, c_2^2, and c_1^3 to the first vertices of each clause constrainer of C^1, C^2, and C^3, respectively (Fig. 9.8; Step 2). Subsequently, move all



Step 1: Move assigned agents to vacation vertices



Step 2: Fill clause constrainers



Step 3: Move unassigned agents one step

Figure 9.8: Construction of a reachable deadlock. The formula is $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \wedge \neg x3)$. The assignment is $x_1 = \mathsf{TRUE}$, $x_2 = \mathsf{TRUE}$, and $x_3 = \mathsf{TRUE}$. Locations of all agents are colored. When an agent departs from its start, the corresponding vertex is grayed out. Bold lines in Step 3 constitute a reachable deadlock.

unassigned agents, c_1^2, c_2^3, c_3^1, and c_3^3, one step (Fig. 9.8; Step 3). Consequently, there is a cyclic deadlock with $c_1^2, c_2^3, c_3^1, c_3^3, c_2^1, c_2^2, c_1^3$, and z, as annotated with bold lines in Fig. 9.8.

C. The formula is satisfiable if a potential cyclic deadlock exists

To form a potential cyclic deadlock, for each variable decider, one or several agents attempt to move along either an upper or a lower path. Consider assigning an opposite value to the path used for the variable. For instance, if c_2^1 and c_2^2 are involved in the deadlock at the variable decider (see Fig. 9.6), then assign FALSE to x_2. This assignment must satisfy the formula because at least one literal in each clause becomes TRUE; otherwise, at least one clause constrainer exists, such that the first vertex is empty, i.e., no deadlock.

D. All potential cyclic deadlocks are reachable

Thus far, the proof has established the claim that a potential cyclic deadlock exists if and only if the formula is satisfiable. Next, it is claimed that all potential cyclic deadlocks are reachable. According to the above discussion, given a potential cyclic deadlock, the corresponding satisfiable assignment exists. Consider the execution of Part B using this assignment, slightly changing Step 2. In this step, arbitrary agents can be selected for each clause constrainer. Therefore, the agents involved in a potential cyclic deadlock can be selected. Consequently, this deadlock is reachable. □

In the proof of Lemma 9.8, multiple edges in a gadget *clause constrainer* for the reduction from 3-SAT are used. Since OTIMAPP assumes a simple graph (i.e., no multiple edges), how to convert it into a *correct* OTIMAPP instance is complemented here. Figure 9.9 shows an example of the clause constrainer for C^2. Recall that a clause constrainer contains all goals for the corresponding clause agents. In this new gadget, intermediate vertices are added for each edge, which can potentially trigger cyclic deadlocks. For each agent c_k^j, a new agent $\hat{c}_k^j$ is introduced. The start point is an intermediate vertex, whereas the goal point is the original goal of c_k^j. Furthermore, a goal for c_k^j is changed to the start of $\hat{c}_k^j$. Consider now replacing all old clause constrainers with this new gadgets. The translation is performed in polynomial time. The remainder of the proof is straightforward, as from Lemma 9.8.

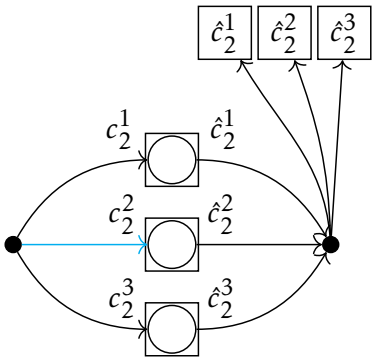


Figure 9.9: Example of *clause constrainer* without multiple edges. Used in the proof of Lemma 9.8.

Finally, the co-NP-completeness is derived as follows.

Theorem 9.9 (complexity of verification)**.** *Verifying a solution of OTIMAPP is co-NP-complete.*

Proof. The necessary condition of Theorem 9.5 states that a solution has non-reachable terminal/cyclic deadlocks. Verifying no reachable terminal deadlocks is in co-NP; indeed, a reachable terminal deadlock is verifiable in polynomial time given an appro-

priate execution schedule. Verifying no reachable cyclic deadlocks is co-NP-complete, according to Lemma 9.8. □

9.5 Cost of Time Independence

This section analyzes OTIMAPP and compares it with other problems for multiple moving agents on graphs, namely the *pebble motion (PM)* problem [Kornhauser *et al.*, 1984], which is a generalization of a sliding puzzle, and MAPF [Stern *et al.*, 2019], which finds "timed" paths on graphs. Because PM, MAPF, and OTIMAPP have the same inputs and similar outputs, i.e., instructions on how agents move, the analysis of the OTIMAPP compared to those two to clarify the characteristics is a sensible option.

A formal description of PM is as follows. Refer to Chap. 3.2.1 for that of MAPF.

Definition 9.10 (PM)**.** *The pebble motion (PM) problem is defined as follows. The inputs are a graph, set of agents, and start-goal pair for each agent. The starts and goals are distinct between agents. In one operation, one agent is moved from its current vertex to an adjacent vacant vertex. A solution is a sequence of operations that makes all agents reach their goals.*

The main observation is that time independence is costly; solvable instances are more restrictive, and the solution cost increases.

9.5.1 Solvability

PM prohibits two agents from moving simultaneously, whereas MAPF allows it. This causes a slight change; MAPF allows a *rotation* of agents, i.e., a set of agents move along a cycle simultaneously, whereas the PM cannot. Emulating a feasible solution for PM using MAPF is possible by moving an agent one by one. In summary;

Proposition 9.11 (solvability: PM vs. MAPF)**.** *Solvable instances for PM are solvable for MAPF. The opposite does not hold.*

OTIMAPP is more restrictive than PM and MAPF.

Proposition 9.12 (solvability: OTIMAPP vs. PM, MAPF)**.** *Solvable instances for OTIMAPP are solvable for PM and MAPF. The opposite does not hold.*

Proof. A solvable instance of OTIMAPP has a solution. This solution operates with any fair execution schedules. Take one of the schedules. The corresponding execution can be emulated by PM, deriving the first claim, along with Prop. 9.11. For the second claim, we have already seen examples in Fig. 9.3. □

9.5.2 Optimality

Next, we consider the solution quality, i.e., the *cost* of the solutions. For the optimization criteria of OTIMAPP, this section uses *the minimum activation counts* wherein all agents reach the goals, e.g., seven in Fig. 9.1 (see right). Let C^*_{OTIMAPP} denote the optimal cost for a given OTIMAPP instance.

The cost of PM is the number of operations, e.g., the optimal cost in Fig. 9.1 is six (see left). Let C^*_{PM} denote the optimal cost of PM. Note that solving PM optimally is NP-hard [Ratner and Warmuth, 1986].

Proposition 9.13 (optimality: OTIMAPP vs. PM)**.** *For any instances wherein OTIMAPP is solvable,* $1 \leq C^*_{OTIMAPP}/C^*_{PM}$*. The bound is tight. For any* $k \in \mathbb{R}$*, there exists an instance where* $C^*_{OTIMAPP}/C^*_{PM} > k$*.*

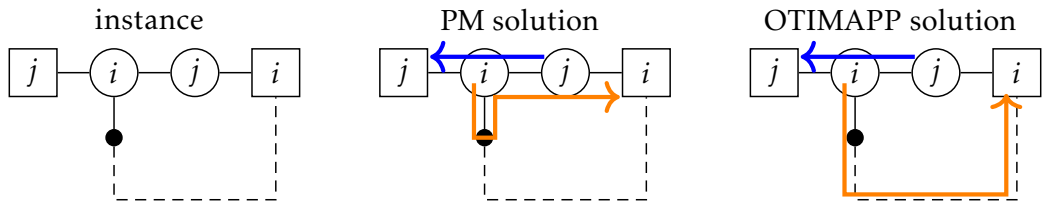


Figure 9.10: Example with a large optimal ratio between PM and OTIMAPP. Regard the dashed line (black) as a sufficiently long path. The optimal PM solution requires six operations, while in OTIMAPP, either i or j has to take the long path.

Proof. $C^*_{\text{OTIMAPP}} \geq C^*_{\text{PM}}$ holds because PM can emulate OTIMAPP execution. Figure 9.10 shows an example where the optimal ratio can be increased arbitrarily. □

The cost of MAPF varies; see Chap. 3.2.2. The most commonly used method is the number of operations (aka. *makespan*). Let C^*_{MAPF} denote the optimal makespan of MAPF. Since MAPF can emulate PM, C^*_{MAPF} never exceeds C^*_{PM}. Furthermore, for solvable PM instances, there exists a polynomial-time procedure [Kornhauser *et al.*, 1984] to obtain a solution that requires $O(V^3)$ operations, where $|V|$ is the number of vertices, concluding as follows.

Proposition 9.14 (optimality: PM vs. MAPF)**.** $1 \leq C^*_{PM}/C^*_{MAPF} \leq O\left(V^3\right)$ *for the same inputs when PM is solvable.*

Proposition 9.15 (optimality: OTIMAPP vs. MAPF)**.** *For any instance in which OTIMAPP is solvable,* $1 \leq C^*_{OTIMAPP}/C^*_{MAPF}$*. The bound is tight. For any* $k \in \mathbb{R}$*, there exists an instance where* $C^*_{OTIMAPP}/C^*_{MAPF} > k$*.*

9.5.3 Complexity

Finally, the computational complexity of the three problems is summarized.

Finding Solutions. Finding a solution to PM or MAPF in *directed* graphs is NP-hard [Nebel, 2020]. Further, OTIMAPP on directed graphs is NP-hard (Thrm. 9.6). In addition, finding a solution to PM or MAPF in *undirected* graphs can be computed in polynomial time. A polynomial-time procedure for solving PM was reported in [Kornhauser *et al.*, 1984; Yu and Rus, 2015]. By contrast, OTIMAPP on undirected graphs is NP-hard (Thrm. 9.7).

Verification. On both directed and undirected graphs, verification of a solution candidate is performed in polynomial time in both PM and MAPF; they belong to NP. In contrast, on both the directed and undirected graphs, the verification of OTIMAPP is co-NP-complete (Thrm. 9.9).

9.6 Solvers

From this section, we shift our focus to solving OTIMAPP.

In practice, using the necessary and sufficient condition (Thrm. 9.5) is challenging because the corresponding schedules must be specified. This motivates us to build a relaxed sufficient condition.

Theorem 9.16 (relaxed sufficient condition)**.** *Given an OTIMAPP instance, a tuple of path* $(\Pi_1, \ldots, \Pi_n)$ *is a solution when there are:*

- *no use of other goals, i.e.,* $g_j \notin \Pi_i$ *for all* $i \neq j$ *except for* $s_i = g_j$*.*

- *no potential cyclic deadlocks.*

Proof. Use Thrm. 9.5; the "no use of other goals" is sufficient for "no reachable terminal deadlocks," whereas "no potential cyclic deadlocks" is sufficient for "no reachable cyclic deadlocks." □

Given a tuple of paths, "no use of other goals" can be easily checked, whereas "no potential cyclic deadlocks" is intractable in computation (Lemma 9.8). Nevertheless, *detecting potential cyclic deadlock is the basis for solving OTIMAPP*. Therefore, this section first explains how to detect potential cyclic deadlocks. Subsequently, two algorithms for solving OTIMAPP are presented.

9.6.1 Detection of Potential Cyclic Deadlocks

First, a notion of *fragment* is introduced, which is a candidate for potential cyclic deadlocks.

Definition 9.17 (fragment)**.** *Given a tuple of paths* $(\Pi_1,\ldots,\Pi_n)$*, a* fragment *is a tuple* $\big((i,j,k,\ldots,l),(t_i,t_j,t_k,\ldots,t_l)\big)$ *such that* $\Pi_i[t_i+1]=\Pi_j[t_j]\wedge\Pi_j[t_j+1]=\Pi_k[t_k]\wedge\ldots=\Pi_l[t_l]$*. The elements of the first tuple are without duplicates.*

A fragment *starts* from a vertex u when $\Pi_i[t_i]=u$ and *ends* at a vertex v when $\Pi_l[t_l+1]=v$. A fragment ending at its start (i.e., $\Pi_l[t_l+1]=\Pi_i[t_i]$) is a potential cyclic deadlock.

Using fragments, Alg. 9.1 detects a potential cyclic deadlock in a tuple of paths, provided it exists. In the pseudocode, fragments are denoted by a pair of two lists: "agents" and (progress) "indexes." '+' operation generates a new list by concatenating elements while maintaining the order, e.g., $i+(j,k)\rightarrow(i,j,k)$, $(i,j)+k\rightarrow(i,j,k)$, and $(i)+j+(k)\rightarrow(i,j,k)$. The intuition for the algorithm is as follows.

1. The algorithm checks each path one by one.
2. All the fragments created thus far are stored.
3. For each edge in each path, the algorithm creates new fragments using the existing fragments.
4. If a fragment ends at its start, this is a potential cyclic deadlock.

The algorithm details are described in the proof of completeness.

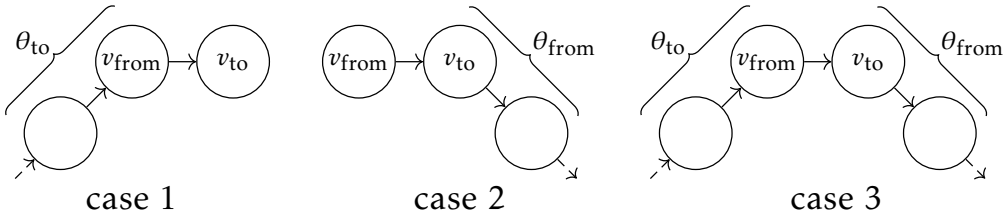


Figure 9.11: Three cases of creating new fragments by extending existing fragments.

Theorem 9.18 (completeness of deadlock detection)**.** *Algorithm 9.1 finds and returns a potential cyclic deadlock if at least one exists; otherwise, it returns* `NOT_FOUND`*.*

Proof. The algorithm uses two tables that store fragments: Θ_{from} and Θ_{to}. Both tables use one vertex as the key. One entry in Θ_{from} stores all the fragments starting from the vertex, while one entry in Θ_{to} stores all the fragments ending at the vertex. A fragment is registered in both tables. The theorem is now derived through induction on Π_i.

Algorithm 9.1 Potential cyclic deadlock detection.

input: tuple of paths $(\Pi_1, \ldots, \Pi_n)$

output: one potential cyclic deadlock or `NOT_FOUND`

1: $\Theta_{\text{from}}, \Theta_{\text{to}} \leftarrow \emptyset$ ▷ *table for fragments, key: vertex*
2: **procedure** REGISTER(θ)
3: $i \leftarrow \theta.\text{agents}[1]$; $j \leftarrow \theta.\text{agents}[-1]$
4: $t_i \leftarrow \theta.\text{indexes}[1]$; $t_j \leftarrow \theta.\text{indexes}[-1]$
5: $\Theta_{\text{from}}[\Pi_i[t_i]].\text{append}(\theta)$
6: $\Theta_{\text{to}}[\Pi_j[t_j + 1]].\text{append}(\theta)$
7: **function** ISDEADLOCK(θ)
8: $i \leftarrow \theta.\text{agents}[1]$; $j \leftarrow \theta.\text{agents}[-1]$
9: $t_i \leftarrow \theta.\text{indexes}[1]$; $t_j \leftarrow \theta.\text{indexes}[-1]$
10: **return** $\Pi_i[t_i] = \Pi_j[t_j + 1]$
11: **for** $i = 1 \ldots n$ **do**
12: **for** $t = 1 \ldots |\Pi_i| - 1$ **do**
13: $v_{\text{from}} \leftarrow \Pi_i[t], v_{\text{to}} \leftarrow \Pi_i[t + 1]$
14: $\theta \leftarrow \{agents : (i), indexes : (t)\}$ ▷ *add a single fragment*
15: REGISTER(θ)
16: **for** $\theta_{\text{to}} \in \Theta_{\text{to}}[v_{\text{from}}]$ **do** ▷ *case 1*
17: **if** $i \in \theta_{\text{to}}.agents$ **then continue**
18: $\theta \leftarrow \{agents : \theta_{\text{to}}.agents + i,\ indexes : \theta_{\text{to}}.indexes + t\}$
19: **if** ISDEADLOCK(θ) **then return** θ
20: REGISTER(θ)
21: **for** $\theta_{\text{from}} \in \Theta_{\text{from}}[v_{\text{to}}]$ **do** ▷ *case 2*
22: **if** $i \in \theta_{\text{from}}.agents$ **then continue**
23: $\theta \leftarrow \{agents : i + \theta_{\text{from}}.agents,\ indexes : t + \theta_{\text{from}}.indexes\}$
24: **if** ISDEADLOCK(θ) **then return** θ
25: REGISTER(θ)
26: **for** $\theta_{\text{to}} \in \Theta_{\text{to}}[v_{\text{from}}], \theta_{\text{from}} \in \Theta_{\text{from}}[v_{\text{to}}]$ **do** ▷ *case 3*
27: **if** $i \in \theta_{\text{to}}.agents \cup \theta_{\text{from}}.agents$ **then continue**
28: **if** $\theta_{\text{to}}.agents \cap \theta_{\text{from}}.agents \neq \emptyset$ **then continue**
29: $\theta \leftarrow \{agents : \theta_{\text{to}}.agents + i + \theta_{\text{from}}.agents,$
$indexes : \theta_{\text{to}}.indexes + t + \theta_{\text{from}}.indexes\}$
30: **if** ISDEADLOCK(θ) **then return** θ
31: REGISTER(θ)
32: **return** `NOT_FOUND`

Base Case. In the first iteration of the loop (Lines 11–31), all fragments for $\{\Pi_1\}$ are registered in Θ_{from} and Θ_{to} because of Lines 14–15. No potential cyclic deadlocks exist for $\{\Pi_1\}$.

Induction Hypothesis. Assume that there are no potential cyclic deadlocks for $\{\Pi_1, \ldots, \Pi_{i-1}\}$, and all their fragments are registered in Θ_{from} and Θ_{to}.

Induction Step. Now, the property for i is derived. Otherwise, a potential cyclic deadlock exists for $\{\Pi_1, \ldots, \Pi_i\}$, and consequently the algorithm returns it. All the new fragments of Π_i are categorized as follows:

- A fragment with only Π_i.
- A fragment that extends other fragments on Θ_{from} and Θ_{to} using $(v_{\text{from}}, v_{\text{to}}) \in \Pi_i$.

The former is preserved because of Lines 14–15. The latter is further categorized into three cases:

1. A fragment that ends at v_{from}.
2. A fragment that starts from v_{to}.
3. A fragment connecting two existing fragments, one that ends at v_{from} and another that starts from v_{to}.

See also Fig. 9.11. Each case corresponds to Lines 16–20, Lines 21–25, and Lines 26–31, respectively. Consequently, all fragments are to register on Θ_{from} and Θ_{to}. Otherwise, a potential cyclic deadlock exists and the algorithm returns it (Lines 19, 24 and 30). □

Table 9.1 provides an example update of Θ_{from} using Alg. 9.1.

induction	key	new fragments
$\{\pi_1\}$	u	$[(1),(1),(u,v)]$
	v	$[(1),(2),(v,w)]$
$\{\pi_1,\pi_2\}$	u	$[(1,2),(1,1),(u,v,x)]$
	v	$[(2),(1),(v,x)]$
	x	$[(2),(2),(x,y)]$
$\{\pi_1,\pi_2,\pi_3\}$	u	$[(1,2,3),(1,1,2),(u,v,x,u)]$
	v	$[(2,3),(1,2),(v,x,u)]$
		$[(2,3,1),(1,2,1),(v,x,u,v)]$
	x	$[(3),(2),(x,u)]$, $[(3,1),(2,1),(x,u,v)]$
		$[(3,1,2),(2,1,1),(x,u,v,x)]$
	z	$[(3),(1),(z,x)]$, $[(3,2),(1,2),(z,x,y)]$

Table 9.1: Example of detecting potential cyclic deadlocks. The update of Θ_{from} for $\Pi_1 = (u,v,w)$, $\Pi_2 = (v,x,y)$, and $\Pi_3 = (z,x,u)$ is described. See also Fig. 9.12. The table uses $[(\text{agents}),(\text{progress indexes}),(\text{path})]$ as a notation of fragment, where "path" is a corresponding sequence of vertices of the fragment. Detected potential cyclic deadlocks are blue-colored. Note that the algorithm halts when it finds one of them.

The time complexity does not contradict the NP-completeness in detecting potential deadlocks (Lemma 9.8).

Observation 9.19 (space and time complexity)**.** *Algorithm 9.1 requires* $\Omega\left(2^{|A|}\right)$ *both for space and time complexity in the worst case.*

Proof. Consider the example in Fig. 9.13. In any solution, the number of fragments starting from u becomes $\Omega\left(2^{|A|}\right)$, implying the statement. □

Although Alg. 9.1 does not run in polynomial time, it works sufficiently fast in a sparse environment such that not many paths use the same vertices.

Next, we see two algorithms that use Alg. 9.1 as a sub-procedure to solve OTIMAPP.

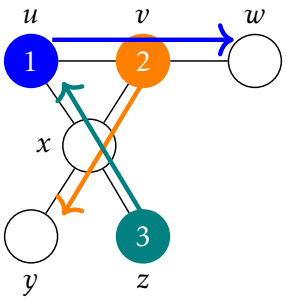


Figure 9.12: Situation of Table 9.1.

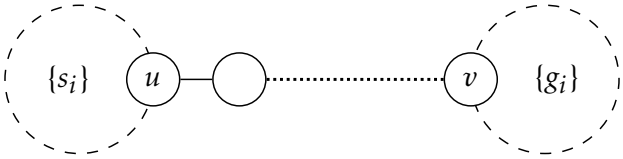


Figure 9.13: Example that requires huge space and time to detect potential deadlocks. All starts are on the left. All goals are on the right. Two zones are connected by a sufficiently long path whose length exceeds $|A|$.

9.6.2 Prioritized Planning (PP)

As appeared several times so far, prioritized planning (PP) [Erdmann and Lozano-Perez, 1987] is neither complete nor optimal. However, it is computationally inexpensive, hence, it is a popular approach to MAPF. PP plans paths sequentially while avoiding collisions with the previously planned paths. Instead of inter-agent collisions, solvers for OTIMAPP must consider potential cyclic deadlocks.

Algorithm 9.2 is PP for OTIMAPP. When planning a single-agent path, PP avoids using *(i)* the goals of other agents and *(ii)* edges that trigger potential cyclic deadlocks (Line 3). The latter is detected by storing all fragments created by the previously computed paths. For this purpose, PP uses the adaptive version of Alg. 9.1. A path that satisfies these constraints can be found using ordinary pathfinding algorithms. If not, PP returns FAILURE. The correctness of PP is derived from the relaxed sufficient condition (Thrm. 9.16).

Algorithm 9.2 PP for OTIMAPP.

input: OTIMAPP instance

output: solution $(\Pi_1, \ldots, \Pi_n)$ or FAILURE

1: $\Theta_{\text{from}}, \Theta_{\text{to}} \leftarrow \emptyset$
2: **for** $i = 1 \ldots |A|$ **do**
3: $\Pi_i \leftarrow$ a path for agent i while avoiding the use of
- $g_j, \forall j \neq i$, except for s_i
- $(u, v) \in E$ s.t. $\exists \theta \in \Theta_{\text{to}}[u]$ and θ starts from v

▷ avoiding cyclic deadlocks for $\{\Pi_j\}_{j<i}$
4: **if** Π_i is not found **then return** FAILURE
5: update Θ_{from} and Θ_{to} with Π_i using Alg. 9.1
6: **return** $(\Pi_1, \ldots, \Pi_N)$

PP is simple albeit incomplete. In particular, the planning order of agents is crucial; an instance may or may not be solved, as illustrated in Fig. 9.14. One resolution involves the repetition of PP with random priorities until the problem is solved. Let's call this PP$^+$. However, finding good orders can be challenging because there are $|A|!$ patterns. This motivates us to develop a search-based solver as described in the next.

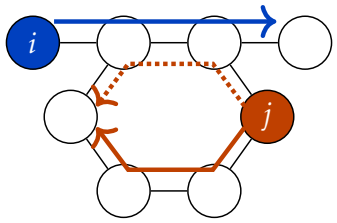


Figure 9.14: Example instance that the planning order affects the solvability. When i plans prior to j, PP results in success with solid lines. PP fails if j plans first and takes the dotted line.

9.6.3 Deadlock-based Search (DBS)

Next, *deadlock-based search (DBS)* to solve OTIMAPP is presented, based on a popular MAPF solver, CBS [Sharon *et al.*, 2015]. CBS uses a two-level search. A high-level search manages collisions between agents. When a collision occurs between two agents at a certain time and location, there are two possible resolutions depending on which agent gets to use the location at that time. Following this observation, CBS constructs a binary tree where each node includes constraints prohibiting the use of space-time pairs for certain agents. In a low-level search, agents find a single path constrained by the corresponding high-level node.

Instead of collisions, DBS considers potential cyclic deadlocks. When detecting a deadlock in a tuple of paths, one of the agents in the deadlock avoids using the edge. Thus, the constraints identify which agents prohibit using which edges in which orientations.

Algorithm 9.3 describes the high-level search of DBS. Each node in the high-level search contains *constraints*, a list of tuples comprising one agent and two vertices (representing "from vertex" and "to vertex"), and *paths* as a solution candidate. The root node has no constraints (Line 1). Its paths are computed following "no use of other goals" in Thrm. 9.16 (Line 2). The node is then inserted into a priority queue *Open* (Line 3). In the main loop (Lines 4–11), DBS repeats;

1. Selecting one node (Line 5).
2. Checking a deadlock and creating constraints (Line 6).
3. Returning a solution if the paths contain no deadlocks (Line 7).
4. If not, creating successors and inserting them into *Open* (Lines 8–11).

DBS returns `NOT_FOUND` when *Open* becomes empty (Line 12). Several complementary details are provided below.

- Line 5: *Open* is a priority queue and needs the order of nodes. Although DBS works in any order, good orders reduce the search effort. For effective heuristics, our implementation uses the descending order of the number of deadlocks with two agents, which is computed within a reasonable time.

Algorithm 9.3 Deadlock-based search (DBS) for OTIMAPP.

input: OTIMAPP instance

output: solution $(\Pi_1, \ldots, \Pi_n)$ or `NOT_FOUND`

1: $\mathcal{N}^{\text{init}}.constraints \leftarrow \emptyset$
2: $\mathcal{N}^{\text{init}}.paths \leftarrow$ find paths with "no use of other goals"
3: $Open.\text{push}\left(\mathcal{N}^{\text{init}}\right)$ ▷ *Open: priority queue*
4: **while** $Open \neq \emptyset$ **do**
5: $\quad \mathcal{N} \leftarrow Open.\text{pop}()$
6: $\quad C \leftarrow$ get constraints of $\mathcal{N}$ using Alg. 9.1
7: $\quad$ **if** $C = \emptyset$ **then return** $\mathcal{N}.paths$
8: $\quad$ **for** $(i, u, v) \in C$ **do**
9: $\quad\quad \mathcal{N}^{\text{new}} \leftarrow \{constraints : \mathcal{N}.constraints + (i, u, v),\ paths : \mathcal{N}.paths\}$
10: $\quad\quad$ update Π_i in $\mathcal{N}^{\text{new}}.paths$ to follow $\mathcal{N}^{\text{new}}.constraints$
11: $\quad\quad$ **if** Π_i is found **then** $Open.\text{push}(\mathcal{N}^{\text{new}})$
12: **return** `NOT_FOUND`

- Line 6: Let $\left((i, j, k, \ldots, l), (t_i, t_j, t_k, \ldots, t_l)\right)$ be the deadlock returned by Alg. 9.1. Then, constraints $(i, \Pi_i[t_i], \Pi_i[t_i+1]), (j, \Pi_j[t_j], \Pi_j[t_j+1]), \ldots, (l, \Pi_l[t_l], \Pi_l[t_l+1])$ are created.
- Line 10 forces one path Π_i in the node to follow the new constraints. This low-level search must follow "no use of other goals," furthermore, all edges in the constraints for i. If not found, DBS discards the corresponding successor.

Theorem 9.20 (DBS)**.** *DBS returns a solution when solutions satisfying Thrm. 9.16 exist; otherwise, it returns* `NOT_FOUND`*.*

Proof. Assume that there is a solution $\boldsymbol{\Pi}$ that satisfies the relaxed sufficient condition (Thrm. 9.16). At each cycle of Lines 4–11, at least one node in *Open* is *consistent* with $\boldsymbol{\Pi}$, i.e., its constraints allow searching $\boldsymbol{\Pi}$. This is derived from the following induction: *(i)* the initial node $\mathcal{N}^{\text{init}}$ is consistent with $\boldsymbol{\Pi}$, and *(ii)* the nodes generated from a consistent node with $\boldsymbol{\Pi}$ must include at least one consistent node. The search space, i.e., which agents are prohibited from using which edges in which directions, is finite. Therefore, DBS eventually returns $\boldsymbol{\Pi}$ (or another solution); otherwise, no such solutions exist. □

Example

An example of DBS is described using Fig. 9.14. Assume that the initial path of i is the solid blue line and the path for j is the dashed red line (Line 2). This node is inserted into *Open* (Line 3) and is expanded immediately (Line 5). There is one potential cyclic deadlock in the paths. Consequently, two constraints are created: either i or j avoids using the shared edge (Line 10). Two child nodes are generated; however, the node that changes i's path is invalid because there is no such path without using the goal of j. The other is valid: j takes the solid red line. Therefore, a node is added to *Open* from the root node. In the next iteration, the newly added node is expanded. There are no potential cyclic deadlocks at this node. Thus, DBS returns its paths as a solution.

Optimization

Although the chapter focuses on a decision problem, DBS can apply to optimization problems. The total and maximum path lengths in a solution can be defined as the objective functions (i.e., sum-of-fuels and maximum-moves in MAPF). These optimization problems can be optimally solved using DBS when it prioritizes high-level search nodes with smaller scores, as is commonly performed in CBS. Note that metrics that assess time aspects, such as the total traveling time used in MAPF studies, are significantly affected by execution schedules. Thus, adaptation is not trivial.

9.6.4 PP vs. DBS

DBS has the theoretical guarantee of finding solutions (Thrm. 9.20) while PP does not. Indeed, there are instances solvable for DBS but unsolvable for PP even with any planning order. Figure 9.15a shows such an example. Observe first that this instance has a solution satisfying the sufficient condition (Thrm. 9.16), as shown in Fig. 9.15b. DBS eventually returns it.

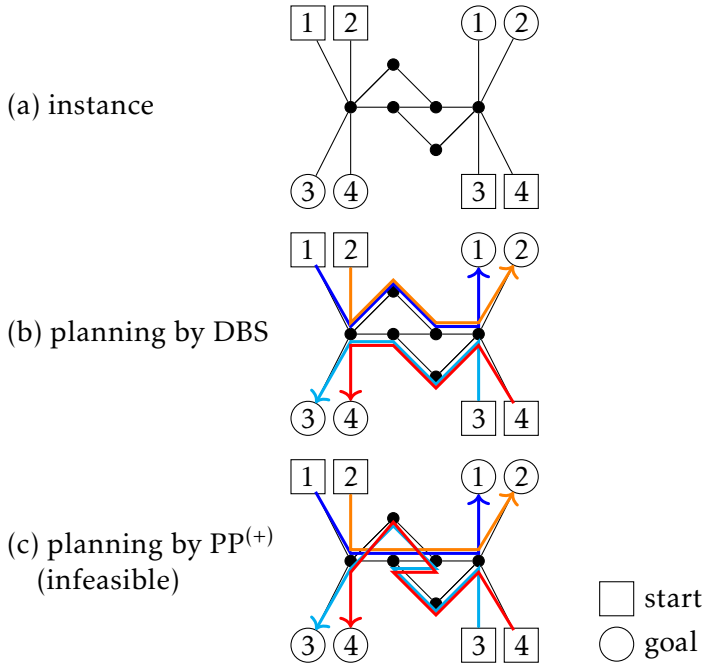


Figure 9.15: Instance solvable for DBS but unsolvable for PP$^{(+)}$.

In contrast, PP$^{(+)}$ fails to solve the instance. Suppose that the single-agent pathfinding in PP prefers to use the middle two vertices of the instance, as illustrated in the path of agent-1 in Fig. 9.15c. Consider the planning order of $(1, 2, 3, 4)$. PP assigns paths with the middle two vertices (blue and orange) to agents 1 and 2. Then, it assigns a path shown in Fig. 9.15c to agent 3 (7 steps; cyan), reflecting avoiding cyclic deadlocks with agents 1 and 2. Now, agent 4 has no path without cyclic deadlocks. For instance, consider the red path in Fig. 9.15c. Then, there are reachable cyclic deadlocks for the combination of these paths, e.g., $((1, 2, 3, 4), (4, 3, 3, 2))$ using the notation of Def. 9.1. Due to the symmetry of the instance, regardless of planning orders, the last planning agent has always no path without deadlocks.

As a technical point, the assumption that single-agent pathfinding prefers to use the middle two vertices follows the typical implementation of PP for MAPF which plans the shortest paths for each agent. Note, the path length of Fig. 9.15b (say, path-b) and Fig. 9.15c (path-c) of agents $\{1, 2\}$ do not differ. However, the instance can introduce

auxiliary vertices and agents to make the path-b longer than path-c while maintaining the structure above. Figure 9.15 just presents a minimum example.

Although PP has no guarantee of finding solutions, in general, the planning burden tends to be smaller compared to that of DBS. In other words, PP is faster than DBS. This is because PP seeks solutions in a *decoupled* search space that does not consider the joint actions of multiple agents. The above discussion corresponds to the discussion of PP vs. CBS in the MAPF literature, i.e., PP is faster than CBS in general while compromising the guarantee of finding solutions. We will later see empirical results that justify this trend.

9.7 Relaxation of Feasibility

OTIMAPP is unfortunately computationally intractable (Thrm. 9.6 and 9.7). Moreover, detecting potential cyclic deadlocks itself, which is a core of solving OTIMAPP, is computationally intractable (Lemma 9.8). Therefore, one realistic approach in large problem instances is to relax the solution concept of OTIMAPP. More precisely, it is practical to find a set of paths that is unlikely to trigger something bad (i.e., deadlock). Following this perspective, we introduce a relaxed solution concept as follows.

Definition 9.21 (m-tolerant solution)**.** *A tuple of paths is an* m-tolerant *solution when*

- *No reachable terminal deadlocks.*
- *No reachable cyclic deadlocks with m agents or fewer.*

This motivation stems from the fact that reachable deadlocks with many agents rarely occur. For instance, in grids, deadlocks with more than eight agents are unlikely to occur with schedules generated uniformly at random (see Chap. 9.8.3). It should be noted that when a tuple of paths is $|A|$-tolerant, it is a solution to OTIMAPP.

To find m-tolerant solutions, a procedure for detecting potential cyclic deadlocks of up to m agents is required. This is constructed directly from Alg. 9.1; abandon all fragments with m agents or more, unless they are potential cyclic deadlocks. In addition, a fragment whose first vertex is m' steps apart from its last vertex can be discarded, without using the vertices in the fragment, when the number of agents in the fragment plus m' exceeds m. This fragment never produces potential cyclic deadlocks with m or fewer agents. Since stored fragments are dramatically reduced, which is the bottleneck for detecting potential cyclic deadlocks, a significant reduction in the computational burden is expected for both PP and DBS introduced in Chap. 9.6.

Unfortunately, the complexity of finding m-tolerant solutions remains intractable.

Theorem 9.22 (m-tolerant solutions)**.** *Finding 2-tolerant solutions is NP-hard.*

Proof. The proof of NP-hardness on undirected graphs (Thrm. 9.7) already restricts deadlocks with three agents (i.e., 3-tolerant solutions). Here, the proof further replaces the clause constrainer as Fig. 9.16 to restrict 2-tolerant solutions. This new gadget cannot be used simultaneously by the three agents. Otherwise, agent c_2^j has a reachable cyclic deadlock with either c_1^j or c_3^j (at red colored edges), or if either c_1^j or c_3^j meet terminal deadlocks with c_2^j. The translation is still in polynomial time. □

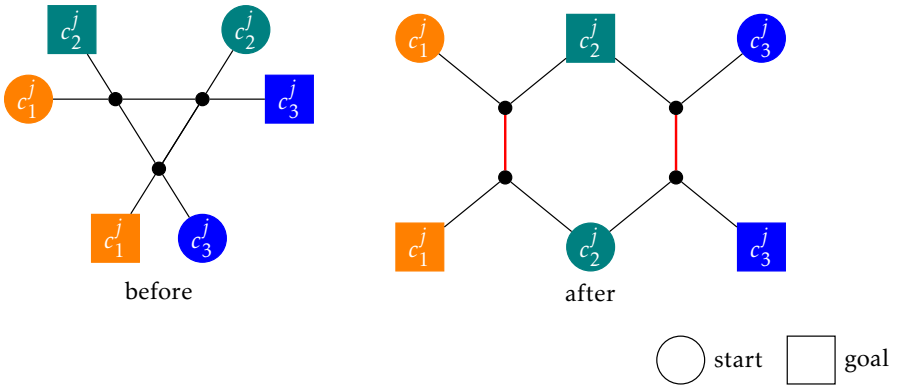


Figure 9.16: Equivalent gadget of the *clause constrainer* for a clause C^j.

9.8 Evaluation

This section empirically demonstrates that OTIMAPP solutions are computable to a certain extent (Chap. 9.8.1), and they are useful in adverse environments regarding timings (Chap. 9.8.2) through simulation experiments. This section also presents how m-tolerant solutions relax computational effort while incurring the risk of execution failure (Chap. 9.8.3), as well as OTIMAPP execution with mobile robots (Chap. 9.8.4). The code was written in C++, and the experiments were run on a desktop PC with Intel Core i9 2.8 GHz CPU and 64 GB RAM.

9.8.1 Stress Test

Setup. Each solver was tested with a timeout of 5 min on four-connected undirected grids picked up from [Stern *et al.*, 2019] as a graph G. In addition, the random graphs were tested. All instances were generated by setting random start s_i and goal g_i, while ensuring that s_i and g_i have at least one path without the use of other goals; otherwise, it violates the "no use of other goals" condition of Thrm. 9.16. However, unsolvable instances may still be included.

Result. Figures 9.17 and 9.18 present the results. Since DBS detects unsolvable instances regarding the relaxed sufficient condition of Thrm. 9.16, the figures additionally show the corresponding scores for the sum of the numbers of solved instances and detected unsolvable instances. The corresponding scores are marked as DBS*. The main findings of the results are as follows.

- Both solvers can solve instances with tens of agents in various maps within a reasonable timeframe. The scalability of DBS is partially due to focusing on decision problems rather than optimization problems, unlike usual CBS studies in MAPF.
- PP frequently fails because of priority orders (e.g., Fig. 9.14), whereas PP$^+$ and DBS can overcome such limitations to some extent. Recall that PP$^+$ repeats PP with random order until finding solutions or reaching the timeout.
- The bottleneck of each solver is the procedure for detecting potential cyclic deadlocks, an NP-hard problem (Lemma 9.8). This also led to similar success rates for PP$^+$ and DBS.
- The bottom of Fig. 9.17 displays how many random attempts in PP$^+$ were done for the solved instances. PP$^+$ solved instances with small numbers of the trials (at most 17 in the displayed results); otherwise, it failed.

- In experiments on random graphs, it becomes easier to find solutions as the edge connection probability p increases. This is attributed to an increase in the average degree of the graphs and a decrease in their diameter; both factors contribute reasonably to finding deadlock-free paths.
- As seen in runtime results of Fig. 9.17, PP$^{(+)}$ has the speed advantage over DBS, i.e., finding solutions with smaller computational burdens compared to those of DBS. Meanwhile, PP^{+} misses the detection of unsolvable instances. Indeed, in Fig. 9.18, there were many unsolvable instances detected by DBS (see differences between DBS and DBS*), while PP^{+} did not tell anything and just reached the timeout. These observations are aligned with the discussion in Chap. 9.6.4.

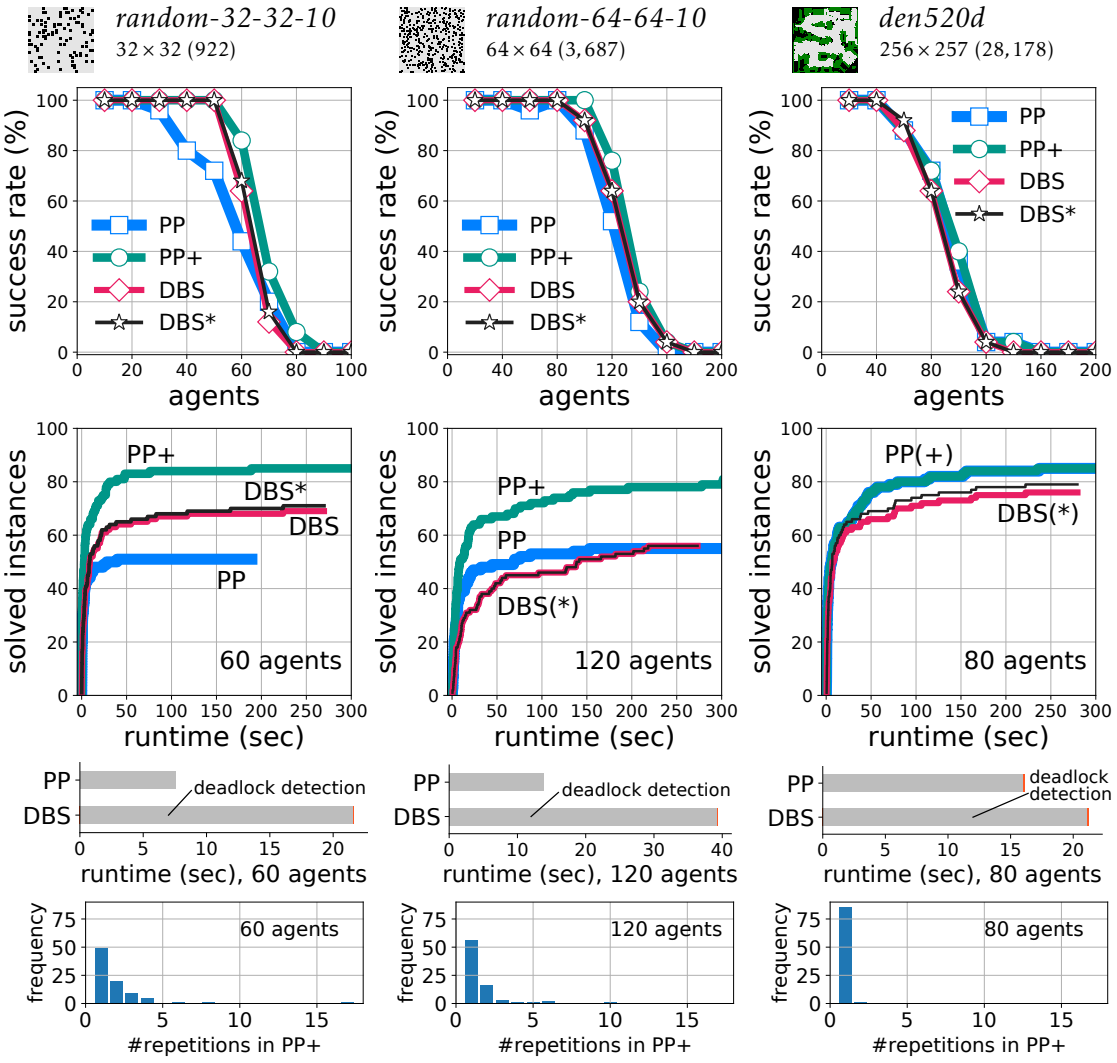


Figure 9.17: Stress test on four-connected grids. The success rate is based on 25 identical instances. DBS* includes detected instances that are unsolvable for DBS before timeout, which is not possible for PP$^{(+)}$. The figure also presents accumulated runtime with a fixed number of agents over 100 instances, and runtime profiling (median) of each solver over success instances for both solvers. At the bottom, histograms of the numbers of random attempts in PP^{+} are additionally reported for solved instances. The same instances for runtime profiling were used.

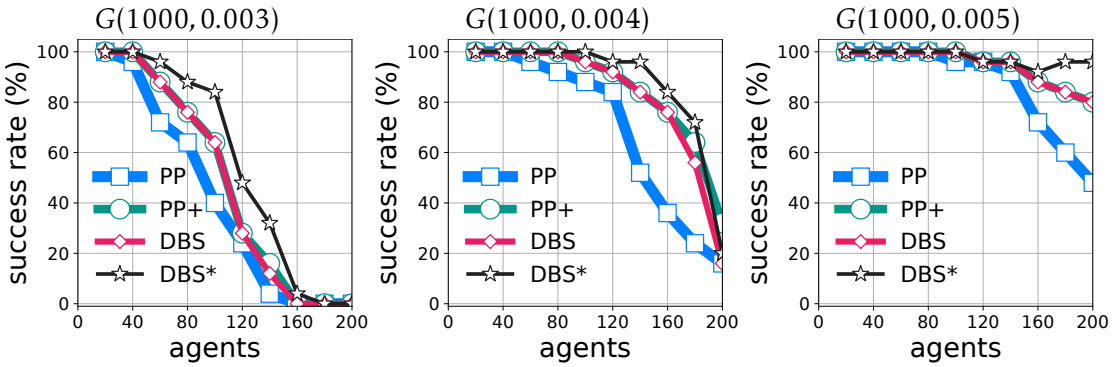


Figure 9.18: Stress test on random graphs. The success rate is based on 25 identical instances. Results on random graphs $G(n, p)$ are shown, where n is the number of vertices and every possible edge occurs independently with probability p.

