Offline Time-Independent Multiagent Path Planning

Keisuke Okumura*, Graduate Student Member, IEEE, François Bonnet*,
Yasumasa Tamura*, Member, IEEE, and Xavier Défago*

Abstract—This study examines a novel planning problem for multiple agents that cannot share holding resources, named Offline Time-Independent Multiagent Path Planning (OTIMAPP). 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. This motivation stems from timing uncertainties, including the reality gaps between planning and robot execution. In contrast to conventional solution concepts of multirobot path planning that rely on timings, once OTIMAPP solutions are obtained, they can be executed without any synchronization between robot actions. Moreover, there is a theoretical guarantee that all robots eventually reach their destinations, provided they avoid interrobot collisions. This study attempts to establish OTIMAPP both theoretically and practically. Specifically, we present a formalization of the problem, solution conditions based on a categorization of deadlocks, computational complexities showing that OTIMAPP is computationally intractable, practical relaxation of the solution concept, two algorithms to solve OTIMAPP based on multiagent pathfinding algorithms, empirical results showing large OTIMAPP instances can be solved to some extent, as well as robot demonstrations of asynchronous OTIMAPP execution.

Index Terms—Asynchronous execution, deadlock prevention, multirobot coordination, timing uncertainties, path planning.

I. INTRODUCTION

When multiple mobile robots operate in a shared workspace, the fundamental challenge is to design a coordination strategy that navigates them to their respective destinations. This is of particular importance considering each robot has a physical body and hence exclusively occupies a certain region in the workspace. Without coordination, a robot may be blocked by other robots and fail to reach its destination. A naive approach consists of preparing a set of collision-free trajectories before the robots start moving and then letting each robot follow the prepared trajectory precisely at runtime. The preparation phase is commonly formulated as multiagent path finding (MAPF) [1], which aims to find collision-free paths on graphs (i.e., discretized workspaces). In MAPF, a plan (i.e., a solution of offline planning, a set of paths) specifies the locations that each agent can use and the time at which it can be used. If an MAPF plan is followed, 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 [2]), 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 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. More specifically, the latter corresponds to the nonexistence 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 robots because the actions of the robots temporally depend on each other.

Consequently, we require a methodology to cope with the timing uncertainties of plan execution on real robots.

One countermeasure to timing uncertainties is online intervention at runtime, as shown in [3], [4], [5], and [6]. Considering an MAPF plan computed offline as input, these approaches use 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, these approaches require runtime effort. Further, they require additional and costly infrastructure, such as steady networks and monitoring systems, to manage the robots’
status in real time. Moreover, the realization of such schemes in large systems is not trivial. Another countermeasure involves the use of offline planning that incorporates timing uncertainties [7], [8], [9], [10]. Such approaches model uncertainties based on the probabilities of traveling time or assuming maximum delays. However, once failing to maintain the system status in predefined uncertainty models owing to black swan events (i.e., events that are unpredictable but bring fatal results), the system behavior is unpredictable and uncontrollable.

Instead of computing “timed” paths vulnerable to timing uncertainties, this study investigates a novel planning problem wherein robots spontaneously act without any timing assumptions. The problem requires a set of paths (i.e., solutions), ensuring that all robots eventually reach their destinations without permanently blocking each other. To observe this, consider the situation in Fig. 1 (left). This plan runs the risk of an execution failure. If robot $j$ is delayed for any reason while robot $i$ moves two steps to the right, then each robot blocks the other and neither agent can progress on its respective path. In contrast, in Fig. 1 (right), regardless of how the two robots are scheduled, both robots eventually reach their destinations unless they permanently stop the progression. We call the corresponding problem Offline Time-Independent Multiagent Path Planning (OTIMAPP).

Once OTIMAPP solutions are obtained, asynchronous execution of the solutions can be performed, with a theoretical guarantee that all robots eventually reach their destinations, provided the robots avoid collisions. Here, asynchronous execution refers to those without cumbersome action synchronization between multiple robots, 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), multirobot navigation can be performed only with local interactions without any central control, again with the theoretical guarantee as above.