Solvability of OTIMAPP vs. MAPF. Recall that OTIMAPP and MAPF have the same input structure. To see the difference in the difficulty of solving instances, we applied an MAPF algorithm to all grid instances above. Specifically, a state-of-the-art sub-optimal MAPF algorithm PIBT$^+$ [Okumura *et al.*, 2022b] solved all instances at most within 300 ms. The solved instances by PIBT$^+$ for MAPF include all detected unsolvable instances by DBS for OTIMAPP. This result highlights the difficulty of solving OTIMAPP; filling the gap of both algorithmic speed and solvability between MAPF and OTIMAPP is one primary future challenge.

9.8.2 Delay Tolerance

Next, it is demonstrated that OTIMAPP solutions (if found) are useful in a simulated environment with stochastic delays of agent moves built on conventional MAPF, called MAPF-DP (with delay probabilities) [Ma *et al.*, 2017a]. MAPF-DP emulates the imperfect execution of MAPF by introducing the possibility p_i of unsuccessful moves to agent i (remaining there).

Setup. The delay probabilities p_i were chosen uniformly at random from $[0, \bar{p}]$, where $\bar{p}$ is the upper bound of p_i. A higher $\bar{p}$ means that agents frequently delay and vice versa. The metric is the total traveling time of the agents; smaller values indicate less wasted time at runtime. The following two baselines were also tested.

- **MCPs** [Ma *et al.*, 2017a] force agents to preserve the order relations of visiting one vertex in an offline MAPF plan at the runtime. The plan was obtained using ECBS [Barer *et al.*, 2014], a bounded sub-optimal version of the CBS algorithm. The sub-optimality was set to 1.05 to obtain plausible solutions in a short time.
- **Causal-PIBT** [Okumura *et al.*, 2021b] is an online time-independent planning method, that is, each agent repeats one-step planning and action adaptively to the surrounding current situations.

Result. Table 9.2 reveals that the execution of OTIMAPP solutions outperforms the alternatives when there are delays in agents' motions. This is because:

- Unlike MCPs, OTIMAPP solutions are free from the temporal dependencies of offline plans in which one-agent delays are possibly fatal.

- Unlike Causal-PIBT, agents follow long-term plans and avoid possible congested locations, which is a positive side effect of avoiding deadlocks in OTIMAPP solutions.

Note however that, without delays (i.e., $\bar{p} = 0$), MCPs scored better than OTIMAPP solutions. This is because agents in MCPs can follow "optimized" offline planning precisely, provided by ECBS.

Discussion. Although finding OTIMAPP solutions is challenging, Table 9.2 motivates us to compute them. Meanwhile, other approaches can solve larger instances with more agents (e.g., $|A| = 200$) and with a much shorter planning time than solving OTIMAPP. Moreover, there are situations where OTIMAPP has no solutions, whereas the others can find feasible plans because OTIMAPP assumes no intervention at runtime, as discussed in Chap. 9.5.1. In association with this discussion of solvability, we additionally display the success rate of solving MAPF-DP by each approach in Table 9.3. With denser situations (e.g., $|A| = 60$), the OTIMAPP algorithm (i.e., PP$^+$) often failed to find solutions whereas the other approaches solved all. One promising direction for OTIMAPP is to fill these gaps.

fixing $|A|$

$\|A\| = 35$	$\bar{p} = 0.2$		$\bar{p} = 0.5$		$\bar{p} = 0.8$	
MCPs+ECBS	1015	(1004,1026)	1422	(1404,1440)	2551	(2507,2596)
Causal-PIBT	986	(976,995)	1238	(1225,1250)	1841	(1816,1866)
OTIMAPP	**941**	(931,951)	**1178**	(1165,1190)	**1730**	(1707,1752)

fixing $\bar{p}$

$\bar{p} = 0.5$	$\|A\| = 20$		$\|A\| = 40$		$\|A\| = 60$	
MCPs+ECBS	724	(711,736)	1698	(1678,1716)	2938	(2911,2964)
Causal-PIBT	662	(653,671)	1466	(1453,1479)	2425	(2405,2444)
OTIMAPP	**639**	(631,648)	**1395**	(1383,1408)	**2328**	(2311,2345)
$\bar{p} = 0.0$	$\|A\| = 20$		$\|A\| = 40$		$\|A\| = 60$	
MCPs+ECBS	**449**	(408,489)	**934**	(891,983)	**1438**	(1385,1489)
Causal-PIBT	**472**	(467,478)	1042	(1033,1050)	1725	(1713,1738)
OTIMAPP	**458**	(452,464)	993	(985,1001)	1628	(1617,1638)

Table 9.2: Total traveling time on MAPF-DP. All settings used *random-32-32-10*. For each setting, ten instances that OTIMAPP solutions were found by PP$^+$ were first picked up. For each instance and approach, then 50 trials were performed while changing the random seed. For all approaches, all execution trials succeeded. Thus, the scores are means on 500 executions, accompanied by 95% confidence intervals. *upper*: Results of changing $\bar{p}$ while fixing $|A|$. *lower*: Results of changing $|A|$ while fixing $\bar{p}$. Note that the probability that someone delays increases with more agents. As reference records, the table also presents scores without delays, i.e., $\bar{p} = 0$.

9.8.3 m-tolerant Solutions

Recall that a tuple of paths is m-tolerant when there are no reachable cyclic deadlocks with m or fewer agents. Next, the study empirically evaluates how computational effort is relaxed by introducing m-tolerant solutions, as well as the risk of execution failure.

$\bar{p} = 0.5$	$\|A\| = 20$	$\|A\| = 40$	$\|A\| = 60$
MCPs+ECBS	**1.00**	**1.00**	**1.00**
Causal-PIBT	**1.00**	**1.00**	**1.00**
OTIMAPP (PP$^+$)	**1.00**	0.80	0.44

Table 9.3: Success rate of solving MAPF-DP. For each $|A|$, we used 25 instances; they are same as instances in Fig. 9.17. OTIMAPP and MCPs never fail in the execution phase if offline solutions are obtained, therefore, the scores presented here are equivalent to the planning success rate. The scores of Causal-PIBT were calculated from 50 trials for each instance (i.e., 1,250 executions).

Setup. Both PP$^+$ and DBS were used in the same experimental setting as in Chap. 9.8.1. Further, for each successful planning outcome, the outcome was simulated with randomly generated 100 execution schedules, and then the number of executions that triggered actual deadlocks was counted. An execution was regarded as a failure when it triggered deadlocks because several agents never reach their destinations eternally. In this way, the execution failure rate was calculated.

Result. Figure 9.19 summarizes the result. The results emphasize a tradeoff between the computational burden and the risk of execution failure. In both PP$^+$ and DBS, m-tolerant solutions are easier to compute than exact $|A|$-tolerant solutions, particularly when m is sufficiently small (e.g., ≤ 6). On the other hand, with a smaller m (e.g., two), the risk of execution failure increases. In practice, m-tolerant solutions are useful because OTIMAPP is computationally difficult; however, the parameter m should be adjusted considering the risk of execution failure.

9.8.4 Robot Demonstrations

Finally, the section presents two OTIMAPP execution demonstrations with centralized and decentralized mobile robots. The video is available at `https://kei18.github.io/otimapp/`. Figure 9.20 shows snapshots. The OTIMAPP solution was prepared using DBS.

In both cases, although the robots moved without any synchronization procedures, they were ensured to eventually reach their goals owing to the nature of OTIMAPP. Moreover, for the latter, any actor had no methods to know the entire configuration at runtime, which cannot be addressed by conventional execution strategies. The implementation details are as follows.

Centralized Execution

Toio robots (`https://toio.io/`) were used, similar to demonstrations in Chap. 8.4. The robots evolve on a specific playmat and can be controlled by instructions of absolute coordinates. A virtual grid was prepared on the playmat and the robots followed the grid. Prior to the demonstration, it was informally confirmed that there is a non-negligible action delay between robots when simultaneously sending instructions to several robots (e.g., ten robots, see the movie). Therefore, one-shot execution — robots move alone without communication after the receipt of plans — will result in collisions and risk of execution failure. In this demonstration, a central server (laptop) managed the locations of all robots and issued instructions (i.e., where to go) to each robot step by step. The instructions were issued *synchrony* between robots while avoiding collisions.

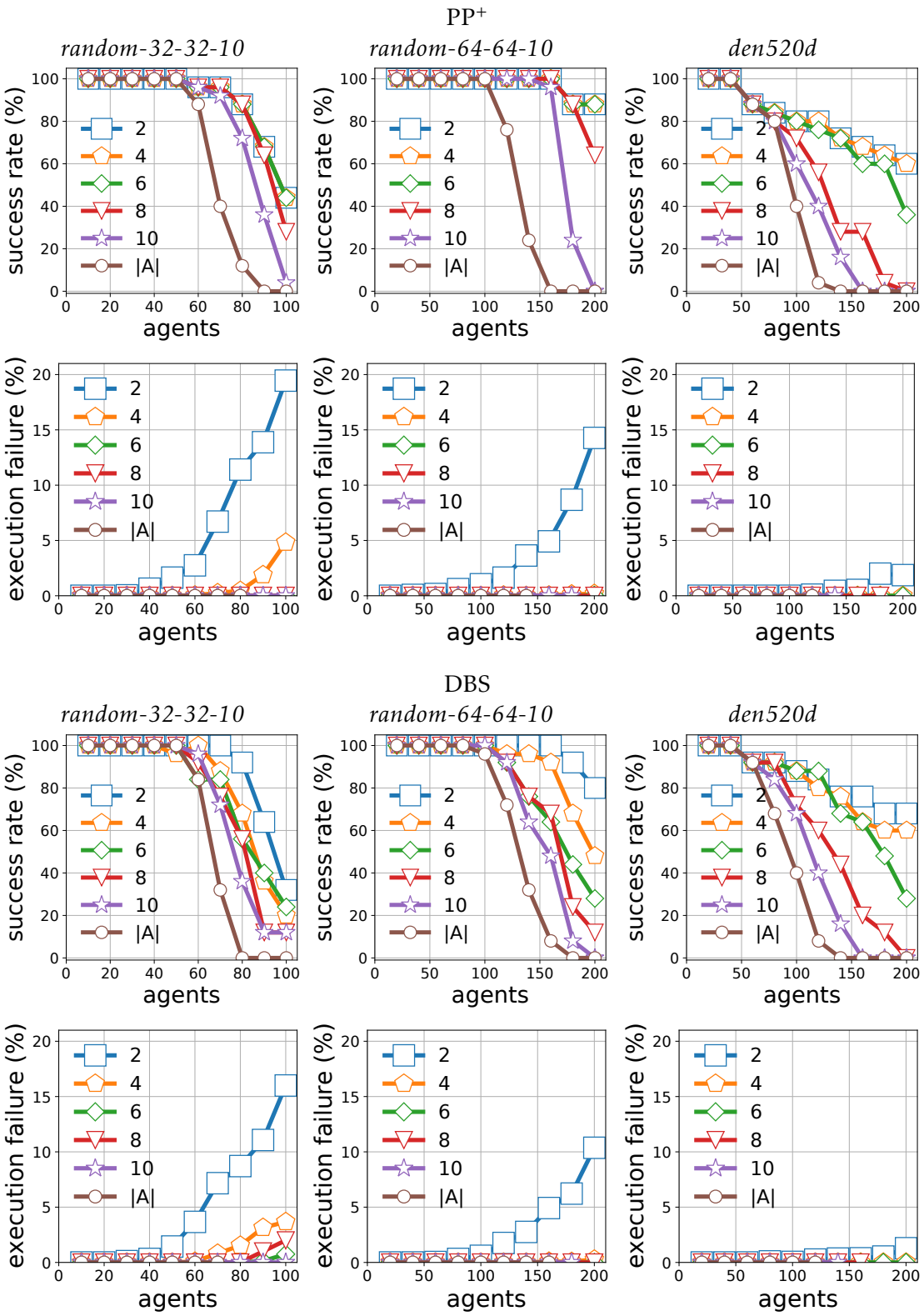


Figure 9.19: Results of *m*-tolerant solutions. The planning success rate (upper) is based on the same 25 identical instances as Fig. 9.17. The time limit was set to 5 min. The execution failure rate (lower) is based on 100 execution for each successful plan.

Decentralized Execution

The *AFADA* platform [Kameyama *et al.*, 2021] was used. It has an architecture comprising mobile robots that evolve over an active environment made of flat *cells*, each equipped

Figure 9.20: OTIMAPP execution with 10 **robots in an** 8×8 **grid.** Colored arrows represent an OTIMAPP solution.

with a computing unit. Adjacent cells can communicate with each other via a serial interface. Further, cells form the environment in two ways: as a two-dimensional physical grid and as a communication network. In addition, a cell can communicate with robots on it via near-field communication (NFC). Using these local communication schemes, we implemented collision avoidance only with local interactions between actors. Each robot spontaneously acted; hence, the system was fully asynchronous, and no actor knew the entire configuration at runtime.

9.9 Related Work

Before concluding the chapter, this section summarizes related studies to OTIMAPP.

9.9.1 Deadlock

A *deadlock* [Coffman *et al.*, 1971] is a widely recognized phenomenon that is not limited to robotics. It is a system state wherein several components claim resources held by others and then block each other permanently. Strategies to cope with deadlocks are categorized as prevention, detection/recovery, and avoidance [Silberschatz *et al.*, 2006; Fanti and Zhou, 2004]. Deadlock prevention prevents deadlock situations by constraining how the requests for resources can be made to suppress one of the known conditions necessary for deadlock [Silberschatz *et al.*, 2006]. Deadlock detection/recovery examines the system state at runtime to detect when a deadlock occurs and, if found, corrects it by applying predefined procedures. Deadlock avoidance prevents the occurrence of deadlocks by avoiding risky states through on-demand interventions such as in the Banker's algorithm [Silberschatz *et al.*, 2006]. *OTIMAPP is based on prevention*;[3] it aims to ensure deadlock-free status at runtime by determining which agent visits which resources in which order (i.e., path) prior to execution. A non-deadlock state from which reaching deadlocks is "inevitable" is referred to as *unsafe* [Silberschatz *et al.*, 2006]. Meanwhile, reachable deadlocks of OTIMAPP correspond to states from which reaching deadlocks may be "possible." The notion of a potential terminal deadlock is related to well-formed instances of MAPF [Čáp *et al.*, 2015], that is, for each start-goal pair, a path exists that traverses no other starts and goals. The relaxed sufficient condition of OTIMAPP (Thrm. 9.16) requires that each agent has at least one path without using the goals of the others. The notion of a reachable cyclic deadlock is referred to as nonlive states/sets for deadlock management in automated manufacturing systems [Fanti and Zhou, 2004] or a multi-robot scheduling problem [Mannucci *et al.*, 2021].

[3] OTIMAPP essentially aims at removing the possibility of *cyclic waiting*, one of the four necessary conditions for deadlocks to happen [Silberschatz *et al.*, 2006].

9.9.2 Path Planning for Multiple Agents

Path planning for multiple robots has been studied extensively. These approaches are typically categorized as *reactive* or *deliberative* approaches, as repeatedly stated throughout the dissertation.

In reactive approaches, as we can see in [Van Den Berg *et al.*, 2011; Lalish and Morgansen, 2012; Senbaslar *et al.*, 2018], robots continuously react to situations at runtime to avoid collisions while heading to their destination. This class is computationally inexpensive; however, deadlock-free systems are difficult to realize owing to the shortsightedness of time evolution. Moreover, reactive approaches require rich and no-delay observations, such as accurate positions and velocities of the surrounding robots for each robot. Thus, implementing this in highly distributed environments may pose non-trivial challenges. In contrast, OTIMAPP execution assumes only the mutual exclusion of locations. This requirement is expected to be much easier to implement than the observation assumptions of reactive approaches.

Deliberative approaches use longer planning horizons to plan collision-free trajectories. This problem is formulated as MAPF or MRMP. In a typical MAPF, the inputs are a graph and a set of start-goal pairs for agents. The objective is to find a list of "timed" paths because MAPF assumes that all agents act synchronously. Both optimal and suboptimal algorithms for MAPF have been extensively studied (see Chap. 3.2.4), although these methods rely heavily on timing assumptions and are fragile to action delays in robot execution at runtime. Therefore, many studies on MAPF consider timing uncertainties. However, current methods still largely rely on additional assumptions on the travel speed of agents or delays to follow certain probability distributions (see Chap. 3.5.2). Failure to represent the inherent uncertainty in the domain means that the system behavior can be unpredictable. In contrast, OTIMAPP can tolerate any type of action delay owing to disclaiming any timing assumptions.

OTIMAPP contains both reactive and deliberative faces. It is reactive because it relies on (online) collision avoidance, which is assumed to be performed by each agent. It is deliberative because it plans the entire trajectories prior to execution. Combining these two properties, OTIMAPP provides a unique concept for achieving multi-robot coordination.

Alternative approaches include a combination of offline deliberative approaches and online intervention during execution, for example, forcing agents to preserve the temporal dependencies of offline planning [Ma *et al.*, 2017a; Hönig *et al.*, 2019; Atzmon *et al.*, 2020b] or continuously synthesize deadlock-free scheduling [O'Donnell and Lozano-Pérez, 1989; Čáp *et al.*, 2016; Coskun and O'Kane, 2019; Mannucci *et al.*, 2021]. However, these approaches suffer from inherent limitations of centralized execution, e.g., requiring costly runtime effort and additional infrastructure (e.g., steady networks and global monitoring systems) to continuously manage the status of all robots. In contrast, OITMAPP does not require such facilities. Once a solution is obtained, it is ensured that all robots reach their destinations by following their respective paths while avoiding collisions locally.

The notion of time independence was originally considered from [Okumura *et al.*, 2021b], being polished in Chap. 8 (and [Okumura and Défago, 2022b]). Time-independent planning represents the entire system with multiple agents on graphs as a transition system. The study presents online planning that incrementally moves agents while resolving deadlocks on demand. In contrast, OTIMAPP is offline planning aimed at without or with less runtime effort.

In graph theory, the (vertex) disjoint path problem and its variants [Robertson and Seymour, 1985] are partly related to ours in the sense that a set of disjoint paths clearly satisfies the solution condition of OTIMAPP; however, the reverse does not.

9.10 Concluding Remarks

This chapter studied a novel path planning problem called OTIMAPP, motivated by the timing uncertainties critical for plan execution on real robots. OTIMAPP is an offline planning problem that considers every possible schedule of agent behaviors at runtime. The chapter presented both theoretical and practical aspects, including the solution condition, computational complexities, solvers, and the relaxed solution concept.

OTIMAPP exemplifies execution methods such that offline planning guides reactive execution. In the next chapter, we see another example that considers *crash faults*.

9.10.1 Interesting Directions

Some interesting directions for the development of OTIMAPP are discussed below.

- **Variants of OTIMAPP**: For instance, an unlabeled version of OTIMAPP, wherein agents can achieve one of the goals while ensuring all goals are eventually occupied by agents, is helpful for robotic pattern formation. The previous chapter introduced an online time-independent planning method for the unlabeled problem; however, offline planning remains missing.
- **Continuous spaces**: This chapter studied discretized environments but extending the work to continuous spaces has practical values. For this direction, definitions of potential/reachable deadlocks in continuous spaces should be elaborated like [Gregoire *et al.*, 2013].
- **Enhancing each solver**: This paper presented two basic solvers based on MAPF studies, which is a very active research field. Using the state-of-the-art MAPF techniques such as LaCAM, both solvers are expected to be more powerful.
- **Applications to other multi-agent planning domains**: Since OTIMAPP, after all, asks for a sequential resource allocation problem, it is interesting to leverage OTIMAPP to other resource allocation problems with mutual exclusion such as distributed databases [Knapp, 1987].

Chapter 10

Offline Planning to Overcome Crash Faults in Execution

This chapter studies multi-agent path planning (MAPP) assuming crashes of agents. Here, following the terminology in the previous chapter of OTIMAPP, *MAPP* is used as a generalized term for path planning for multiple agents, not limited to the formalization of MAPF. The study uses the same strategy as what we saw for timing uncertainties, namely, considering offline planning so that agents can reactively act according to runtime situations. The problem is formulated as *MAPP with crash faults (MAPPCF)*, which originally appeared in [Okumura and Tixeuil, 2023].[1]

10.1 Chapter Overview

The objective of the chapter is to establish offline planning followed by reactive execution that overcomes crash faults at runtime, embodied as MAPPCF. The chapter provides both theoretical foundations and practical methods.

10.1.1 What is MAPPCF

MAPPCF is a novel graph path planning problem for multiple agents that may crash at runtime. The crashed agents then forever block part of the workspace. Correct agents (i.e., non-crashed ones) can detect crashes through *local observations* and then switch their executing path on the fly, based on this observation. The objective is to find a collection of paths and their switching rules for each agent, such that correct agents can reach their destinations regardless of crash patterns.

Throughout the chapter, MAPPCF assumes *local observations* that permit to immediately detect a crash if it occurs in a neighboring location. Therefore, significantly different from conventional MAPP studies, the challenge here is to design a safe planning methodology that avoids collisions or deadlocks, under the assumption that agents behave following their plan, and *their own observed information* about other agents' crashes.

10.1.2 Why MAPPCF is Attractive

As discussed in Chap. 3.5.3, building robust and resilient multi-robot systems is an emerging and important topic. Since MAPP is the foundation of multi-robot systems, designing robust and resilient approaches to MAPP is a critical component to realize reliable multi-robot systems. Nevertheless, cutting-edge MAPP studies largely overlook

[1] Work done during an internship at LIP6, Sorbonne University, France. I really enjoyed the collaboration, as well as the beautiful city of Paris!

this aspect and assume that agents perfectly follow the offline planning that is prepared without any fault assumptions. Therefore, the chapter is positioned as the first step of fault-tolerant MAPP.

Similar to OTIMAPP, once a solution to MAPPCF is obtained, decentralized execution is possible. That is, each agent can take actions spontaneously while sensing its surrounding situation. Furthermore, all correct agents are guaranteed to reach their destination (i.e., liveness) despite unforeseen crashes.

10.1.3 What will be Presented

Just like the previous chapter, this chapter contains both theoretical and practical parts.

Theoretical Part

The MAPPCF problem is formalized. The formalization includes the conventional synchronous execution model of MAPF where all agents take actions simultaneously, as well as the asynchronous execution model of OTIMAPP. In the formalization, agents can switch executing paths on the fly according to local observation results. The observation is done using a *failure detector*, a blackbox function that tells an agent whether an adjacent location is occupied by a crashed agent, occupied by a correct agent, or vacant. The chapter considers anonymous and named failure detectors; the former cannot identify a crashed agent (only a crashed location). After characterizing relationships between execution models and failure detector variants, the computational complexities of MAPPCF are provided. The main results are that finding a solution is NP-hard, and verifying a solution is co-NP-complete.

Practical Part

A method to solve MAPPCF is presented, called *decoupled crash faults resolution framework (DCRF)*. DCRF resolves the effects of crashes one by one by preparing backup paths. Then, DCRF with the named failure detector was evaluated in grid environments. The experiments show that DCRF can address more problem instances compared to computing a list of vertex disjoint paths, i.e., a trivially fault-tolerant approach since correct agents can reach their destinations regardless of crash patterns. Moreover, we see that the difficulty of finding solutions stems both from the problem instance size (e.g., the number of agents), and from the number of crashes to be tolerated.

10.1.4 Chapter Organization

- Chapter 10.2 formalizes MAPPCF.
- Chapter 10.3 presents preliminary analyses.
- Chapter 10.4 presents analyses on computational complexity.
- Chapter 10.5 describes DCRF.
- Chapter 10.6 presents empirical results.
- Chapter 10.7 reviews related work.
- Chapter 10.8 concludes the chapter.

The code is available at `https://kei18.github.io/mappcf`.

10.1.5 Notations and Assumptions

$G = (V, E)$	(undirected) graph, a set of vertices, and a set of edges
$A = \{1, 2, \ldots, n\}$	a tuple of agents
s_i, g_i	start and goal of agent i
f	the maximum number of crashes
p_i	plan for agent i

Caution

Similar to the previous chapter of OTIMAPP, in this chapter, *all indexes start from one*. The chapter uses the representation by paths.

10.2 Offline Problem

10.2.1 Problem Formulation

MAPPCF Instance. An *MAPPCF instance* is given by a graph $G = (V, E)$, a set of agents $A = \{1, 2, \ldots, n\}$, the maximum number of crashes $f \in \mathbb{N}_{\geq 0}$, injective initial-state function $s : A \mapsto V$, and injective goal-state function $g : A \mapsto V$. Let s_i and g_i denote $s(i)$ and $g(i)$, respectively. An MAPPCF instance on digraphs is similar to the undirected case.

Plan. A *plan* for one agent comprises a list of paths each defined on G and *transition rules*. At runtime, the agent moves along one path in the plan, called *executing path*, while always occupying one vertex. Meanwhile, the agent switches its executing path following the transition rules. A plan contains one special path called *primary path* which is initially executed. A transition rule is defined with a *failure detector* and *progress index*, explained below.

Crash and Failure Detector. During plan execution, agents are potentially crashed. *Crashed* agents eternally remain in their occupying vertices. *Correct* agents are those who are not crashed. Correct agents cannot pass where crashed agents are located. However, a correct agent can use a *failure detector* at runtime to change its executing path. Doing so enables the correct agent to reach its goal under crash faults. A failure detector tells a correct agent about the existence of an agent on an adjacent vertex, and if so, whether it has crashed or not. Two types of detectors are considered. A detector is called *named (NFD)* when it can identify who is crashed, otherwise *anonymous (AFD)*. Formal definitions are as follows. Assume that an agent i is at $v \in V$. Then, $\mathtt{AFD} : \mathtt{neigh}(v) \mapsto \{\circ, \times, \perp\}$ and $\mathtt{NFD} : \mathtt{neigh}(v) \mapsto \{\circ, \perp\} \cup A$. Here, $\circ$ and $\times$ respectively correspond to a correct or crashed agent, otherwise $\perp$ is returned (no agent is there). NFD returns an agent instead of $\times$.

Progress Index. A *progress index* $clock_i \in \mathbb{N}_{>0}$ specifies location; the agent i is at $\langle clock_i \rangle$–th vertex of its executing path. For each transition of executing paths, the progress index is initialized with one. It increases up to the length of the executing path.

Transition Rule. The rule specifies the next executing path given the current executing path, progress index, and results of the failure detector. Two execution models that differ in how to increment progress indexes are then described.

Synchronous Execution Model (SYN). In this model, all agents take actions simultaneously. More precisely, all the correct agents perform the following at the same time:

1. Agents may crash.
2. Correct agents change executing paths if necessary, using a failure detector.
3. Move to their next vertices.
4. Increment their progress indexes.

Two types of collisions must be prohibited by plans:

- vertex collisions: two agents are on the same vertex simultaneously.
- swap collisions: two agents swap their hosting vertices simultaneously.

Note that an agent can remain at its hosting vertex if its executing path contains the same vertex consecutively.

Sequential Execution Model (SEQ). In this model, agents take actions sequentially but we cannot control how agents are scheduled at runtime. More precisely, given an infinite sequence of agents $\mathcal{E}$ called *execution schedule*, the agents are *activated* in turn according to $\mathcal{E}$. Upon activating, the agent performs the following:

1. Change executing paths if necessary.
2. Move to its next vertex specified by the progress index if the vertex is unoccupied by other agents.
3. Increment its progress index if moved.

The agent remains on its hosting vertex when the next vertex is occupied. Agents crash at any time, except for the duration of the procedures for activation.[2] The execution schedule $\mathcal{E}$ is unknown when offline planning, but the chapter assumes that every agent appears infinitely-many times in $\mathcal{E}$.

Solution. Given an MAPPCF instance, a *solution for SYN* is a tuple of plans $(p_1, p_2, \ldots, p_n)$ respectively for each agent, such that:

1. The primary path of p_i begins with a start s_i.
2. For each path that is not primary, the path begins with a vertex where the agent changes its executing path.
3. The agent i is ensured to reach its goal g_i provided that i follows p_i, regardless of other agents' crashes, when the total number of crashes is up to f.

A *solution for SEQ* is similar to SYN but the third condition should be satisfied for any execution schedule. Figure 10.1 presents solution examples for both models. Without crash assumptions (i.e., when $f = 0$), solutions for SYN are equivalent to those of classical MAPF of Chap. 3.2.1, and solutions for SEQ are equivalent to those of OTIMAPP (Chap. 9.2.1). The reminder assumes $f > 0$.

[2] Hence, activation is regarded as atomic.

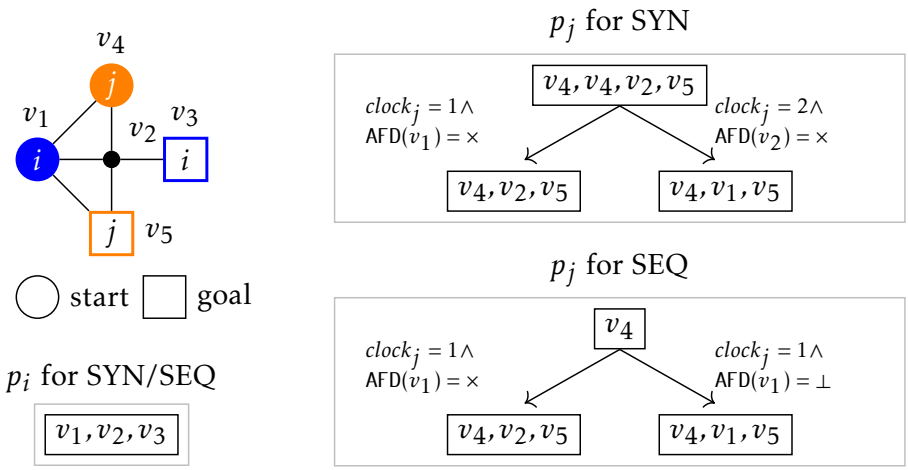


Figure 10.1: Solution example with AFD.

10.2.2 Remarks

As discussed in the online time-independent planning (Chap. 8.2.2), in SYN, a solution must prevent collisions, while in SEQ, collisions are assumed to be prevented by agents' runtime behaviors. Implementations of failure detectors depend on applications, e.g., using heartbeats as commonly used in distributed network systems [Felber *et al.*, 1999] or multi-robot platforms such that environments can detect robot faults [Kameyama *et al.*, 2021]. Herein, failure detectors are assumed to be blackbox functions.

10.3 Preliminary Analysis

To begin with, two fundamental analyses are provided to grasp the characteristics of MAPPCF, namely, the model power and the necessary condition for instance to include a solution.

10.3.1 Model Power

A model X is *weakly stronger* than another model Y when all solvable instances in Y are also solvable in X. X is *strictly stronger* than Y when it is weakly stronger than Y and there exists an instance that is solvable in X but unsolvable in Y. Two models are *equivalent* when both are respectively weakly stronger than another.

A *model* of MAPPCF is specified by two components:

- whether the failure detector is anonymous (AFD) or named (NFD), and
- whether the execution model is synchronous (SYN) or sequential (SEQ).

Characterizing model power, i.e., which model is stronger than another, is important because it can be instrumental when implementing the algorithm. For instance, AFD is intuitively easier to implement than NFD. So, if those two models have equivalent power, then we may not need to realize NFD.

The main results are summarized in Fig. 10.2. Several relationships are still open questions, e.g., whether NFD is strictly stronger than AFD in SYN. In what follows, three theorems for the model power analysis are presented.

Theorem 10.1 (NFD vs. AFD)**.** *When using the same execution model, NFD is weakly stronger than AFD.*

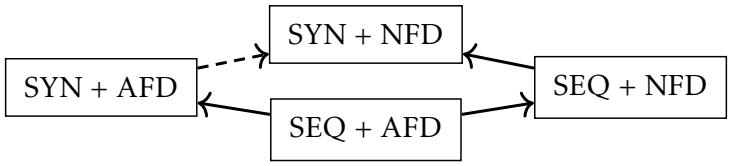


Figure 10.2: Relationship of models. '$X \to Y$' denotes that model Y is strictly stronger than model X. A dashed arrow means a weakly stronger relationship.

Proof. NFD can emulate AFD by dropping "who." □

Theorem 10.2 (SYN vs. SEQ)**.** *When using the same failure detector types, SYN is strictly stronger than SEQ.*

Proof. Figure 10.3 shows an instance that is solvable for SYN but unsolvable for SEQ. In SYN, the agent j can wait until i passes the middle two vertices, and according to crash patterns, j can change its path toward its goal. However, in SEQ, there are execution schedules that j enters either of the middle two vertices prior to i because j cannot distinguish whether i is on its start or goal. If j is crashed there and i still remains at its start, i cannot reach its goal.

Next, it is proven that every solvable instance in SEQ is solvable in SYN. Consider constructing a new solution Z_{syn} in SYN given a solution Z_{seq} in SEQ. This is achieved by, considering one execution schedule (e.g., (1, 2, . . . , n, 1, 2, . . . , n, . . .)) and allowing Z_{syn} to move agents in their turn. For instance, at timestep one, only agent-1 is allowed to move, and at timestep two, only agent-2 is allowed to move, and so forth. With appropriate modifications of paths specified Z_{seq}, since Z_{seq} solves the original instance, Z_{syn} also solves the instance in SYN. □

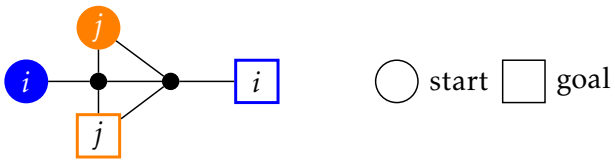


Figure 10.3: Solvable instance in SYN but unsolvable in SEQ.

Theorem 10.3 (NFD vs. AFD in SEQ)**.** *In SEQ, NFD is strictly stronger than AFD.*

Proof. Figure 10.4a is an instance that is only solvable with NFD in SEQ. Regardless of the failure detector types, i needs to move to the center vertex as the first action (Fig. 10.4b); otherwise, i may crash at the goal vertex of another agent, and this agent never reaches its goal due to i's crash. The same argument holds for j and k.

Assume that j detects someone is crashed at the center vertex. Then j needs to move to either the goal of i or the goal of k. Consider that j moves to k's goal, and then crashes (Fig. 10.4c). If the crashed agent at the center vertex is i, the remaining correct agent k cannot reach its goal. A symmetric situation occurs if j moves to i's goal but the crashed agent at the center vertex is k. As a result, this instance is unsolvable with AFD. In contrast, with NFD, j can choose the "correct" vertex according to detected crashes, e.g., i's goal when i is crashed at the center vertex. Even if j is further crashed at i's goal, k can reach its goal (Fig. 10.4d). □

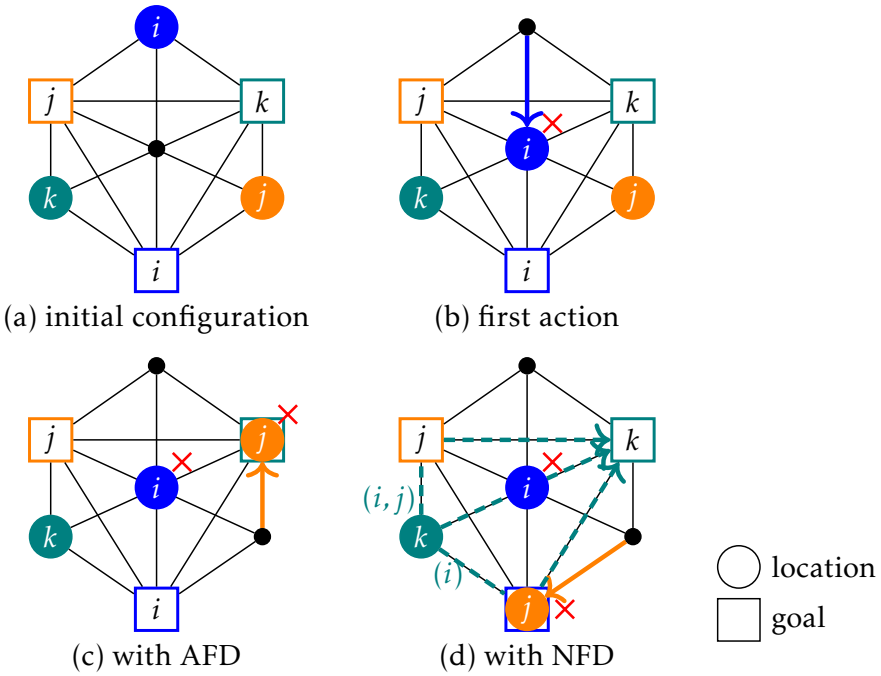


Figure 10.4: Instance that is solvable with NFD, but unsolvable for AFD in SEQ. A red cross corresponds to a crashed agent. In (d), part of the plan of k is visualized by dashed arrows, annotated with detected crashes.

10.3.2 Necessary Condition

Theorem 10.4. *Regardless of execution models and failure detector types, two conditions are necessary for instances to contain solutions.*

- *No use of other goals: For each agent i, there exists a path in G from s_i to g_i that does not include any g_j, for all $j \in A, j \neq i$.*
- *Limitation of other starts: For each agent i, for each $B \subset A$ where $i \notin B$ and $|B| = f$, there exists a path in G from s_i to g_i that does not include s_j, for all $j \in B$.*

Proof. Each item is respectively proven as follows:

- *No use of other goals:* Suppose that there exists an agent i that needs to pass through one of the goals g_j. If i crashes at g_j, then j cannot reach g_j.
- *Limitation of other starts:* Suppose that there exists an agent i that needs to pass through one of the starts of B. If all agents in B are crashed at their starts, then i cannot reach g_i.

□

When $f = |A| - 1$, the conditions in Thrm. 10.4 are equivalent to a *well-formed instance* [Čáp *et al.*, 2015] for MAPF; an instance such that every agent has at least one path that uses no others' start and goal vertices. The condition of Thrm. 10.4 slightly differs in the limitation of starts because agents can change their behavior at runtime, according to failure detectors.

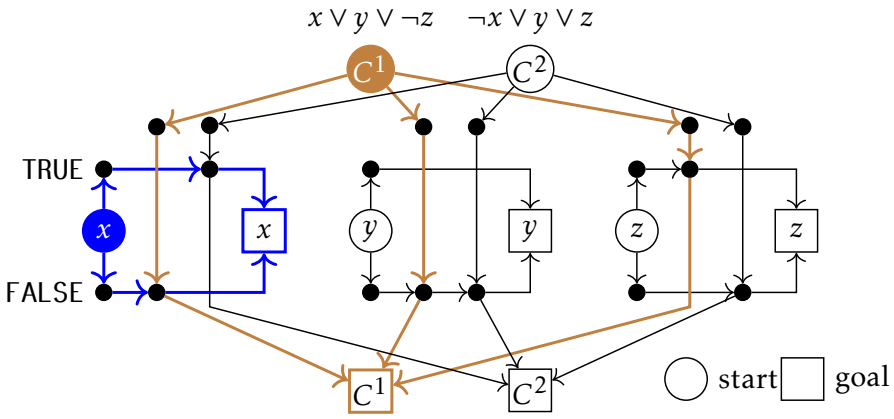


Figure 10.5: MAPPCF instance on a directed graph in SEQ reduced from SAT. The corresponding formula is $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$.

10.4 Computational Complexity

This section discusses the complexity of MAPPCF. Specifically, two questions are considered, namely, the difficulty of finding solutions and that of verifying solutions. The primary result is that both problems are computationally intractable; the former is NP-hard and the latter is co-NP-complete. Both proofs are based on reductions of the 3-SAT problem, determining the satisfiability of a formula in conjunctive normal form with three literals in each clause.

10.4.1 Finding Solutions

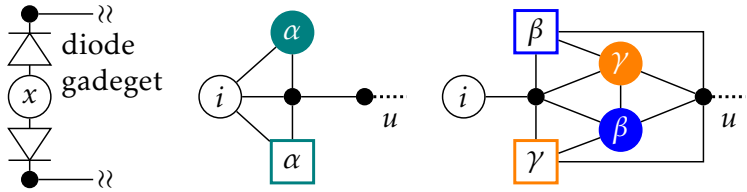
Theorem 10.5 (complexity of the decision problem)**.** *MAPPCF is NP-hard regardless of models.*

Proof. It is first proven that MAPPCF on digraphs in SEQ is NP-hard. The proof is done by reduction of the SAT problem and works regardless of failure detector types. Throughout the proof, the following example is used: $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$. The reduction is depicted in Fig. 10.5.

A. Construction of an MAPPCF Instance

To begin with, a *variable agent* is introduced for each variable x_i. Figure 10.5 highlights the corresponding agent of the variable x as blue-colored. The reduced instance has two paths for each variable agent: *upper* or *lower* paths. Both paths include at least one vertex (just above/below the start in Fig. 10.5) and additional vertices depending on clauses of the formula.

Next, a *clause* agent is introduced for each clause C^j of the formula. Each clause agent has multiple paths, corresponding to each literal in the clause. Those paths contain two vertices excluding the start and the goal: one vertex unique to each literal, and another vertex shared with the corresponding variable agent. The shared vertex is located on the lower (or upper) path of the variable agent when the literal is positive (resp. negative). In Fig. 10.5, the corresponding agent of the clause $C^1 = x \vee y \vee \neg z$ is highlighted as brown-colored. The translation from the formula into an MAPPCF instance is clearly done in polynomial time.



(a) usage (b) SEQ (c) SYN

Figure 10.6: Diode gadgets.

B. MAPPCF has a solution if the formula is satisfiable

Given a satisfiable assignment, a solution for MAPPCF is built as follows. When a variable is assigned TRUE (or FALSE), let the corresponding agent takes the upper (resp. lower) path. Each clause agent then has at least one path (vertex) disjoint with any variable agent; otherwise, the clause is unsatisfied. Let the clause agent take this path. These paths constitute a solution because all paths are disjoint.

C. The formula is satisfiable if MAPPCF has a solution

In every solution, a plan for each agent inherently consists of a single path, due to the instance construction. These paths should be (vertex) disjoint; otherwise, a crash of one agent blocks another from reaching its goal. Then, build an assignment as follows. Assign a variable TRUE (or FALSE) when the corresponding variable agent uses the upper (resp. lower) path. This assignment is satisfiable because it ensures at least one literal is satisfied in all clauses.

D. Extending the reduction to undirected graphs

The aforementioned proof is extended to the undirected case by introducing a *diode* gadget to the starts of every agent, as partially shown in Fig. 10.6a. This gadget prevents back to the start once the agent passes through the gadget (i.e., reaching vertex u in Fig. 10.6b–c). Therefore, the same proof procedure (i.e., finding disjoint paths) is applied to other models. The gadget for SEQ and SYN are shown in Fig. 10.6b–c, respectively. The proofs of their properties are delivered into Observations 10.6 and 10.7. □

Observation 10.6 (diode gadget for SEQ)**.** *Assuming SEQ, in Fig. 10.6b, agent i cannot go back to its starting location once it has reached vertex u.*

Proof. Observe that agent α cannot move until i moves to the center vertex; otherwise, i cannot reach u if α crashes at the center vertex. Once i reaches u, α moves to the center vertex. If α crashes there, then i cannot go back to its starting location. Note that regardless of the crashed status of i, α can reach its goal. □

Observation 10.7 (diode gadget for SYN)**.** *Assuming SYN, in Fig. 10.6c, agent i cannot go back to its starting location once it has reached vertex u.*

Proof. Figure 10.7 shows how the agent i reaches the vertex u. In the initial configuration (Fig. 10.6c), β needs to wait for i to move from the starting location; otherwise, if β crashes there, either i or γ cannot reach its goal. The core observation is that γ (or β) must use the right neighboring vertex of i's starting location in Step 2 of Fig. 10.7. Therefore, if γ crashes at Step 2, i cannot go back to the starting location. Note that regardless of the crashed status of i, β, and γ, the remaining agents can reach their goals. □

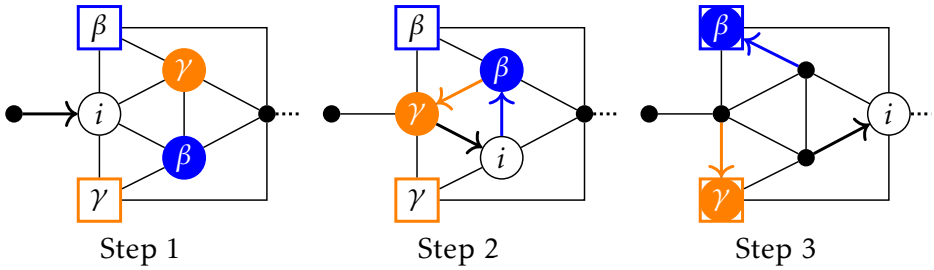


Figure 10.7: Execution in the diode gadget for SYN without crashes.

Although MAPPCF is intractable in general, it is notable that the difficulties might be relaxed in certain classes, e.g., instances on planner graphs.

10.4.2 Verification

Theorem 10.8 (complexity of verification)**.** *Verifying a solution of MAPPCF is co-NP-complete regardless of models.*

Proof. The below part focuses on the case of SEQ with AFD; however, it is easily applicable to other models such as those with SYN or with NFD. The proof is done by reduction of the SAT problem, i.e., constructing a combination of an MAPPCF instance and a tuple of plans such that the plans constitute a solution if and only if the corresponding formula is unsatisfiable. Since the infeasibility check of the plan is done in polynomial time with a proper execution schedule and crash patterns, the verification is co-NP-complete. Throughout the proof, the proof uses the following example:

$$(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y) \tag{10.1}$$

The reduction is depicted in Fig. 10.8. Let denote the number of clauses in the formula as l.

A. Construction of Instance and Plans

First, a *variable agent* is introduced for each variable x_i of the formula. Figure 10.8 highlights the corresponding agent of the variable x as blue-colored. A plan for a variable agent is just to move one step to the right.

Next, a *clause* agent is introduced for each clause C^j. Figure 10.8 highlights the corresponding agent of the clause $C^2 = \neg x \vee y \vee z$ as brown-colored. A plan for the clause agent consists of two sub-plans:

- A *primary plan* passes through: *(i)* either the start or the goal of a variable agent, *(ii)* a *rest vertex*, *(iii)* one of the *bottleneck vertices*, and *(iv)* its goal (so, four steps in total).
- An *contingency plan* directly reaches the goal in one step. This plan is used when the clause agent cannot take the first step of the primary plan.

Each rest vertex is unique to each clause agent. The start of a clause agent is connected to a rest vertex via a start (or goal) of a variable agent when the clause contains a negative (resp. positive) literal of the variable. Bottleneck vertices are shared between clause agents, and their number is $l-1$. Each rest vertex and goal of a clause agent is connected to all bottleneck vertices.

The reduction further introduces *observation edges* (dotted lines) between each start of a clause agent and start of a variable agent. These edges are not included in any plans

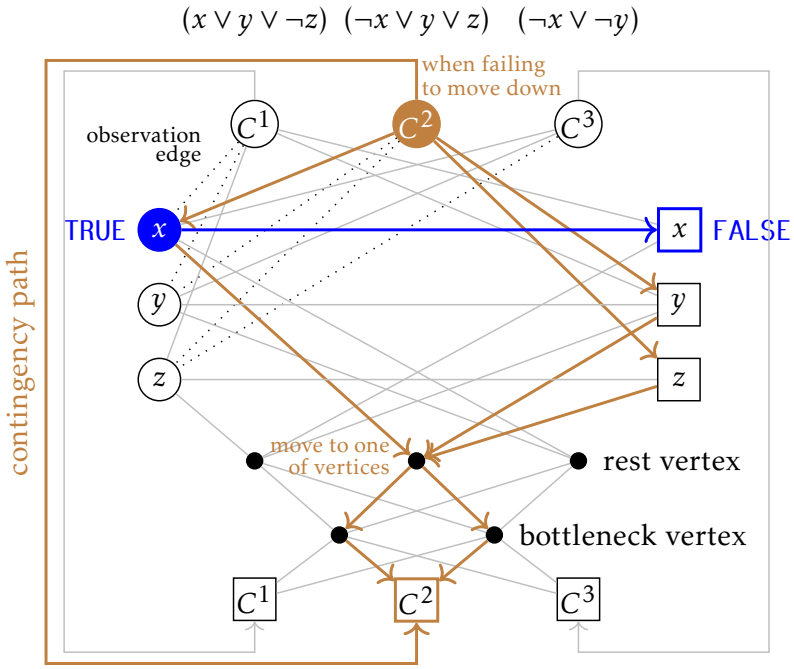


Figure 10.8: MAPPCF instance and solution reduced from SAT. The corresponding formula is $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \vee (\neg x \vee \neg y)$. Dotted lines are edges that are not used in the plans.

but they enable clause agents not to enter the goals of variable agents when the variable agents are correct, and they are on their start locations.

The translation from the formula into an MAPPCF instance and the plans is clearly done in polynomial time.

B. The formula is unsatisfiable when the plans are the solution

Observe that any assignment can be built by the execution of MAPPCF. Specifically, assign a variable TRUE when the corresponding variable agent is crashed at its start; otherwise FALSE, i.e., when the variable agent enters its goal.

Assume now that all clause agents use their primary plan, and are currently on their rest vertex. If $l-1$ clause agents move to bottleneck vertices, and crash there, then the remaining agent cannot reach its goal following the plan. Therefore, for the set of plans to be a solution, it is necessary that at least one clause agent takes its contingency plan. Consider now that clause agent. This agent i cannot take the primary plan because some variable agents are blocking i either due to a crash at their start, or upon reaching their goal. The corresponding assignment is unsatisfiable for the clause. For instance, in Fig. 10.8, assume that the clause agent C^2 takes the contingency plan. This happens when the variable agent x is crashed at its start, and both y and z have entered their goals. The assignment is then $x = \mathsf{TRUE}$, $y = \mathsf{FALSE}$, and $z = \mathsf{FALSE}$, which is unsatisfiable for C^2.

C. The plans constitute a solution when the formula is unsatisfiable

When the formula is unsatisfiable, it never happens that all agents take their primary plan and all correct agents are ensured to reach their goals. □

10.5 Solving MAPPCF

From this section, we focus on the practical part, namely, describing how to solve MAPPCF. The primary challenge is how to manage crash awareness differences among agents. For instance, agent i may observe a crash of agent j while at a neighboring position, and change its path accordingly. However, another correct agent k, located further away from j, might not be aware that j has crashed. To preserve safety and liveness, a plan of k requires avoiding collisions and deadlocks with both before-after paths of i (that is, regardless of the crash of j). Any planning algorithm solving MAPPCF thus has to cope with different awareness of crash patterns.

The proposed method, *decoupled crash faults resolution framework (DCRF)*, returns an MAPPCF solution. Herein, the study focuses only on NFD, but DCFR is also applicable to AFD.

10.5.1 Framework Description

Algorithm 10.1 presents DCFR. Figure 10.9 illustrates a running example in SYN. The below part first describes DCRF using this example, followed by detailed parts of the implementation.

Algorithm 10.1 DCRF.

input: MAPPCF instance I

output: solution $\mathcal{P}$ or `FAILURE`

1: $\mathcal{P} \leftarrow$ `get_initial_plans`(I)
2: $\mathcal{U} \leftarrow$ `get_initial_unresolve_events`$(\mathcal{I}, \mathcal{P})$
3: **while** $\mathcal{U} \neq \emptyset$ **do**
4: $\quad e \leftarrow \mathcal{U}$.pop() ▷ *event; pair of crash and effect*
5: $\quad \pi \leftarrow$ `find_backup_path`$(I, \mathcal{P}, e)$
6: $\quad$ **if** $\pi = \perp$ **then return** `FAILURE`
7: $\quad \mathcal{U}$.push(`get_new_unresolved_events`$(I, \mathcal{P}, \pi)$)
8: $\quad$ update $\mathcal{P}$ with π
9: **return** $\mathcal{P}$

Finding Initial Plans. DCRF first obtains an initial plan for each agent (i.e., a path) (Line 1). This process is equivalent to solving MAPF with existing solvers. In the example, the initial plans for agents i, j, and k are respectively depicted in Fig. 10.9d, 10.9b, and 10.9c.

Identifying Unresolved Events. DCRF next identifies *unresolved events*. An event is a pair of *crash* (i.e., who crashes where and when), and *effect* (i.e., whose path is affected where and when). Finding unresolved events is done by finding shared vertices in a tuple of paths. In the example (Fig. 10.9d), the initial plans contain two unresolved events:

- e^1: $\mathcal{P}_i$ cannot use $\langle v_2; t = 2\rangle$ if j is crashed at $\langle v_2; t = 1\rangle$
- e^2: $\mathcal{P}_i$ cannot use $\langle v_3; t = 3\rangle$ if k is crashed at $\langle v_3; t = 2\rangle$

These events should be resolved by preparing *backup paths*. DCRF resolves events in a decoupled manner as follows.

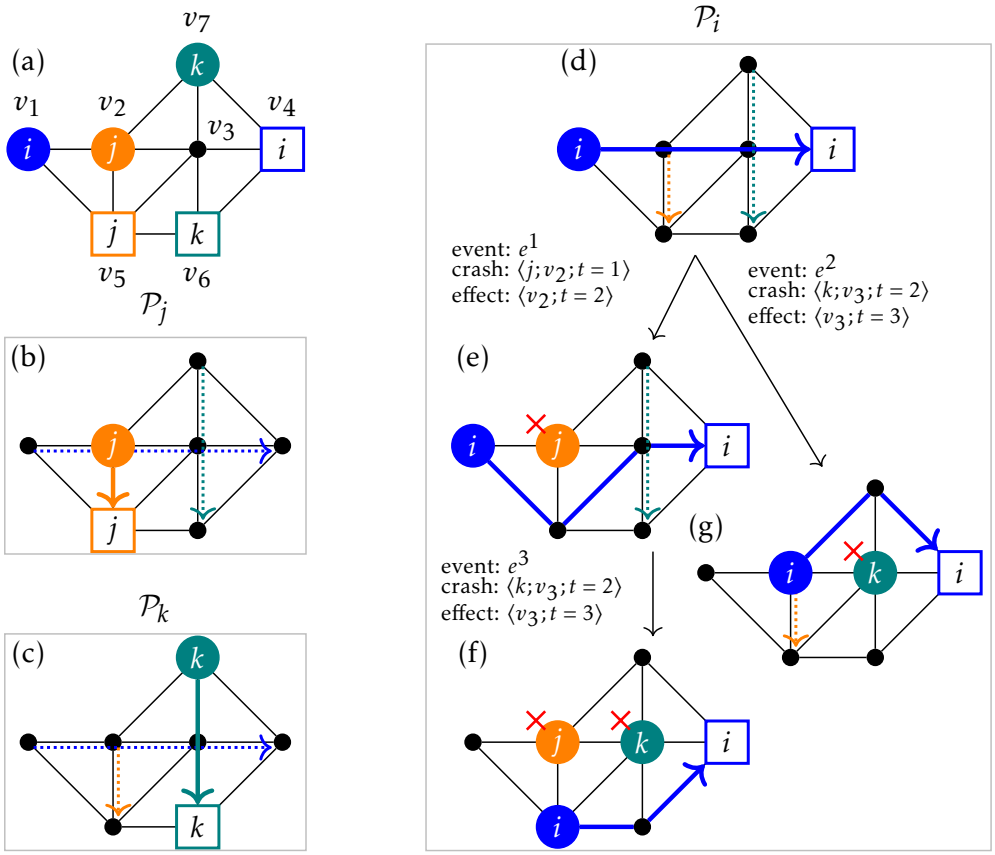


Figure 10.9: Running example of DCRF in SYN. A red cross corresponds to an observed crashed agent. Dotted lines are paths with which the planning should avoid collisions.

Resolving Events. Unresolved events are stored in a priority queue $\mathcal{U}$ (named from "unresolved") and handled one by one (Lines 3–8). For each event, DCRF tries to find a backup path (Line 5). This is a single-agent pathfinding problem, whose goal location is the same as the initial path, while the start location is one-step before where the crashed agent is. The pathfinding is constrained to avoid collisions with observed crashed agents and other already constructed plans. If failed to find such a path, DCRF reports FAILURE. Otherwise, DCRF identifies new unresolved events with the backup path, and updates the plan. The event is now resolved. When all events are resolved, the framework returns a solution (Line 9). In the example, when the event e^1 is popped from $\mathcal{U}$, DCRF computes the backup path shown in Fig. 10.9e. Observe that this backup path must *not* use v_6 to avoid collisions with p_k. DCRF then identifies and registers a new unresolved event e^3, and updates i's plan. DCRF continues applying the same procedure for the events e^2 (Fig. 10.9g) and e^3 (Fig. 10.9f).

10.5.2 Implementation Details

Discarding Unnecessary Events. When one crash affects a given path several times, only the first effect should be resolved. This happens when the path is not simple. For this purpose, the queue $\mathcal{U}$ stores events in ascending order of their occurring time. DCRF then discards events when encountering already resolved crashes.

Inconsistent Crash Patterns. When preparing a backup path for agent i, pathfinding does not necessarily need to avoid collisions with all others' paths. For instance, Fig. 10.9e assumes j has crashed. Therefore, in descendant backup paths for i, i can ne-

glect j's plan. Similarly, when two different paths assume crashes at distinct locations of the same agent, or when two paths assume more than f crashes in total, those two paths cannot be executed during the same execution. Consequently, DCRF does not need to identify unresolved events between these paths.

Refinement of Initial Paths. Reducing the number of events is crucial for constructing solutions, as a higher number of events yields a higher number of paths for each agent. Consequently, when preparing a backup path, the pathfinding process tries to avoid collisions with those many paths, and may fail to find a suitable path. To circumvent this problem, when preparing initial plans, we introduce a refinement phase that minimizes the use of shared vertices between agents. This is done by adapting the technique to improve MAPF solutions, presented in Chap. 7.

Implementation for SEQ. DCRF is applicable to SEQ by simply replacing 'collision' and 'MAPF' by 'deadlocks' and 'OTIMAPP,' respectively.

Implementation for AFD. For AFD, the same workflow is available by assuming that observed crashes are anonymous.

10.5.3 Limitations

DCRF is incomplete, i.e., it does not guarantee to return a solution even though an instance is solvable. Figure 10.10 presents a failure planning example of DCRF in SYN. An MAPPCF instance is shown in Fig. 10.10a. Assume that vertices v_1 and v_5 are connected through a long path including v_{11}. Observe first that this instance contains a solution (shown in Fig. 10.10h); these paths are vertex disjoint, hence all agents reach their goals regardless of crash patterns.

Consider now that DCRF provides initial plans as Fig. 10.10e, 10.10b, and 10.10d respectively for agent i, j, and k at Line 1. Then, the planning eventually fails because there is a crash pattern that i cannot prepare a backup path, as shown in Fig. 10.10g.

10.6 Evaluation

This section evaluates DCRF in both SYN and SEQ with NFD, and presents a variety of aspects including merits to consider MAPPCF and bottlenecks of the planning.

10.6.1 Setup

Baseline. DCRF was compared with a procedure to obtain pairwise vertex-disjoint paths. The rationale is that disjoint paths are trivially fault-tolerant; regardless of crash patterns, correct agents can always reach their destinations. On the other hand, with more agents, it is expected that finding such paths becomes impossible. The disjoint paths were obtained by an adapted version of CBS [Sharon *et al.*, 2015]. The adapted one is complete, i.e., eventually returning disjoint paths if they exist, otherwise, reporting non-existence.

Experimental Design. MAPPCF has two critical factors: the number of agents $|A|$ and crashes f. To investigate those effects on the planning, two scenarios were prepared: *(i)* fixing f while changing $|A|$, or *(ii)* fixing $|A|$ while changing f. Each scenario was tested on two four-connected grids (size: 32×32 and 64×64) obtained from [Stern *et al.*, 2019]. These grids contain randomly placed obstacles (10%), , as shown in Fig. 10.11. For

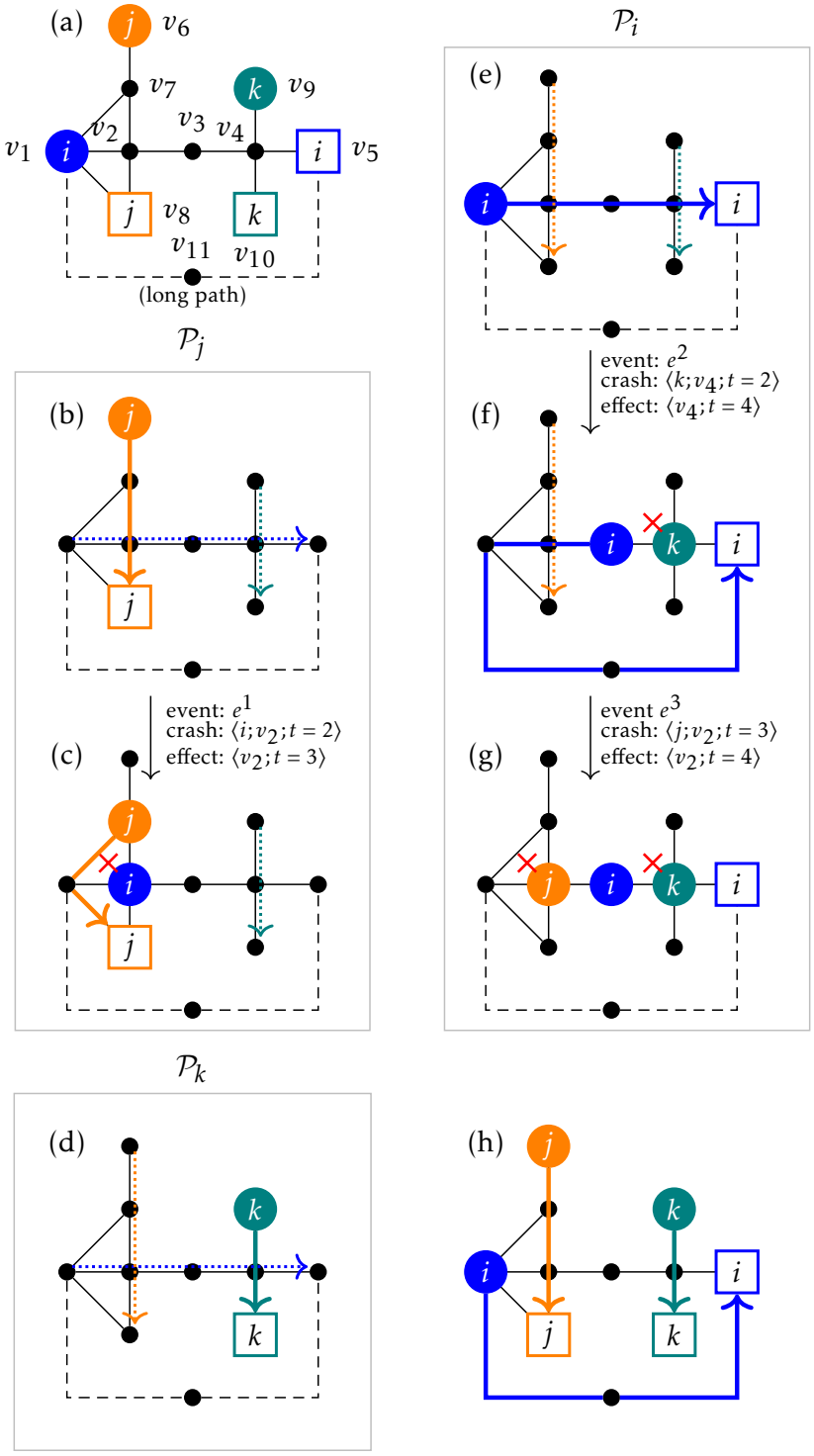


Figure 10.10: Failure example of DCRF in SYN.

each scenario, grid, and $|A|$, 25 well-formed instances [Čáp *et al.*, 2015] were prepared, considering the necessary condition for solutions to exist (see Chap. 10.3.2). However, note that unsolvable instances might still be included because it is not sufficient. The identical instances were used in both SYN and SEQ.

Planning Failure. A method is regarded as succeeding in solving an instance when it returns a solution before the timeout of 30 s; otherwise, the attempt is a failure.

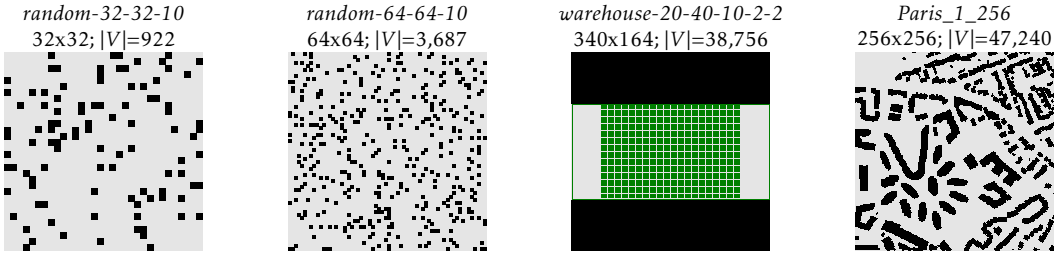


Figure 10.11: Used grids in the experiments.

Implementation of DCRF. The initial paths were obtained by prioritized planning [Čáp *et al.*, 2015; Okumura *et al.*, 2022a], respectively for SYC and SEQ. Single-agent pathfinding was implemented by A*, adding a heuristic that penalizes the use of common vertices with other agents' paths. It was informally observed that this manipulation improves the success rate due to a smaller number of events. The refinement over initial paths (Chap. 10.5.2) was applied for SYN but not for SEQ because the effect was subtle (see Chap. 10.6.3).

Evaluation Environment. The code was written in Julia. The experiments were run on a desktop PC with Intel Core i9-7960X 2.8 GHz CPU and 64 GB RAM. 32 different instances were run in parallel using multi-threading.

10.6.2 Results

Figures 10.12 and 10.13 summarize the results. In the figures, three types of charts are included.

- **failure rate**: Failure reasons of MAPPCF are also presented by stacking graphs. "no_backup" means that MAPPCF failed to prepare a backup path. "timeout" means that MAPPCF reaches the time limit. "init_paths" means that MAPPCF fails to prepare initial paths.
- **runtime**: The average runtime of successful instances is presented, accompanied by minimum and maximum values shown by transparent regions. Runtime profiling is also presented, which is categorized into preparing initial paths ("init_paths") and computing backup paths ("backup").
- **cost**: This rates solution quality when crashes do not happen. The detailed description of solution quality will be complemented later. The average, minimum, and maximum values of the successful instances are shown. Note that finding disjoint paths are irrelevant from f.

The main findings are below:

- *Regardless of models, finding solutions become difficult to compute as the number of agents |A| or crashes f increase.* With larger $|A|$ or f, DCRF needs to manage a huge number of crash patterns. Consequently, DCRF often fails to find backup paths or reaches the timeout.
- *MAPPCF can address more crash situations compared to just finding disjoint paths.* Note however that the gaps in the success rate between DCRF and finding disjoint paths becomes smaller in SEQ. This is partially due to finding deadlock-free paths as they are difficult to compute in SEQ, as seen in Chap. 9.

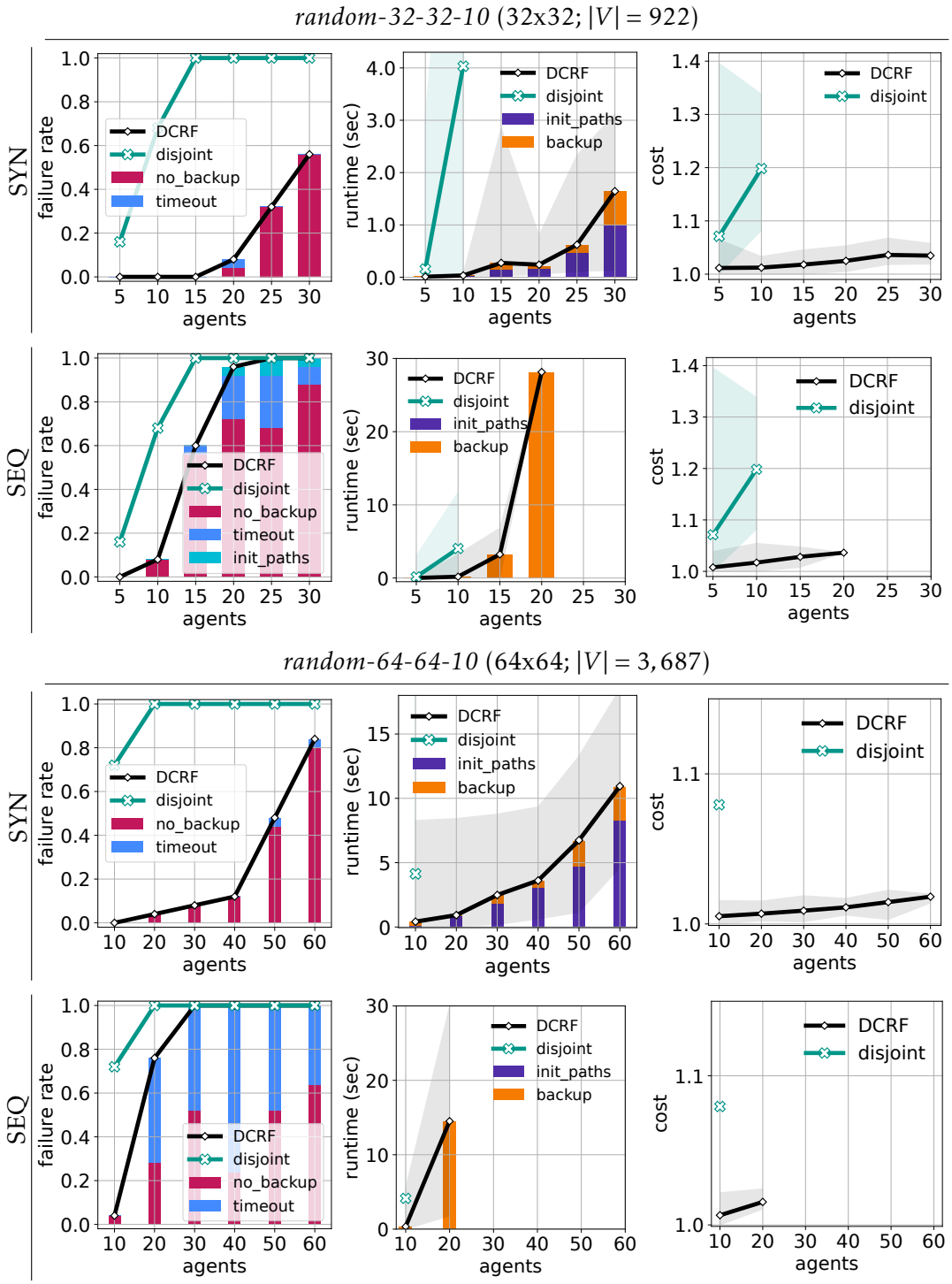


Figure 10.12: Results with a fixed number of crashes ($f = 1$).

- *Without crashes, DCRF provides solutions with smaller costs, compared to disjoint paths.* Figures 10.12 and 10.13 also present the cost of paths that agents are to follow if there is no crash. For SYN, a cost is total traveling time (aka. sum-of-costs). For SEQ, a cost is the sum of path distance (aka. sum-of-fuels). Both scores were normalized by the sum of distances between start-goal pairs; hence the minimum is one. Note that the cost is identical with different f if $|A|$ and start-goal locations are the same. The result indicates that DCRF provides better planning that suppresses redundant agents' motions when the entire system operates correctly.

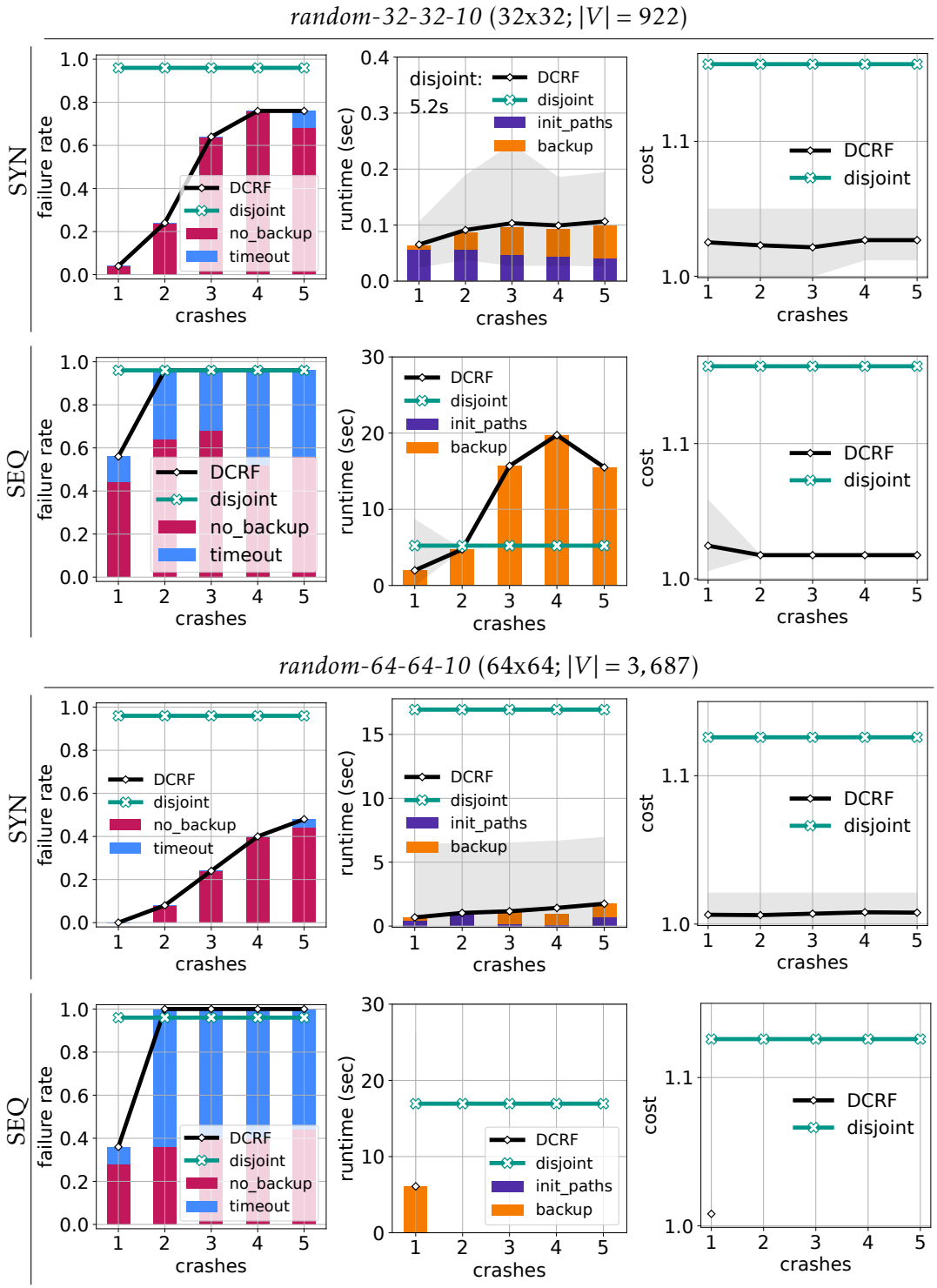


Figure 10.13: Results with a fixed number of agents ($|A| = 15$).

10.6.3 Effect of Refinement

This part complements the effect of refinement over the initial paths introduced in Chap. 10.5.2. Figure 10.14 presents the number of solved instances given a certain time, over the identical instances to those of Fig. 10.12 (the scenario of fixed crashes). The timeout was set to 30 s. We can see a slight effect in SYN while not so in SEQ. This is owing to that finding deadlock-free (backup) path itself is difficult.

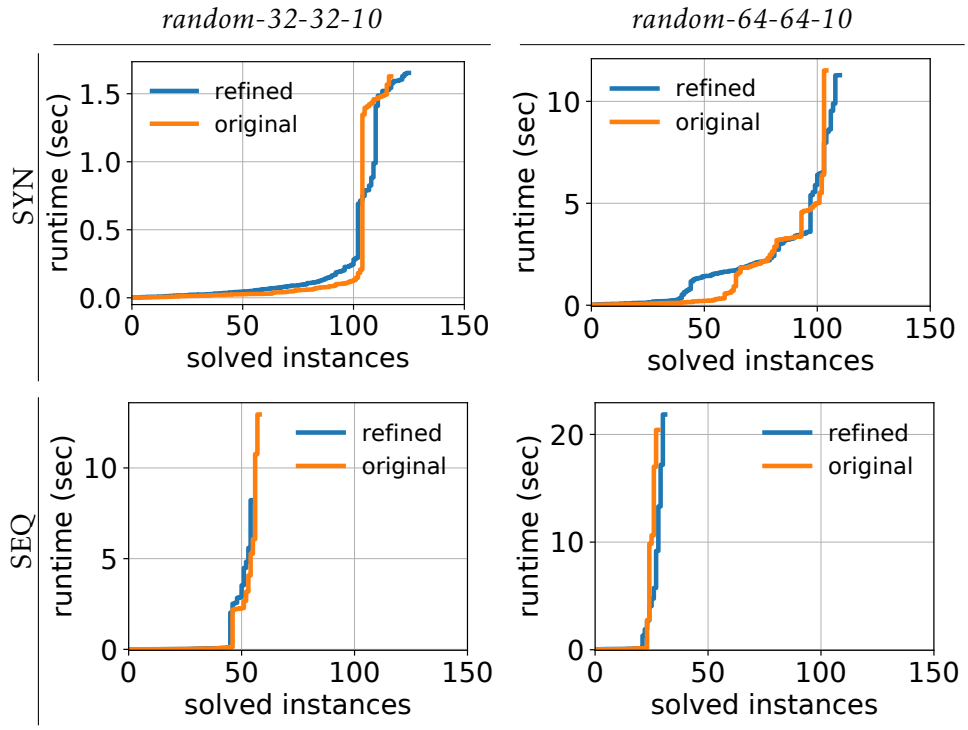


Figure 10.14: Effect of refinement. The number of solved instances until a certain time is visualized. The identical instances to those of Fig. 10.12 were used. The maximum number of crashes f was fixed to one while the number of agents varied from 5 to 30 in *random-32-32-10* and from 10 to 60 in *random-64-64-10*.

10.6.4 Results of Large Instances

Setup. Lastly, DCRF was evaluated in larger grids compared to the experiment in Chap. 10.6. Scenarios with fixed the number of crashes f as one were prepared, while changing the number of agents up to 80. 25 well-formed instances in *warehouse-20-40-10-2-2* and *Paris_1_256* were prepared.

	$\|A\|$	20	40	60	80
warehouse-20-40-10-2-2	SYN	1.00	1.00	1.00	0.76
	SEQ	0.64	0.00	0.00	0.00
Paris_1_256	SYN	1.00	0.84	0.52	0.04
	SEQ	0.24	0.00	0.00	0.00

Table 10.1: Success rate of large instances. The number of crashes is fixed as $f = 1$.

Result. Table 10.1 presents the success rate of DCRF with the 5 min timeout. In SYC, DCRF solved a moderate number of instances even with tens of agents, while in SEQ, DCRF failed in most instances. These gaps stem from the difficulty of finding deadlock-free paths in SEQ.

10.6.5 Discussion

Since this is the first study of MAPPCF where agents may observe different information at runtime, there is room for algorithmic improvements. Therefore, further improve-

ments in DCRF or the development of new MAPPCF algorithms are promising directions. Specifically, as seen in the results, a large amount of failure reasons is failing to prepare backup paths. DCRF takes a decoupled approach that sequentially plans a path for each agent. Instead, developing coupled approaches may decrease the failure rate.

10.7 Related Work

Before concluding the chapter, this section summarizes related studies to MAPPCF.

10.7.1 Path Planning for Multiple Agents

Reactive vs. Deliberative. Similar to OTIMAPP, MAPPCF is based on the discussion of *reactive* and *deliberative* approaches. Reactive approaches can deal with unexpected events such as crash failures. However, provably deadlock-free systems are difficult to realize due to the shortsightedness of time evolution. Deliberative approaches use a longer planning horizon to plan collision/deadlock-free trajectories. Recent studies [Atzmon *et al.*, 2020b; Shahar *et al.*, 2021; Okumura *et al.*, 2022a] focus on robust MAPF for timing uncertainties, i.e., where agents might be delayed at runtime. On the other hand, those studies assume that agents never crash and eventually take actions; significantly different from MAPPCF. MAPPCF is on the deliberative side but also has reactive aspects because agents change their behaviors at runtime according to failure detectors.

Two-level Search Scheme. DCRF can be regarded as a two-level search, akin to popular MAPF algorithms [Sharon *et al.*, 2013; Sharon *et al.*, 2015; Surynek, 2019; Lam *et al.*, 2022]. Those algorithms manage collisions at a high-level, and perform single-agent pathfinding at a low level. Instead of collision management, DCRF manages unresolved crash faults at the high-level.

Local Observation Assumption. MAPP with *local observations* is not new in the literature [Wiktor *et al.*, 2014; Zhang *et al.*, 2016]. Typically, previous studies aim at avoiding collisions or deadlocks by applying ad-hoc rules of agents' behavior, according to observation results at runtime, without assumptions of crash fault. It is notable that learning-based MAPF approaches like [Sartoretti *et al.*, 2019] also assume local observations of each agent (e.g., field-of-view).

10.7.2 Resilient Multi-Robot Systems

Studies on multi-robot systems sometimes assume robot crashes at runtime, e.g., for target tracking [Zhou *et al.*, 2018], orienteering [Guangyao Shi, 2020], and task assignment [Schwartz and Tokekar, 2020]. In these studies, however, crashed robots do not disturb correct robots as we assume in this chapter. In the context of pathfinding, a few studies focus on system designs for potentially non-cooperative agents [Bnaya *et al.*, 2013; Strawn and Ayanian, 2021] where those agents can pretend to be crashed. However, they do not provide safe paths as presented in this paper.

10.7.3 Failure Detector

The notion of a failure detector is inspired by a popular abstraction in theoretical distributed algorithms [Chandra and Toueg, 1996], introduced to enable consensus solvability in an asynchronous setting. With respect to the original concept, this paper assumes the detector to be both *accurate* (i.e., it never suspects correct agents) and *complete* (i.e., it always suspects crashed agents). Removing these assumptions is an interesting

future direction. Contrary to the seminal paper, we assume failure detectors to be *local*, providing localized information about failures, rather than *global*, providing information about all agents.

10.8 Concluding Remarks

This chapter studied a graph path planning problem for multiple agents that may crash at runtime, and block part of the workspace. Different from conventional MAPP studies, each agent can change its executing path according to local crash failure detection; hence a collection of paths constitute a solution for each agent. This chapter presented a safe approach to ensure that correct agents reach their destinations regardless of crash patterns, including a series of theoretical analyses.

Similar to OTIMAPP, MAPPCF exemplifies execution methods such that offline planning guides reactive execution. Therefore, it is possible to perform decentralized execution, provided that failure detectors are available, e.g., by [Kameyama *et al.*, 2021]. Then, MAPPCF can overcome the inherent limitations of centralized execution.

10.8.1 Interesting Directions

Some interesting directions for the development of MAPPCF are listed below.

- **Developing complete algorithms** that can address the crash awareness differences among agents.
- **Optimization:** The chapter focused on the feasibility problem (i.e., the decision problem of whether a given instance contains a solution); optimization problems are not addressed. One potential objective is minimizing the worst-case makespan (i.e., the maximum traveling time), which is convenient to practical situations.
- **Global Failure Detector:** The chapter assumed that an agent detects crashes only when it is adjacent to crashed agents. Extending this observation range might improve the planning success rate because agents can determine its behavior based on supplementary knowledge about crashes. On the other hand, collecting crash information at a wider range is likely to induce a delay in propagating this information further away, inducing more inconsistent observations between agents. Of course, if the crash information propagation is immediate and the span of the observation is the entire graph, MAPPCF becomes a centralized problem, since every agent has the same information.

Part III

Representation

Image by Clker-Free-Vector-Images / Pixabay License

Chapter 11

Building Representation from Learning

From this chapter, we consider the representation perspective, ultimately aiming to solve multi-robot motion planning (MRMP). As discussed in the dissertation strategy described in Chap. 3.6.6, the primary challenge of MRMP is how to represent the environment for subsequent planning algorithms so as to derive collision-free paths efficiently. This chapter, therefore, studies the representation of multi-agent path planning (MAPP) in continuous 2D spaces. The proposed approach is a *machine learning (ML)-based* method that constructs *cooperative timed roadmaps (CTRM)*. CTRM originally appeared in [Okumura *et al.*, 2022c].[1]

This chapter studies a simplified version of MRMP, therefore, the problem studied is called "MAPP in continuous spaces."

11.1 Chapter Overview

The objective of the chapter is to see the power of representation for MAPP, embodied as CTRM, built by a data-driven approach.

11.1.1 Representation Issue

Recall that one possible approach to MAPP in continuous spaces is *two-phase planning*, consisting of the following two phases:

1. Approximate the spaces by constructing graphs called *roadmaps*.
2. Derive collision-free paths using MAPF algorithms on those roadmaps.

While such approaches have been widely used for single-agent planning [LaValle, 2006], doing the same for multiple agents is non-trivial. This is primarily due to the necessity of constructing sparse roadmaps; otherwise, dense roadmaps would make it dramatically difficult to find collision-free paths due to the necessity to manage a higher number of collisions. Nevertheless, there is a tradeoff between roadmap density and solution quality (see Fig. 3.11), that is, roadmaps should be sufficiently dense to ensure a high planning success rate and better solutions. Consequently, we need to consider what are and how to construct such roadmaps.

[1] Work done during an internship at OMRON SINIC X. I really enjoyed the internship and collaboration!

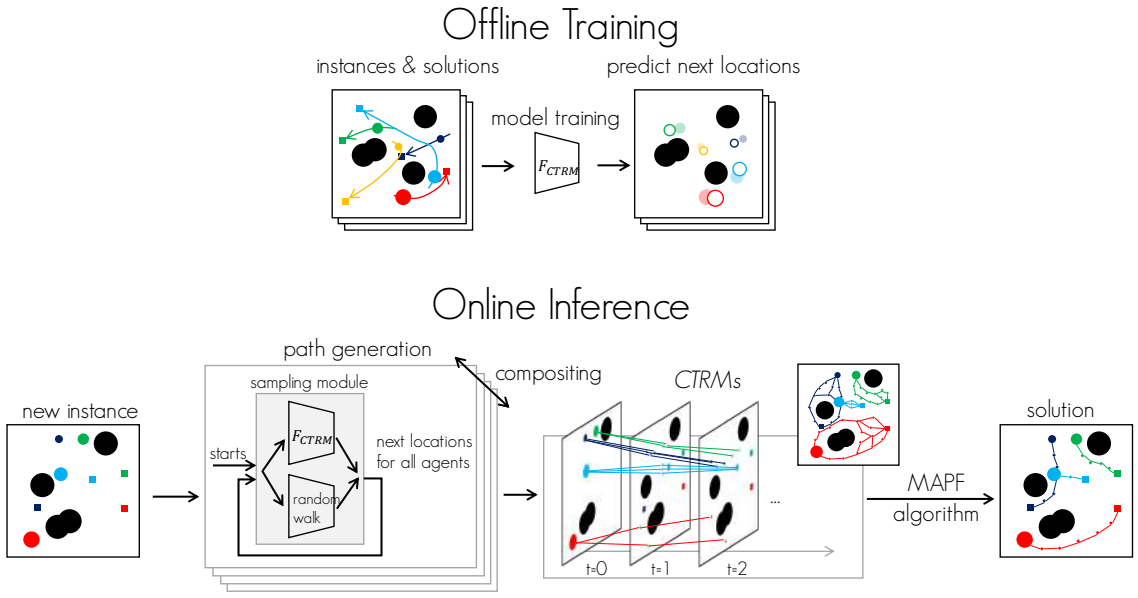


Figure 11.1: Learning to construct CTRM. *upper*: From MAPP demonstrations, we learn a conditional variational autoencoder (CVAE) that predicts how agents move to their goals while avoiding collisions with others. *lower*: For a new problem instance, we use the learned model as a vertex sampler F_{CTRM} to construct CTRM, on which an MAPF algorithm is invoked to derive solution paths efficiently.

11.1.2 What is CTRM

To handle this tradeoff, this chapter presents a novel concept of graph representations of the space called *cooperative timed roadmaps (CTRM)*. CTRM consists of directed acyclic graphs constructed in the following fashion.

- *Agent-specific*: Each CTRM is specialized for respective agents to focus on their important locations.
- *Cooperative*: Each CTRM is aware of the behaviors of other agents so as to make it easier for subsequent planning to find collision-free paths.
- *Timed*: Each vertex in CTRM is augmented in the time direction to represent not only "where," as commonly done in conventional roadmaps, but also "when," because solutions of MAPP are a list of "timed" paths.

By considering these properties collectively, CTRM aims to provide a small search space that still contains plausible solutions.