Applications of OTIMAPP include multirobot systems, such as fleet operations of warehouses [11], intersection management for self-driving cars [12], and multirobot 3D printing systems [13], 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 [14], 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 [15], 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. In the remainder, we thus mainly use “agent” instead of “robot.”

A. Contributions

The contribution of this study is the establishment of the foundation of OTIMAPP both theoretically and practically. Specifically, the contributions of this study are categorized into theoretical and practical parts as follows:

1) Theoretical Part: We first formalize OTIMAPP, and then derive a necessary and sufficient condition for a solution, i.e., a set 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, we perform computational complexity analyses and reveal that 1) finding a solution is NP-hard, and 2) verifying a solution is co-NP-complete. Both proofs are by reductions of the 3-SAT problem. We further analyze 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 problem [16] (i.e., a generalization of a sliding puzzle) and MAPF. Against the others, OTIMAPP is restricted in solvability and results in higher optimal costs.

2) 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 [17], [18] and relied on a newly developed procedure to detect deadlocks within a set of paths. We further present 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 real robots, we evaluate these algorithms and demonstrate the following four:

1) Either PP or DBS can compute large OTIMAPP instances to a certain extent.
2) OTIMAPP solutions cause robots to move efficiently in an adverse environment for timing assumptions compared to existing approaches with runtime support [4], [19];
3) A relaxation of the solution concept ($m$-tolerant solutions) moderately offloads the burden of computation but with a risk of execution failure.
4) 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.

B. Paper Organization

Section II formulates the OTIMAPP problem. Section III identifies a necessary and sufficient condition. Section IV conducts computational complexity analyses. Section V analyzes the cost of time independence. Section VI presents how to
solve OTIMAPP. Section VII presents a relaxed solution concept. Section VIII presents empirical results including robot demonstrations. Section IX discusses related work. Section X concludes the paper with discussion of interesting directions to extend OTIMAPP. The code and movie are available at https://kei18.github.io/otimapp.

C. Difference From the Conference Version

OTIMAPP was originally presented in our preliminary paper [20]. In this article, in addition to improving the description and presentation of OTIMAPP, the following major differences are discussed.

1) In our prior work, the NP-hardness for feasibility on undirected graphs was applied only to solutions with simple paths. This article removes this limitation using a new proof (see Section IV-A).

2) This article analyzes and discusses the cost of time independence compared to other well-known multiagent planning problems (see Section V).

3) This article presents a deadlock detection procedure that is at the heart of solving OTIMAPP (see Section VI-A).

4) This article introduces the concept of relaxed solutions, called m-tolerant solutions, which are analyzed (see Section VII) and evaluated (see Section VIII-C).

5) This article discusses the pros and cons of PP and DBS, both qualitatively and quantitatively (see Section VI-D and VIII-A).

6) This article provides further empirical results of the scalability and delay tolerance of OTIMAPP for deeper insights (see Sections VIII-A and VIII-B).

III. Solution Analysis

Given a set of paths, the first question is whether it is a solution. This section derives the necessary and sufficient condition for the solutions. For this purpose, we introduce four types of deadlocks, 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. For instance, two paths ( . . . , u, v, . . . ) and ( . . . , v, u, . . . ), where u, v ∈ V, has a potential cyclic deadlock. Two paths ( . . . , v) and ( . . . , v, . . . ) has a potential terminal deadlock. Formal definitions are the following.

Definition 1 (Potential Cyclic Deadlock): Given an OTIMAPP instance and a set of paths {π1, . . . , πN}, a potential cyclic deadlock is a pair of tuples ((i, j, k, . . . , l), (t1, t2, t3, . . . , tk)) such that πi[t1 + 1] = πj[t2] ∧ πj[t2 + 1] = πk[t3] ∧ . . . ∧ πi[tk + 1] = πi[tk]. The elements of the first tuple are not duplicated.