11.1.3 How to Construct CTRM

The proposed approach to construct CTRM is machine learning (ML)-based. Figure 11.1 outlines the workflow. Suppose that a collection of MAPP demonstrations is given, which is a pair of problem instances and plausible solution paths. These demonstrations can be obtained by intensive offline computation with conventional roadmaps, e.g., those constructed with uniform random sampling. Next, the ML model learns from how agents behave cooperatively at each unit of time (i.e., timestep), which is implicitly embedded in the demonstrations, to predict how they will move in the next timesteps (Fig. 11.1; *upper*). Then, for a new, previously unseen problem instance, the learned model serves as a *vertex sampler* that samples a small set of agent-specific vertices (i.e., space-time pairs) for generating multiple solution path candidates. These candidate paths are then

composited and constitute CTRM. As a result, CTRM can serve a smaller but promising search space and enable the planning to derive a solution more efficiently than when using conventional roadmaps (Fig. 11.1; *lower*).

Machine Learning Model. Technically, the proposed ML-model extends a class of generative models called the conditional variational autoencoder (CVAE) [Sohn *et al.*, 2015], a popular choice for ML sampling-based motion planning [Ichter *et al.*, 2018; Kumar *et al.*, 2019], to learn a conditional probability distribution of the vertices of CTRM for each agent, given the observations of multiple agents. The model can work with an arbitrary number of agents and even within a heterogeneous setting where agents are diverse in their spatial size and motion constraints.

11.1.4 Performance

An extensive evaluation of CTRM was performed, on a variety of MAPP problems with several different setups in terms of the number of agents (21–40), the presence of obstacles, and the heterogeneity in agent sizes and motion speeds. The results consistently demonstrate that, compared to standard roadmap construction strategies, planning by learning to construct CTRM is several orders of magnitude more efficient in the planning effort (e.g., assessed by search node expansions or runtime), while maintaining a comparable planning success rate and solution quality, with acceptable overheads.

11.1.5 Chapter Organization

- Chapter 11.2 re-provides formulation of MAPP. This is a limited version of that presented in Chap. 3.3.1.
- Chapter 11.3 explains how to train the ML model and its components.
- Chapter 11.4 describes the methodology to construct CTRM using the trained model.
- Chapter 11.5 evaluations path planning using CTRM while comparing other roadmap construction methods.
- Chapter 11.6 reviews studies closely related to CTRM.
- Chapter 11.7 concludes the chapter.

The code is available at `https://omron-sinicx.github.io/ctrm/`.

11.1.6 Notations and Assumptions

$\perp$	not found, undefined
$D_i = (V, E)$	timed roadmap (DAG) for agent i
I	problem instance
$\mathcal{W}$	2D workspace
$A = \{1, 2, \dots, n\}$	a set of agents
$\mathcal{R}_i(q) \subset \mathcal{W}$	occupied region by agent i at position $q \in \mathcal{W}$
$\mathcal{O} \subset \mathcal{W}$	obstacle
$\mathcal{C}_i^{\mathbf{free}}, \mathcal{C}_i^{\mathbf{obs}} \subseteq \mathcal{W}$	free and obstacle spaces for i
$\mathcal{M}_i$	motion constraints
$s_i, g_i \in \mathcal{W}$	start and goal of i
Π_i	path for i

Caution

- This chapter uses the representation by paths. *Those indexes start from zero*.
- The assumed problem in this chapter is a limited version of MRMP defined in Chap. 3.3.1. For instance, this chapter assumes that the configuration spaces of each agent are equivalent to the workspace.

11.2 Preliminaries

11.2.1 Problem Formulation

Problem Instance. We consider a problem of path planning for a team of n agents $A = \{1, 2, \dots, n\}$ in the 2D continuous space $\mathcal{W} \subset \mathbb{R}^2$. Each agent i has a body modeled by a convex region $\mathcal{R}_i(q) \in \mathcal{W}$ around its position $q \in \mathcal{W}$, which remains fixed regardless of position but may vary from agent to agent. The space $\mathcal{W}$ may contain a set of convex obstacles $\{\mathcal{O}_1, \dots, \mathcal{O}_m\} \subset \mathcal{W}$, where $\mathcal{O} = \bigcup_{j \in \{1,\dots,m\}} \mathcal{O}_j$. Then, the obstacle space for an agent i is represented by $\mathcal{C}_i^{\text{obs}} = \mathcal{O} \ominus \mathcal{R}_i(\mathbf{0})$, where $\ominus$ is the Minkowski difference and $\mathbf{0}$ is the origin of $\mathbb{R}^2$. The free space for i is $\mathcal{C}_i^{\text{free}} = \mathcal{W} \setminus \mathcal{C}_i^{\text{obs}}$. A trajectory for agent i is defined by a continuous mapping $\sigma_i : \mathbb{R}_{\geq 0} \mapsto \mathcal{W}$. Each agent i has its motion constraints $\mathcal{M}_i$, e.g., each agent has a maximum velocity and moves based on a constant acceleration model. A *problem instance* of MAPP is defined by a tuple $I = (A, W, \mathcal{O}, \mathcal{R}, \mathcal{M}, \mathcal{S}, \mathcal{G})$, where $\mathcal{R} = (\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n)$ and $\mathcal{M} = (\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_n)$. $\mathcal{S} = (s_1, \dots, s_n \mid s_i \in \mathcal{W})$ and $\mathcal{G} = (g_1, \dots, g_n \mid g_i \in \mathcal{W})$ are a set of initial and goal positions, respectively.

Solution. A *solution* for the problem instance is a tuple of n trajectories $(\sigma_1, \dots, \sigma_n)$ that satisfy the following four conditions:

- endpoint: $\sigma_i(0) = s_i \wedge \sigma_i(t_{\text{end}}) = g_i,\ \forall i \in A$
- obstacle-free: $\sigma_i(\tau) \in \mathcal{C}_i^{\text{free}},\ 0 \leq \tau \leq t_{\text{end}}, \forall i \in A$
- inter-agent collision-free: $\mathcal{R}_i\big(\sigma_i(\tau)\big) \cap \mathcal{R}_j\big(\sigma_j(\tau)\big) = \emptyset,\ \forall i, j \in A, i \neq j, 0 \leq \tau \leq t_{\text{end}}$
- constraints-aware: σ_i satisfies constraints of $\mathcal{M}_i$

Here, $t_{\text{end}} \in \mathbb{R}_{\geq 0}$ denotes the makespan. Under these conditions, the quality of a solution is measured by *sum-of-costs*: $\sum_{i \in A} t_i$ where $\forall i \in A, t_i \leq \tau \leq t_{\text{end}} : \sigma_i(\tau) \in g_i$.

11.2.2 Assumed Artifacts

Local Planner. As assumed in many studies of planning in continuous spaces [LaValle, 2006], the chapter assumes that each agent i has a *local planner* $\texttt{connect}_i$. Given two positions $q^{\text{from}}, q^{\text{to}} \in \mathcal{W}$, this function returns a collision-free continuous mapping $\mu : [0, 1] \mapsto \mathcal{C}_i^{\text{free}}$ that starts from q^{from} and ends at q^{to} while following $\mathcal{M}_i$. In other words, $\mu(0) = q^{\text{from}}$, $\mu(1) = q^{\text{to}}$, and $\mu(\tau) \in \mathcal{C}_i^{\text{free}}$ for $0 \leq \tau \leq 1$. When such mapping does not exist, $\texttt{connect}_i$ returns $\perp$ as "not found." For instance, $\texttt{connect}_i$ may return $(1 - \tau)q^{\text{from}} + \tau q^{\text{to}}$ or Dubins paths [Dubins, 1957].

Timed Roadmaps. To solve MAPP problems, our approach relies on a *timed roadmap* D_i for each agent i, which approximates the original space $\mathcal{C}_i^{\text{free}}$ with a finite set of vertices augmented in the time direction, similar to [Erdmann and Lozano-Perez, 1987; Fraichard, 1998]. Each timed roadmap D_i is defined as a directed acyclic graph (DAG) $D_i = (V_i, E_i)$, where each vertex $v = (p, t) \in V_i$ is a pair of space $p \in \mathcal{W}$ and discrete time

$t \in \mathbb{N}_{\geq 0}$, representing that an agent i is at location p at timestep t. An edge $(u, v) \in E_i$ exists only when $u = (p, t)$ and $v = (p', t+1)$. The roadmap D_i is regarded as *consistent* with agent i when:

- $p \in \mathcal{C}_i^{\text{free}}$ for all $(p, t) \in V_i$, and
- the local planner returns a continuous mapping for all $\big((p, t), (p', t+1)\big) \in E_i$.

Multi-Agent Path Planner. On the tuple of timed roadmaps that are consistent with respective agents, $(D_1, \dots, D_n)$, a *multi-agent path planner* is invoked to find a tuple of paths defined on discrete and synchronized time: $\boldsymbol{\Pi} = (\Pi_1, \dots, \Pi_n)$, where $\Pi_i = \big(\Pi_i[0], \dots, \Pi_i[\lceil t_{\text{end}} \rceil]\big) \in \big(\mathcal{C}_i^{\text{free}}\big)^{\lceil t_{\text{end}} \rceil}$. We can then obtain a trajectory σ_i that meets the solution conditions by applying the local planner that interpolates between consecutive points in Π_i. The local planner is also used to check the inter-agent condition. Any typical MAPF algorithm, such as CBS [Sharon *et al.*, 2015], PP [Erdmann and Lozano-Perez, 1987; Silver, 2005], or algorithms presented in Part I, is potentially applicable for a multi-agent path planner.

11.3 Learning Generative Model

The main technical challenge of the chapter lies in the construction of timed roadmaps for each agent in a way that effectively reduces the computational effort of multi-agent path planners. We want the roadmaps to provide the planners with a small search space that contains a solution path with plausible quality. To this end, the chapter argues that roadmaps should be "cooperative" in the sense that planners can easily find a collision-free path by taking into account the presence of other agents during the roadmap construction. This leads to the proposed CTRM, which are agent-specific timed roadmaps aware of agent interactions.

11.3.1 Entire Process

This chapter casts roadmap construction as an ML problem. Suppose we are given a collection of MAPP demonstrations consisting of problem instances and their solutions. The solutions can be obtained by offline intensive computation of MAPF algorithms on sufficiently dense roadmaps constructed with uniform random or grid sampling (Fig. 11.2a). Using MAPP demonstrations as training data, a generative model can learn an approximation of a probability distribution for vertices constituting a solution path for each agent. The learned model can then be used as a vertex sampler to construct CTRM that are more effective for solving a new, previously unseen problem instance efficiently.

11.3.2 ML-Model

Feature Extraction. As a generative model, the conditional variational auto-encoder (CVAE) [Sohn *et al.*, 2015] is extended, which is known to be effective for sequence modeling and prediction [Salzmann *et al.*, 2020; Ivanovic *et al.*, 2021]. The proposed CVAE model takes a feature vector $\boldsymbol{x}$ extracted from the observations of agent i and its neighbors at current timestep t in a solution $\boldsymbol{\Pi}$. This provides a conditional probability distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ of the vector $\boldsymbol{y}$ that informs how that agent should move in the next timestep $t+1$ (Fig. 11.2b). Specifically, using a pair of positions $\Pi_i[t]$ and $\Pi_i[t+1]$ extracted from solution path Π_i, the objective is to learn the distribution of $\boldsymbol{y} = \xi(\Pi_i[t+1] - \Pi_i[t])$, where $\xi(v) = \Big[|v|, (v/|v|)^\top\Big]^\top$ is the magnitude and relative direction of $\Pi_i[t+1]$ with respect to $\Pi_i[t]$.

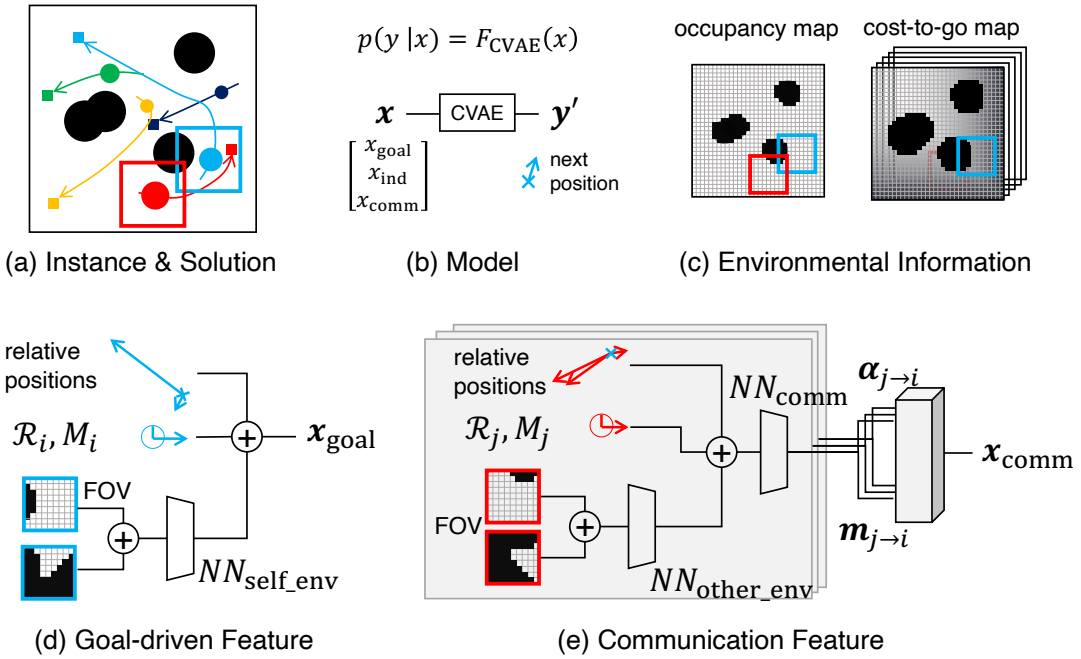


Figure 11.2: The model architecture and its components. "⊕" represents the concatenation of multiple vectors.

Encoder and Decoder. The ML model consists of two neural networks: an *encoder* and a *decoder*. The encoder $\text{Enc}(\boldsymbol{x};\theta)$, parameterized by θ, takes $\boldsymbol{x}$ as input to produce a conditional probability distribution $p_\theta(\boldsymbol{z} \mid \boldsymbol{x})$. The variable $\boldsymbol{z}$ is referred to as a *latent variable* that represents $\boldsymbol{x}$ in a lower-dimensional space. Then, the decoder $\text{Dec}(\boldsymbol{x},\boldsymbol{z};\phi)$, parameterized by ϕ, accepts $\boldsymbol{x}$ and $\boldsymbol{z}$ drawn from $p_\theta(\boldsymbol{z} \mid \boldsymbol{x})$ to form another conditional distribution $p_\phi(\boldsymbol{y} \mid \boldsymbol{x},\boldsymbol{z})$. The output distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ can be obtained by $p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{\boldsymbol{z}} p_\theta(\boldsymbol{z} \mid \boldsymbol{x}) p_\phi(\boldsymbol{y} \mid \boldsymbol{x},\boldsymbol{z})$. Doing so is particularly effective when $\boldsymbol{x}$ is high-dimensional, as in our case.

Using ML-Model as a Sampler. Given a collection of input features $\boldsymbol{x}_k$ and ground-truth outputs $\boldsymbol{y}_k$, i.e., $\{(\boldsymbol{x}_k,\boldsymbol{y}_k)\}_{k=1}^K$, extracted across agents and timesteps from problem instances in the training data, we can train the encoder and decoder jointly in a supervised learning fashion (see Appendix E.1 for the details). Once learned, CVAE can be used as a vertex sampler taking $\boldsymbol{x}$ as input to yield a particular sample $\boldsymbol{y}'$ in accordance with the learned probability distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$. In the following sections, this sampling process with the learned CVAE is referred to as a sampler function $\boldsymbol{y}' \sim \mathsf{F}_{\text{CTRM}}(\boldsymbol{x})$.

11.3.3 Features

Extracting informative features to form $\boldsymbol{x}$ is essential for constructing effective CTRM. An inherent challenge in MAPP is that each agent needs to move toward its goal while avoiding collisions with other agents that are also in motion. The proposed ML model consists of three types of features:

- *Goal-driven* features that inform agent i of the way to its goal.
- *Communication* features extracted using an attention network that encodes the information of other agents.
- An *indicator feature* that takes into account high-level choices for which direction to move in when obstacles and other agents are present.

Goal-driven Features

The most basic feature that helps agent i move toward its goal g_i is the magnitude and relative direction of g_i with respect to the current position $\Pi_i[t]$, i.e., $\xi(g_i - \Pi_i[t])$. The features also include a single-step motion history $\xi(\Pi_i[t-1] - \Pi_i[t])$, parameters of body region $\mathcal{R}_i$, and motion constraints $\mathcal{M}_i$, as they affect how agent i moves.

Furthermore, environmental information is encoded into the field of view (FOV) of agent i around its position $\Pi_i[t]$ to take into account nearby obstacles. To do so, first, the original world $\mathcal{W} \subset \mathbb{R}^2$ is discretized by the L^2 grid, where $L \in \mathbb{N}_{>0}$. Then the FOV is created around $\Pi_i[t]$ with size l^2, where $l \in \mathbb{N}_{>0}$. Within the FOV, two binary maps are extracted (illustrated in Fig. 11.2c: *local occupancy map*), indicating if each grid cell is occupied by obstacles and a *cost-to-go-map* that shows if each cell is closer to the goal compared to the current position, pre-computed by means of a breadth-first search on the grid. These two maps are fed to a neural network $\mathtt{NN}_{\text{self\_env}}$ to transform them into a compact vector. As shown in Fig. 11.2d, these features are then concatenated to form a goal-driven feature vector $\boldsymbol{x}_{\text{goal}}$.

Communication Features

To learn the cooperative behaviors between agents observed in ground-truth paths, it is critical to extract features about other agents j, such as:

- where they are in the current and previous timesteps, i.e., $\xi(\Pi_j[t] - \Pi_i[t]), \xi(\Pi_j[t-1] - \Pi_i[t])$,
- where they will be moving, i.e., their goal position $\xi(g_j - \Pi_i[t])$, and
- how their trajectory could be affected by agent body $\mathcal{R}_j$ and motion constraints $\mathcal{M}_j$.

The features also take into account the occupancy and cost-to-go maps around the position of agent j by feeding them into a neural network $\mathtt{NN}_{\text{other\_env}}$ to be represented by a compact vector. Let $\boldsymbol{x}_{j\to i}$ be a feature vector concatenating all this information about agent j with respect to agent i, as shown in Fig. 11.2e.

Suppose now that a collection of features $\{\boldsymbol{x}_{j\to i} \mid j \in \mathcal{N}_i\}$ is given for all the agents nearby target i, where $\mathcal{N}_i$ is a set of the predefined number of neighboring agents of i. The question then is how to aggregate them as a part of feature vector $\boldsymbol{x}$. Obviously, just concatenating them all is not scalable and would not be able to deal with problem instances affecting variable numbers of agents. Instead, this study leverages the recent progress in multi-agent interaction modeling that deals with agent communications using an attention network [Hoshen, 2017]. Specifically, $\boldsymbol{x}_{j\to i}$ is fed to a neural network $\mathtt{NN}_{\text{comm}}$ that outputs two variables: attention vector $\boldsymbol{\alpha}_{j\to i}$ and message vector $\boldsymbol{m}_{j\to i}$. Message vectors are then aggregated while weighted using attention vectors to provide a communication feature vector $\boldsymbol{x}_{\text{comm}}$, as follows:

$$\begin{aligned} \boldsymbol{x}_{\text{comm}} &:= \sum_{j\in\mathcal{N}_i} \boldsymbol{m}_{j\to i} \cdot w_{j\to i}, \\ w_{j\to i} &:= \frac{\exp\left[-\|\boldsymbol{\alpha}_{j\to i} - \boldsymbol{\alpha}_{i\to i}\|^2\right]}{\sum_{k\in\mathcal{N}_i} \exp\left[-\|\boldsymbol{\alpha}_{k\to i} - \boldsymbol{\alpha}_{i\to i}\|^2\right]} \end{aligned} \tag{11.1}$$

where $w_{j\to i}$ is a scalar weight for the message vector $\boldsymbol{m}_{j\to i}$, which is defined by the L2 distance between two attention vectors $\boldsymbol{\alpha}_{j\to i}$ and $\boldsymbol{\alpha}_{i\to i}$ normalized across $j \in \mathcal{N}_i$ using the softmax function. With Eq. (11.1), agent i can consider message vectors only from selected agents who are close to i in terms of attention vectors.

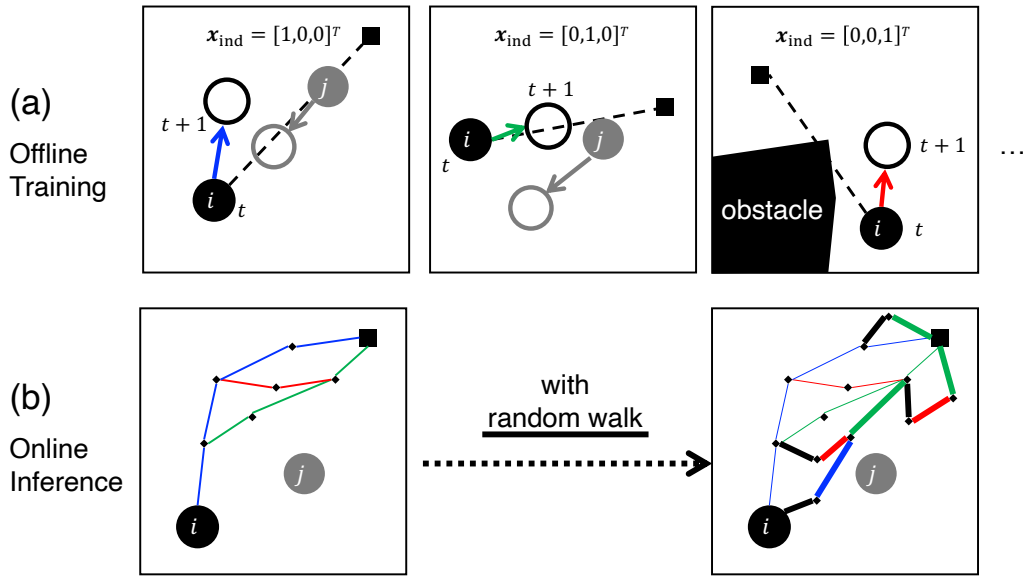


Figure 11.3: Illustration of the indicator function. Agents are represented by solid black/gray circles.

Indicator feature

Figure 11.3a depicts typical situations where we want to sample the next motion of target agent i in the presence of another agent j or obstacles in front of i. As shown in these examples, agents are often presented with multiple and discrete choices about their moving direction (such as moving left, straight, or right) depending on the layout of the surrounding agents and obstacles. As such, it is ideal that the sampling of the next motions is affected by such high-level choices, which is indeed crucial for constructing effective CTRM, as we will empirically see in the experiments.

To this end, the study proposes augmenting an input feature $\boldsymbol{x}$ with a discrete feature called an *indicator feature* $\boldsymbol{x}_{\text{ind}}$ that is learned to indicate promising choices of the next moving directions based on the current observations. Specifically, $\boldsymbol{x}_{\text{ind}}$ is defined by the relative direction of $\Pi_i[t+1]$ from $\Pi_i[t]$ with respect to the goal direction $g_i - \Pi_i[t]$, in the one-hot form such as left, straight, or right, respectively indicated by $[1,0,0]^\top$, $[0,1,0]^\top$, and $[0,0,1]^\top$. Concrete implementations are presented in Chap. 11.5.2. While calculating this feature from ground-truth path Π_i in the training phase, the method also learns a neural network $\texttt{NN}_{\text{ind}}$ taking the concatenation of $\boldsymbol{x}_{\text{goal}}$ and $\boldsymbol{x}_{\text{comm}}$ as input to predict it. The learned network is then used in the inference phase to provide $\boldsymbol{x}_{\text{ind}}$, where $\Pi_i[t+1]$ is unknown.

Putting Everything Together

Finally, all features are concatenated to form $\boldsymbol{x}$, i.e., $\boldsymbol{x} = \left[\boldsymbol{x}_{\text{goal}}^\top, \boldsymbol{x}_{\text{comm}}^\top, \boldsymbol{x}_{\text{ind}}^\top\right]^\top$. Note that the neural networks used for the feature extraction, i.e., $\texttt{NN}_{\text{self\_env}}$, $\texttt{NN}_{\text{other\_env}}$, $\texttt{NN}_{\text{comm}}$, and $\texttt{NN}_{\text{ind}}$, can be trained end-to-end with the CVAE.

11.4 Constructing Roadmaps with Learned Model

This section explains how to construct CTRM while exploiting the learned model as a vertex sampler.

11.4.1 Algorithm Overview

Algorithm 11.1 previews the proposed method, taking a problem instance I as input and then building CTRM consistent with respective agents, i.e., $D = (D_1, D_2, \ldots, D_n)$. It generates a sequence of vertices (i.e., path) with a maximum length $t_{\max} \in \mathbb{N}_{>0}$ for each agent using `sample_next_vertex` (explained in Chap. 11.4.2), which is repeated $n_{\text{traj}} \in \mathbb{N}_{>0}$ times to construct CTRM. Each path being generated is temporally stored in the table loc, where $loc_i[t]$ refers to the location of agent i sampled for timestep t. This table is used by the learned model in `sample_next_vertex` to sample the next vertices for each agent while being aware of the locations of other agents, that is, $loc_i[t]$ is used as a proxy of $\Pi_i[t]$ for the feature extraction described in Chap. 11.3.3.

Algorithm 11.1 `construct_CTRM`.

input: problem instance I

output: a tuple of timed roadmaps $D = (D_1, D_2, \ldots, D_n)$

hyperparameters: $n_{\text{traj}} \in \mathbb{N}_{>0}, t_{\max} \in \mathbb{N}_{>0}$

1: $D_i \leftarrow \Big(\{(s_i, 0)\}, \emptyset\Big)$: for each $i \in A$
2: $t_{\text{makespan}} \leftarrow 0$
3: **for** $j = 1, \ldots, n_{\text{traj}}$ **do**
4: initialize table loc ▷ *$loc_i[t]$ is location of agent-i at timestep t*
5: $loc_i[0] \leftarrow s_i$: for each $i \in A$
6: **for** $t = 1, \ldots, t_{\max} - 1$ **do** ▷ *create one set of trajectories*
7: **for** $i = 1, \ldots, n$ **do**
8: $p \leftarrow$ `sample_next_vertex`(I, t, i, loc) ▷ $p \in C_i^{free}$
9: $q \leftarrow$ `find_compatible_vertex`(I, t, i, p, D_i)
10: **if** $q = \perp$ **then**
11: $loc_i[t] \leftarrow p$; `insert`(p, t, D_i)
12: **else**
13: $loc_i[t] \leftarrow q$
14: **if** `check_reachability_to_goals`(I, t, loc) **then**
15: $t_{\text{makespan}} \leftarrow \max\{t + 1, t_{\text{makespan}}\}$; **break**
16: `insert`(g_i, t, D_i): for each $t = 1, \ldots, t_{\text{makespan}}$, $i \in A$
17: **return** $D = (D_1, \ldots, D_n)$

During each path generation, the `insert` function is invoked to add a sampled vertex (p, t) to the current CTRM $D_i = (V_i, E_i)$ and update E_i properly (Line 11). This procedure is done only when the algorithm confirms that the sampled vertex is not compatible with those it already has in V_i using `find_compatible_vertex` (see Chap. 11.4.3). Further, it keeps updating t_{makespan} to be the smallest number of timesteps to move all the agents to their goals by checking whether they can reach their goals from their last locations with the `check_reachability_to_goals` function (Line 14). After all the iterations, the algorithm inserts each agent's goal g_i into D_i for time $t = 1, \ldots, t_{\text{makespan}}$ (Line 16) and returns CTRM D.

11.4.2 Combining Sampling from Learned Model with Random Walk

In the sub-routine `sample_next_vertex`, presented in Alg. 11.2, a learned model $\mathsf{F}_{\mathrm{CTRM}}$ is used to sample vertices such that CTRM can efficiently cover a variety of possible solution paths. A sample is mainly obtained from the trained model $\mathsf{F}_{\mathrm{CTRM}}$ (Line 2), but the algorithm also introduces a random walk centered at the last location (Line 5) with the probability of $1 - p_{\mathrm{model}}$. Once a vertex (p, t) is sampled either from the model or randomly, the algorithm validates whether this is reachable from the current location $loc_i[t-1]$ using the local planner. If p is invalid, it repeats the random-walk sampling up to the maximum times specified for resampling, and returns $p = loc_i[t-1]$ (which is trivially valid) if p is still invalid.

Algorithm 11.2 `sample_next_vertex`**.**

input: instance I, timestep t, agent i, location table loc

output: one sample $p \in \mathcal{C}_i^{\mathrm{free}}$

hyperparameters: $0 \leq p_{\mathrm{model}} \leq 1$, $n_{\mathrm{retry}} \in \mathbb{N}$

1: **if** with probability p_{model} **then**
2: $\quad p \sim \mathsf{F}_{\mathrm{CTRM}}(\cdot)$
3: $\quad$ **if** `valid_edge`$(loc_i[t-1], p, I, i)$ **then return** p
4: **for** $k = 1 \ldots n_{\mathrm{retry}}$ **do**
5: $\quad p \leftarrow$ random walk from $loc_i[t-1]$
6: $\quad$ **if** `valid_edge`$(loc_i[t-1], p, I, i)$ **then return** p
7: **return** $loc_i[t-1]$

Incorporating Random Walk. Algorithm 11.2 replaces the learned model with a random walk centered at the current location ($loc_i[t-1]$) at probability $1 - p_{\mathrm{model}}$. This contributes to improving the expressiveness of CTRM beyond what has been learned from the training data, as illustrated in Fig. 11.3b. Similar techniques are commonly introduced in learning-based sampling for SBMP [Ichter *et al.*, 2018; Ichter *et al.*, 2020; Chen *et al.*, 2020]. In practice, p_{model} is set to be low in the initial timestep and gradually increases from there.

11.4.3 Finding Compatible Vertices

At Line 9 of the roadmap construction (Alg. 11.1), the algorithm invokes `find_compatible_vertex` to search the current CTRM for a vertex that is compatible with the sampled vertex (p, t). Replacing the original vertex (p, t) with a compatible one $(q, t) \in V_i$ can reduce the time required for connectivity checking in the `insert` function and, more importantly, reduce the search space for multi-agent path planners.

The specific algorithm of `find_compatible_vertex` is presented in Chap. 11.4.3, which is partially inspired by the post-processing technique of roadmap sparsification for single-agent motion planning [Salzman *et al.*, 2014]. Intuitively, two vertices p, q are regarded as compatible when they *(i)* are spatially close enough and *(ii)* share the same connectivity to other vertices. The former is determined by whether $\|q - p\| \leq \delta \in \mathbb{R}_{\geq 0}$ holds (Line 4) In the experiments, δ was set to be the tenth of the maximum speed of each agent. The latter needs to check the relationship between the parents and children of these samples. Specifically, the algorithm initially obtains potential parents and children for a new sample p at timestep t by two functions `get_parents_candidates` and

Algorithm 11.3 `find_compatible_vertex`.

input: instance I, timestep t, agent i, location p, roadmap D

output: location $q \in \mathcal{C}_i^{\text{free}}$ or `NOT_FOUND`

hyperparameters: $\delta \in \mathbb{R}$, heuristics $\mathsf{h} : \mathcal{W} \mapsto \mathbb{R}$

1: $V_{\text{parents}} \leftarrow$ `get_parents_candidates`(p, I, t, i, D)
2: $V_{\text{children}} \leftarrow$ `get_children_candidates`(p, I, t, i, D)
3: **for** $v = (q, t') \in V_i$ s.t. $t' = t$ **do**
4: **if** $\|q - p\| > \delta$ **then continue**
5: **if** $\left(V_{\text{parents}} = \text{parents of } v\right) \wedge \left(V_{\text{children}} = \text{children of } v\right)$ **then**
6: **if** $\mathsf{h}(p) < \mathsf{h}(q)$ **then**
7: replace q of v by p
8: **return** p
9: **else**
10: **return** q
11: **if** $\left(V_{\text{parents}} \subseteq \text{parents of } v\right) \wedge \left(V_{\text{children}} \subseteq \text{children of } v\right)$ **then**
12: **return** q
13: **if** $\left(\text{parents of } v \subseteq V_{\text{parents}}\right) \wedge \left(\text{children of } v \subseteq V_{\text{children}}\right)$ **then**
14: replace q of v by p
15: replace parents of v by V_{parents}
16: replace children of v by V_{children}
17: **return** p
18: **return** `NOT_FOUND`

`get_children_candidates` (Lines 1–2), using the local planner. Then, for each vertex $v = (q, t) \in V_i$ of the current roadmap, the algorithm checks whether p and q are compatible. If so, it returns p or q based on their structures of parents and children (Lines 3–17), and otherwise returns `NOT_FOUND` (Line 18). The algorithm considers three cases for p and q:

1. p and q have the same parents and children (Lines 5–10): Select one of them based on the heuristics $\mathsf{h}(p)$, which in our experiment were given by the distance from p to the goal.
2. q contains p's edge structure (Lines 11–12): The algorithm simply returns q, as it can account for the edge structure of p.
3. p contains q's edge structure (Lines 13–17): After replacing q of v with p and updating the edge structure, the algorithm returns p.

11.5 Evaluation

Next, this section evaluates CTRM in various MAPP problems and clarifies its merits.

11.5.1 Experimental Setups

MAPP Problems. Recall that an MAPP problem instance is a tuple $I = (A, \mathcal{W}, \mathcal{O}, \mathcal{R}, \mathcal{M}, \mathcal{S}, \mathcal{G})$; see Chap. 11.2. For all the MAPP problem instances, agent bodies $\mathcal{R}$ and obstacles $\mathcal{O}$ were modeled as a circle in the 2D closed continuous space $\mathcal{W} = [0,1]^2$ to detect collisions easily. Hence, an agent body $\mathcal{R}_i \in \mathcal{R}$ is characterized by a scalar value r_i representing the radius. As motion constraints $\mathcal{M}_i \in \mathcal{M}$, it was assumed that each agent i has the maximum velocity v_i and moves linearly in two vertices (space-time pairs) at constant velocity. With these settings, 100 different instances were generated, hereafter referred to as *test* instances, for each of the following five scenarios of MAPP (see also the leftmost of Fig. 11.5).

- **Basic Scenario:** A baseline scenario referred to as ***(i)* Basic** that corresponds to a discrete setting of a 32×32 grid and contains 21–30 homogeneous agents as A and ten non-uniform obstacles as $\mathcal{O}$. For each timestep, an agent can move a maximum distance of $v_i = 1/32$. The radius of agents is set to half of their maximum speed, i.e., $r_i = 1/64$. For each problem instance, the number of agents $|A|$ was randomly determined from the range $|A| \in \{21, 22, \ldots, 30\}$. The initial positions $\mathcal{S}$, goal positions $\mathcal{G}$, and the positions of the obstacles were set randomly for each instance.
- **Variants of the Basic Scenario:** Three scenarios that each change one of the settings from those of **Basic**: ***(ii)* More Agents** with $|A| \in \{31, 32, \ldots, 40\}$, ***(iii)* No Obstacles** with no obstacles, and ***(iv)* More Obstacles** with 20 non-uniform obstacles as $\mathcal{O}$, to see how robust the proposed approach against these parameters.
- **Heterogeneous agent scenario:** An advanced scenario was included, called ***(v)* Hetero Agents**, containing $|A| \in \{21, 22, \ldots, 30\}$ agents whose size r_i and speed v_i are each multiplied randomly by $\times 1$, $\times 1.25$, or $\times 1.5$ from the original ones defined in **Basic**.

Baselines for Roadmap Construction Methods. To evaluate the effectiveness of CTRM, four other roadmap construction methods (see also Fig. 11.4) that are non-timed and non-cooperative were implemented, which have commonly been used for single-agent SBMP and previous work on MAPP in continuous spaces.

- **Random sampling (random)** that samples agent locations uniformly at random from the space. This is equivalent to a simplified version of PRM [Karaman and Frazzoli, 2011] and has been used as part of the procedure to solve MAPP [Van Den Berg and Overmars, 2005; Solis *et al.*, 2021]. The numbers of samples were set to $\{3,000; 5,000; 7,000\}$.
- **Grid sampling (grid)** like the ones the conventional MAPF assumes as a discretized environment [Stern *et al.*, 2019]. The grid sizes were set to $\{32 \times 32, 64 \times 64, 84 \times 84\}$.
- **SPARS (SPArse Roadmap Spanner algorithm)** [Dobson and Bekris, 2014], an algorithm for roadmap construction that attempts to reduce both vertices and edges. SPARS has been developed for single-agent planning and has also been used in MAPP [Hönig *et al.*, 2018b]. The implementation was obtained from the Open Motion Planning Library [Şucan *et al.*, 2012].
- **Square sampling (square)** as a variant of the random sampling focusing on the square region with its diagonal line given by the start and goal for a single agent (with a margin). This sampling was introduced as a simpler approach to providing a tuple of agent-specific roadmaps. The number of samples was determined by the length of the diagonal line times a parameter given adaptively to generate low-, middle-, and high-density roadmaps.

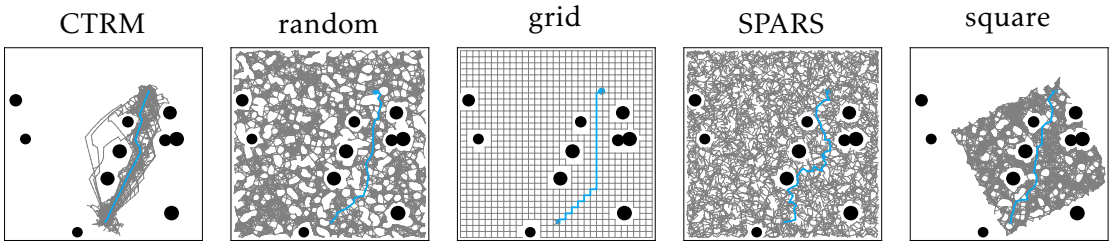


Figure 11.4: Constructed roadmaps for one agent in the Basic scenario. The blue line is a path in a solution. Parameters are: *CTRM*: $n_{\text{traj}} = 25$; *random*: 3,000 samples; *grid*: 32×32; *square*: the middle density.

More details for SPARS and square sampling are presented in Chap. 11.5.3. Given a set of samples as vertices, roadmaps were built by creating edges at two vertices where agents can travel within a single timestep given their size r_i and speed v_i parameters. For the former three methods, a single roadmap was shared across all the agents in the scenarios (*i–iv*) since their size and speed were set identically. For scenario *(v)* with heterogeneous agents, different roadmaps for individual agents were constructed, as done for CTRM and the square sampling.

Metrics. Since the objective of MAPP is to plan collision-free trajectories, the effectiveness of roadmap construction methods should be evaluated in terms of the subsequent planning results. Standard prioritized planning (PP) [Silver, 2005; Van Den Berg and Overmars, 2005] was used as the planner. The following metrics were computed.

- **Success rate of the planning:** Percentage of successful planning among 100 instances. Two cases were regarded as a failure: when the PP yielded failure or when the planning time reached a 10 min timeout.
- **Sum-of-costs:** Solution quality defined in Chap. 11.2 averaged across all problem instances that resulted in planning success.
- **The number of expanded search nodes in the planning:** Metric of the planning effort (the smaller, the better), which was also averaged across all problem instances that resulted in planning success. This metric is used as a proxy of runtime, since actual runtimes rely heavily on implementations. Specifically, the number of search nodes expanded in PP was counted.
- **Runtime:** Reference record of the execution time for the roadmap construction and the subsequent planning.

Evaluation Environment. All methods were implemented in Python with partial use of C++. The results were obtained on a desktop PC with Intel Core i7-8700 CPU and 32 GB RAM.

11.5.2 Implementation Details for CTRM

Model Architectures. All neural networks used in our method are standard multilayer perceptrons; see Chap. 11.5.3 for details. The FOV was set to $l = 19$ and $L = 160$. The number of neighboring agents of $\mathcal{N}_i$ was set to 15 based on the distance between the current locations of agents, $\Pi_i[t]$. The indicator feature $\boldsymbol{x}_{\text{ind}}$ was defined by a three-dimensional one-hot vector indicating left, straight, and right, which were determined based on whether the sine of the angle between two vectors $g_i - \Pi_i[t]$ and $\Pi_i[t+1] - \Pi_i[t]$ was in $[-1, -1/3]$, $(-1/3, 1/3]$, or $(1/3, 1]$.

Training Setups. 1,000 problem instances were prepared as training data and 100 instances for validation, where the trained model with the minimum loss on the validation data was stored and used to construct CTRM for test instances. Regardless of test scenarios, the same training and validation data were used so that the parameters followed the **Hetero-Agent** scenario. This is because the Hetero-Agent scenario naturally contains diverse agent behaviors that could also be observed in the other scenarios, which helps reduce the time needed for training with little empirical performance degradation. Note that no identical instances were shared among training, validation, and test instances. Solutions to those instances were obtained by means of roadmap construction based on random sampling with 3,000 samples and path planning by PP. Model parameters were updated using the Adam optimizer [Kingma and Ba, 2015] with the mini-batch size, the number of epochs, and the learning rate set to $(50; 1000; 0.001)$. The data creation and model training were conducted on a single workstation equipped with an Intel Xeon CPU E5-2698 v4 and NVIDIA Tesla V100 GPU to complete the procedures in a reasonable amount of time (data creation: 1 day with ×40 multiprocessing; training: 2 hours).

Weighted Loss. Prior to the experiments, it was observed that the training and validation data created as above contained many agents that trivially moved along the shortest path from their start to goal positions, while it was expected to observe actions of agents dodging each other to learn their interactions. As such data imbalance typically makes training difficult [Lin *et al.*, 2017; Lu *et al.*, 2018], therefore, a weighted loss technique was introduced. The technique gives different weights to each training sample based on the angle Δ between $g_i - \Pi_i[t]$ and $\Pi_i[t+1] - \Pi_i[t]$ with the following criterion: $1 - \exp\left(-\gamma\Delta^2\right)$, where $\gamma = 50$. Intuitively, this gave smaller weights to samples that just move agents forward.

Roadmap Construction. The parameters for constructing CTRM in Alg. 11.1 were set as follows. n_{traj} was set to $\{25, 50, 75, 100\}$, and $t_{\max}$ was 64. In `sample_next_vertex`, p_{model} was adaptively set based on timestep t with $1 - \exp\left(-\gamma t / \min(t_{\max}, t_{\text{makespan}})\right)$, where $\gamma = 5$, until each agent arrives at its goal. p_{model} was then fixed to 0.1 after agents reach their goal so that they can move around to avoid other agents still moving toward their goal.

11.5.3 Further Details of Experimental Setups

Parameters for Baselines

- **SPARS:** The implementation in the Open Motion Planning Library [Şucan *et al.*, 2012] has four hyperparameters: `dense_delta_fraction`, `spars_delta_fraction`, `stretch_factor`, and time limits. These parameters were set to 0.1, 0.01, 1.3, and 30 s, respectively.
- **Square Sampling:** The number of samples of each agent is determined by the length of the diagonal line l (between its start and goal) times a given parameter c. Specifically, this was determined by the number of c/v_i, where v_i is the maximum velocity of agent i. c was set to 50 (low), 75 (middle), and 100 (high). The margin of the square region was set to $v_i/5$.

Parameters for CTRM

All the neural networks used in the method, i.e., $\mathsf{NN}_{\text{self\_env}}$, $\mathsf{NN}_{\text{other\_env}}$, $\mathsf{NN}_{\text{comm}}$, NN_{ind}, along with the encoders $\mathsf{Enc}(\boldsymbol{x}; \theta)$, $\mathsf{Enc}(\boldsymbol{x}, \boldsymbol{y}; \psi)$ and decoder $\mathsf{Dec}(\boldsymbol{z}; \phi)$ in CVAE, were defined by a standard multilayer perceptron with two fully connected layers and 32 chan-

nels, followed by ReLU activation [Nair and Hinton, 2010] except for the last layer. The encoders and decoder also had batch normalization [Ioffe and Szegedy, 2015] between the fully connected layers and ReLU to stabilize the training. The output of $\mathtt{NN}_{\mathrm{ind}}$ was activated by the softmax function to predict the one-hot indicator feature $\boldsymbol{x}_{\mathrm{ind}}$, where the argmax function was further applied in the inference phase. The dimensions of message vectors $\boldsymbol{m}_{j\to i}$, attention vectors $\boldsymbol{\alpha}_{j\to i}$, and the latent variables of CVAE $\boldsymbol{z}$ were respectively set to 32, 10, and 64.

11.5.4 Results

Figure 11.4 visualizes examples of constructed roadmaps. Figure 11.5 shows the main results. Overall, we can see that CTRM substantially reduced the planning effort to obtain plausible solutions while maintaining a high success rate and sum-of-costs. Specific findings are summarized below.

CTRM contains solutions in a small search space. The size of the search space can be assessed by the number of vertices in each roadmap per agent and per timestep, as compared to the success rate in the second column of Fig. 11.5. CTRM has significantly fewer vertices compared to the others but still allows the subsequent planner to successfully find a solution at a high rate. The roadmaps created by the random, grid, and SPARS sampling provided much larger search spaces since they are shared among agents and should thus cover the entire space. The square sampling was a bit better than those baselines, but was consistently outperformed by CTRM.

While keeping solution quality, CTRM contributes to reducing the planning effort by several orders of magnitude. This is simply due to the small search space compared to the baselines, as shown in the third and fourth columns. The sum-of-cost metric normalized by the number of agents was comparable to the baseline methods and even better than when the number of samples for the baseline methods is limited. This is primarily because CTRM-approach learns where to sample vertices from the solution paths of MAPP demonstrations.

CTRM achieves efficient MAPP solving from the end-to-end perspective. The total runtime with the overhead from roadmap constructions is shown in the fourth column. Although the runtime relies heavily on the implementation, the proposed method can produce CTRM in a realistic computation time.[2] Once CTRM is constructed, the planning can be finished immediately, unlike the baselines, despite the use of the same planning algorithm and implementation. Even so, there is room for further optimization in the roadmap construction, including improvement to the implementation of connectivity checks between vertices, as it is often a bottleneck of SBMP [Elbanhawi and Simic, 2014].

CTRM-approach is generalizable for various scenarios. Even though the ML model was trained on just a single scenario with heterogeneous agents, the model once trained can be used for other scenarios with different numbers of agents and obstacles.

11.5.5 Ablation Study

Here, the effect of each technical component used in CTRM is further assessed. Specifically, the study evaluated degraded versions of the approach that omit

[2] The latest implementation is indeed much faster, see https://github.com/omron-sinicx/jaxmapp.

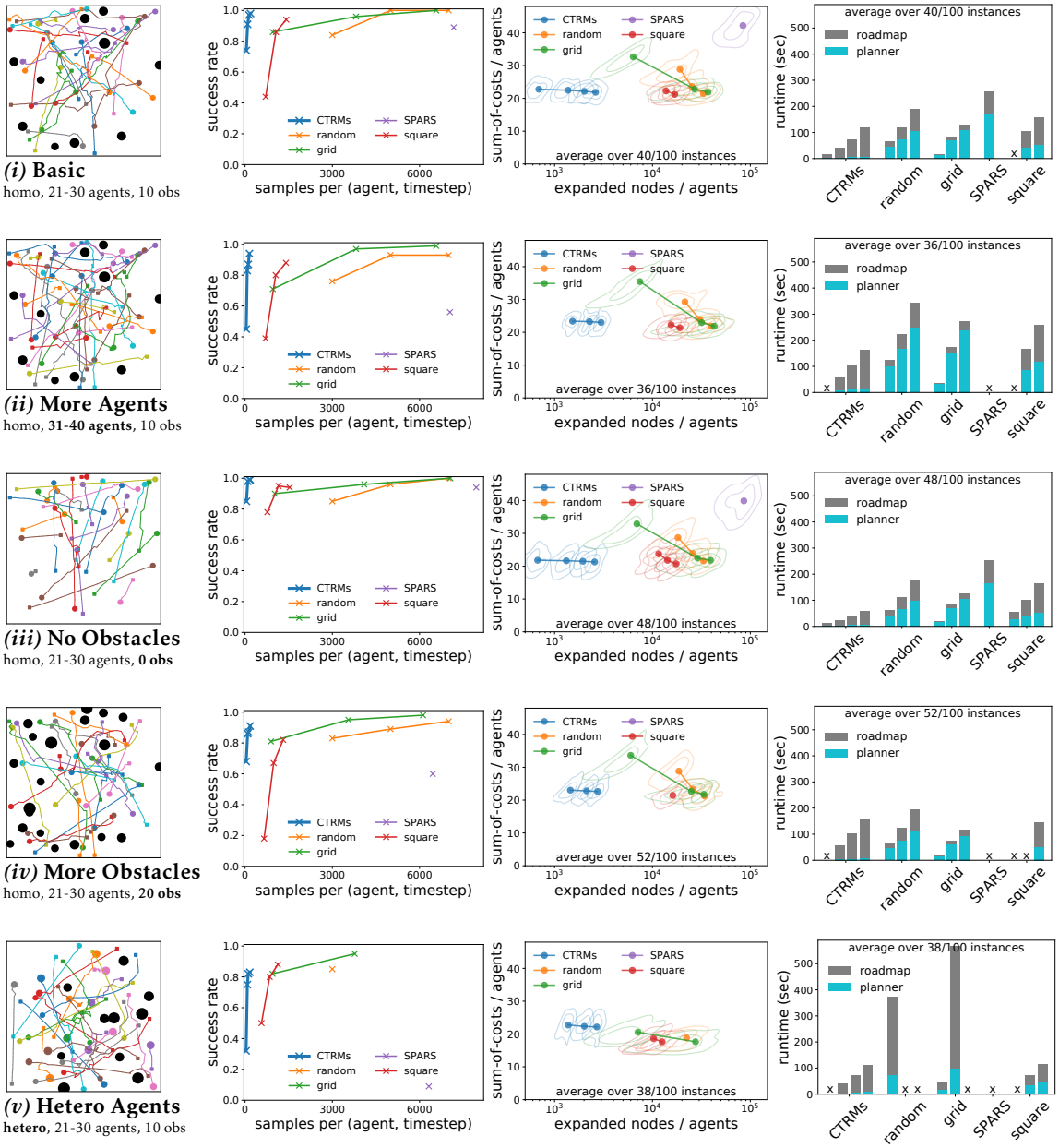


Figure 11.5: Summary of results. The rows correspond to respective scenarios. For each method, plots to the right correspond to the denser roadmaps (e.g., CTRM: $n_{\text{traj}} = 25 \rightarrow n_{\text{traj}} = 100$, grid: $32 \times 32 \rightarrow 84 \times 84$). *First column*: Solution examples planned by PP on CTRM ($n_{\text{traj}} = 25$). *Second column*: Success rate vs. the number of samples per agent and per timestep in the constructed roadmaps. *Third column*: Sum-of-costs vs. the number of expanded search nodes in the planning phase. Each value is normalized by dividing by the number of agents. *Fourth column*: Overall runtime, consisting of the planning and the roadmap construction phases. For the results in the third and fourth rows, in order to obtain meaningful insights, several methods with success rates below 70% were excluded; the scores have calculated the average of the metrics for problem instances where all the remaining methods were successful in planning. Some baselines with specific parameter settings are also omitted due to extremely long computation times (over 15 min per problem instance) from the evaluation of the Hetero Agent scenario. All these omitted methods and settings are marked with "x" in the fourth row.

- communication features (no $\boldsymbol{x}_{\text{comm}}$) or
- indicator features (no $\boldsymbol{x}_{\text{ind}}$) when learning the model, and
- no random walk (i.e., p_{model} =1) when constructing CTRM.

All the variants were evaluated on the Basic scenario in terms of success rate, sum-of-costs, and the number of expanded nodes with $n_{\text{traj}} = 25$. As shown in Table 11.1, involving the random walk is particularly critical to ensure a high success rate. This is consistent with the common practice of existing studies [Ichter *et al.*, 2018; Ichter *et al.*, 2020; Chen *et al.*, 2020] that use learned sampling and random sampling together. On top of that, the lack of communication feature $\boldsymbol{x}_{\text{comm}}$ greatly limits the success rate, sum-of-costs, and the number of expanded nodes, as agents are then required to avoid others solely by random walks. Furthermore, we can see that the indicator feature $\boldsymbol{x}_{\text{ind}}$ is equally important to improve the overall performance.

	success rate	sum-of-costs		expanded nodes	
CTRMs ($n_{\text{traj}} = 25$)	**0.80**	**21.2**	(20.3, 22.0)	**612.4**	(547.4, 674.2)
no $\boldsymbol{x}_{\text{comm}}$	0.23	28.7	(27.3, 30.2)	996.3	(923.1, 1068.5)
no $\boldsymbol{x}_{\text{ind}}$	0.33	31.3	(30.5, 32.2)	1058.6	(993.8, 1117.3)
no random walk	0.03	N/A		N/A	

Table 11.1: Results of the ablation study performed on the Basic scenario. Sum-of-costs and expanded nodes are normalized by the number of agents and averaged over 11 instances that succeeded in all methods except "no random walk" (resulting in an extremely low success rate), accompanied by 95% confidence intervals.

11.5.6 Limitations

Although CTRM generally worked well in various scenarios, it was informally confirmed that the method sometimes failed in "bug trap" situations where SBMP generally struggles [LaValle, 2006]. In general, the proposed approach is applicable only if each agent can find an obstacle-free path to reach the goal when constructing CTRM. Failing to do so will make it impossible to perform MAPF in the later stage. One possible resolution would be to introduce bi-directional path generation like a variant of RRT [Kuffner and LaValle, 2000]. Improving the overall performance by replacing the multi-agent path planner from PP such as LaCAM is another promising direction.

11.6 Related Work

This study is broadly categorized into MAPP, which has evolved in multiple directions and also shares some techniques with single-agent planning, as described below.

11.6.1 Learning-based MAPF

MAPF is a problem of MAPP in discrete spaces and has been studied extensively over the 2010s. More recently, some studies have attempted to leverage ML techniques to solve MAPF. These techniques learn from planning demonstrations collected offline to directly predict the next actions of agents given the current observations by means of imitation learning or reinforcement learning [Sartoretti *et al.*, 2019; Li *et al.*, 2020; Damani *et al.*, 2021; Ma *et al.*, 2021; Li *et al.*, 2021e]. Despite such progress, it remains challenging to determine how these techniques should be applied to MAPP in continuous spaces due to

the inherent limitation that assumes the search space to be given *a priori* (typically as a grid map).

11.6.2 MAPP in Continuous Spaces

Path planning in continuous spaces has long been studied for the single-agent case, typically in the form of sampling-based motion planning (SBMP) [Elbanhawi and Simic, 2014] which iteratively samples state points from the space to seek a solution. Popular approaches include PRM [Kavraki *et al.*, 1996], RRT [LaValle, 1998], or their asymptotically optimal versions (PRM*, RRT*) [Karaman and Frazzoli, 2011]. Most of the studies on MAPP in continuous spaces have been built on these techniques. Roadmaps (or simply pre-defined grid maps) are either constructed or given *a priori* to be shared across agents [Van Den Berg and Overmars, 2005; Čáp *et al.*, 2015; Hönig *et al.*, 2018b; Yakovlev and Andreychuk, 2017] or individually for each agent [Solis *et al.*, 2021; Gharbi *et al.*, 2009; Kumar and Chakravorty, 2012]. In contrast to these previous attempts, CTRM explores a different research direction that delves into what roadmap representations are effective for solving MAPP in continuous spaces and how they should be constructed.

11.6.3 Learning-based SBMP

The proposed approach learns how to construct effective roadmaps from MAPP demonstrations. Such a learning-based approach is gaining attention as a promising method for biased sampling in SBMP [Ichter *et al.*, 2018; Ichter *et al.*, 2020; Chen *et al.*, 2020; Qureshi *et al.*, 2020; Kumar *et al.*, 2019] to sample state points from important regions of given problem instances to derive a solution efficiently. In contrast, only a few studies have examined how learning can improve roadmaps for multi-agent cases. The most relevant work is [Arias *et al.*, 2021b], which proposed learning to bias sampling of PRMs so as to effectively avoid collisions with dynamic obstacles. However, this approach can only be applied to MAPP with homogeneous agents having the same size and motion constraints, since it generates a single roadmap shared across all agents. Moreover, it biases samples based on learned obstacle layouts (e.g., narrow passages), making it hard to run in obstacle-free or random-obstacle environments. Another related work [Henkel and Toussaint, 2020] optimizes already constructed roadmaps by means of online learning, which is orthogonal to the above studies that focus on roadmap construction.

11.7 Concluding Remarks

This chapter presented cooperative timed roadmaps (CTRM), a novel graph representation tailored to MAPP in continuous spaces. CTRM is constructed to provide the subsequent planner with a small search space for each agent while simultaneously being aware of other agents to avoid potential collisions and contain plausible solution paths. This can be done by learning a generative model from relevant demonstrations and using it as an effective vertex sampler. The experimental results demonstrate the effectiveness of CTRM for a variety of MAPP problems.

There are several interesting directions to develop CTRM, such as anytime planning, planning in continuous time, or multi-agent motion planning in higher-dimensional spaces. Among them, the next chapter dives into an advanced idea, that is, integrating representation and planning perspectives. This is a consequence of the primary observation of the chapter; *representation is important for planning*.

Chapter 12

Building Representation while Planning

The previous chapter demonstrates the power of representation. Following this direction, this chapter goes one step further by integrating planning and representation, aiming at solving multi-robot motion planning (MRMP) efficiently. The resulting algorithm is called *simultaneous sampling-and-search planning (SSSP)* and can solve various types of MRMP. SSSP originally appeared in the technical paper [Okumura and Défago, 2022a].

12.1 Chapter Overview

The aim of the chapter is to see the power of integration between planning and representation.

12.1.1 What is SSSP

SSSP is an algorithm solving MRMP. Recall that a standard approach to solve MRMP is *two-phase planning*, which first constructs roadmaps and then performs the multi-agent search, i.e., finding collision-free paths on those roadmaps. The previous chapter is indeed an example of two-phase planning. In contrast, SSSP *performs roadmap construction and multi-agent search at the same time*.

Motivation

Since two-phase planning decouples roadmap construction and multi-agent search, it always has the possibility to develop roadmaps in unnecessary regions wherein search never uses. That is, the roadmap construction unexpectedly may prepare huge search spaces. Meanwhile, combining roadmap construction and the search makes it possible to develop a small but effective search space that the search is willing to use.

Mechanism

In short, SSSP performs an exhaustive search on agent-wise roadmaps while constructing them by random walks. It is directly inspired by two algorithms listed below, respectively for SBMP and MAPF:

- *Expansive Space Trees (EST)* [Hsu *et al.*, 1997], an example of SBMP, that solves single-robot motion planning by constructing a query tree growing with random walks.

- *A* with operator decomposition* [Standley, 2010] that solves MAPF efficiently by decomposing successors in the search tree such that at most one agent takes an action, rather than all agents taking actions simultaneously.

SSSP realizes rapid MRMP by combining these two techniques.

12.1.2 Properties and Performance

Theoretical Properties

SSSP eventually finds a sequential solution for solvable instances. An MRMP solution is called *sequential* when two or more robots never take actions simultaneously. This property corresponds to probabilistic completeness in SBMP; i.e, as time passes, the probability of finding solutions approaches to one.

Empirical Performance

Extensive evaluations on various scenarios with diverse degrees of freedom and kinematic constraints demonstrate that SSSP significantly outperforms standard approaches of two-phase planning, i.e., solving more problem instances much faster. This chapter also provides a planning demo with 32 ground robots (64-DOF in total) in a dense situation.

12.1.3 Chapter Organization

- Chapter 12.2 re-organizes the problem formulation and explains assumptions to solve MRMP.
- Chapter 12.3 describes the SSSP algorithm and its analyses.
- Chapter 12.4 presents empirical results.
- Chapter 12.5 concludes the chapter.

The code and movie are available at `https://kei18.github.io/sssp`.

12.1.4 Disclaimer

Configuration. Following the SBMP literature, this chapter uses the term "configuration" as a robot state unlike Parts I and II that refer to it as a tuple of locations.

Agent vs. Robot. There is no primary difference between the terms "agent" and "robot," however, to be consistent with the literature, the term "robot" is used in this chapter.

Optimization. Throughout the chapter, since solving MRMP itself is challenging, we focus on the decision problem rather than optimization problems. Optimal SSSP is discussed at the end.

Kinodynamic Constraints. As discussed in Chap. 2.6.5, robot motions are typically categorized into kinematic or dynamics constraints. These constraints are collectively called kinodynamic constraints and limit directions that a robot can take in the configuration space. In this chapter, since this study is an early-stage attempt at combining sampling and search, *we consider only kinematic constraints and retain dynamics constraints* for simplicity. That is, SSSP is presented with kinematic constraints only, and all empirical results presented consider only kinematic constraints. However, as like SBMP algorithms

such as RRT [LaValle and Kuffner Jr, 2001] or EST [Hsu *et al.*, 1997] which are applicable to kinodynamic planning, SSSP has a high potential for kinodynamic MRMP; the discussion of kinodynamic planning will appear at the end of this chapter.

Caution

The chapter uses the representation by configurations (i.e., locations for all agents).

12.2 Preliminaries

12.2.1 Problem Formulation

Unlike the previous chapter (Chap. 11), this chapter considers MRMP presented in Chap. 3.3.1; i.e., we do not restrict the configuration spaces of robots. Recall that notations of *geometric* MRMP are as follows:

$A = \{1, 2, \ldots, n\}$	a set of robots
$\mathcal{W} \subset \mathbb{R}^3$	workspace
$\mathcal{O} \subset \mathcal{W}$	obstacles
$\mathcal{C}_i$	configuration space for robot i
$\mathcal{R}_i(q) \subset \mathcal{W}$	body of i at configuration $q \in \mathcal{C}_i$
$\mathcal{C}_i^{\mathbf{obs}}, \mathcal{C}_i^{\mathbf{free}}$	obstacle and free spaces for i
$\sigma_i : \mathbb{R}_{\geq 0} \mapsto \mathcal{C}_i$	trajectory
$q_i^{\mathbf{init}} \in \mathcal{C}_i$	initial configuration for i
$\mathcal{Q}_i^{\mathbf{goal}} \subset \mathcal{C}_i$	goal configurations for i

This chapter furthermore considers *kinematic* MRMP. Following Chap. 3.3.1, the chapter discretizes the time by introducing $\Delta \in \mathbb{R}_{>0}$ as the small amount of time. Each robot i has *local planner* $\mathtt{connect}_i$. Following kinematic constraints, this blackbox function returns an obstacle-free trajectory for the amount of Δ time that connects two configurations if exist. For instance, $\mathtt{connect}_i\left(q^{\mathrm{from}}, q^{\mathrm{to}}\right)$ may output $(1-\tau)q^{\mathrm{from}} + \tau q^{\mathrm{to}}$ or Dubins paths [Dubins, 1957]. Doing so hides the primary differences between geometric and kinematic MRMP.

Using the representation by paths, a solution of MRMP is then a tuple of paths $\boldsymbol{\Pi} = (\Pi_1, \Pi_2, \ldots, \Pi_n)$, where $\Pi_i = (q_0, q_1, \ldots, q_k \mid q_t \in \mathcal{C}_i)$ and k is common between robots, satisfying the following conditions:

- *endpoint*: $\Pi_i[0] = q_i^{\mathrm{init}} \wedge \Pi_i[-1] \in \mathcal{Q}_i^{\mathrm{goal}},\ \forall i \in A$
- *consistent path*: $\mathtt{connect}_i\Big(\Pi_i[t], \Pi_i[t+1]\Big) \neq \bot,\ t \in \{0, 1, \ldots, k-1\}, \forall i \in A$
- *inter-robot collision-free*:

$$\begin{aligned} &\mathcal{R}_i\Big(\sigma_i(\tau)\Big) \cap \mathcal{R}_j\Big(\sigma_j(\tau)\Big) = \emptyset \\ &\qquad \forall i, j \in A, i \neq j, t \in \{0, 1, \ldots k-1\}, 0 \leq \tau \leq \Delta \\ &\qquad \sigma_{\{i,j\}} = \mathtt{connect}_{\{i,j\}}\Big(\Pi_{\{i,j\}}[t], \Pi_{\{i,j\}}[t+1]\Big) \end{aligned} \tag{12.1}$$

12.2.2 Roadmap

A *roadmap* for robot i is a directed graph $G_i = (V_i, E_i)$ which approximates the original space $\mathcal{C}_i^{\mathrm{free}}$ with a finite set of vertices. Each vertex $q \in V_i$ corresponds to one configuration of $\mathcal{C}_i$, thus simply denoted as $q \in \mathcal{C}_i$. The roadmap G_i must satisfy $q \in \mathcal{C}_i^{\mathrm{free}}$ for all $q \in V_i$ and $\mathtt{connect}_i(q, q') \neq \bot$ for all $(q, q') \in E_i$. Here, E_i contains (q, q) for all $q \in V_i$ because this chapter considers planing without dynamics constraints.

12.2.3 Utility Functions

Below, four "blackbox" utility functions are introduced to solve MRMP in a practical manner. The first three are common in motion planning studies [Choset *et al.*, 2005; LaValle, 2006], whereas the last one is specific to MRMP.

Sampling. The function $\texttt{sample}_i$ randomly samples a configuration $q \in \mathcal{C}_i$, where q is not necessarily in $\mathcal{C}_i^{\text{free}}$.

Distance. The function $\texttt{dist}_i : \mathcal{C}_i \times \mathcal{C}_i \mapsto \mathbb{R}_{\geq 0}$ defines a distance of two configurations of robot i. It is not necessary for this function to consider obstacles or inter-robot collisions, however, we assume that $p = q \Leftrightarrow \texttt{dist}_i(p,q) = 0$, $\texttt{dist}(p,q) = \texttt{dist}(q,p)$, and $\texttt{dist}_i$ satisfies the triangle inequality. For instance, the experiment used Euclidean distance.

Steering. The steering function for robot i takes two configurations $q^{\text{from}}, q^{\text{to}} \in \mathcal{C}_i^{\text{free}}$ then returns a "closer" configuration $q \in \mathcal{C}_i^{\text{free}}$ to q^{to}. With a prespecified parameter $\epsilon > 0$, formally:

$$\begin{aligned} &\texttt{steer}_i\left(q^{\text{from}}, q^{\text{to}}\right) := \operatorname*{argmin}_{q \in U_\epsilon} \texttt{dist}_i\left(q, q^{\text{to}}\right) \\ &\text{where} \\ &U_\epsilon := \left\{ q \in \mathcal{C}_i^{\text{free}} \mid \texttt{dist}_i\left(q^{\text{from}}, q\right) \leq \epsilon, \texttt{connect}_i\left(q^{\text{from}}, q\right) \neq \perp \right\} \end{aligned} \tag{12.2}$$

In practice, `steer` can be implemented by binary search.

Inter-robot Collision. For two robots i, j, given four configurations $q^{\text{from}}_{\{i,j\}}, q^{\text{to}}_{\{i,j\}} \in \mathcal{C}^{\text{free}}_{\{i,j\}}$ such that $\texttt{connect}_{\{i,j\}}\left(q^{\text{from}}_{\{i,j\}}, q^{\text{to}}_{\{i,j\}}\right) \neq \perp$, the function $\texttt{collide}_{(i,j)}\left(q_i^{\text{from}}, q_i^{\text{to}}, q_j^{\text{from}}, q_j^{\text{to}}\right)$ returns `TRUE` when there is a collision if two robots simultaneously change their configurations from $q^{\text{from}}_{\{i,j\}}$ to $q^{\text{to}}_{\{i,j\}}$. Here, a *collision* is defined similarly to Eq. (12.1). The function otherwise returns `FALSE`. For convenience, the chapter uses $\texttt{collide}\left(\mathcal{Q}^{\text{from}}, \mathcal{Q}^{\text{to}}\right)$, where $\mathcal{Q}^{\{\text{from, to}\}} = \left(q_i^{\{\text{from, to}\}} \in \mathcal{C}_i^{\text{free}}\right)^{i=1\ldots n}$. This shorthand notation returns `TRUE` if and only if there is a pair (i,j) for which $\texttt{collide}_{(i,j)}\left(q_i^{\text{from}}, q_i^{\text{to}}, q_j^{\text{from}}, q_j^{\text{to}}\right)$ returns `TRUE`.

Domain Independence. To sum up, we solve MRMP using only five blackbox functions: `connect`, `sample`, `dist`, `steer`, and `collide`. Doing so makes our formalism non-restrictive to specific robotic systems, rather it is applicable to many planning domains as we will see in the experiments.

12.3 Algorithm Description

In a nutshell, SSSP performs a best-first search using operator decomposition [Standley, 2010] while simultaneously growing robot-wise roadmaps via random walks [Hsu *et al.*, 1997]. This section first explains the core idea, followed by the pseudocode, theoretical analysis of completeness, and postprocessing to obtain better solutions.

12.3.1 Core Idea

As a high-level description, SSSP constructs a search tree while expanding robot-wise roadmaps. Each search node in the tree contains a tuple of configurations $\mathcal{Q} = \left(q_j \in \mathcal{C}_j^{\text{free}}\right)^{j \in A}$

and robot i that will take the next action. When this node is selected during the search process, the algorithm does *vertex expansion* and *search node expansion* in order. The former expands the roadmap G_i for robot i by random walks from $Q[i]$. The latter creates the successors of the node by transiting the configuration of robot i from $Q[i]$ to its neighboring configurations on G_i, and then passing the turn for i to $i+1$. Figure 12.1 illustrates this procedure with initial roadmap construction explained later, while Fig. 12.2 shows an example of constructed roadmaps.

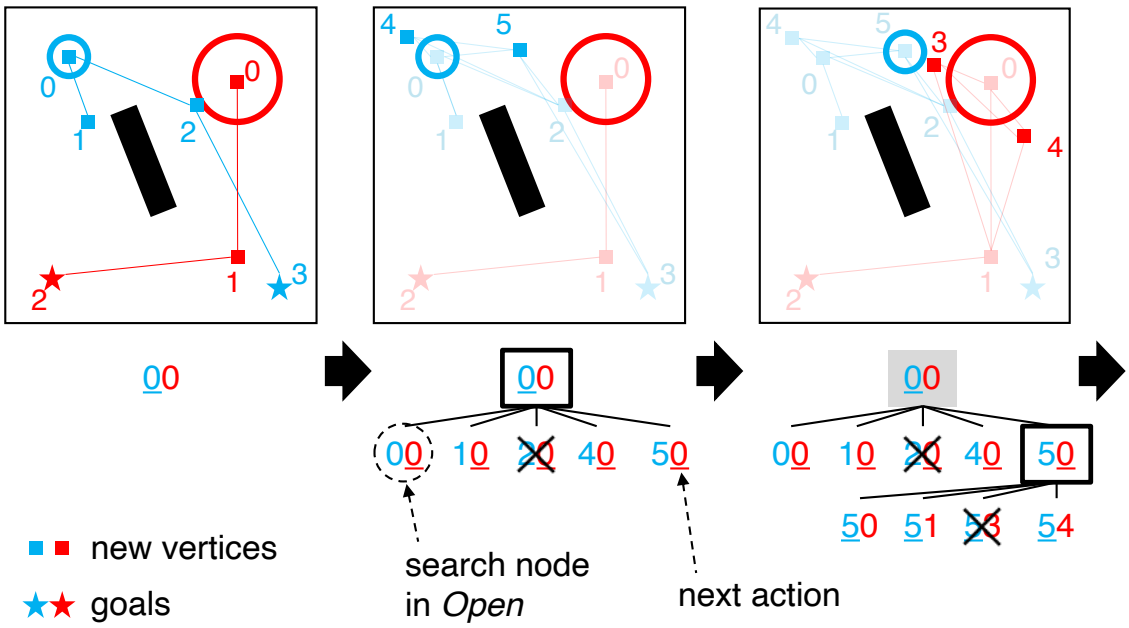


Figure 12.1: Illustration of SSSP. The algorithm progresses from left to right. *top*: Robot-wise roadmaps and search situations. Two robots are shown by colored circles and an obstacle by a black rectangle. *bottom*: Search trees. A node 'xy' corresponds to a situation where the blue and red robots are respectively at vertices 'x' and 'y,' and the blue robot will take the next action. *left*: Initial roadmaps. The search starts from the node '00.' *middle*: Vertex expansion and search node expansion for the blue robot. The node '20' is not expanded due to inter-robot collision. *right*: The red robot's turn. The search continues until all robots reach their goals.

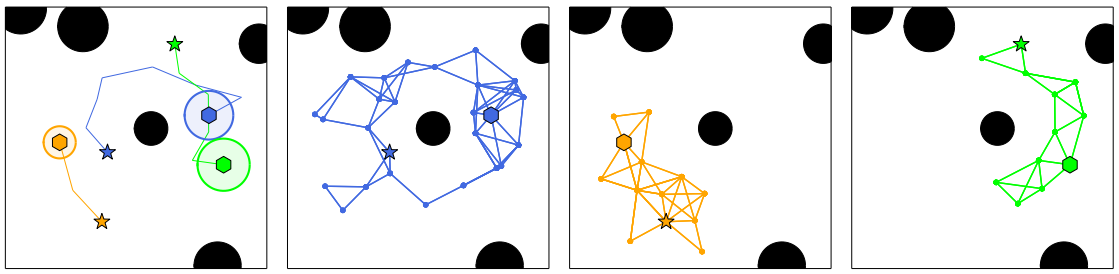


Figure 12.2: Example of constructed roadmaps for 2D point robots. A solution is depicted at the leftmost. The others show respective roadmaps for each robot.

12.3.2 Details

Algorithm 12.1 presents the pseudocode of SSSP. Some artifacts are complimented as follows.

Algorithm 12.1 SSSP.

input: MRMP instance I

output: solution for MRMP

hyperparameters: number of sampling $m \in \mathbb{N}_{>0}$, random sampling prob. $\lambda \in (0,1]$
threshold distances $\theta_i \in \mathbb{R}_{>0}$, decay rate $\gamma \in (0,1)$

1: $G_i = (V_i, E_i) \leftarrow$ `init_roadmap`(I, i); for each $i \in A$
2: **while** TRUE **do**
3: initialize *Open*, *Explored* ▷ *priority queue, set*
4: $\mathcal{N}^{\text{init}} \leftarrow \left\langle \mathcal{Q} : \mathcal{Q}^{\text{init}},\ next : 1,\ parent : \perp \right\rangle$
5: $Open$.push$\left(\mathcal{N}^{\text{init}}\right)$; $Explored$.append$\left(\left(\mathcal{Q}^{\text{init}}, 1\right)\right)$
6: **while** $Open \neq \emptyset$ **do**
7: $\mathcal{N} \leftarrow Open$.pop()
8: **if** $\forall i \in A, \mathcal{N}.\mathcal{Q}[i] \in \mathcal{Q}_i^{\text{goal}}$ **then return** `backtrack`$(\mathcal{N})$
9: $i \leftarrow \mathcal{N}.next$; $q^{\text{from}} \leftarrow \mathcal{N}.\mathcal{Q}[i]$
10: **for** $1, 2, \ldots, m$ **do** ▷ *vertex expansion via sampling*
11: $q^{\text{new}} \leftarrow \texttt{sample}_i()$
12: with probability $(1-\lambda)$: $q^{\text{new}} \leftarrow \texttt{steer}_i(q^{\text{from}}, q^{\text{new}})$
13: **if** $\min_{q \in V_i} \texttt{dist}_i(q, q^{\text{new}}) > \theta_i$ **then** $V_i \leftarrow V_i \cup \{q^{\text{new}}\}$; update E_i
14: $j \leftarrow i+1$ **if** $i \neq |A|$ **else** 1 ▷ *search node expansion*
15: **for** $q^{\text{to}} \in \left\{ q \in V_i \mid \left(q^{\text{from}}, q\right) \in E_i \right\}$ **do**
16: $\mathcal{Q}^{\text{new}} \leftarrow \texttt{copy}(\mathcal{N}.\mathcal{Q})$; $\mathcal{Q}^{\text{new}}[i] \leftarrow q^{\text{to}}$
17: **if** $(\mathcal{Q}^{\text{new}}, j) \in Explored$ **then continue**
18: **if** `collide`$(\mathcal{N}.\mathcal{Q}, \mathcal{Q}^{\text{new}})$ **then continue**
19: $\mathcal{N}^{\text{new}} \leftarrow \langle \mathcal{Q} : \mathcal{Q}^{\text{new}}, next : j, parent : \mathcal{N} \rangle$
20: $Open$.push$(\mathcal{N}^{\text{new}})$; $Explored$.append$((\mathcal{Q}^{\text{new}}, j))$
21: $\theta_i \leftarrow \gamma\theta_i$; for each $i \in A$

Initial Roadmap Construction (Line 1) is done by conventional single-agent SBMP such as RRT-Connect [Kuffner and LaValle, 2000]. The objective is to secure at least one valid path from initial to goal configurations for each robot.

Best-first Search (Lines 3–20) realizes the core idea by maintaining a priority queue *Open* that stores generated nodes and a set *Explored* that stores already expanded search situations. For each iteration, SSSP checks whether the popped node from *Open* satisfies the goal condition (Line 8). If so, it returns a solution by backtracking the node.

Vertex Expansion (Lines 10–13) is implemented by steering from the target configuration to newly sampled ones. To guarantee the completeness, SSSP also uses vanilla random sampling with a small probability λ (0.01 in the experiments). Each new vertex must satisfy a constraint of distance threshold, which suppresses roadmaps being too dense; otherwise, the search space dramatically increases, making it difficult to find a

solution. Similar techniques are seen in SBMP studies [Kala, 2013; Dobson and Bekris, 2014].

Search Node Expansion (Lines 14–20) adds successors that *(i)* have not appeared yet in the search process and *(ii)* do not collide with other robots.

Search Iteration (Lines 2–21). The search iterates until a solution is found while decreasing the distance thresholds for vertex expansion by multiplying $0 < \gamma < 1$ by current ones.

Search Node Scoring. The heart of the best-first search is how to score each node that determines which node is popped from *Open* (Line 7). Given a tuple of configurations $\mathcal{Q}$, SSSP scores the node by summation over each robot i's shortest path distance from $\mathcal{Q}[i]$ to one of the configurations in Q_i^{goal} on the roadmap G_i. Here, each edge $\left(q^{\text{from}}, q^{\text{to}}\right) \in E_i$ is weighted by $\texttt{dist}\left(q^{\text{from}}, q^{\text{to}}\right)$. The path distance is calculated by ignoring inter-agent collisions, as common in heuristics for MAPF studies [Silver, 2005].

12.3.3 Properties

Let $k \in \mathbb{N}_{>0}$ be the number of search iterations of Lines 2–21. Moreover, let G_i^k be the roadmap G_i at the beginning of k-th iteration (i.e., G_i at Line 3) and $G^k := (G_i^k)^{i \in A}$. A solution is called *sequential* if two or more robots never transit their configurations simultaneously.

Lemma 12.1. *SSSP finds a solution in the k-th search iteration if G^k contains a sequential solution.*

Proof. For each k-th iteration, G_i is not infinitely increasing due to the distance threshold θ_i. Thus, the search space is finite: $O(|V_1| \times \ldots \times |V_n| \times |A|)$. Thanks to brute-force search in finite space, SSSP finds a solution if G^k contains one. □

Theorem 12.2. *For the geometric MRMP problem (Def. 3.5), the probability that SSSP finds a solution approaches one as k approaches ∞, provided the instance is solvable.*

Proof. Here, the discussion is limited to geometric MRMP wherein each robot can move in arbitrary directions in its configuration space. The proof is based on analysis in [Švestka and Overmars, 1998], which claims that if an instance is solvable, *(i)* there is a sequential solution and *(ii)* sufficiently dense robot-wise roadmaps constructed uniformly at random sampling (e.g., PRM [Kavraki *et al.*, 1996]) contain a sequential solution. According to Lemma 12.1, SSSP finds a solution once roadmaps holding claim-(ii) are obtained. SSSP eventually constructs such roadmaps due to as follows.

For each search iteration, each robot i tries to develop its roadmap with at least $m \geq 1$ new samples. With the probability $1 - \lambda > 0$, some of them are outcomes of uniformly at random sampling. Each iteration terminates in finite time (see Lemma 12.1), therefore, each robot does not stop attempts of uniformly at random sampling until finding solutions. Moreover, each iteration decreases the distance threshold, enabling robots to construct denser roadmaps. □

12.3.4 Postprocessing

SSSP returns only sequential solutions and compromises solution quality such as the maximum traveling time. Thus, this section briefly complements how to realize parallel execution that enables two or more robots to move simultaneously. Specifically, the

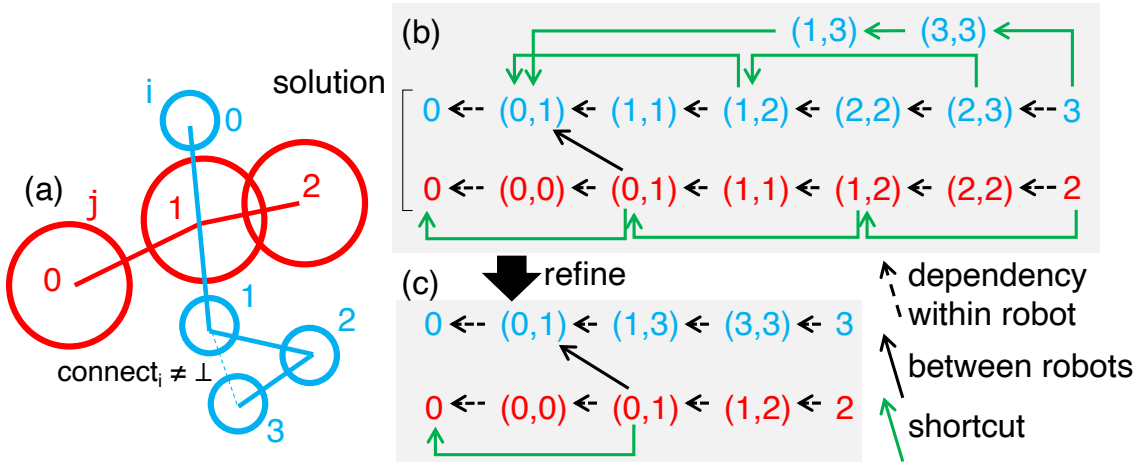


Figure 12.3: Postprocessing to refine a solution..

section considers postprocessing to refine solutions. Smoothing solution trajectories by postprocessing is common in single-robot SBMP [Geraerts and Overmars, 2007]. However, MRMP additionally takes care of inter-robot collisions.

The proposed method is described with Fig. 12.3a. Suppose that SSSP outputs a sequential solution of $\Pi_i = (0,1,1,2,2,3)$ (blue) and $\Pi_j = (0,0,1,1,2,2)$ (red). The method repeats the next two steps until a given solution metric has not improved.

1. Construct a *temporal plan graph (TPG)* [Hönig *et al.*, 2016] of the solution. TPG is a directed acyclic graph that records temporal dependencies of each robot's motions. Moreover, possible "shortcut" motions are attached to TPG. Figure 12.3b shows an example. There is an arc between the motions $(0,1)$ of robots $\{i,j\}$; these motions must happen in order due to collision avoidance. Several shortcut arcs exist, for example, i can skip using vertex-2 by directly going from vertex-1 to vertex-3.
2. Remove redundant motions in TPG while keeping the dependencies between robots. Figure 12.3c shows an example. For j, the motions $(1,1)$ and $(2,2)$ are removed but $(0,0)$ survives to keep the dependency with i.

In this example, we finally obtain a refined solution $\Pi_i = (0,1,3,3)$ and $\Pi_j = (0,0,1,2)$.

The above refinement is applicable to any MRMP solutions not limited to those from SSSP. Indeed, the experiments applied the refinement for solutions obtained by all methods.

12.4 Evaluation

This section extensively evaluates SSSP on a variety of MRMP problems and demonstrates that it can solve MRMP rapidly compared to other standard approaches. The experiments further assess solution quality, scalability about the number of agents $|A|$, and which components are essential for SSSP, followed by ground robot demos in a dense situation.

12.4.1 Experimental Setups

Benchmarks. As illustrated in Fig. 12.4 and 12.5, various scenarios were prepared with diverse degrees of freedom and kinematic constraints, in closed workspaces $\mathcal{W} \in [0,1]^{\{2,3\}}$. To focus on characteristics specific to MRMP, these scenarios were modeled with simple geometric patterns (e.g., spheres or lines) and reduced the effort of the

`connect` and `collide` functions. For each scenario, 100 instances were prepared by randomly generating initial/goal configurations and obstacle layouts. For each instance, the number of robots $|A|$ was chosen from the interval $\{2, 3, \ldots, 10\}$. Robots' body parameters (e.g., radius and arm length) were also generated randomly and differed between robots. These parameters were adjusted so that robots are sufficiently congested, otherwise, the instances become easy to solve. Note that unsolvable instances might be included, though obviously unsolvable instances were excluded, e.g., initial configurations with inter-robot collisions. In summary, each instance consists of a team of heterogeneous robots and each robot has a different configuration space; a shared roadmap is unavailable.

Baselines. The following well-known approaches to MRMP are carefully selected as baselines, which are applicable to MRMP defined in Chap. 12.2.

- **Probabilistic roadmap (PRM)** [Kavraki *et al.*, 1996] is a celebrated SBMP. For MRMP, PRM samples a composite state of $|A|$ robots directly from $O(|A|)$ dimensional spaces and constructs a single roadmap, then derives a solution by pathfinding on it.
- **Rapidly-exploring Random Tree (RRT)** [LaValle, 1998] is another popular SBMP, focusing on single-query situations. Similar to PRM, RRT for MRMP samples a composite state and constructs a tree roadmap rooted in a composite state of the initial configurations of all robots.
- **RRT-Connect (RRT-C)** [Kuffner and LaValle, 2000] is a popular extension of RRT, which accelerates finding a solution by bi-directional search from both initial and goal configurations.
- **Prioritized Planning (PP)** [Erdmann and Lozano-Perez, 1987; Silver, 2005; Van Den Berg and Overmars, 2005] is a standard approach to MAPF in that robots sequentially plan paths while avoiding collisions with already planned paths. PP was applied on roadmaps constructed by robot-wise PRMs, as taken in [Le and Plaku, 2018]. PP was repeated with random priorities until the problem is solved.
- **Conflict-Based Search (CBS)** [Sharon *et al.*, 2015] is another popular algorithm for MAPF. CBS is applicable to MRMP when roadmaps are given [Solis *et al.*, 2021]. CBS was applied on robot-wise roadmaps constructed by PRM. Moreover, the heuristics of CBS were manipulated such that avoids collisions as much as possible during the search. Doing so loses the optimality of CBS but speeds up finding solutions significantly like [Barer *et al.*, 2014].

SSSP used RRT-Connect [Kuffner and LaValle, 2000] to obtain initial robot-wise roadmaps (Line 1). Note that this is irrelevant to RRT-Connect in the baselines. PP/CBS were tested with PRM rather than RRT-Connect because otherwise constructed roadmaps do not include detours, which is essential for solving MAPF in the second phase of two-phase planning. Since all methods rely on non-determinism, we tested each method with 10 different random seeds for each instance (1,000 trials in total).

Hyperparameter Adjustment. All hyperparameters of each method including SSSP were adjusted prior to the experiments, e.g., the maximum distance to connect two vertices in the PRM-based two-phase planning methods (PP and CBS). For each scenario and each algorithm, hyperparameters were adjusted to maximize the number of successful instances within 30 s of 50 instances, among randomly chosen 50 pairs of parameters. Tie-break was based on average runtime. The used instances were generated following

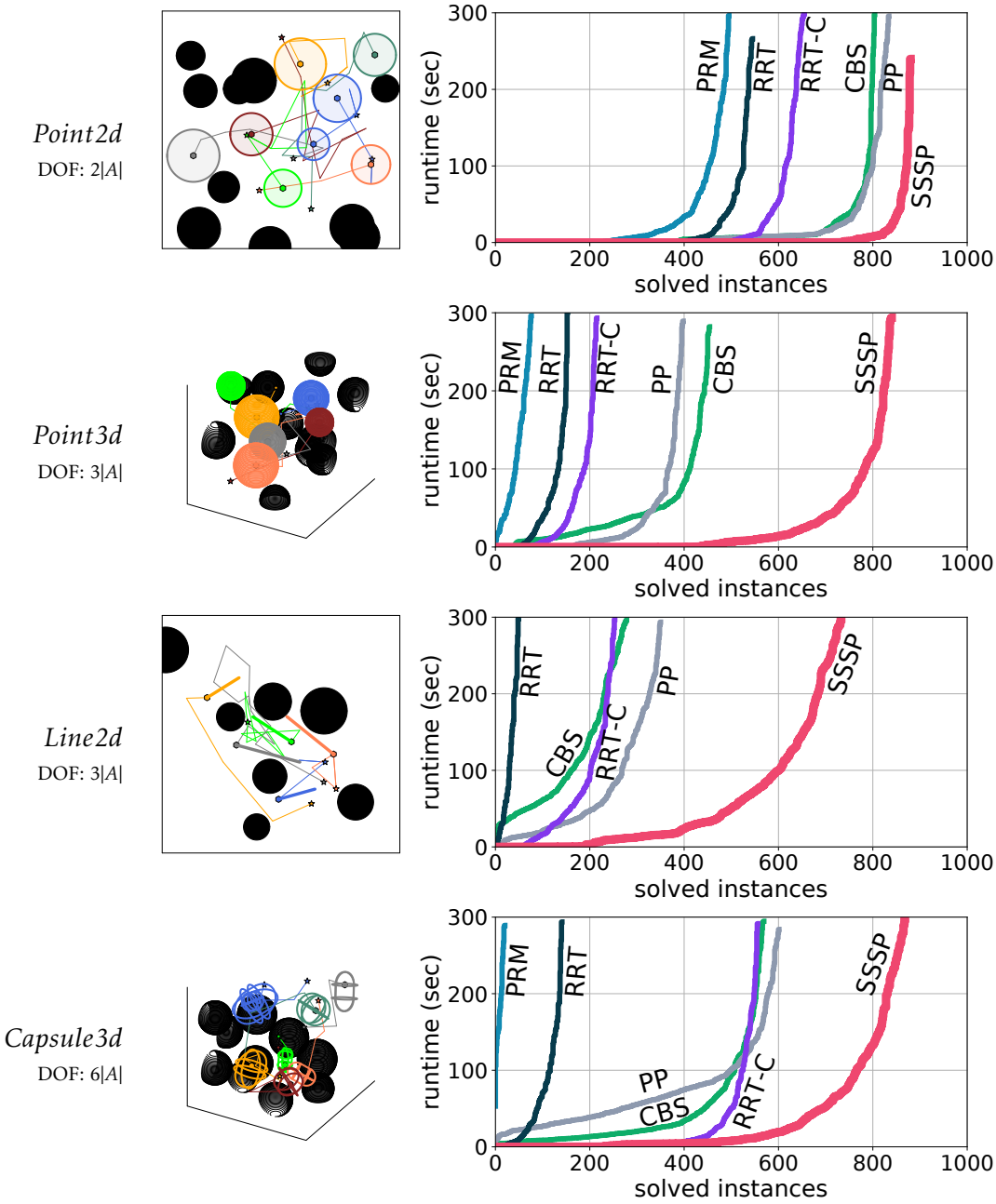


Figure 12.4: Summary of results (1/2). Each scenario is visualized with robots' bodies (colored circles, spheres, or lines), obstacles (black-filled circles or spheres), and a solution example (thin lines). Degrees of freedom (DOF) are denoted below scenario names, where $|A|$ is the number of robots. The runtime of PP and CBS includes roadmap construction. The runtime of SSSP also includes the initial roadmap construction.

the same parameters of the experiment but differed in random seeds. The used parameters are included in the configuration files of the code repository.