Definition 2 (Potential Terminal Deadlock): Given an OTIMAPP instance and a set of paths {π1, . . . , πN}, a potential terminal deadlock is a tuple (i, j, t) such that πi[−1] = πj[t] and i ̸= j.

Definition 3 (Reachable Cyclic Deadlock): A potential cyclic deadlock ((i, j, . . . , l), (t1, t2, t3, . . . , tk)) is reachable when there is an execution schedule that leads to a situation, where clock
t_i \land \text{clock}_j = t_j \land \ldots \land \text{clock}_1 = t_1. \] This deadlock is called a reachable cyclic deadlock.

**Definition 4 (Reachable Terminal Deadlock):** A potential terminal deadlock \((i, j, t_j)\) is reachable when there is an execution schedule that leads to a situation where \(\text{clock}_i = |\pi_i| \land \text{clock}_j = t_j - 1\). This deadlock is called a reachable terminal deadlock.

We refer to both reachable (or potential) cyclic/terminal deadlocks as reachable (resp. potential) deadlocks and simply use “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. 1 (left), a schedule \((i, i, \ldots)\) is evidence. A potential deadlock is not always reachable as illustrated in Fig. 2.

**Theorem 5 (Necessary and Sufficient Condition):** Given an OTIMAPP instance, a set of paths \(\{\pi_1, \ldots, \pi_N\}\) is a solution if and only if there are:

1) No reachable terminal deadlocks.
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 cornerstones are necessary.

Now, consider the proof for these two conditions being sufficient. Given a solution candidate \(\{\pi_1, \ldots, \pi_N\}\) with no reachable deadlocks, we consider the potential function \(\phi := \sum_{i \in A} (|\pi_i| - \text{clock}_i)\) defined over a configuration \(\{\text{clock}_1, \ldots, \text{clock}_N\}\). As the progress indexes \(\text{clock}_i\) are non-decreasing, \(\phi\) is non-increasing, and \(\phi = 0\) means that all agents have reached their goals. Furthermore, when \(\phi > 0\), \(\phi\) 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 \(\phi\) 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 \(i \in B, \pi_i[\text{clock}_i + 1]\) 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 \(\pi_j[\text{clock}_j + 1] = \pi_k[\text{clock}_k]\). 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.

---

**Fig. 3. Unsolvable OTIMAPP instances.** To visualize instances, the figures distinguish three types of vertices: start (square), goal (circle), 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.

When reachable deadlocks are inevitable, OTIMAPP instances have no solution. Fig. 3 shows such examples.

---

**IV. COMPUTATIONAL COMPLEXITY**

This section discusses the complexity of OTIMAPP. In particular, we address two questions: the difficulty of finding solutions (see Section IV-A) and that of verifying solutions (see Section IV-B). 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.

**A. Finding Solutions**

We first derive the NP-hardness for directed graphs, then extend the proof to the case of undirected graphs. The following proof is partially inspired by the NP-hardness of MAPF in digraphs [22].

**Theorem 6 (Complexity on Digraphs):** OTIMAPP on directed graphs is NP-hard.

**Proof:** The proof is a reduction of the 3-SAT problem. Fig. 4 is an example of the reduction from a formula \((x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3)\).

1) **Construction of an OTIMAPP Instance:** We introduce 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 $\chi_i$ 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 $\neg x_i$, we further add one agent $c^j_k$ to the gadget. Its start and goal are positioned on the right side from $\chi_i$ when the literal is a negation; otherwise, they are positioned on the left side. $c^j_k$ 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 $\chi_i$ 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 $c^j_k$. The gadget contains a triangle. Each literal $c^j_k$ 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.

2) The Formula is Satisfiable if OTIMAPP has a Solution: The use of a 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. 4, if $c^j_k$ avoids using the clause constrainer, $\chi_3$ must take the right path. This sets $x_3$ to true, thus satisfying clause $C_2$.