Metrics. The objective here is to find solutions as quickly as possible. Therefore, how many instances are solved within given time limits (maximum: 5 min) was rated.

Evaluation Environment. The simulator and all methods were coded in Julia. The experiments were run on a desktop PC with Intel Core i9-7960X 2.8 GHz CPU and 64 GB

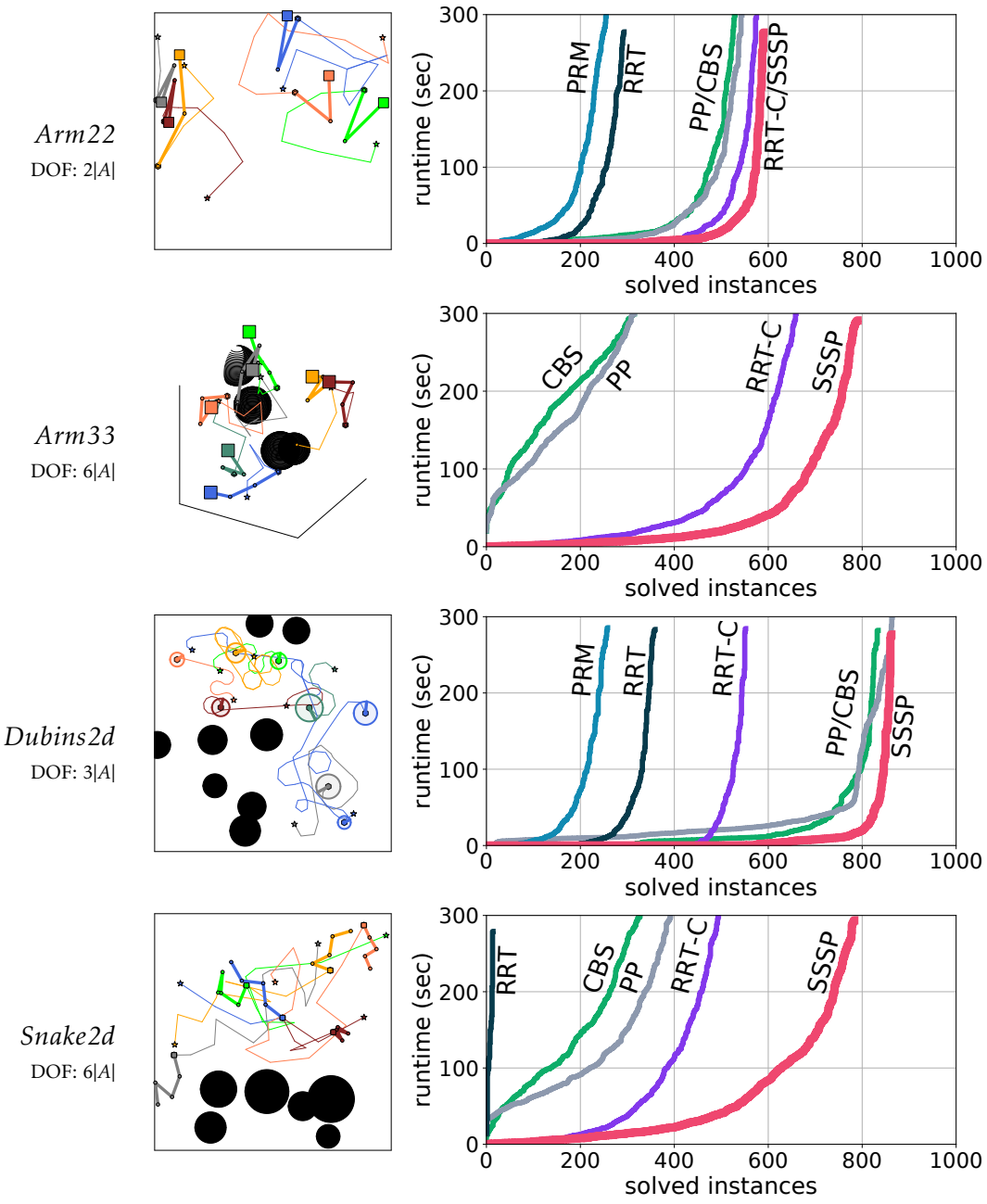


Figure 12.5: Summary of results (2/2). See also the caption of Fig. 12.4. In *Arm22* and *Arm33*, each robot has a fixed root, represented as colored boxes. Those two and *Snake2d* prohibit self-colliding. In *Dubins2d*, the robots must follow Dubins paths [Dubins, 1957].

RAM. 32 different instances were run in parallel using multi-threading. All methods used exactly the same implementations of `connect`, `collide`, `sample`, and `dist`.

12.4.2 Results of Variety of MRMP Problems

Figures 12.4 and 12.5 summarize the results. In short, SSSP significantly outperforms the other baselines in all tested scenarios, i.e., solving more instances much faster. We acknowledge that runtime performance heavily relies on implementations, however, these results indicate that SSSP is very promising. The results of SSSP in *Arm22* and *Dubins2d* are relatively non-remarkable but we guess this is due to many unsolvable instances, which could be easily generated in these scenarios. The reason of why SSSP is quick will

be discussed at the end of this chapter.

12.4.3 Solution Quality

As reference records of solution quality, Table 12.1 shows the expected total traveling time of all robots (aka. sum-of-costs), after applying the postprocessing introduced in Chap. 12.3.4 to all methods. Compared to the other baselines, the total traveling time of SSSP is not the best but comparable.

	SSSP	PRM	RRT	RRT-C	PP	CBS
Point2d 54/100	1.93 (1.75, 2.10)	3.67 (2.95, 4.25)	3.42 (3.02, 3.81)	3.04 (2.62, 3.43)	**1.45** (1.37, 1.52)	**1.43** (1.36, 1.49)
Point3d 26/100	2.88 (2.56, 3.16)	N/A	N/A	4.58 (4.14, 4.99)	**1.91** (1.73, 2.06)	**1.72** (1.59, 1.84)
Line2d 34/100	2.68 (2.42, 2.92)	N/A	N/A	5.89 (5.14, 6.62)	1.80 (1.69, 1.90)	**1.62** (1.56, 1.69)
Capsule3d 61/100	2.34 (2.12, 2.53)	N/A	N/A	2.89 (2.55, 3.23)	**1.84** (1.79, 1.90)	2.33 (2.24, 2.43)
Arm22 30/100	**1.57** (1.35, 1.75)	2.46 (2.13, 2.78)	2.23 (1.96, 2.48)	**1.73** (1.43, 2.00)	**1.60** (1.43, 1.73)	**1.51** (1.35, 1.64)
Arm33 94/100	**2.31** (2.22, 2.40)	N/A	N/A	2.97 (2.72, 3.21)	2.74 (2.64, 2.83)	2.75 (2.68, 2.82)
Dubins2d 30/100	1.52 (1.42, 1.62)	3.84 (3.31, 4.32)	3.06 (2.78, 3.33)	2.07 (1.80, 2.32)	1.28 (1.23, 1.32)	1.39 (1.34, 1.45)
Snake2d 55/100	**3.17** (2.91, 3.42)	N/A	N/A	4.62 (4.20, 5.02)	**3.28** (3.08, 3.48)	**3.25** (3.04, 3.44)

Table 12.1: Total traveling time. Scores are averaged over instances successfully solved by all methods, and normalized by $\sum_{i \in A} \texttt{dist}(q_i^{\text{init}}, q \in \mathcal{Q}_i^{\text{goal}})$. The numbers below on the scenario names are those numbers of instances. To obtain meaningful values, some cases are excluded from the calculation due to the low success rates (≤20%; marked as N/A). 95% confidence intervals of means are also presented. Bold characters are based on the overlap of the confidence intervals.

12.4.4 Scalability Test

Next, the scalability of SSSP about the number of robots $|A|$ was evaluated, varied by 10 increments. For each $|A|$, a hundred *Point2d* instances were prepared with smaller robots' radius (see Fig. 12.6). PP with tuned parameters was also tested as a baseline, which relatively scored high among the other baselines in the scenario with many robots. Figure 12.6 shows that, with larger $|A|$, SSSP takes longer but still acceptable time for planning, compared to PP. Note that PP relies on robot-wise PRMs covering the entire space; the roadmap construction itself becomes a bottleneck with larger $|A|$.

Hyperparameter Adjustment. The parameters were manually adjusted and applied them to all instances regardless of $|A|$. SSSP used $m = 10$, $\theta_i = 0.05$, and $\epsilon = 0.2$ (in the `steer` function). PP was run on PRMs with 500 samples and the maximum distance to connect two vertices was 0.1. These PRM's values were adjusted such that roadmaps sufficiently cover the entire workspace while being not too dense; otherwise, the roadmap construction itself takes too much time and the search space will become too huge. Figure 12.7 shows two roadmaps created by PP (PRM) and SSSP. We informally observed that those PP parameters were sensitive to obtain consistently good results.

Remark for Experimental Environment. It is noted that the scalability test was heavily affected by experimental environments. We informally confirmed that both PP and SSSP

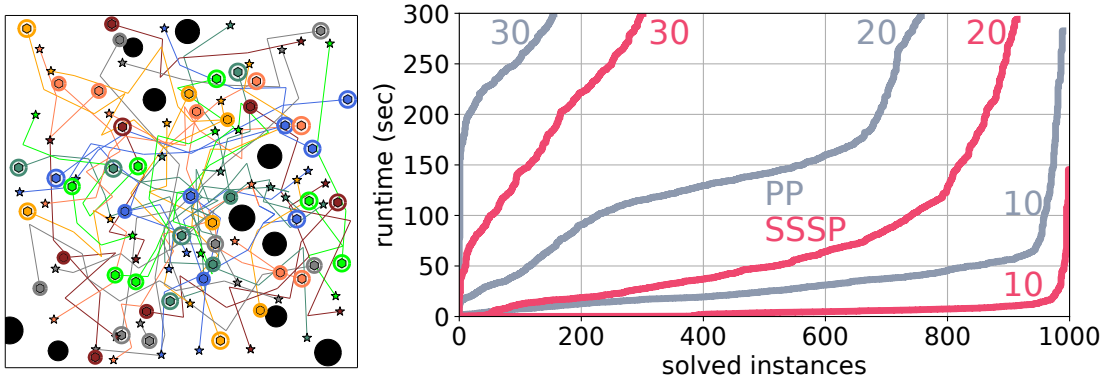


Figure 12.6: Scalability test in *Point2d*. *left*: An example instance ($|A| = 30$) and its solution from SSSP. *right*: The number of solved instances with annotation of $|A|$.

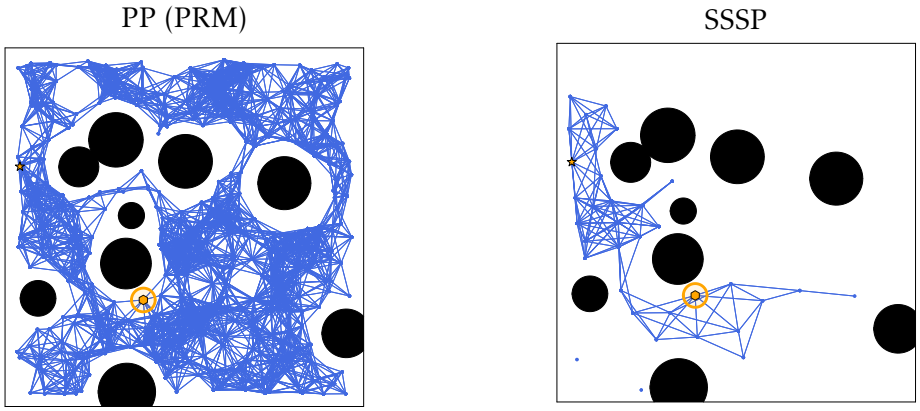


Figure 12.7: Constructed roadmaps for the scalability test.

could be faster (≥x2) with another environment. Nevertheless, SSSP generally worked better than the baseline. For instance, SSSP solved instances with 50 agents (≤5 min) while PP failed the same instances.

12.4.5 Ablation Study – Which Elements are Essential?

The next experiment addresses another question that asks which technical components are essential to SSSP. Specifically, degraded versions were tested that omit the following components in several scenarios.

- initial roadmap constructions (Line 1)
- search node scoring (replaced by random values)
- vertex expansion (Lines 10–13)
- distance thresholds check (Line 13)
- steering (by setting λ=1)
- integrated sampling and search

The last one rated SSSP without vertex expansion ($m = 0$) on robot-wise PRMs.

Results of Solvability. Table 12.2(*upper*) reveals that all these components contribute to the performance of SSSP. Among them, involving the appropriate node scoring is particularly critical to achieving high success rates within a limited time, as well as vertex expansion. The initial roadmap construction is effective when single-robot motion

planning itself is difficult (*Snake2d*). The steering effect was non-dramatic because the workspace is small relative to robots; this is further investigated below.

Results of Solution Quality. Table 12.2(*lower*) shows total traveling time of the ablation study. From the table, we can see that initial roadmap construction contributes to improving solution quality. This is because, with initial roadmaps, each roadmap is expanded mainly in neighboring regions of a valid path from start to goal included in the initial one. On the other hand, such a "guide" does not exist without initial roadmaps, resulting in non-efficient trajectories.

number of solved instances within 5 min

	SSSP	random score	no init roadmap	no vertex expansion	no `dist` check	λ=1	on PRM
Point2d	**880**	512	793	498	737	875	807
Arm22	**586**	101	565	388	523	584	398
Snake2d	**710**	0	379	674	519	677	39

total traveling time

	SSSP	random score	no init roadmap	no vertex expansion	no `dist` check	λ=1	on PRM
Point2d 54/100	**1.96** (1.75, 2.15)	7.49 (5.44, 9.28)	8.75 (4.78, 11.84)	**1.89** (1.63, 2.08)	**1.70** (1.58, 1.81)	2.02 (1.80, 2.21)	**1.78** (1.58, 1.95)
Arm22 52/100	**1.76** (1.58, 1.91)	N/A	7.38 (5.71, 8.95)	**1.87** (1.64, 2.06)	**1.69** (1.55, 1.82)	**1.74** (1.58, 1.88)	**1.93** (1.72, 2.12)
Snake2d 63/100	3.35 (3.08, 3.62)	N/A	50.11 (40.72, 58.44)	3.78 (3.41, 4.13)	N/A	**2.77** (2.56, 2.97)	3.64 (3.35, 3.93)

Table 12.2: Results of ablation studies. See also the caption of Table 12.1 for the bottom table. To obtain meaningful values, some cases are excluded from the calculation due to the low success rates (≤20%; marked as N/A).

Effect of Steering. SSSP was additionally evaluated with different λ (probability of vanilla random sampling) in *point2d* with 20 robots, which were the same instances as Chap. 12.4.4. To illustrate the steering effect clearly, the hyperparameters of SSSP were set as $m = 5$, $\theta_i = 0.02$, and $\epsilon = 0.05$. Figure 12.8 shows the results, demonstrating the decrease in success rate with a higher rate of vanilla random sampling.

12.4.6 Robot Demonstration

Finally, SSSP was applied to 32 robots (`https://toio.io/`) in a dense situation (Fig. 12.9). The robots evolve on a specific playmat and are controllable by instructions of absolute coordinates. The planning module was the same one used in the previous experiments and the robots were modeled as *Point2d*. Even though an experimenter randomly placed the robots as an initial configuration (see the movie), the planning was done in about 30 s, then all robots eventually reached their destinations. Importantly, the planning was performed without prepared roadmaps like grids, rather, it was based on only five utility functions (`connect`, `sample`, `dist`, `steer`, and `collide`).

12.4.7 Discussion – Why is SSSP Quick?

Below, two qualitative explanations regarding the quickness of SSSP are provided: one from the MAPF side, and another from the SBMP side. The performance of SSSP relies on both factors.

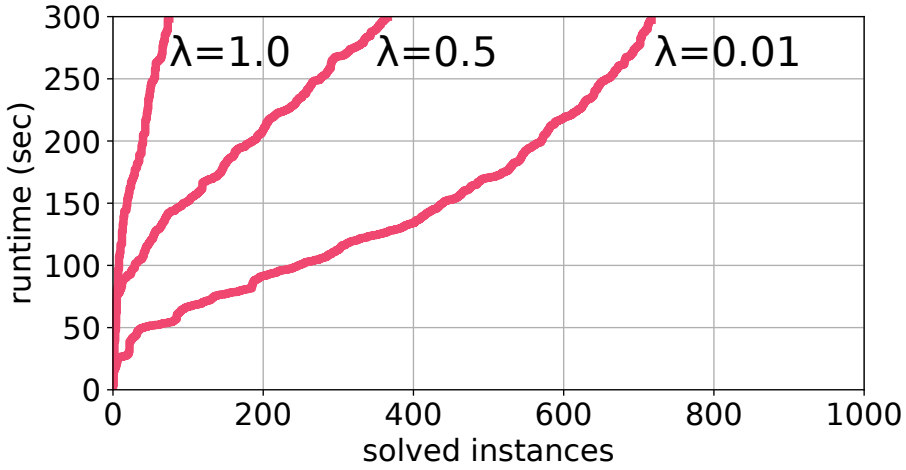


Figure 12.8: Effect of steering. For each instance of *Point2d* among 100 instances, where $|A| = 20$, SSSP was run 10 times with different random seeds.

Figure 12.9: Robot demo. From left to right, the pictures show the initial, intermediate, and final configurations, which eventually constitute characters "SSSP."

Branching Factor. During the search, SSSP decomposes successors into search nodes corresponding to at most one robot taking a motion. Compared to search styles allowing all robots to move at the same time, the decomposition significantly reduces *branching factor*, i.e., the number of successors at each node (likewise LaCAM in Chap. 6). In general, the average branching factor b largely determines the search effort [Edelkamp and Schrodl, 2011]. Assume that each robot has k possible motions from each state on average. Coupled with perfect heuristics and a perfect tie-breaking strategy, allowing all robots to move simultaneously results in $b = k^{|A|}$ and generates $\left(k^{|A|}\right) \cdot l$ nodes, where l is the depth of the search. In contrast, SSSP results in $b = k$ and enables to search of the equivalent node with only $(k|A|) \cdot l$ nodes generation. This is a key trick of A* with operator decomposition [Standley, 2010] for MAPF; SSSP is based on this idea.

Imbalanced Roadmaps. PRM-based methods (i.e., PRM, PP, and CBS in the experiments) have no choice other than to construct roadmaps uniformly spread in each configuration space, making search spaces huge. Such drawbacks might be relieved with biased sampling but representing good bias for MRMP is not trivial; indeed, existing studies use machine learning [Arias *et al.*, 2021a; Okumura *et al.*, 2022c]. In contrast, owing to carefully-designed components, SSSP naturally constructs sparse roadmaps in important regions for each robot like in Fig. 12.2. As a result, the search space for SSSP is kept small, which also contributes to quick MRMP.

12.5 Concluding Remarks

The chapter introduced the SSSP algorithm that rapidly solves MRMP. The main idea behind the algorithm was to unite techniques developed for SBMP and search techniques for MAPF. The former, corresponding to the representation perspective, had been studied mainly in the robotic community, while the latter, corresponding to the planning perspective, in the AI community. Bringing them together brings promising results, as extensively demonstrated in the experiments.

12.5.1 Kinodynamic MRMP

This chapter untreated dynamics constraints, therefore, *kinodynamic MRMP* is an interesting future direction. Similar to RRT or EST that are applicable to kinodynamic planning, SSSP has a high potential for such planning. The adaptation is by considering planning with a *state* x instead of a configuration $q \in \mathcal{C}_i$ for robot i, which comprises a configuration q and its derivative $\dot{q}$ (i.e., $x := (q, \dot{q})$). Some parts require care; in usual kinodynamic MRMP, $\texttt{connect}_i(x, x) = \bot$ as we see in cars that cannot stop instantly. This means that sequential solutions are not allowed. In this case, it is necessary to regard $|A|$ successive search nodes in SSSP as "one block" to enable concurrent motions of multiple robots, as taken in the original A* with operator decomposition. Moreover, it may be necessary to change the sampling strategy as follows.

Sampling from Control Spaces

For kinodynamic motion planning, depending on what constraints f_i are posed for robot i, given arbitrary two configurations, the local planner may rarely return a trajectory due to boundary condition issues. SSSP can address such planning problems by changing the steering at Line 12. The below shortly complements this aspect.

Algorithm 12.1 assumes steered sampling obtained uniformly at random from $\mathcal{C}_i$. Instead, we can always get a connected configuration by considering sampling from the control space as follows. Let denote $\texttt{sample}_i^{\mathcal{U}}$ sampling function from the control space $\mathcal{U}_i$ for robot i. Let further denote $\texttt{dist}_i^{\mathcal{U}}$ distance function in $\mathcal{U}_i$. Then, replace Line 12 by:

$$q^{\text{new}} \leftarrow \texttt{steer}_i^{\mathcal{U}}\left(q^{\text{from}}, \texttt{sample}_i^{\mathcal{U}}()\right) \tag{12.3}$$

where (12.4)

$$\texttt{steer}_i^{\mathcal{U}}(q, \xi) := \texttt{emulate}_i(q, \xi^{\text{near}}) \tag{12.5}$$

$$\texttt{emulate}_i(q, \xi) := q + f_i(q, \xi) \cdot \Delta \tag{12.6}$$

$$\xi^{\text{near}} := \operatorname*{argmin}_{\xi' \in U_\xi} \texttt{dist}_i^{\mathcal{U}}(\xi', \xi) \tag{12.7}$$

$$U_\xi := \left\{\xi \in \mathcal{U} \mid \texttt{connect}_i\Big(q, \texttt{emulate}_i(q, \xi)\Big) \neq \bot\right\} \tag{12.8}$$

The updated one outputs steered sampling with feasible control inputs.

12.5.2 Interesting Directions

Some other interesting directions for the development of SSSP are listed below.

Optimal MRMP. SSSP prioritizes solving MRMP itself, rather than solution quality. However, it is straightforward to take into account quality by modifying node scoring, which is currently designed as a greedy best-first search. With terminologies of

A* search [Hart *et al.*, 1968], the chapter only used h-value (i.e., estimation of cost-to-go). Incorporating g-value (i.e., cost-to-come) allows SSSP to solve MRMP optimally for a set of roadmaps G^k at k-th search iteration. Therefore, it is expected that a modified version of SSSP is asymptotically optimal, together with a discussion of eventual completeness. Another methodology to achieve optimal MRMP may be rewriting the search tree, as taken in LaCAM* (Chap. 6).

Further Integration of SBMP and MAPF. The concept behind the chapter was developing robot-wise roadmaps according to the multi-agent search progress, turning out to be promising. Therefore, this direction should be further investigated. A very-recent study [Kottinger *et al.*, 2022] explores this direction for CBS. A bunch of powerful MAPF algorithms exists not limited to A* with operator decomposition or CBS, as presented in Chap. 3.2.4 and Part I. Therefore, integrating them with SBMP may fruit practical methodologies for MRMP.

Chapter 13

Conclusion and Discussion

The dissertation aims at developing quick, scalable, near-optimal, robust, domain independent, and end-to-end multi-agent navigation from computational aspects. To this end, three perspectives were introduced, namely, planning, execution, and representation; they were respectively studied. Throughout the dissertation, the presented studies provided methodologies to break tradeoffs existing in each perspective. This final chapter concludes the dissertation while highlighting some remarkable things in the dissertation, rather than reviewing it all in detail. Each highlight is followed by interesting future directions.

13.1 Planning

13.1.1 Summary of Contributions

The crux of the planning part is graph path planning for multiple agents, formulated as MAPF. Therefore, Part I devoted to advancing MAPF technologies.

The advancement by this dissertation is impressively visualized in Fig. 13.1, corresponding to the previous one presented in Fig. 3.3. Recall that this figure presents how many instances have been successfully solved by each algorithm, where instances were retrieved from the MAPF benchmark [Stern *et al.*, 2019]. In past, the dissertation pointed out the difficulty of ensuring completeness and solution quality while suppressing the planning effort. Indeed, Fig. 3.3 was evidence of this tradeoff.

Now, let us confirm the result of the developed algorithms in the dissertation, in Fig. 13.1. The PIBT$^{(+)}$ algorithm (Chap. 4) achieved good performance itself, i.e., solving many instances speedily. However, due to its incompleteness, it has room for improvements in success rate. Meanwhile, the LaCAM* algorithm (Chap. 6) with improved PIBT, a complete algorithm, solved 99% of instances while guaranteeing to converge to optima. This result is clearly breaking the tradeoff in planning! Furthermore, as presented in Chap. 6, LaCAM can solve a variety of MAPF instances in a very short time. For instance, it solved puzzle-like complicated instances (near-)optimally while it also solved instances with 10,000 agents in a few of tens seconds. *Such algorithms never exist so far.*

13.1.2 Future Directions

Many enhancements and applications of Part I can be considered. Among them, two directions that I am especially interested in are discussed.

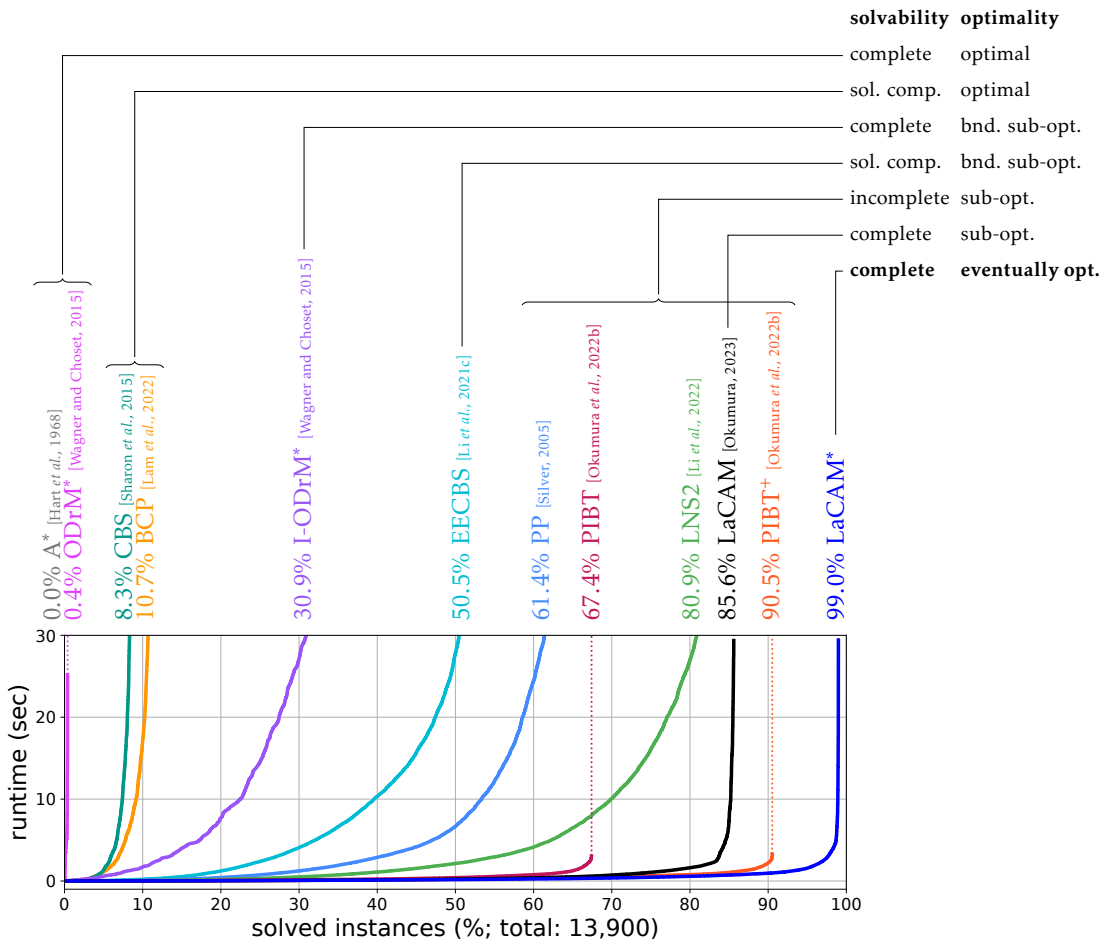


Figure 13.1: Comparison of various algorithms on the MAPF benchmark.

Integrating Planning and Learning

The current PIBT implementation (Chap. 6.6), used in LaCAM*, incorporates a manually adjusted pattern detector to improve its performance. This will be a bottleneck for domain independence since the detector is designed for the conventional MAPF setting (four-connected grids). However, we can learn something important from here, namely, improving vertex scoring of PIBT used in LaCAM can significantly improve the performance of the search. Therefore, it has value to seek how to design "good" vertex scoring automatically, while considering domain independence in mind.

One promising direction might be exploring machine learning, as we see successful results in various domains [Mnih *et al.*, 2015; Krizhevsky *et al.*, 2017; Jumper *et al.*, 2021]. In particular, I expect that learning good short-horizon planning in a full-scratch manner, like what AlphaZero did for board games [Silver *et al.*, 2018], will be a very powerful technology. Integrating ML into planning may also improve planning quality (e.g., flowtime and makespan in MAPF). In the MAPF literature, several studies have already started to integrate planning and learning, e.g., [Huang *et al.*, 2021; Virmani *et al.*, 2021; Huang *et al.*, 2022].

Though LaCAM, as well as PIBT, resulted in very promising results, they are newcomers to MAPF algorithms. In other words, they are not extensively examined for their potential. Therefore, I believe integrating ML into these two fruit remarkable achievements.

Leverage LaCAM to Other Graph Pathfinding Problems

After all, *lazy constraint addition search*, a concept behind LaCAM, is just a graph pathfinding algorithm. Therefore, I believe it can leverage to search problems beyond MAPF. Indeed, unlabeled-MAPF is easily captured by LaCAM by changing a configuration generator from PIBT to TSWAP (Chap. 5). The necessary components of LaCAM to apply other domains are a "good" configuration generator (i.e., the role of PIBT/TSWAP) and the manner of constructing a constraint tree. If these two components are designed appropriately, LaCAM may be a powerful search scheme for domains wherein the branching factor is very huge like MAPF.

13.2 Execution

13.2.1 Summary of Contributions

In the execution part (Part II), aiming at overcoming uncertainties, the dissertation explored the non-trivial integration of planning and execution. That is, *agents reactively behave at runtime though planning itself is offline*. The concept is exemplified by OTIMAPP for timing uncertainties (Chap. 9) and MAPPCF for crash faults (Chap. 10). This novel style can attach good theoretical properties to execution containing uncertainties. Indeed, decentralized execution with real robots was presented, while ensuring liveness, without any central intervention at runtime, and without any global interactions (Chap. 9.8.4). This kind of demonstration can be achieved neither by conventional centralized execution styles that rely on global monitoring systems nor by decentralized execution styles that lack centralized planners. In short, the proposed integrated planning-execution style provides a unique concept for achieving multi-agent navigation.

13.2.2 Future Directions

Two interesting directions are discussed below.

Hybrid of Deliberative and Reactive Approaches

The primary observations of theoretical analyses in OTMAPP and MAPPCF are that these problems are computationally challenging. Empirically, for both problems, we have seen algorithms that can handle instances with tens of agents. However, with much larger instances like we saw in Part I, it is significantly challenging to find solutions due to both computational difficulty and the non-existence of feasible solutions. Therefore, it makes sense to consider relaxed versions of the problems, which are unlikely to trigger something bad.

As a nature of planning problems discussed in Chap. 3.6.3, the difficulty of such problems partially stems from their length of planning horizons. Therefore, it might be effective to consider relaxations by limiting the length of planning horizons. More precisely, I suggest exploring a planning-execution style visualized in Fig. 13.2; this is where I did not deeply survey.

The primary questions will be, with such styles, whether attaching something good properties (e.g., liveness) to execution is possible, as what OTIMAPP and MAPPCF did. Moreover, whether such styles really relax computational burdens is unclear.

Dynamic Obstacles

The dissertation assumed static obstacles while did not assume *dynamic* obstacles such that their locations are time-varying. Dynamic obstacles are not rare in reality. For in-

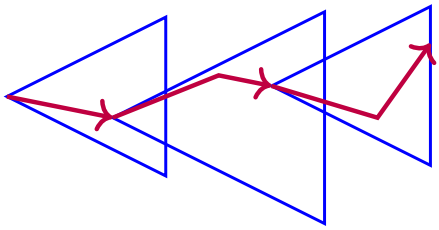


Figure 13.2: Hybrid of deliberative and reactive approaches.

stance, in warehouse applications, humans themselves are dynamic obstacles to robots. This begs a new research question, namely, how to realize robust execution while ensuring something good, even with dynamic obstacles. I envision that similar approaches to OTIMAPP or MAPPCF are possible to answer this question.

13.3 Representation

13.3.1 Summary of Contributions

Considering representation issues is necessary to realize end-to-end multi-agent navigation; hence studied in Part III. The primary challenge in representation was how to construct small but effective search spaces (i.e., roadmaps) for subsequent planning. The dissertation first considered two-phase planning, that is, decoupled planning style that first constructs roadmaps and then solves MAPF on those roadmaps. Especially, with the enhancements of ML techniques (Chap. 11), it had been demonstrated that such small but effective roadmaps can be built. With this result, an advanced concept that couples roadmap construction and multi-agent search has been tested as the SSSP algorithm (Chap. 12) to solve MRMP. SSSP solved various planning instances in a domain-independent way within short timeframes, including the ground robot demo. Though its scalability is not remarkable compared to Part I, I believe this direction is promising for the future development of MRMP.

13.3.2 Future Directions

Two interesting directions are discussed below.

Integration of SBMP, MAPF, and ML

SSSP uses random walks when sampling. This technical component is able to be replaced with biased sampling enhanced by ML techniques, as done in CTRM. Doing so provides much smaller but effective roadmaps, expected to bring scalable MRMP algorithms.

Observe that this idea integrates three technical components: SSSP itself is a composition of SBMP and MAPF, furthermore, the idea poses the integration of ML into SSSP. Like this, I strongly envision the integration of SBMP, MAPF, and ML technologies fruit very powerful and practical approaches for MRMP. For instance, integration of MAPF and ML has been seen in, e.g., [Huang *et al.*, 2021; Zhang *et al.*, 2022c]. That of SBMP and MAPF has been seen in, e.g., [Solis *et al.*, 2021; Kottinger *et al.*, 2022]. That of ML and SBMP has been seen in, e.g., [Qureshi *et al.*, 2020; Ichter *et al.*, 2020; Chen *et al.*, 2020]. However, to the best of my knowledge, the integration of three components is under-explored. This is where further research is promising.

Environment Optimization

The dissertation treated the environment as untouchable, however, in many practical scenarios, the *environment is configurable*. For instance, in warehouse operations, the layout of shelf and delivery locations can be manipulated. This begs a new research question that asks for a good representation of configurable environments. Some early studies have appeared in the literature [Bellusci *et al.*, 2020; Vainshtain and Salzman, 2021; Gao and Prorok, 2022].

13.4 Final Notes

The dissertation implicitly assumes that agents are physically embedded (even with animated agents). However, after all, what was presented is *multi-agent sequential decision-making* where agents potentially interfere with each other. Therefore, I expect that killer applications of multi-agent navigation exist not limited to robotics. I am not sure about this point, but it is always nice to seek something unimaginable thus far.

References

[Aakash and Saha, 2022] Aakash and Indranil Saha. It costs to get costs! a heuristic-based scalable goal assignment algorithm for multi-robot systems. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2022. (Cited on page 99)

[Achá *et al.*, 2022] Roberto Asín Achá, Rodrigo López, Sebastian Hagedorn, and Jorge A Baier. Multi-agent path finding: A new boolean encoding. *Journal of Artificial Intelligence Research (JAIR)*, 2022. (Cited on page 28)

[Adler *et al.*, 2015] Aviv Adler, Mark De Berg, Dan Halperin, and Kiril Solovey. Efficient multi-robot motion planning for unlabeled discs in simple polygons. 2015. (Cited on pages 36 and 99)

[Ahuja *et al.*, 1993] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. 1993. (Cited on pages 89 and 93)

[Ahuja *et al.*, 2002] Ravindra K Ahuja, Özlem Ergun, James B Orlin, and Abraham P Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 2002. (Cited on pages 31 and 132)

[Alonso-Mora *et al.*, 2012] Javier Alonso-Mora, Andreas Breitenmoser, Martin Rufli, Roland Siegwart, and Paul Beardsley. Image and animation display with multiple mobile robots. *International Journal of Robotics Research (IJRR)*, 2012. (Cited on pages 1 and 35)

[Altisen *et al.*, 2019] Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. Introduction to distributed self-stabilizing algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2019. (Cited on pages 147 and 156)

[Andreychuk and Yakovlev, 2018] Anton Andreychuk and Konstantin Yakovlev. Two techniques that enhance the performance of multi-robot prioritized path planning. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2018. (Cited on page 117)

[Andreychuk *et al.*, 2022] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. *Artificial Intelligence (AIJ)*, 2022. (Cited on page 32)

[Ardizzoni *et al.*, 2022] Stefano Ardizzoni, Irene Saccani, Luca Consolini, and Marco Locatelli. Multi-agent path finding on strongly connected digraphs: feasibility and solution algorithms. *arXiv preprint*, 2022. (Cited on page 25)

[Arias *et al.*, 2021a] Felipe Felix Arias, Brian Ichter, Aleksandra Faust, and Nancy M Amato. Avoidance critical probabilistic roadmaps for motion planning in dynamic environments. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2021. (Cited on page 238)

[Arias *et al.*, 2021b] Felipe Felix Arias, Brian Andrew Ichter, Aleksandra Faust, and Nancy M. Amato. Avoidance critical probabilistic roadmaps for motion planning in dynamic environments. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2021. (Cited on page 223)

[Atzmon *et al.*, 2020a] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R Sturtevant, and Sven Koenig. Probabilistic robust multi-agent path finding. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2020. (Cited on pages 38 and 153)

[Atzmon *et al.*, 2020b] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research (JAIR)*, 2020. (Cited on pages 38, 182, and 203)

[Atzmon *et al.*, 2020c] Dor Atzmon, Yonathan Zax, Einat Kivity, Lidor Avitan, Jonathan Morag, and Ariel Felner. Generalizing multi-agent path finding for heterogeneous agents. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2020. (Cited on page 32)

[Augugliaro *et al.*, 2014] Frederico Augugliaro, Sergei Lupashin, Michael Hamer, Cason Male, Markus Hehn, Mark W Mueller, Jan Sebastian Willmann, Fabio Gramazio, Matthias Kohler, and Raffaello D'Andrea. The flight assembled architecture installation: Cooperative construction with flying machines. *IEEE Control Systems Magazine*, 2014. (Cited on page 2)

[AutoStore, 2021] AutoStore. Reliability, 2021. https://fr.autostoresystem.com/benefits/reliable. (Cited on page 39)

[Avis, 1983] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 1983. (Cited on page 93)

[Azarm and Schmidt, 1997] Kianoush Azarm and Günther Schmidt. Conflict-free motion of multiple mobile robots based on decentralized motion planning and negotiation. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 1997. (Cited on pages 30 and 76)

[Balyo *et al.*, 2012] Tomá Balyo, Roman Bartak, and Pavel Surynek. Shortening plans by local re-planning. 2012. (Cited on page 132)

[Banfi *et al.*, 2017] Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. Intractability of time-optimal multirobot path planning on 2d grid graphs with holes. *IEEE Robotics and Automation Letters (RA-L)*, 2017. (Cited on pages 3, 26, and 130)

[Barer *et al.*, 2014] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2014. (Cited on pages 30, 138, 177, and 232)

[Barták *et al.*, 2021] Roman Barták, Marika Ivanová, and Jiří Švancara. From classical to colored multi-agent path finding. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2021. (Cited on page 37)

[Bellusci *et al.*, 2020] Matteo Bellusci, Nicola Basilico, and Francesco Amigoni. Multi-agent path finding in configurable environments. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2020. (Cited on page 245)

[Ben-Ari and Mondada, 2017] Mordechai Ben-Ari and Francesco Mondada. *Elements of robotics*. Springer Nature, 2017. (Cited on page 38)

[Bennewitz *et al.*, 2002] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, 2002. (Cited on pages 30 and 76)

[Bishop and Nasrabadi, 2006] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Springer, 2006. (Cited on pages 19 and 277)

[Bnaya and Felner, 2014] Zahy Bnaya and Ariel Felner. Conflict-oriented windowed hierarchical cooperative A*. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2014. (Cited on page 76)

[Bnaya *et al.*, 2013] Zahy Bnaya, Roni Stern, Ariel Felner, Roie Zivan, and Steven Okamoto. Multi-agent path finding for self interested agents. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2013. (Cited on pages 39 and 203)

[Botea *et al.*, 2018] Adi Botea, Davide Bonusi, and Pavel Surynek. Solving multi-agent path finding on strongly biconnected digraphs. *Journal of Artificial Intelligence Research (JAIR)*, 2018. (Cited on page 25)

[Bouzy, 2013] Bruno Bouzy. Monte-carlo fork search for cooperative path-finding. In *Computer Games Workshop at IJCAI*, 2013. (Cited on page 31)

[Boyarski *et al.*, 2015] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. Icbs: improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2015. (Cited on pages 29, 40, 67, 135, and 138)

[Boyrasky *et al.*, 2015] Eli Boyrasky, Ariel Felner, Guni Sharon, and Roni Stern. Don't split, try to work it out: bypassing conflicts in multi-agent pathfinding. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2015. (Cited on page 67)

[Burkard and Rendl, 1991] Rainer E Burkard and Franz Rendl. Lexicographic bottleneck problems. *Operations Research Letters (ORL)*, 1991. (Cited on pages 37 and 99)

[Cachin *et al.*, 2011] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer, 2011. (Cited on page 4)

[Călinescu *et al.*, 2008] Gruia Călinescu, Adrian Dumitrescu, and János Pach. Reconfigurations in graphs and grids. *SIAM Journal on Discrete Mathematics (SIDMA)*, 2008. (Cited on pages 37 and 99)

[Canny, 1988] John Canny. *The complexity of robot motion planning*. MIT press, 1988. (Cited on page 17)

[Cáp *et al.*, 2013] Michal Cáp, P. Novák, J. Vokrínek, and M. Pechoucek. Multi-agent rrt: Sampling-based cooperative pathfinding. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2013. (Cited on page 31)

[Čáp *et al.*, 2015] Michal Čáp, Peter Novák, Alexander Kleiner, and Martin Selecký. Prioritized planning algorithms for trajectory coordination of multiple mobile robots. *IEEE Transactions on Automation Science and Engineering (T-ASE)*, 2015. (Cited on pages 27, 30, 76, 138, 181, 190, 198, 199, and 223)

[Čáp *et al.*, 2016] Michal Čáp, Jean Gregoire, and Emilio Frazzoli. Provably safe and deadlock-free execution of multi-robot plans under delaying disturbances. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016. (Cited on pages 38 and 182)

[Chandra and Toueg, 1996] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 1996. (Cited on page 203)

[Chen *et al.*, 2017] Yu Fan Chen, Miao Liu, Michael Everett, and Jonathan P How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2017. (Cited on page 39)

[Chen *et al.*, 2020] Binghong Chen, Bo Dai, Qinjie Lin, Guo Ye, Han Liu, and Le Song. Learning to plan in high dimensions via neural exploration-exploitation trees. In *Proceedings of the International Conference on Learning and Representation (ICLR)*, 2020. (Cited on pages 49, 215, 222, 223, and 244)

[Choset *et al.*, 2005] Howie Choset, Kevin M Lynch, Seth Hutchinson, George A Kantor, and Wolfram Burgard. *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005. (Cited on pages 16, 34, and 227)

[Chris J. Maddison, 2017] Yee Whye Teh Chris J. Maddison, Andriy Mnih. The concrete distribution: A continuous relaxation of discrete random variables. In *Proceedings of the International Conference on Learning and Representation (ICLR)*, 2017. (Cited on page 278)

[Coffman *et al.*, 1971] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 1971. (Cited on page 181)

[Cohen *et al.*, 2018a] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernández, Ariel Felner, TK Satish Kumar, and Sven Koenig. Anytime focal search with applications. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2018. (Cited on pages 121, 125, 128, 132, and 141)

[Cohen *et al.*, 2018b] Liron Cohen, Glenn Wagner, David Chan, Howie Choset, Nathan Sturtevant, Sven Koenig, and TK Satish Kumar. Rapid randomized restarts for multi-agent path finding solvers. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2018. (Cited on page 117)

[Cohen *et al.*, 2019] Liron Cohen, Tansel Uras, TK Satish Kumar, and Sven Koenig. Optimal and bounded-suboptimal multi-agent motion planning. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2019. (Cited on page 35)

[Cormen *et al.*, 2022] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022. (Cited on page 12)

[Coskun and O'Kane, 2019] Adem Coskun and Jason M O'Kane. Online plan repair in multi-robot coordination with disturbances. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2019. (Cited on page 182)

[Cplex, 2009] IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 2009. (Cited on pages 28, 41, and 121)

[Damani *et al.*, 2021] Mehul Damani, Zhiyao Luo, Emerson Wenzel, and Guillaume Sartoretti. Primal_2: Pathfinding via reinforcement and imitation multi-agent learning-lifelong. *IEEE Robotics and Automation Letters (RA-L)*, 2021. (Cited on pages 31 and 222)

[Daniel *et al.*, 2010] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research (JAIR)*, 2010. (Cited on page 32)

[Dayan *et al.*, 2021] Dror Dayan, Kiril Solovey, Marco Pavone, and Dan Halperin. Near-optimal multi-robot motion planning with finite sampling. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2021. (Cited on page 35)

[De Wilde *et al.*, 2014] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research (JAIR)*, 2014. (Cited on pages 3, 6, 27, 30, 67, 76, 79, 118, 128, and 132)

[Diestel, 2017] Reinhard Diestel. *Graph Theory*. Springer, 2017. (Cited on page 10)

[Dijkstra, 1959] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959. (Cited on pages 15 and 89)

[Dobson and Bekris, 2014] Andrew Dobson and Kostas E Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. *International Journal of Robotics Research (IJRR)*, 2014. (Cited on pages 217 and 230)

[Donald *et al.*, 1993] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 1993. (Cited on page 18)

[Dresner and Stone, 2008] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research (JAIR)*, 2008. (Cited on page 2)

[Dubins, 1957] Lester E Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 1957. (Cited on pages 34, 209, 226, and 234)

[Edelkamp and Schrodl, 2011] Stefan Edelkamp and Stefan Schrodl. *Heuristic search: theory and applications*. Elsevier, 2011. (Cited on pages 14 and 238)

[Elbanhawi and Simic, 2014] Mohamed Elbanhawi and Milan Simic. Sampling-based robot motion planning: A review. *IEEE Access*, 2014. (Cited on pages 17, 220, and 223)

[Erdem *et al.*, 2013] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2013. (Cited on page 28)

[Erdmann and Lozano-Perez, 1987] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 1987. (Cited on pages 30, 42, 52, 67, 75, 109, 121, 154, 170, 209, 210, and 232)

[Eric Jang, 2017] Ben Poole Eric Jang, Shixiang Gu. Categorical reparameterization with gumbel-softmax. In *Proceedings of the International Conference on Learning and Representation (ICLR)*, 2017. (Cited on page 278)

[Fanti and Zhou, 2004] Maria Pia Fanti and MengChu Zhou. Deadlock control methods in automated manufacturing systems. *IEEE Transactions on systems, man, and cybernetics-part A: systems and humans*, 2004. (Cited on page 181)

[Felber *et al.*, 1999] Pascal Felber, Xavier Défago, Rachid Guerraoui, and Philipp Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications*, 1999. (Cited on page 188)

[Felner *et al.*, 2017] Ariel Felner, Roni Stern, Solomon Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2017. (Cited on page 27)

[Felner *et al.*, 2018] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, TK Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2018. (Cited on pages 29, 40, 67, and 138)

[Financial Times, 2021] Financial Times. Ocado's largest warehouse shut after robot collision causes fire, 2021. https://www.ft.com/content/aaddf4b1-a78b-4289-b42f-fd3f5cd7f176. (Cited on page 39)

[Floyd, 1962] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 1962. (Cited on page 72)

[Ford and Fulkerson, 1956] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 1956. (Cited on pages 89 and 269)

[Fraichard, 1998] Thierry Fraichard. Trajectory planning in a dynamic workspace: A 'state-time space' approach. *Advanced Robotics (AR)*, 1998. (Cited on page 209)

[Gammell *et al.*, 2014] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014. (Cited on pages 18 and 49)

[Gammell *et al.*, 2015] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2015. (Cited on pages 18 and 49)

[Gao and Prorok, 2022] Zhan Gao and Amanda Prorok. Environment optimization for multi-agent navigation. *arXiv preprint*, 2022. (Cited on page 245)

[Geft and Halperin, 2022] Tzvika Geft and Dan Halperin. Refined hardness of distance-optimal multi-agent path finding. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2022. (Cited on pages 3, 26, and 130)

[Geraerts and Overmars, 2007] Roland Geraerts and Mark H Overmars. Creating high-quality paths for motion planning. *International Journal of Robotics Research (IJRR)*, 2007. (Cited on page 231)

[Gerkey and Matarić, 2004] Brian P Gerkey and Maja J Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research (IJRR)*, 2004. (Cited on page 37)

[Ghallab *et al.*, 2016] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016. (Cited on pages 14 and 42)

[Gharbi *et al.*, 2009] Mokhtar Gharbi, Juan Cortés, and Thierry Siméon. Roadmap composition for multi-arm systems path planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009. (Cited on pages 35 and 223)

[Goldenberg *et al.*, 2014] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert Holte, and Jonathan Schaeffer. Enhanced partial expansion A∗. *Journal of Artificial Intelligence Research (JAIR)*, 2014. (Cited on pages 28, 114, and 128)

[G'omez *et al.*, 2020] Rodrigo N G'omez, Carlos Hern'andez, and Jorge A Baier. Solving sum-of-costs multi-agent pathfinding with answer-set programming. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2020. (Cited on pages 28 and 271)

[Goodfellow *et al.*, 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. (Cited on page 19)

[Goraly and Hassin, 2010] Gilad Goraly and Refael Hassin. Multi-color pebble motion on graphs. *Algorithmica*, 2010. (Cited on pages 37 and 99)

[Gregoire *et al.*, 2013] Jean Gregoire, Silvère Bonnabel, and Arnaud de La Fortelle. Optimal cooperative motion planning for vehicles at intersections. *arXiv preprint*, 2013. (Cited on page 183)

[Gross, 1959] O Gross. The bottleneck assignment problem. Technical report, RAND CORP SANTA MONICA CALIF, 1959. (Cited on pages 65, 79, 88, and 270)

[Grothe *et al.*, 2022] Francesco Grothe, Valentin N Hartmann, Andreas Orthey, and Marc Toussaint. St-rrt∗: Asymptotically-optimal bidirectional motion planning through space-time. *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2022. (Cited on page 35)

[Guangyao Shi, 2020] Lifeng Zhou Guangyao Shi, Pratap Tokekar. Robust multiple-path orienteering problem: Securing against adversarial attacks. In *Proceedings of International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2020. (Cited on pages 39 and 203)

[Gurobi Optimization, 2022] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2022. (Cited on page 28)

[Han *et al.*, 2018] Shuai D Han, Edgar J Rodriguez, and Jingjin Yu. Sear: A polynomial-time multi-robot path planning algorithm with expected constant-factor optimality guarantee. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018. (Cited on page 35)

[Hart *et al.*, 1968] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968. (Cited on pages 15, 40, 109, 120, 126, and 240)

[Hearn and Demaine, 2005] Robert A Hearn and Erik D Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science (TCS)*, 2005. (Cited on pages 2 and 34)

[Henkel and Toussaint, 2020] Christian Henkel and Marc Toussaint. Optimized directed roadmap graph for multi-agent path finding using stochastic gradient descent. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, pages 776–783, 2020. (Cited on page 223)

[Henkel *et al.*, 2019] Christian Henkel, Jannik Abbenseth, and Marc Toussaint. An optimal algorithm to solve the combined task allocation and path finding problem. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019. (Cited on page 37)

[Hirata *et al.*, 2021] Mai Hirata, Manabu Tsukada, Keisuke Okumura, Yasumasa Tamura, Hideya Ochiai, and Xavier Défago. Roadside-assisted cooperative planning using future path sharing for autonomous driving. In *Proceedings of the Vehicular Technology Conference (VTC)*, 2021. (Cited on page 2)

[Hönig *et al.*, 2016] Wolfgang Hönig, TK Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2016. (Cited on pages 32, 67, and 231)

[Hönig *et al.*, 2018a] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph Durham, and Nora Ayanian. Conflict-based search with optimal task assignment. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2018. (Cited on pages 37, 97, and 150)

[Hönig *et al.*, 2018b] Wolfgang Hönig, James A Preiss, TK Satish Kumar, Gaurav S Sukhatme, and Nora Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics (T-RO)*, 2018. (Cited on pages 4, 28, 35, 217, and 223)

[Hönig *et al.*, 2019] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W Durham, and Nora Ayanian. Persistent and robust execution of mapf schedules in warehouses. *IEEE Robotics and Automation Letters (RA-L)*, 2019. (Cited on pages 38 and 182)

[Hopcroft and Karp, 1973] John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing (SICOMP)*, 1973. (Cited on page 89)

[Hopcroft *et al.*, 1984] John E Hopcroft, Jacob Theodore Schwartz, and Micha Sharir. On the complexity of motion planning for multiple independent objects; pspace-hardness of the warehouseman's problem. *International Journal of Robotics Research (IJRR)*, 1984. (Cited on pages 2 and 34)

[Hoshen, 2017] Yedid Hoshen. Vain: Attentional multi-agent predictive modeling. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017. (Cited on page 212)

[Hsu *et al.*, 1997] David Hsu, J-C Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 1997. (Cited on pages 50, 224, 226, and 227)

[Hsu, 2000] David Hsu. *Randomized single-query motion planning in expansive spaces*. Stanford University, 2000. (Cited on page 50)

[Hu *et al.*, 2021] Shuli Hu, Daniel D Harabor, Graeme Gange, Peter J Stuckey, and Nathan R Sturtevant. Jump point search with temporal obstacles. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2021. (Cited on page 30)

[Hu *et al.*, 2022] Shuli Hu, Daniel D Harabor, Graeme Gange, Peter J Stuckey, and Nathan R Sturtevant. Multi-agent path finding with temporal jump point search. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2022. (Cited on page 30)

[Huang *et al.*, 2021] Taoan Huang, Bistra N. Dilkina, and Sven Koenig. Learning node-selection strategies in bounded-suboptimal conflict-based search for multi-agent path finding. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2021. (Cited on pages 242 and 244)

[Huang *et al.*, 2022] Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra N. Dilkina. Anytime multi-agent path finding via machine learning-guided large neighborhood search. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2022. (Cited on page 242)

[Husár *et al.*, 2022] Matej Husár, Jirí Svancara, Philipp Obermeier, Roman Barták, and Torsten Schaub. Reduction-based solving of multi-agent pathfinding on large maps using graph pruning. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2022. (Cited on page 28)

[Hwang *et al.*, 1992] Yong Koo Hwang, Narendra Ahuja, et al. A potential field approach to path planning. *IEEE Transactions on Robotics and Automation*, 1992. (Cited on page 48)

[Ichter *et al.*, 2018] Brian Ichter, James Harrison, and Marco Pavone. Learning sampling distributions for robot motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pages 7087–7094, 2018. (Cited on pages 49, 208, 215, 222, 223, and 278)

[Ichter *et al.*, 2020] Brian Ichter, Edward Schmerling, Tsang-Wei Edward Lee, and Aleksandra Faust. Learned critical probabilistic roadmaps for robotic motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pages 9535–9541, 2020. (Cited on pages 49, 215, 222, 223, and 244)

[Intel Corporation, 2020] Intel Corporation. Travel intel's autonomous superhighway, 2020. https://newsroom.intel.com/news/travel-intels-autonomous-superhighway. (Cited on pages 1 and 4)

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015. (Cited on page 220)

[Ivanovic *et al.*, 2021] Boris Ivanovic, Karen Leung, Edward Schmerling, and Marco Pavone. Multimodal deep generative models for trajectory prediction: A conditional variational autoencoder approach. *IEEE Robotics and Automation Letters (RA-L)*, 2021. (Cited on pages 210 and 277)

[Jiang *et al.*, 2019] Yuqian Jiang, Harel Yedidsion, Shiqi Zhang, Guni Sharon, and Peter Stone. Multi-robot planning with conflicts and synergies. *Autonomous Robots (AURO)*, 2019. (Cited on page 75)

[Jumper *et al.*, 2021] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 2021. (Cited on pages 31 and 242)

[Kala, 2013] Rahul Kala. Rapidly exploring random graphs: motion planning of multiple mobile robots. *Advanced Robotics (AR)*, 2013. (Cited on page 230)

[Kameyama *et al.*, 2021] Shota Kameyama, Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Active modular environment for robot navigation. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pages 8636–8642, 2021. (Cited on pages 65, 180, 188, and 204)

[Karaman and Frazzoli, 2011] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research (IJRR)*, 2011. (Cited on pages 114, 128, 217, and 223)

[Kasaura *et al.*, 2022] Kazumi Kasaura, Mai Nishimura, and Ryo Yonetani. Prioritized safe interval path planning for multi-agent pathfinding with continuous time on 2d roadmaps. *IEEE Robotics and Automation Letters (RA-L)*, 2022. (Cited on page 32)

[Kautz *et al.*, 2002] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. 2002. (Cited on page 117)

[Kavraki *et al.*, 1996] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 1996. (Cited on pages 18, 34, 48, 223, 230, and 232)

[Khorshid *et al.*, 2011] Mokhtar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2011. (Cited on page 76)

[Kingma and Ba, 2015] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning and Representation (ICLR)*, 2015. (Cited on page 219)

[Kingma and Welling, 2014] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *Proceedings of the International Conference on Learning and Representation (ICLR)*, 2014. (Cited on page 278)

[Knapp, 1987] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys (CSUR)*, 1987. (Cited on pages 153 and 183)

[Kornhauser *et al.*, 1984] Daniel Martin Kornhauser, Gary Miller, and Paul Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. Master's thesis, MIT, 1984. (Cited on pages 3, 25, 30, 36, 37, 99, 154, 165, and 166)

[Kottinger *et al.*, 2022] Justin Kottinger, Shaull Almagor, and Morteza Lahijanian. Conflict-based search for multi-robot motion planning with kinodynamic constraints. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022. (Cited on pages 35, 240, and 244)

[Krizhevsky *et al.*, 2017] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 2017. (Cited on pages 31 and 242)

[Kuffner and LaValle, 2000] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2000. (Cited on pages 222, 229, and 232)

[Kuhn, 1955] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 1955. (Cited on pages 37, 65, and 88)

[Kumar and Chakravorty, 2012] Sandip Kumar and Suman Chakravorty. Multi-agent generalized probabilistic roadmaps: Magprm. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012. (Cited on page 223)

[Kumar *et al.*, 2019] Rahul Kumar, Aditya Mandalika, Sanjiban Choudhury, and Siddhartha Srinivasa. Lego: Leveraging experience in roadmap generation for sampling-based planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019. (Cited on pages 208 and 223)

[Lalish and Morgansen, 2012] Emmett Lalish and Kristi A Morgansen. Distributed reactive collision avoidance. *Autonomous Robots (AURO)*, 2012. (Cited on pages 21, 39, and 182)

[Lam *et al.*, 2022] Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter J Stuckey. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research (COR)*, 2022. (Cited on pages 6, 27, 29, 41, 53, 67, 121, 126, 128, 130, 132, and 203)

[Latombe, 1991] Jean-Claude Latombe. *Robot motion planning*. Kluwer Academic Publishers, 1991. (Cited on page 48)

[Laurent *et al.*, 2021] Florian Laurent, Manuel Schneider, Christian Scheller, Jeremy Watson, Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Konstantin Makhnev, Oleg Svidchenko, et al. Flatland competition 2020: Mapf and marl for efficient train coordination on a grid world. In *NeurIPS 2020 Competition and Demonstration Track*, 2021. (Cited on page 2)

[LaValle and Kuffner Jr, 2001] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *International Journal of Robotics Research (IJRR)*, 2001. (Cited on page 226)

[LaValle, 1998] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Computer Science Department, Iowa State University (TR 98–11), 1998. (Cited on pages 18, 34, 128, 223, and 232)

[LaValle, 2006] Steven M LaValle. *Planning algorithms*. Cambridge University Press, 2006. (Cited on pages 16, 17, 34, 35, 206, 209, 222, and 227)

[Le and Plaku, 2018] Duong Le and Erion Plaku. Cooperative, dynamics-based, and abstraction-guided multi-robot motion planning. *Journal of Artificial Intelligence Research (JAIR)*, 2018. (Cited on pages 34 and 232)

[Le and Plaku, 2019] Duong Le and Erion Plaku. Multi-robot motion planning with dynamics via coordinated sampling-based expansion guided by multi-agent search. *IEEE Robotics and Automation Letters (RA-L)*, 2019. (Cited on page 34)

[Le Goc *et al.*, 2016] Mathieu Le Goc, Lawrence H Kim, Ali Parsaei, Jean-Daniel Fekete, Pierre Dragicevic, and Sean Follmer. Zooids: Building blocks for swarm user interfaces. In *Proceedings of the annual symposium on user interface software and technology (UIST)*, 2016. (Cited on page 1)

[Li *et al.*, 2019a] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2019. (Cited on pages 29, 40, 67, 138, and 141)

[Li *et al.*, 2019b] Jiaoyang Li, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig. Disjoint splitting for multi-agent path finding with conflict-based search. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2019. (Cited on pages 40, 67, and 128)

[Li *et al.*, 2019c] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, TK Satish Kumar, and Sven Koenig. Multi-agent path finding for large agents. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2019. (Cited on page 32)

[Li *et al.*, 2020] Qingbiao Li, Fernando Gama, Alejandro Ribeiro, and Amanda Prorok. Graph neural networks for decentralized multi-robot path planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. (Cited on pages 27, 31, and 222)

[Li *et al.*, 2021a] Jiaoyang Li, Zhe Chen, Daniel Harabor, P Stuckey, and Sven Koenig. Anytime multi-agent path finding via large neighborhood search. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2021. (Cited on pages 128 and 132)

[Li *et al.*, 2021b] Jiaoyang Li, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence (AIJ)*, 2021. (Cited on pages 6, 29, 40, 53, 67, 121, and 130)

[Li *et al.*, 2021c] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2021. (Cited on pages 30, 41, 53, 67, 101, 109, 121, and 126)

[Li *et al.*, 2021d] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2021. (Cited on pages 12 and 108)

[Li *et al.*, 2021e] Qingbiao Li, Weizhe Lin, Zhe Liu, and Amanda Prorok. Message-aware graph attention networks for large-scale multi-robot path planning. *IEEE Robotics and Automation Letters (RA-L)*, 2021. (Cited on pages 31 and 222)

[Li *et al.*, 2022] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Mapf-lns2: Fast repairing for multi-agent path finding via large neighborhood search. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2022. (Cited on pages 6, 31, 42, 101, 109, 121, and 126)

[Lin *et al.*, 2017] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceeding of the International Conference on Computer Vision (ICCV)*, 2017. (Cited on page 219)

[Liu *et al.*, 2019] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. Task and path planning for multi-agent pickup and delivery. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2019. (Cited on pages 37 and 269)

[Lozano-Pérez and Wesley, 1979] Tomás Lozano-Pérez and Michael A Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 1979. (Cited on page 48)

[Lu *et al.*, 2018] Xiankai Lu, Chao Ma, Bingbing Ni, Xiaokang Yang, Ian Reid, and Ming-Hsuan Yang. Deep regression tracking with shrinkage loss. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018. (Cited on page 219)

[Luis *et al.*, 2020] Carlos E Luis, Marijan Vukosavljev, and Angela P Schoellig. Online trajectory generation with distributed model predictive control for multi-robot motion planning. *IEEE Robotics and Automation Letters (RA-L)*, 2020. (Cited on page 39)

[Luna and Bekris, 2011] Ryan Luna and Kostas E Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2011. (Cited on pages 30, 53, 67, 76, 109, 118, 121, and 128)

[Ma and Koenig, 2016] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2016. (Cited on page 37)

[Ma *et al.*, 2016] Hang Ma, Craig Tovey, Guni Sharon, TK Kumar, and Sven Koenig. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2016. (Cited on pages 3, 26, 36, 99, 130, and 269)

[Ma *et al.*, 2017a] Hang Ma, TK Satish Kumar, and Sven Koenig. Multi-agent path finding with delay probabilities. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2017. (Cited on pages 38, 149, 154, 177, and 182)

[Ma *et al.*, 2017b] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2017. (Cited on pages 37, 52, 65, 72, 99, 118, and 131)

[Ma *et al.*, 2019a] Hang Ma, Daniel Harabor, Peter J Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2019. (Cited on pages 6, 30, and 76)

[Ma *et al.*, 2019b] Hang Ma, Wolfgang Hönig, TK Satish Kumar, Nora Ayanian, and Sven Koenig. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2019. (Cited on page 32)

[Ma *et al.*, 2021] Ziyuan Ma, Yudong Luo, and Hang Ma. Distributed heuristic multi-agent path finding with communication. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2021. (Cited on pages 31 and 222)

[Ma, 2022] Hang Ma. Graph-based multi-robot path finding and planning. *Current Robotics Reports*, 2022. (Cited on page 27)

[MacAlpine *et al.*, 2015] Patrick MacAlpine, Eric Price, and Peter Stone. Scram: scalable collision-avoiding role assignment with minimal-makespan for formational positioning. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2015. (Cited on pages 2, 35, and 99)

[MacCormick, 2018] John MacCormick. *What can be computed?: A practical guide to the theory of computation*. Princeton University Press, 2018. (Cited on page 12)

[Mannucci *et al.*, 2021] Anna Mannucci, Lucia Pallottino, and Federico Pecora. On provably safe and live multirobot coordination with online goal posting. *IEEE Transactions on Robotics (T-RO)*, 2021. (Cited on pages 3, 181, and 182)

[Mansouri *et al.*, 2019] Masoumeh Mansouri, Bruno Lacerda, Nick Hawes, and Federico Pecora. Multi-robot planning under uncertain travel times and safety constraints. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2019. (Cited on pages 38 and 153)

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015. (Cited on pages 31 and 242)

[Morris *et al.*, 2016] Robert Morris, Corina S Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, TK Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *In Proceedings of AAAI Workshop: Planning for Hybrid Systems*, 2016. (Cited on page 2)

[Nair and Hinton, 2010] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010. (Cited on page 220)

[Nebel, 2020] Bernhard Nebel. On the computational complexity of multi-agent pathfinding on directed graphs. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2020. (Cited on pages 25, 158, and 166)

[Nguyen *et al.*, 2019] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. Generalized target assignment and path finding using answer set programming. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2019. (Cited on page 37)

[O'Donnell and Lozano-Pérez, 1989] Patrick A O'Donnell and Tomás Lozano-Pérez. Deadlock-free and collision-free coordination of two robot manipulators. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 1989. (Cited on page 182)

[Oh *et al.*, 2015] Kwang-Kyo Oh, Myoung-Chul Park, and Hyo-Sung Ahn. A survey of multi-agent formation control. *Automatica*, 2015. (Cited on page 99)

[Okoso *et al.*, 2019] Ayano Okoso, Keisuke Otaki, and Tomoki Nishi. Multi-agent path finding with priority for cooperative automated valet parking. In *Proceedings of IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019. (Cited on page 2)

[Okumura and Défago, 2022a] Keisuke Okumura and Xavier Défago. Quick multi-robot motion planning by combining sampling and search. *arXiv preprint*, 2022. (Cited on page 224)

[Okumura and Défago, 2022b] Keisuke Okumura and Xavier Défago. Solving simultaneous target assignment and path planning efficiently with time-independent execution. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, pages 270–278, 2022. ©2022 AAAI, https://doi.org/10.1609/icaps.v32i1.19810. (Cited on pages 36, 78, 145, and 182)

[Okumura and Tixeuil, 2023] Keisuke Okumura and Sebastien Tixeuil. Fault-tolerant offline multi-agent path planning. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2023. (Cited on page 184)

[Okumura *et al.*, 2018] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Amoeba exploration: Coordinated exploration with distributed robots. In *International Conference on Awareness Science and Technology (iCAST)*, 2018. (Cited on page 2)

[Okumura *et al.*, 2019] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2019. (Cited on pages 52 and 62)

[Okumura *et al.*, 2020] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. winpibt: Extended prioritized algorithm for iterative multi-agent path finding. In *IJCAI Workshop on Multi-Agent Path Finidng (WoMAPF)*, 2020. (Cited on page 52)

[Okumura *et al.*, 2021a] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Iterative refinement for real-time multi-robot path planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021. ©2021 IEEE. https://doi.org/10.1109/IROS51168.2021.9636071. (Cited on pages 128 and 130)

[Okumura *et al.*, 2021b] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Time-independent planning for multiple moving agents. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2021. (Cited on pages 145, 154, 177, and 182)

[Okumura *et al.*, 2022a] Keisuke Okumura, Franccois Bonnet, Yasumasa Tamura, and Xavier Défago. Offline time-independent multi-agent path planning. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2022. https://doi.org/10.24963/ijcai.2022/645. (Cited on pages 145, 152, 199, and 203)

[Okumura *et al.*, 2022b] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence (AIJ)*, 2022. https://doi.org/10.1016/j.artint.2022.103752. (Cited on pages 12, 52, 101, 109, 121, 126, and 177)

[Okumura *et al.*, 2022c] Keisuke Okumura, Ryo Yonetani, Mai Nishimura, and Asako Kanezaki. Ctrms: Learning to construct cooperative timed roadmaps for multi-agent path planning in continuous spaces. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2022. https://dl.acm.org/doi/10.5555/3535850.3535959, https://www.ifaamas.org/Proceedings/aamas2022/. (Cited on pages 206 and 238)

[Okumura, 2023] Keisuke Okumura. Lacam: Search-based algorithm for quick multi-agent pathfinding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2023. (Cited on pages 100, 121, 126, and 132)

[Park *et al.*, 2022] Jungwon Park, Dabin Kim, Gyeong Chan Kim, Dahyun Oh, and H Jin Kim. Online distributed trajectory planning for quadrotor swarm with feasibility guarantee using linear safe corridor. *IEEE Robotics and Automation Letters (RA-L)*, 2022. (Cited on page 39)

[Peasgood *et al.*, 2008] Mike Peasgood, Christopher Michael Clark, and John McPhee. A complete and scalable strategy for coordinating multiple robots within roadmaps. *IEEE Transactions on Robotics (T-RO)*, 2008. (Cited on page 30)

[Pecora *et al.*, 2018] Federico Pecora, Henrik Andreasson, Masoumeh Mansouri, and Vilian Petkov. A loosely-coupled approach for multi-robot coordination, motion planning and control. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2018. (Cited on page 3)

[Peltzer *et al.*, 2020] Oriana Peltzer, Kyle Brown, Mac Schwager, Mykel J Kochenderfer, and Martin Sehr. STT-CBS: A conflict-based search algorithm for multi-agent path finding with stochastic travel times. *arXiv preprint*, 2020. (Cited on pages 38 and 153)

[Petersen *et al.*, 2019] Kirstin H Petersen, Nils Napp, Robert Stuart-Smith, Daniela Rus, and Mirko Kovac. A review of collective robotic construction. *Science Robotics*, 2019. (Cited on page 2)

[Phillips and Likhachev, 2011] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2011. (Cited on page 30)

[Pivtoraiko *et al.*, 2009] Mihail Pivtoraiko, Ross A Knepper, and Alonzo Kelly. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics (JFR)*, 2009. (Cited on page 48)

[Preparata and Shamos, 1985] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer, 1985. (Cited on page 48)

[Prorok *et al.*, 2021] Amanda Prorok, Matthew Malencia, Luca Carlone, Gaurav S Sukhatme, Brian M Sadler, and Vijay Kumar. Beyond robustness: A taxonomy of approaches towards resilient multi-robot systems. *arXiv preprint*, 2021. (Cited on pages 5 and 39)

[Qureshi *et al.*, 2020] Ahmed H Qureshi, Yinglong Miao, Anthony Simeonov, and Michael C Yip. Motion planning networks: Bridging the gap between learning-based and classical motion planners. *IEEE Transactions on Robotics (T-RO)*, pages 1–9, 2020. (Cited on pages 49, 223, and 244)

[Ratner and Warmuth, 1986] Daniel Ratner and Manfred K Warmuth. Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 1986. (Cited on pages 26 and 165)

[Rego *et al.*, 2011] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research (EJOR)*, 2011. (Cited on page 44)

[Reif, 1979] John H Reif. Complexity of the mover's problem and generalizations. In *Proceeding of IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1979. (Cited on page 34)

[Ren *et al.*, 2021] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Loosely synchronized search for multi-agent path finding with asynchronous actions. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021. (Cited on page 32)

[Robertson and Seymour, 1985] Neil Robertson and Paul D Seymour. Disjoint paths–a survey. *SIAM Journal on Algebraic Discrete Methods*, 1985. (Cited on page 182)

[Röger and Helmert, 2012] Gabriele Röger and Malte Helmert. Non-optimal multi-agent pathfinding is solved (since 1984). In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2012. (Cited on page 25)

[Ryan, 2010] Malcolm Ryan. Constraint-based multi-robot path planning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2010. (Cited on page 28)

[Sajid *et al.*, 2012] Qandeel Sajid, Ryan Luna, and Kostas E Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2012. (Cited on pages 67 and 76)

[Salzman *et al.*, 2014] Oren Salzman, Doron Shaharabani, Pankaj K Agarwal, and Dan Halperin. Sparsification of motion-planning roadmaps by edge contraction. *International Journal of Robotics Research (IJRR)*, 2014. (Cited on page 215)

[Salzmann *et al.*, 2020] Tim Salzmann, Boris Ivanovic, Punarjay Chakravarty, and Marco Pavone. Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020. (Cited on page 210)

[Sánchez and Latombe, 2002] Gildardo Sánchez and Jean-Claude Latombe. On delaying collision checking in prm planning: Application to multi-robot coordination. *International Journal of Robotics Research (IJRR)*, 2002. (Cited on page 34)

[Sartoretti *et al.*, 2019] Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, TK Satish Kumar, Sven Koenig, and Howie Choset. Primal: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters (RA-L)*, 2019. (Cited on pages 27, 31, 203, and 222)

[Schmerling and Pavone, 2019] Edward Schmerling and Marco Pavone. Kinodynamic planning. *Encyclopedia of Robotics. Springer,*, 2019. (Cited on page 18)

[Schwartz and Tokekar, 2020] Russell Schwartz and Pratap Tokekar. Robust multi-agent task assignment in failure-prone and adversarial environments. In *Proceedings of International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2020. (Cited on pages 39 and 203)

[Senbaslar *et al.*, 2018] Baskin Senbaslar, Wolfgang Hönig, and Nora Ayanian. Robust trajectory execution for multi-robot teams using distributed real-time replanning. In *Proceedings of International Symposium on Distributed Robotic Systems (DARS)*, 2018. (Cited on pages 21, 39, and 182)

[Sha *et al.*, 1990] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 1990. (Cited on pages 52 and 54)

[Shahar *et al.*, 2021] Tomer Shahar, Shashank Shekhar, Dor Atzmon, Abdallah Saffidine, Brendan Juba, and Roni Stern. Safe multi-agent pathfinding with time uncertainty. *Journal of Artificial Intelligence Research (JAIR)*, 2021. (Cited on page 203)

[Sharon *et al.*, 2013] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*, 2013. (Cited on pages 29, 128, 135, and 203)

[Sharon *et al.*, 2015] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*, 2015. (Cited on pages 11, 27, 29, 40, 53, 67, 109, 121, 126, 128, 138, 154, 171, 197, 203, 210, and 232)

[Sheehy, 2015] Justin Sheehy. There is no now. *Communications of the ACM*, 2015. (Cited on page 3)

[Shome *et al.*, 2020] Rahul Shome, Kiril Solovey, Andrew Dobson, Dan Halperin, and Kostas E Bekris. drrt$*$: Scalable and informed asymptotically-optimal multi-robot motion planning. *Autonomous Robots (AURO)*, 2020. (Cited on pages 35 and 114)

[Silberschatz *et al.*, 2006] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. Deadlocks. In *Operating system concepts*, chapter 7. 2006. (Cited on page 181)

[Silver *et al.*, 2018] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 2018. (Cited on page 242)

[Silver, 2005] David Silver. Cooperative pathfinding. In *Proceedings of AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2005. (Cited on pages 1, 6, 27, 30, 42, 53, 67, 76, 90, 101, 108, 109, 121, 126, 139, 210, 218, 230, and 232)

[Skrynnik *et al.*, 2021] Alexey Skrynnik, Alexandra A. Yakovleva, Vasilii Davydov, Konstantin S. Yakovlev, and Aleksandr I. Panov. Hybrid policy learning for multi-agent pathfinding. *IEEE Access*, 2021. (Cited on page 31)

[Sohn *et al.*, 2015] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2015. (Cited on pages 208, 210, and 277)

[Sokkalingam and Aneja, 1998] PT Sokkalingam and Yash P Aneja. Lexicographic bottleneck combinatorial problems. *Operations Research Letters (ORL)*, 1998. (Cited on page 37)

[Solis *et al.*, 2021] Irving Solis, James Motes, Read Sandström, and Nancy M Amato. Representation-optimal multi-robot motion planning using conflict-based search. *IEEE Robotics and Automation Letters (RA-L)*, 2021. (Cited on pages 35, 217, 223, 232, and 244)

[Solovey and Halperin, 2016] Kiril Solovey and Dan Halperin. On the hardness of unlabeled multi-robot motion planning. *International Journal of Robotics Research (IJRR)*, 2016. (Cited on page 36)

[Solovey *et al.*, 2016] Kiril Solovey, Oren Salzman, and Dan Halperin. Finding a needle in an exponential haystack: Discrete rrt for exploration of implicit roadmaps in multi-robot motion planning. *International Journal of Robotics Research (IJRR)*, 2016. (Cited on page 35)

[Solovey, 2020] Kiril Solovey. Complexity of planning. *arXiv preprint*, 2020. (Cited on page 34)

[Spirakis and Yap, 1984] Paul Spirakis and Chee K Yap. Strong np-hardness of moving many discs. *Information Processing Letters*, 1984. (Cited on pages 2 and 34)

[Srinivasan *et al.*, 1990] Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K Brayton. Algorithms for discrete function manipulation. In *Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD)*, 1990. (Cited on page 135)

[Standley and Korf, 2011] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2011. (Cited on pages 31, 128, and 132)

[Standley, 2010] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2010. (Cited on pages 27, 28, 50, 101, 109, 114, 120, 126, 128, 132, 225, 227, and 238)

[Stern *et al.*, 2019] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2019. (Cited on pages 7, 22, 24, 37, 40, 53, 67, 68, 73, 92, 101, 110, 123, 126, 137, 138, 165, 175, 197, 217, and 241)

[Stern, 2019] Roni Stern. Multi-agent path finding–an overview. *Artificial Intelligence: 5th RAAI Summer School*, 2019. (Cited on page 27)

[Strawn and Ayanian, 2021] Kegan Strawn and Nora Ayanian. Byzantine fault tolerant consensus for lifelong and online multi-robot pickup and delivery. In *Proceedings of International Symposium on Distributed Robotic Systems (DARS)*, 2021. (Cited on pages 39 and 203)

[Sturtevant, 2012] Nathan R Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games (T-CIAIG)*, 2012. (Cited on page 137)

[Şucan *et al.*, 2012] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine (RAM)*, 2012. (Cited on pages 217 and 219)

[Surynek *et al.*, 2016] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient sat approach to multi-agent path finding under the sum of costs objective. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, 2016. (Cited on pages 27, 28, and 271)

[Surynek *et al.*, 2022] Pavel Surynek, Roni Stern, Eli Boyarski, and Ariel Felner. Migrating techniques from search-based multi-agent path finding solvers to sat-based approach. *Journal of Artificial Intelligence Research (JAIR)*, 2022. (Cited on page 29)

[Surynek, 2009] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2009. (Cited on pages 6, 27, 30, and 79)

[Surynek, 2010] Pavel Surynek. An optimization variant of multi-robot path planning is intractable. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2010. (Cited on page 26)

[Surynek, 2012] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *Proceedings of Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, 2012. (Cited on page 28)

[Surynek, 2013] Pavel Surynek. Redundancy elimination in highly parallel solutions of motion coordination problems. *International Journal on Artificial Intelligence Tools (IJAIT)*, 2013. (Cited on pages 128 and 132)

[Surynek, 2019] Pavel Surynek. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2019. (Cited on pages 128 and 203)

[Švancara *et al.*, 2019] Jiří Švancara, Marek Vlk, Roni Stern, Dor Atzmon, and Roman Barták. Online multi-agent pathfinding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2019. (Cited on page 131)

[Švestka and Overmars, 1998] Petr Švestka and Mark H Overmars. Coordinated path planning for multiple robots. *Robotics and autonomous systems*, 1998. (Cited on pages 35 and 230)

[Tel, 2000] Gerard Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000. (Cited on page 153)

[Tordesillas and How, 2021] Jesus Tordesillas and Jonathan P How. Mader: Trajectory planner in multiagent and dynamic environments. *IEEE Transactions on Robotics (T-RO)*, 2021. (Cited on page 39)

[Turpin *et al.*, 2014] Matthew Turpin, Kartik Mohta, Nathan Michael, and Vijay Kumar. Goal assignment and trajectory planning for large teams of interchangeable robots. *Autonomous Robots (AURO)*, 2014. (Cited on pages 35, 37, and 99)

[Vainshtain and Salzman, 2021] David Vainshtain and Oren Salzman. Multi-agent terraforming: Efficient multi-agent path finding via environment manipulation. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2021. (Cited on page 245)

[Van Den Berg and Overmars, 2005] Jur P Van Den Berg and Mark H Overmars. Prioritized motion planning for multiple robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005. (Cited on pages 30, 42, 67, 76, 108, 109, 121, 217, 218, 223, and 232)

[Van Den Berg *et al.*, 2011] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics Research*. 2011. (Cited on pages 21, 39, and 182)

[Vedder and Biswas, 2021] Kyle Vedder and Joydeep Biswas. X$*$: Anytime multi-agent path finding for sparse domains using window-based iterative repairs. *Artificial Intelligence (AIJ)*, 2021. (Cited on pages 128 and 132)

[Velagapudi *et al.*, 2010] Prasanna Velagapudi, Katia Sycara, and Paul Scerri. Decentralized prioritized planning in large multirobot teams. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010. (Cited on page 75)

[Virmani *et al.*, 2021] Lakshay Virmani, Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Subdimensional expansion using attention-based learning for multi-agent path finding. *arXiv preprint*, 2021. (Cited on page 242)

[Wagner and Choset, 2015] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence (AIJ)*, 2015. (Cited on pages 11, 27, 28, 35, 40, 41, 114, 120, 126, and 128)

[Wagner and Choset, 2017] Glenn Wagner and Howie Choset. Path planning for multiple agents under uncertainty. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2017. (Cited on pages 38 and 153)

[Wagner *et al.*, 2012a] Glenn Wagner, Howie Choset, and Nora Ayanian. Subdimensional expansion and optimal task reassignment. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2012. (Cited on page 37)

[Wagner *et al.*, 2012b] Glenn Wagner, Minsu Kang, and Howie Choset. Probabilistic path planning for multiple robots with subdimensional expansion. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2012. (Cited on page 35)

[Walker *et al.*, 2018] Thayne T Walker, Nathan R Sturtevant, and Ariel Felner. Extended increasing cost tree search for non-unit cost domains. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2018. (Cited on page 32)

[Wang and Botea, 2011] Ko-Hsin Cindy Wang and Adi Botea. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research (JAIR)*, 2011. (Cited on pages 31, 75, and 79)

[Wang *et al.*, 2008] Ko-Hsin Cindy Wang, Adi Botea, et al. Fast and memory-efficient multi-agent pathfinding. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2008. (Cited on page 31)

[Wei *et al.*, 2014] Changyun Wei, Koen V Hindriks, and Catholijn M Jonker. Multi-robot cooperative pathfinding: A decentralized approach. In *Proceedings of International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA-AEI)*, 2014. (Cited on page 76)

[Wen *et al.*, 2022] Licheng Wen, Yong Liu, and Hongliang Li. Cl-mapf: Multi-agent path finding for car-like robots with kinematic and spatiotemporal constraints. *Robotics and Autonomous Systems*, 2022. (Cited on page 32)

[Werfel *et al.*, 2014] Justin Werfel, Kirstin Petersen, and Radhika Nagpal. Designing collective behavior in a termite-inspired robot construction team. *Science*, 2014. (Cited on page 2)

[Wiktor *et al.*, 2014] Adam Wiktor, Dexter Scobee, Sean Messenger, and Christopher Clark. Decentralized and complete multi-robot motion planning in confined spaces. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014. (Cited on pages 67, 76, and 203)

[Wu *et al.*, 2020] Wenying Wu, Subhrajit Bhattacharya, and Amanda Prorok. Multi-robot path deconfliction through prioritization by path prospects. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2020. (Cited on page 30)

[Wurman *et al.*, 2008] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 2008. (Cited on pages 1, 4, 35, 37, 53, and 130)

[Yakovlev and Andreychuk, 2017] Konstantin Yakovlev and Anton Andreychuk. Any-angle pathfinding for multiple agents based on sipp algorithm. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2017. (Cited on pages 32 and 223)

[Yakovlev *et al.*, 2019] Konstantin Yakovlev, Anton Andreychuk, and Vitaly Vorobyev. Prioritized multi-agent path finding for differential drive robots. In *Proceedings of European Conference on Mobile Robots (ECMR)*, 2019. (Cited on page 32)

[Yu and LaValle, 2013a] Jingjin Yu and Steven M LaValle. Multi-agent path planning and network flow. 2013. (Cited on pages 36, 79, 85, 92, 93, 99, 150, 269, and 270)

[Yu and LaValle, 2013b] Jingjin Yu and Steven M LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2013. (Cited on pages 3, 25, 26, 70, 79, and 130)

[Yu and LaValle, 2016] Jingjin Yu and Steven M LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics (T-RO)*, 2016. (Cited on pages 27, 28, and 271)

[Yu and Rus, 2015] Jingjin Yu and Daniela Rus. Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. 2015. (Cited on pages 3, 25, 28, and 166)

[Yu, 2015] Jingjin Yu. Intractability of optimal multirobot path planning on planar graphs. *IEEE Robotics and Automation Letters (RA-L)*, 2015. (Cited on page 26)

[Zhang *et al.*, 2016] Yu Zhang, Kangjin Kim, and Georgios Fainekos. Discof: Cooperative pathfinding in distributed systems with limited sensing and communication range. In *Proceedings of International Symposium on Distributed Robotic Systems (DARS)*. 2016. (Cited on pages 67, 76, and 203)

[Zhang *et al.*, 2018] Xu Zhang, Mingyang Li, Jian Hui Lim, Yiwei Weng, Yi Wei Daniel Tay, Hung Pham, and Quang-Cuong Pham. Large-scale 3d printing by a team of mobile robots. *Automation in Construction*, 2018. (Cited on page 2)

[Zhang *et al.*, 2020] Xinya Zhang, Robert Belfer, Paul G Kry, and Etienne Vouga. C-space tunnel discovery for puzzle path planning. *ACM Transactions on Graphics (TOG)*, 2020. (Cited on page 49)

[Zhang *et al.*, 2022a] Han Zhang, Jiaoyang Li, Pavel Surynek, TK Satish Kumar, and Sven Koenig. Multi-agent path finding with mutex propagation. *Artificial Intelligence (AIJ)*, 2022. (Cited on pages 29, 40, and 67)

[Zhang *et al.*, 2022b] Ketao Zhang, Pisak Chermprayong, Feng Xiao, Dimos Tzoumanikas, Barrie Dams, Sebastian Kay, Basaran Bahadir Kocer, Alec Burns, Lachlan Orr, Christopher Choi, et al. Aerial additive manufacturing with multiple autonomous robots. *Nature*, 2022. (Cited on page 2)

[Zhang *et al.*, 2022c] Shuyang Zhang, Jiaoyang Li, Taoan Huang, Sven Koenig, and Bistra Dilkina. Learning a priority ordering for prioritized planning in multi-agent path finding. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*, 2022. (Cited on pages 30 and 244)

[Zhou *et al.*, 2017] Dingjiang Zhou, Zijian Wang, Saptarshi Bandyopadhyay, and Mac Schwager. Fast, on-line collision avoidance for dynamic vehicles using buffered voronoi cells. *IEEE Robotics and Automation Letters (RA-L)*, 2017. (Cited on page 39)

[Zhou *et al.*, 2018] Lifeng Zhou, Vasileios Tzoumas, George J Pappas, and Pratap Tokekar. Resilient active target tracking with multiple robots. *IEEE Robotics and Automation Letters (RA-L)*, 2018. (Cited on pages 39 and 203)

[Zilberstein, 1996] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 1996. (Cited on pages 12 and 131)

Part IV

Appendix

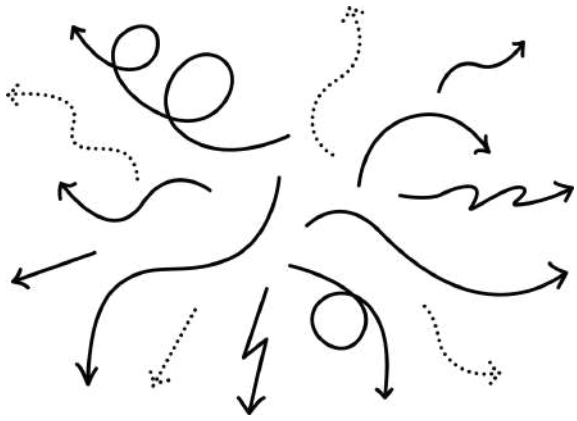


Image by loochanin / Pixabay License

Appendix A

Publications

A.1 Journal Publications

- **Keisuke Okumura**, Manao Machida, Xavier Défago & Yasumasa Tamura. "Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding." *Artificial Intelligence (AIJ)*, 310, p.103752, 2022. DOI: https://doi.org/10.1016/j.artint.2022.103752. Extended version from IJCAI-19.