3) OTIMAPP has a Solution if the Formula is Satisfiable: If satisfiable, let the variable agent $\chi_i$ follow a path that follows the assignment. Let $c^j_k$ take a path within the variable decider when $\chi_i$ follows the opposite direction. Otherwise, let $c^j_k$ use the clause constrainer. For instance, if $x_3$ is set to true, $\chi_3$ takes the right path, $c^j_3$ uses the path within the variable decider. In contrast, $c^j_3$ 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.

Theorem 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 $c^j_k$ corresponding to the $k$th literal in clause $C^j$ when the literal is either $x_i$ or $\neg x_i$. We revise the variable decider by adding a new agent $\zeta^j_k$ for each $c^j_k$ and converting all directed edges of the original variable decider into undirected edges. Fig. 5 shows an example of the revised one for $x_1$. The start and goal of $\zeta^j_k$ are positioned next to the start of $c^j_k$.

Using this revised gadget, we claim that an OTIMAPP instance has a solution if and only if the formula is satisfiable. This claim holds when solution paths for an agent $\chi_i$ 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, $c^j_k$ positioned on the left side must use a clause constrainer if $\chi_i$ 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.

![Figure 5](image-url)

Fig. 5. Variable decider of $x_1$ with undirected edges.

We now prove that $\chi_i$ takes the shortest paths in the OTIMAPP solutions, as follows:

1) $\zeta^j_k$ must take the shortest path (two steps): Observe that $\zeta^j_k$ must pass through at least one goal of another agent if it does not follow the shortest path. For instance, $\zeta^j_1$ must use either goal of $c^j_1, c^j_2,$ or $c^j_3$. Now, consider an execution schedule such that $\zeta^j_3$ is not activated for a sufficiently long time and remains at the start. In such execution schedules, the other agents reach their goal because the start of $\zeta^j_3$ never blocks any other paths. $\zeta^j_3$ has no choice other than to take the shortest path; otherwise, terminal deadlocks exist.

2) $\chi_i$ must take one of the shortest paths (two steps): Assume contrary that $\chi_i$ 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 $\chi_i$ (red-colored in Fig. 5). However, such paths must use one of the goals of $\zeta$ agents. As these $\zeta$ agents take their shortest paths, there are reachable terminal deadlocks.

Strictly speaking, neither $\chi_i$, nor $c^j_k$ must take their shortest paths. For instance, $\chi_1$ 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.

B. Verification

The co-NP completeness of the verification relies on a lemma that states that finding cyclic deadlocks is computationally intractable. Subsequently, the complexity result is derived because the solution has no reachable deadlocks. The entire proof of the lemma is provided in Appendix (Section IV-A).

Lemma 8 (Complexity of Detecting Cyclic Deadlocks): Determining whether a set of paths contains reachable or potential cyclic deadlocks is NP-complete.

Theorem 9 (Complexity of Verification): Verifying a solution of OTIMAPP is co-NP-complete.

Proof: Theorem 5 states that a solution has nonreachable terminal/cyclic deadlocks. Verifying no reachable terminal deadlocks is in co-NP; indeed, a reachable terminal deadlock is verifiable in polynomial time given an appropriate execution schedule. Verifying no reachable cyclic deadlocks is co-NP-complete, according to Lemma 8.
V. 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 [16], which is a generalization of a sliding puzzle, and the multiagent path finding (MAPF) problem [1], 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 the two problems is as follows:

**Definition 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.

**Definition 11 (MAPF):** The multiagent path finding (MAPF) 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, all agents perform their respective actions. Each agent has two options: either staying at the current vertex or moving to an adjacent vertex. Agents must avoid two types of collisions: occupying one vertex simultaneously, or traversing one edge simultaneously. 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.

A. 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. The summary is as follows.

**Proposition 12 (Solvability: PM Versus MAPF):** Solvable instances for PM are solvable for MAPF. The opposite does not hold.

OTIMAPP is more restrictive than PM and MAPF.

**Proposition 13 (Solvability: OTIMAPP Versus 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 schedule. Take one of the schedules. The corresponding execution can be emulated by PM, deriving the first claim, along with Proposition 12. For the second claim, we have already provided examples in Fig. 3.

B. Optimality

Next, we consider the solution quality, i.e., the cost of the solutions. For the optimization criteria of OTIMAPP, we use the minimum activation counts wherein all agents reach the goals, e.g., seven in Fig. 1 (see right). Let $C_{OPTIMAPP}$ 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. 1 is six (see left). Let $C_{PM}$ denote the optimal cost of PM. Note that solving PM optimally is NP-hard [23].

**Proposition 14 (Optimality: OTIMAPP Versus PM):** For any instances wherein OTIMAPP is solvable, $1 \leq C_{OPTIMAPP}/C_{PM}$. The bound is tight. For any $k \in \mathbb{R}$, there exists an instance where $C_{OPTIMAPP}/C_{PM} > k$.

Proof: $C_{OPTIMAPP} \geq C_{PM}$ holds because PM can emulate OTIMAPP execution. Fig. 6 shows an example, where the optimal ratio can be increased arbitrarily.

The cost of MAPF varies. Optimal solving of MAPF is known to be NP-hard for various criteria [24]. The most commonly used method is the number of operations (aka. makespan). Let $C_{MAPF}$ denote the optimal makespan of MAPF. Because MAPF can emulate PM, $C_{OPTIMAPP}$ is lower than $C_{PM}$. Furthermore, for solvable PM instances, there exists a polynomial-time procedure [16] to obtain a solution that requires $O(|V|^3)$ operations, where $|V|$ is the number of vertices. We conclude as follows.

**Proposition 15 (Optimality: PM Versus MAPF):** $1 \leq C_{PM}/C_{MAPF} \leq O(|V|^3)$ for the same inputs when PM is solvable.

**Proposition 16 (Optimality: OTIMAPP Versus MAPF):** For any instance in which OTIMAPP is solvable, $1 \leq C_{OPTIMAPP}/C_{MAPF}$.

C. Complexity

Finally, we summarize the computational complexity of the three problems.

Finding a solution to PM or MAPF in directed graphs is NP-hard [22]. Further, OTIMAPP on directed graphs is NP-hard (Theorem 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 [16]. By contrast, OTIMAPP on undirected graphs is NP-hard (Theorem 7).

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 (Theorem 9).
VI. SOLVERS

In this section, we shift our focus to solving OTIMAPP. In practice, using the necessary and sufficient condition is challenging (Theorem 5) because the corresponding schedules must be specified. This motivates building a relaxed sufficient condition.

Theorem 17 (Relaxed Sufficient Condition): Given an OTIMAPP instance, a set of paths \( \{\pi_1, \ldots, \pi_N\} \) is a solution when there are:

1. no use of other goals, i.e., \( g_j \notin \pi_i \) for all \( i \neq j \) except for \( s_i = g_j \);
2. no potential cyclic deadlocks.

Proof: Use Theorem 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 set of paths, “no use of other goals” can be easily checked, whereas “no potential cyclic deadlocks” is intractable in computation (Lemma 8). Nevertheless, detecting potential cyclic deadlock is the basis for solving OTIMAPP. Therefore, this section first explains the detection of potential cyclic deadlocks. Subsequently, two algorithms for solving OTIMAPP are presented.

A. Detection of Potential Cyclic Deadlocks

First, we introduce a fragment, a candidate for potential cyclic deadlocks.

Definition 18 (Fragment): Given a set of paths \( \{\pi_1, \ldots, \pi_N\} \), a fragment is a tuple \( ((i, j, k, \ldots, l), (t_i, t_j, t_k, \ldots, t_l)) \) such that \( \pi_i[t_i + 1] = \pi_j[t_j] \land \pi_j[t_j + 1] = \pi_k[t_k] \land \ldots = \pi_l[t_l] \). The elements of the first tuple are without duplicates.

We say that a fragment starts from a vertex \( u \) when \( \pi_i[t_i] = u \) and a fragment ends at a vertex \( v \) when \( \pi_i[t_i + 1] = v \). A fragment ending at its start (i.e., \( \pi_i[t_i + 1] = \pi_i[t_i] \)) is a potential cyclic deadlock.