A.2 Conference Proceedings

- **Keisuke Okumura**. "LaCAM: Search-Based Algorithm for Quick Multi-Agent Pathfinding." In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2023. In press.
- **Keisuke Okumura** & Sebastien Tixeuil. "Fault-Tolerant Offline Multi-Agent Path Planning." In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2023. In press.
- **Keisuke Okumura**, François Bonnet, Yasumasa Tamura & Xavier Défago. "Offline Time-Independent Multi-Agent Path Planning." In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 4649–4656, 2022. DOI: https://doi.org/10.24963/ijcai.2022/645.
- **Keisuke Okumura** & Xavier Défago. "Solving Simultaneous Target Assignment and Path Planning Efficiently with Time-Independent Execution." In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 270–278, 2022. DOI: https://doi.org/10.1609/icaps.v32i1.19810. **The ICAPS 2022 Best Student Paper Award**.
- **Keisuke Okumura**, Ryo Yonetani, Mai Nishimura & Asako Kanezaki. "CTRMs: Learning to Construct Cooperative Timed Roadmaps for Multi-agent Path Planning in Continuous Spaces." In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pp. 972–981, 2022. DOI: https://dl.acm.org/doi/10.5555/3535850.3535959.
- **Keisuke Okumura**, Yasumasa Tamura & Xavier Défago. "Iterative Refinement for Real-Time Multi-Robot Path Planning." In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9690-9697, 2021. DOI: https://doi.org/10.1109/IROS51168.2021.9636071.
- **Keisuke Okumura**, Manao Machida, Xavier Défago & Yasumasa Tamura. "Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding." *Proceedings*

of International Joint Conference on Artificial Intelligence (IJCAI), pp. 535-542, 2019. DOI: https://doi.org/10.24963/ijcai.2019/76.

A.3 Under Review Submissions

- **Keisuke Okumura** & Xavier Défago. "Quick Multi-Robot Motion Planning by Combining Sampling and Search." Under review at a conference on AI & robotics. Preprint on ArXiv: https://doi.org/10.48550/arXiv.2203.00315.
- **Keisuke Okumura**. "Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding." Under review at a conference on AI & robotics.
- **Keisuke Okumura**, François Bonnet, Yasumasa Tamura & Xavier Défago. "Offline Time-Independent Multi-Agent Path Planning." Under review at *IEEE Transactions on Robotics (T-RO)*. Extended version from IJCAI-22.
- **Keisuke Okumura** & Xavier Défago. "Solving Simultaneous Target Assignment and Path Planning Efficiently with Time-Independent Execution." Under review at *Artificial Intelligence (AIJ)*. Extended version from ICAPS-22 (**invited**).

A.4 Other Related Publications

- Shota Kameyama, **Keisuke Okumura**, Yasumasa Tamura & Xavier Défago. "Active Modular Environment for Robot Navigation." In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8636-8642, 2021. DOI: https://doi.org/10.1109/ICRA48506.2021.9561111.
- **Keisuke Okumura**, Yasumasa Tamura & Xavier Défago. "Time-Independent Planning for Multiple Moving Agents." In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, pp. 11299-11307, 2021. DOI: https://doi.org/10.1609/aaai.v35i13.17347.
- **Keisuke Okumura**, Yasumasa Tamura & Xavier Défago. "winPIBT: Extended Prioritized Algorithm for Iterative Multi-agent Path Finding." In *IJCAI Workshop on Multi-Agent Path Finidng (WoMAPF)*, 2021. DOI: https://doi.org/10.48550/arXiv.1905.10149.
- **Keisuke Okumura**, Yasumasa Tamura & Xavier Défago. "Amoeba Exploration: Coordinated Exploration with Distributed Robots" In *Proceedings of IEEE International Conference on Awareness Science and Technology (iCAST)*, pp. 191-195, 2018. DOI: https://doi.org/10.1109/ICAwST.2018.8517225.

Appendix B

PIBT

B.1 PIBT in Extremely Dense Situations

This section presents additional experiments on PIBT in extremely dense situations where $|A| > |V|/2$ and G satisfies the graph condition of Thrm. 4.2. The *empty-8-8* map and the same experimental settings were used as those introduced in Chap. 4.4.1.

Table B.1 summarizes the result. Most instances are solved at most 5 ms; otherwise, PIBT failed by reaching the makespan limit. Solution costs dramatically increase with more agents because, in dense situations, most agents cannot take their shortest paths.

$\|A\|$	success rate	runtime (ms)	sum-of-costs	makespan
40	0.96	0.21	3.15	3.46
50	0.84	1.43	7.38	6.94
60	1.00	2.16	12.25	7.86
64	1.00	3.16	21.55	10.01

Table B.1: PIBT in an extremely dense situation. The used map is *empty-8-8*. 25 instances were prepared for each $|A|$. Scores of "sum-of-costs" and "makespan" are upper bounds of sub-optimality in the same way as Fig. 4.6.

Interestingly, PIBT solved all instances with 64 agents (fully occupied), while sometimes failing with 50 agents. This is because of the randomness of the tie-break rule at Line 9 of Alg. 4.2. As a discussion of the implementation level, we used three rules when sorting candidate nodes for the following location:

1. distances to the goal,
2. the presence of agents to avoid unnecessary priority inheritance, and
3. random values, when neither the prior two break a tie.

The rule-2 omitted version was also tested, succeeding for all instances regardless of $|A|$. Since rule-2 loses its meaning in situations where $|A| = |V|$, we consider that the randomness contributes to solving extremely dense situations. We also observed that this trend is the same when testing a 16×16 empty grid, i.e., usual PIBT sometimes failed whereas PIBT that omits the rule-2 tie-break solved all instances regardless of $|A|$. From these observations, it is guessed that in tidy environments like *empty-8-8*, PIBT without the rule-2 can solve one-shot MAPF with a high probability as makespan increases. This is an interesting direction to seek but significantly beyond the dissertation scope.

Note that the aforementioned discussion is not applicable when G does not satisfy the graph condition of Thrm. 4.2. Moreover, rule-2 is usually effective in improving sum-of-costs, as shown in Table B.2.

	success rate		sum-of-costs		makespan	
$\|A\|$	normal	random	normal	random	normal	random
100	1.00	1.00	1.04 (1.04, 1.05)	1.08 (1.07, 1.09)	1.00 (1.00, 1.00)	1.00 (1.00, 1.00)
300	1.00	1.00	1.10 (1.09, 1.10)	1.16 (1.15, 1.16)	1.00 (1.00, 1.00)	1.00 (1.00, 1.00)
500	0.96	1.00	1.15 (1.14, 1.15)	1.22 (1.22, 1.23)	1.00 (1.00, 1.00)	1.00 (1.00, 1.00)
700	0.96	0.96	1.20 (1.20, 1.20)	1.28 (1.27, 1.28)	1.00 (1.00, 1.00)	1.00 (1.00, 1.00)
900	0.88	0.96	1.25 (1.24, 1.25)	1.33 (1.32, 1.34)	1.00 (1.00, 1.00)	1.00 (1.00, 1.01)

Table B.2: Effect of tie-break strategy. "normal" is PIBT and "random" is PIBT without the rule-2 tie-break. The used map was *den520d*. For each $|A|$, 25 random scenarios were prepared, not equivalent to those in Chap. 4.4.1. The scores of sum-of-costs and makespan are the average of the upper bounds of sub-optimality with 95% confidence intervals, based on instances that were solved by both.

Appendix C

TSWAP

C.1 Implementation of the Polynomial-Time Makespan-Optimal Algorithm

In the experiments of TSWAP, the polynomial-time makespan-optimal algorithm [Yu and LaValle, 2013a] was used. This algorithm has several techniques to improve the runtime performance. Thus, prior to the experiments in Chap. 5.4, these techniques were tested to select the best one for each experimental setting. This section describes the details.

C.1.1 Algorithm Description

Given a timestep T, a decision problem of whether an unlabeled-MAPF instance has a solution with makespan T can be solved in polynomial time. This is achieved by a reduction to maximum flow problems on a large graph called *time expanded network* [Yu and LaValle, 2013a].[1] Let denote $\mathcal{N}_T$ be the time expanded network for makespan T. To clarify the context, the following uses "nodes" for the network $\mathcal{N}_T$ and "vertices" for the original graph G.

For each timestep $0 \leq t < T$ and each vertex $v \in V$, the network $\mathcal{N}_T$ has two nodes v^t_{in} and v^t_{out}. In addition, there are two special nodes *source* and *sink* to convert the unlabeled-MAPF instance to the maximum flow problem. $\mathcal{N}_T$ has five types of edges with a unit capacity. The intuitions are the following:

- $(v^t_{\text{in}}, v^t_{\text{out}})$: An agent can stay at v during $[t, t+1]$.
- $(u^t_{\text{in}}, v^t_{\text{out}})$ if $(u, v) \in E$: An agent can move from u to v during $[t, t+1]$.
- $(v^t_{\text{out}}, v^{t+1}_{\text{in}})$: Prevent vertex collisions.
- $(\text{source}, v^0_{\text{in}})$ if $v \in \mathcal{S}$: Initial locations.
- $(v^{T-1}_{\text{out}}, \text{sink})$ if $v \in \mathcal{G}$: Targets.

Figure C.1 shows an example of time expanded networks with the maximum flows. Once the maximum flow with a size equal to $|A|$ is obtained, the solution for the unlabeled-MAPF instance is easily obtained from the flow.

Since many polynomial-time maximum flow algorithms exist, the maximum flow problem for time expanded networks can be solved in polynomial-time. For instance, the time complexity of the Ford-Fulkerson algorithm [Ford and Fulkerson, 1956], a major algorithm for the maximum flow problem, is $O(fE')$ where f is the maximum flow

[1] The structure of the network is slightly changed from the original paper to make the network slim, i.e., removing internal two nodes for preventing edge collisions. In the unlabeled setting, plans with edge collisions can be easily converted to plans without collisions. This technique is used in [Ma *et al.*, 2016; Liu *et al.*, 2019].

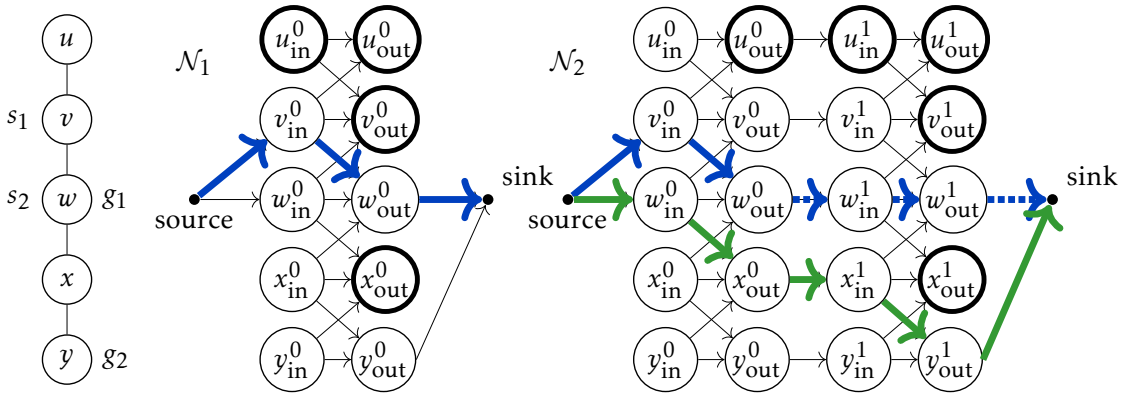


Figure C.1: Examples of time expanded network and two techniques (prune and reuse). The left shows an unlabeled-MAPF instance. The center shows $\mathcal{N}_1$ with the maximum flow (blue). Since the size of the maximum flow is not equal to $|A|$, there is no feasible solution with makespan $T = 1$. The right shows $\mathcal{N}_2$ with the maximum flow (blue and green). The resulting solution is $\Pi_1 = (v, w, w)$ and $\Pi_2 = (w, x, y)$. Since nodes with bold lines, e.g., u^0_{out} in both networks, never reach the sink, they can be pruned during the search for augmenting paths. When extending timestep, the past flow (blue solid line in $\mathcal{N}_1$) can be effectively reused to create a new flow (blue dotted line in $\mathcal{N}_2$).

size and E' denotes edges in the network; the running time in $\mathcal{N}_T$ is $O(AVT)$ with a natural assumption of $E = O(V)$. According to [Yu and LaValle, 2013a], $T = |A| + |V| - 2$ in the worst case, thus, the time complexity is $O\left(AV^2\right)$.

Using the above scheme, the remaining problem is to find an optimal T. This phase has many design choices. The typical one is incremental search (i.e., $T = 1, 2, \ldots$).

C.1.2 Techniques

This part introduces three effective techniques to speed up the optimal algorithm, assuming that the Fold-Furlkerson algorithm is used to find the maximal flow. The first two techniques are about finding an optimal makespan T. The last one is for reducing the search effort of the maximum flow; this is new in the MAPF literature.

Lower Bound

Starting the search for T from the makespan lower bound is expected to reduce the computational effort because the number of solving the maximum flow problems is reduced. A naive approach to obtain the bound is computing $\max_i \min_j \mathrm{h}(s_i, g_j)$.[2] A tighter bound is obtained by solving the bottleneck assignment problem [Gross, 1959], i.e., assigning each agent to one target while minimizing the maximum cost, regarding distances between initial locations and targets as costs. This bound is easily obtained by an adaptive version of Alg. 5.4.

Pruning of Redundant Nodes

During the search for augmenting paths, nodes that never reach the sink can be pruned. In Fig. C.1, such nodes are highlighted by bold lines. The pruning is realized by two processes.

[2] Or, $\mathtt{dist}(s_i, g_j)$ but this is avoided because in most cases the admissible heuristics work well and it is much faster.

- Preprocessing: Before searching optimal makespans, calculate the minimum distance to reach one of the targets from each vertex $v \in V$. Let denote this distance $\lambda(v)$, e.g., $\lambda(u) = 2$ in Fig. C.1. This is computed by a one-shot breadth-first search from all targets; its time complexity is $O(V + E)$, i.e., the overhead of the preprocessing.
- Pruning: During the search of augmenting paths, v^t_{out} such that $t + \lambda(v) \geq T$ is avoided from expanding as successors. This also prevents from expanding v^{t+1}_{in}.

Pruning reduces the search time of the maximum flow algorithm without affecting its correctness and optimality. This concept to flow network can be seen in [Yu and LaValle, 2016], while similar concepts can be seen in other reduction-based approaches to *labeled* MAPF, e.g., SAT-based [Surynek *et al.*, 2016] and ASP-based [G'omez *et al.*, 2020].

Reuse of Past Flows

Consider the incremental search of optimal makespan and expanding the network from $\mathcal{N}_T$ to $\mathcal{N}_{T+1}$. The Ford-Fulkerson algorithm iteratively finds an augmenting path until no such path exists. Thus, a reduction in the iterations is expected to reduce computation time.

A feasible flow of $\mathcal{N}_{T+1}$ with a size equal to the maximum flow of $\mathcal{N}_T$ can be obtained immediately without a search. To see this, let v^{T-1}_{out} be a node used in the maximum flow of $\mathcal{N}_T$. Let this flow extending for $\mathcal{N}_{T+1}$ by using v^{T-1}_{out}, v^T_{in}, v^T_{out}, and the sink. In Fig. C.1, the example of $\mathcal{N}_2$ is shown, highlighted by a blue dotted line started from w^0_{out}. This new flow is trivially feasible in $\mathcal{N}_2$; in general, it is feasible in $\mathcal{N}_{T+1}$. As a result, the Ford-Fulkerson algorithm in $\mathcal{N}_2$ only needs to find one augmenting path (green), rather than two. Hence, the reuse of the past flow contributes to reducing the iterations of the Ford-Fulkerson algorithm.

C.1.3 Evaluation of Techniques

The three techniques were evaluated using a four-connected grid *random-64-64-20*, shown in Fig. 5.7, while changing the number of agents. The simulator and the experimental environment were the same as Chap. 5.4. All instances were created by choosing randomly initial locations and targets.

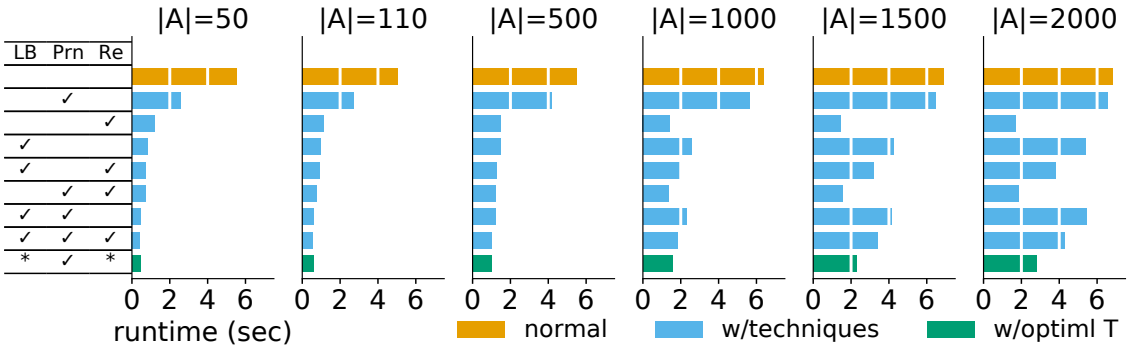


Figure C.2: The average runtime of the optimal algorithm in *random-64-64-20*. Checkmarks at "LB" mean starting the search from the lower bound obtained by Alg. 5.4; otherwise, it is obtained by computing $\max_i \min_j h(s_i, g_j)$. "Prn" stands for pruning. "Re" stands for reusing past flows.

The average runtime over 50 instances is shown in Fig. C.2. The figure additionally shows a single run of the maximum flow algorithm with optimal makespan, unknown be-

fore experiments (green bars). Since all combinations yield optimal solutions, the smaller runtime is better.

As for the technique of the lower bounds, the following two were tested: the conservative one obtained by $\max_i \min_j \mathrm{h}(s_i, g_j)$ (without checkmarks at "LB"), or, the aggressive one obtained by solving the bottleneck assignment problem using Alg. 5.4 (with checkmarks). The runtime includes computing the bounds. The aggressive one has an advantage when the number of agents is small. However, as increasing, solving the assignment problem itself takes time then it loses the advantage. Rather, the conservative one scores smaller runtime.

The other two techniques surely contribute to reducing runtime. Notably, the best runtimes with the proposed techniques (blue) do not differ or are faster than those given the optimal makespan (green).

C.1.4 Implementations in the Experiments

Following the above result, in our experiments, the optimal algorithm used the techniques of the aggressive "LB", "Prn", and "Re" except for $|A| \geq 1000$; in this case, it used the conservative "LB" instead of the aggressive one because Alg. 5.4 becomes costly. *brc202d* is an exception; aggressive "LB" was used even when $|A| \geq 1,000$. Since the map is too large, conservative "LB" more often failed to find solutions within a time limit.

Appendix D

LaCAM

D.1 Detailed Results of MAPF Benchmark

Figures D.1 to D.4 present full results of the MAPF benchmark, complementing Fig. 6.14. In addition to sum-of-loss, the figures also present makespan normalized by $\max_{i \in A} \mathtt{dist}(s_i, g_i)$. Scores of LaCAM* are for initial solutions.

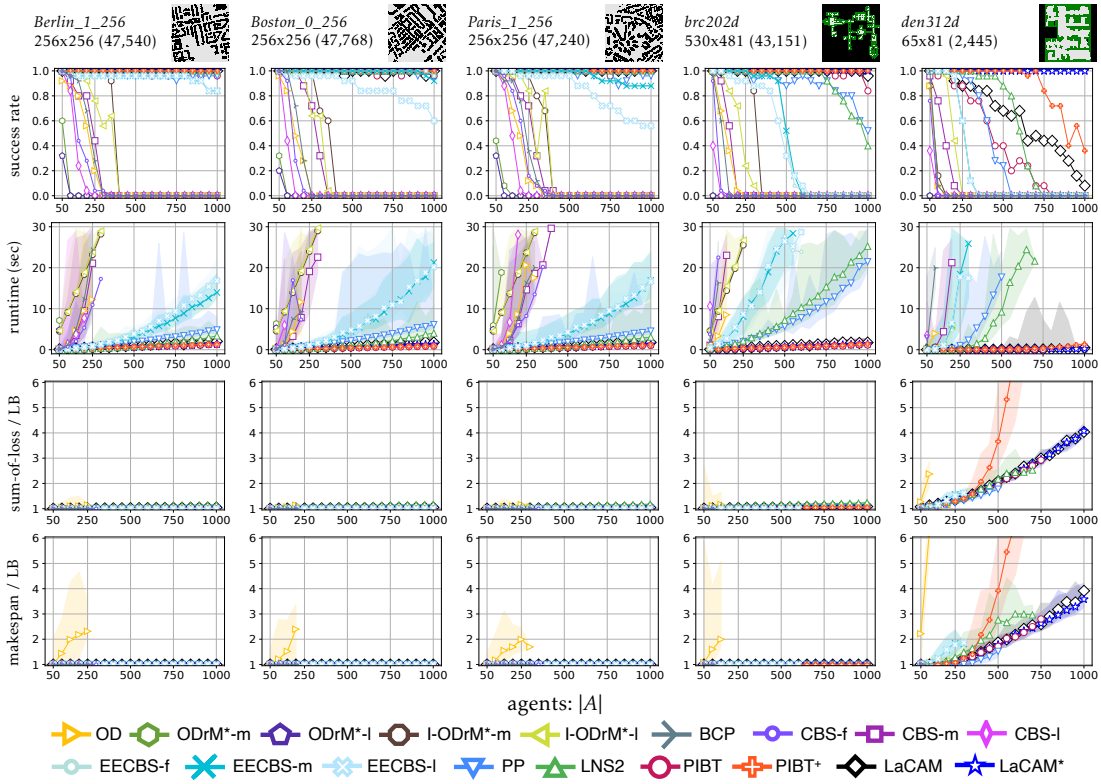


Figure D.1: Result of the MAPF benchmark (1/4). See also the caption of Fig. 6.14. $|V|$ is shown in parentheses.

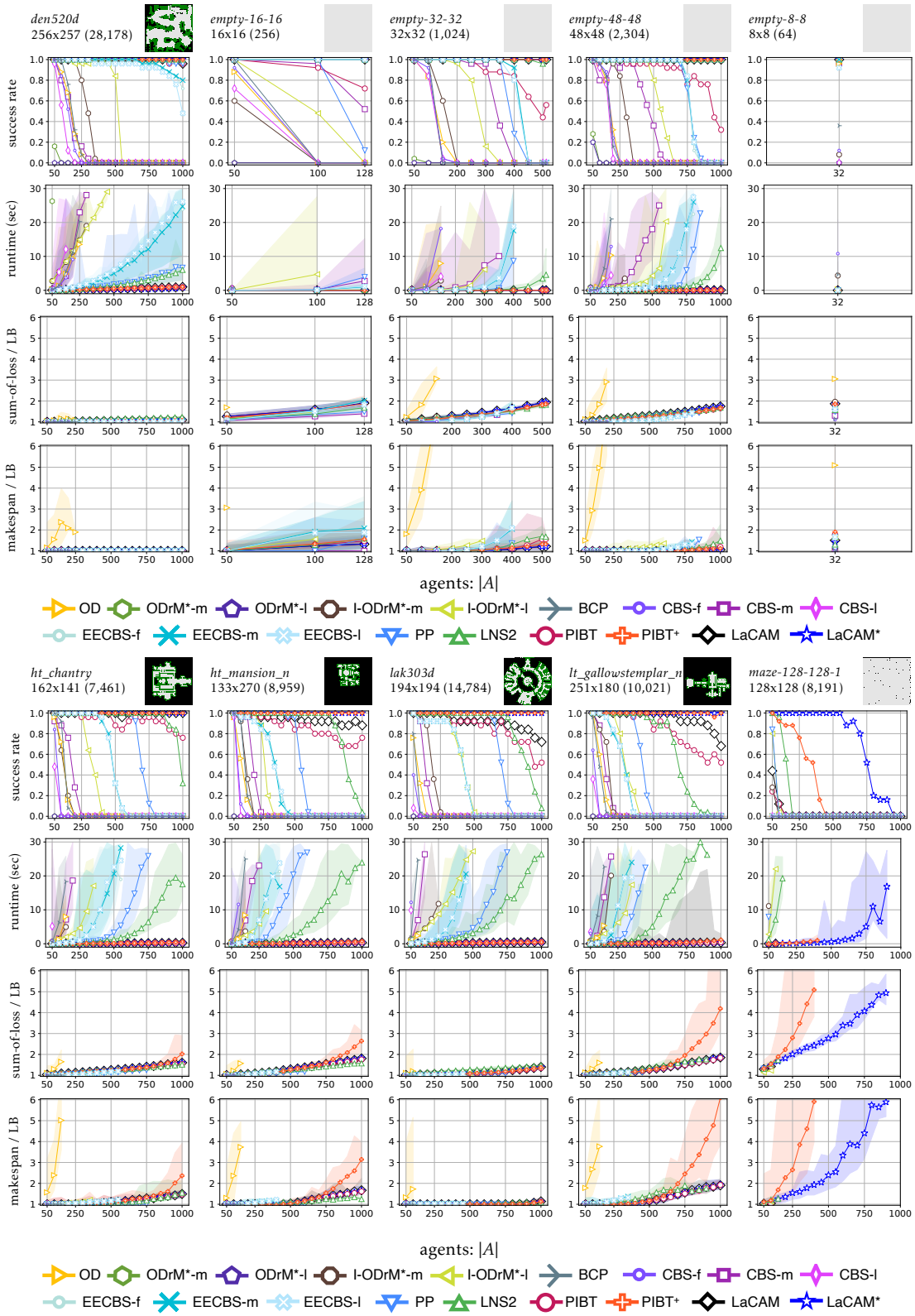


Figure D.2: Result of the MAPF benchmark (2/4).

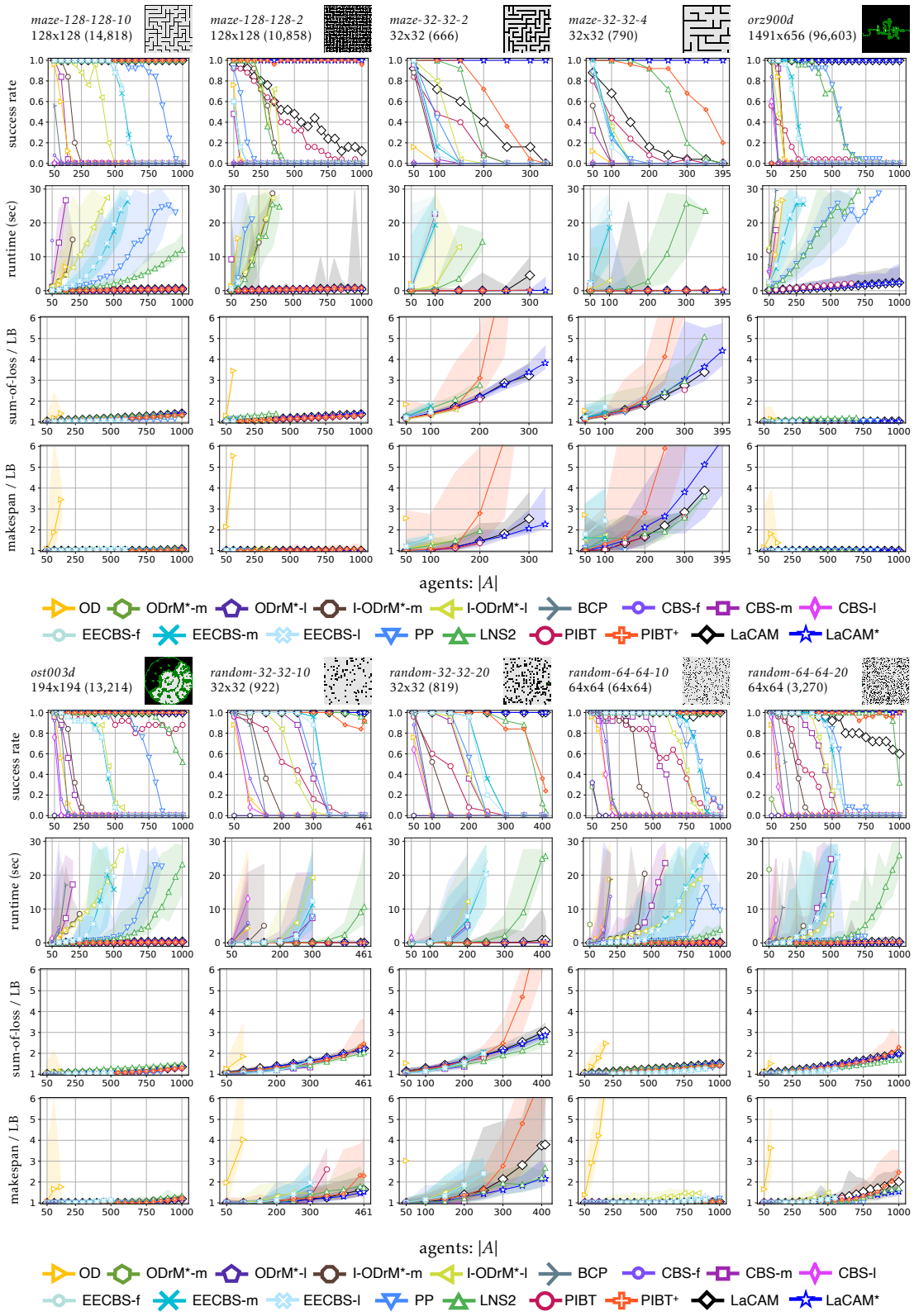


Figure D.3: Result of the MAPF benchmark (3/4).

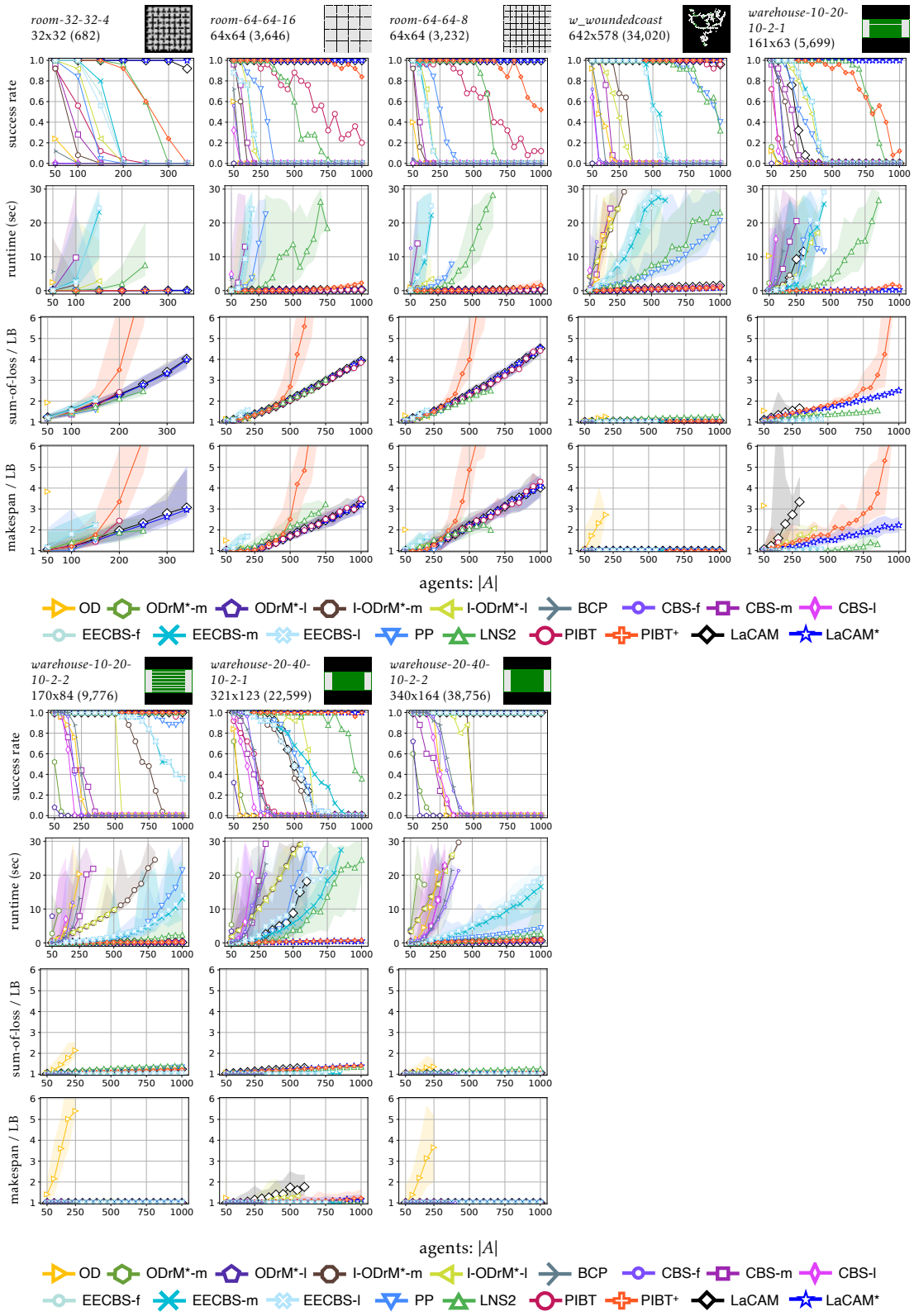


Figure D.4: Result of the MAPF benchmark (4/4).

Appendix E

CTRM

E.1 Conditional Variational Autoencoder and its Training

E.1.1 Basic Formulation of CVAE

The objective of CVAE [Sohn *et al.*, 2015] is to approximate a conditional probability distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ of vector $\boldsymbol{y}$ given another vector $\boldsymbol{x}$. CVAE consists of *encoder* $\mathrm{Enc}(\boldsymbol{x};\theta)$ and *decoder* $\mathrm{Dec}(\boldsymbol{x},\boldsymbol{z};\phi)$, which are typically neural networks parameterized by θ and ϕ, respectively. The encoder $\mathrm{Enc}(\boldsymbol{x};\theta)$ takes $\boldsymbol{x}$ as input to produce a conditional probability distribution $p_\theta(\boldsymbol{z} \mid \boldsymbol{x})$, where $\boldsymbol{z}$ is a latent variable that represents $\boldsymbol{x}$ in a low-dimensional space called latent space. Here, p_θ is modeled by a discrete, categorical distribution following [Ivanovic *et al.*, 2021], but it is also possible to consider a continuous distribution (such as Gaussian distribution). As for the decoder $\mathrm{Dec}(\boldsymbol{x},\boldsymbol{z};\phi)$, it receives a concatenation of $\boldsymbol{x}$ and $\boldsymbol{z}$ sampled from p_θ to approximate another conditional probability distribution $p_\phi(\boldsymbol{y} \mid \boldsymbol{x},\boldsymbol{z})$. By marginalizing out $\boldsymbol{z}$, we can obtain $p(\boldsymbol{y} \mid \boldsymbol{x})$ as follows.

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{\boldsymbol{z}} p_\phi(\boldsymbol{y} \mid \boldsymbol{x},\boldsymbol{z}) p_\theta(\boldsymbol{z} \mid \boldsymbol{x}) \tag{E.1}$$

E.1.2 CVAE with Importance Sampling

To perform the marginalization of Eq. (E.1), we expect $\boldsymbol{z}$ to contain some information about $\boldsymbol{y}$, otherwise $p_\theta(\boldsymbol{z} \mid \boldsymbol{x})$ will contribute very little to $p(\boldsymbol{y} \mid \boldsymbol{x})$. A typical approach to this problem is *importance sampling* [Bishop and Nasrabadi, 2006], where we sample $\boldsymbol{z}$

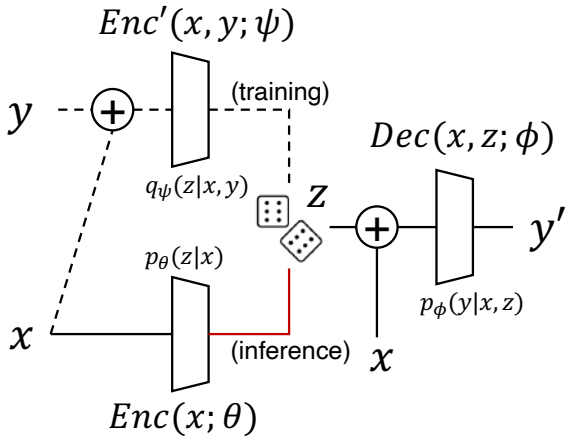


Figure E.1: CAVE architecture.

from a *proposal distribution* $q(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})$ to select a proper $\boldsymbol{z}$. To this end, another neural network-based encoder $\mathrm{Enc}'(\boldsymbol{x}, \boldsymbol{y}; \psi)$ is introduced, parameterized by ψ, which takes the concatenation of $\boldsymbol{x}$ and $\boldsymbol{y}$ to produce the proposal distribution $q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})$. Equation (E.1) can then be rewritten as follows:

$$\begin{aligned} p(\boldsymbol{y} \mid \boldsymbol{x}) &= \sum_{\boldsymbol{z}} \frac{p_\phi(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{z}) p_\theta(\boldsymbol{z} \mid \boldsymbol{x})}{q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})} q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y}) \\ &= \mathbb{E}_{q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})} \left[\frac{p_\phi(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{z}) p_\theta(\boldsymbol{z} \mid \boldsymbol{x})}{q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})} \right] \end{aligned} \tag{E.2}$$

By taking the log of both sides and using Jensen's inequality, the *evidence lower-bound (ELBO)* is obtained on the right-hand side in the following inequality:

$$\log p(\boldsymbol{y} \mid \boldsymbol{x}) \geq \mathbb{E}_{q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})} \left[\log p_\phi(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{z}) \right] - \mathcal{D}_{\mathrm{KL}} \left[q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y}) \parallel p_\theta(\boldsymbol{z} \mid \boldsymbol{x}) \right] \tag{E.3}$$

where $\mathcal{D}_{\mathrm{KL}}[p \parallel q] := \mathbb{E}_p[\log(p(\boldsymbol{x})/q(\boldsymbol{x}))]$ is a Kullback-Leibler (KL) divergence.

E.1.3 Training CVAE

Given a collection of pairs of $\boldsymbol{x}$ and $\boldsymbol{y}$, we want to maximize this ELBO with respect to θ, ϕ, ψ to achieve $p(\boldsymbol{y} \mid \boldsymbol{x})$. More specifically, the training of CVAE proceeds as follows (see also Fig. E.1). First, the encoder $\mathrm{Enc}(\boldsymbol{x}; \theta)$ takes $\boldsymbol{x}$ as input to output the log of $p_\theta(\boldsymbol{z} \mid \boldsymbol{x})$. At the same time, the other encoder $\mathrm{Enc}'(\boldsymbol{x}, \boldsymbol{y}; \psi)$ receives the concatenation of $\boldsymbol{x}$ and $\boldsymbol{y}$ to output the log of $q_\psi(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y})$. Then, a latent variable $\boldsymbol{z}$ is drawn from q_ψ, concatenated with $\boldsymbol{x}$, and fed to the decoder $\mathrm{Dec}(\boldsymbol{x}, \boldsymbol{z}; \phi)$ to obtain $\boldsymbol{y}'$. This $\boldsymbol{y}'$ can be regarded as a sample drawn from $p(\boldsymbol{y} \mid \boldsymbol{x})$ under the current parameters θ, ϕ, and ψ. Therefore, minimizing the discrepancy between $\boldsymbol{y}'$ and $\boldsymbol{y}$ corresponds to maximizing the first log-likelihood term of Eq. (E.3). Here, a reparameterization trick [Kingma and Welling, 2014; Eric Jang, 2017; Chris J. Maddison, 2017] is used for the sampling of latent variable $\boldsymbol{z}$ to enable the end-to-end learning of encoder $\mathrm{Enc}'(\boldsymbol{x}, \boldsymbol{y}; \psi)$ and decoder $\mathrm{Dec}(\boldsymbol{x}, \boldsymbol{z}; \phi)$ by means of back-propagation. Furthermore, it is easy to compute the second KL term of Eq. (E.3) directly from the outputs of $\mathrm{Enc}(\boldsymbol{x}; \psi)$ and $\mathrm{Enc}'(\boldsymbol{x}, \boldsymbol{y}; \psi)$. In practice, the implementation measures the L2 distance between $\boldsymbol{y}'$ and $\boldsymbol{y}$ for the first log-likelihood term, and gives a scalar weight 0.1 to the second KL term [Ichter *et al.*, 2018], as it performed empirically well.

E.1.4 Using CVAE as a Sampler

Once trained, CVAE in the inference time can be used as a conditional sampler taking $\boldsymbol{x}$ as input to generate plausible $\boldsymbol{y}'$, which is denoted by $\mathsf{F}_{\mathrm{CTRM}}(\boldsymbol{x})$ in our work. Note that we do not use $\mathrm{Enc}'(\boldsymbol{x}, \boldsymbol{y})$ anymore for this sampling.

E.1.5 Joint Training with Other Network Modules

As described in Chap. 11.3.3, CVAE is trained with the other network modules $\mathsf{NN}_{\mathrm{self\_env}}$, $\mathsf{NN}_{\mathrm{other\_env}}$, $\mathsf{NN}_{\mathrm{comm}}$, and $\mathsf{NN}_{\mathrm{ind}}$ jointly. Specifically, the study uses the negative log-likelihood (NLL) loss between the softmax output of $\mathsf{NN}_{\mathrm{ind}}$ and the ground-truth values of $\boldsymbol{x}_{\mathrm{ind}}$ computed exactly using solution paths in the training data. Then the multi-task loss is minimized, defined by the sum of CVAE loss and NLL loss scaled by a factor of 0.001. This scaling was necessary to roughly match the magnitude of the two losses. Since the outputs from $\mathsf{NN}_{\mathrm{self\_env}}$, $\mathsf{NN}_{\mathrm{other\_env}}$, and $\mathsf{NN}_{\mathrm{comm}}$ are used directly as inputs to the CVAE and $\mathsf{NN}_{\mathrm{ind}}$, the parameters of those modules can also be updated end-to-end by back-propagating the multi-task loss.