Using fragments, Algorithm 1 detects a potential cyclic deadlock in a set 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 of Algorithm 1 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; and
4. if a fragment ends at its start, this is a potential cyclic deadlock.

We describe the algorithm details in the proof of completeness.

Theorem 19 (Completeness of Deadlock Detection): Algorithm 1 finds and returns a potential cyclic deadlock if at least one exists; otherwise, it returns NONE.

Proof: The algorithm uses two tables that store fragments: \( \Theta_f \) (“from” table) and \( \Theta_t \) (“to” table). Both tables use one vertex as the key. One entry in \( \Theta_f \) stores all the fragments starting from the vertex, while one entry in \( \Theta_t \) stores all the fragments ending at the vertex. A fragment is registered in both tables. We now derive the theorem through induction on \( \pi_i \).

Base case: In the first iteration of the loop [Lines 11–31], all fragments for \( \{\pi_1\} \) are registered in \( \Theta_f \) and \( \Theta_t \) 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 \( \Theta_f \) and \( \Theta_t \).

Induction Step: We now show the property for \( i \); otherwise, a potential cyclic deadlock exists for \( \{\pi_1, \ldots, \pi_i\} \), and consequently the algorithm returns it. All the new fragments of \( \pi_i \) are categorized as follows: 1) a fragment with only \( \pi_i \) or 2) a fragment that extends other fragments on \( \Theta_f \) and \( \Theta_t \) using

---

Algorithm 1: Potential Cyclic Deadlock Detection.

**Input:** a set of paths \( \{\pi_1, \ldots, \pi_N\} \)

**Output:** one potential cyclic deadlock or NONE

1. \( \Theta_f, \Theta_t \leftarrow \emptyset \) \( \triangleright \) table for fragments, key: vertex
2. **procedure REGISTER(\( \theta \))**
   3. \( i \leftarrow \theta.agents[1]; j \leftarrow \theta.agents[-1] \)
   4. \( \pi_i[t_i] \leftarrow \theta.indexes[1]; \pi_j[t_j] \leftarrow \theta.indexes[-1] \)
   5. \( \Theta_f[\pi_i[t_i]].append(\theta) \)
   6. \( \Theta_t[\pi_j[t_j + 1]].append(\theta) \)
7. **function ISDEADLOCK(\( \theta \))** \( \triangleright \) return true or false
   8. \( i \leftarrow \theta.agents[1]; j \leftarrow \theta.agents[-1] \)
   9. \( \pi_i[t_i] \leftarrow \theta.indexes[1]; \pi_j[t_j] \leftarrow \theta.indexes[-1] \)
   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.     \( u \leftarrow \pi_i[t]; v \leftarrow \pi_i[t + 1] \)
14.     \( \theta \leftarrow \{\text{agents: } \theta.agents + i, \text{ indexes: } \theta.indexes + t\} \)
15.     REGISTER(\( \theta \))
16.     for \( \theta_t \in \Theta_t[u] \) do \( \triangleright \) case 1
17.       **if** \( i \in \theta_t.agents \) **then continue**
18.       \( \theta \leftarrow \{\text{agents: } \theta_t.agents + i, \text{ indexes: } \theta_t.indexes + t\} \)
19.       **if** ISDEADLOCK(\( \theta \)) **then return** \( \theta \)
20.     REGISTER(\( \theta \))
21.     for \( \theta_f \in \Theta_f[v] \) do \( \triangleright \) case 2
22.       **if** \( i \in \theta_f.agents \) **then continue**
23.       \( \theta \leftarrow \{\text{agents: } \theta_f.agents + i, \text{ indexes: } \theta_f.indexes + t\} \)
24.       **if** ISDEADLOCK(\( \theta \)) **then return** \( \theta \)
25.     REGISTER(\( \theta \))
26.     for \( \theta_f \in \Theta_f[u], \theta_t \in \Theta_t[v] \) do \( \triangleright \) case 3
27.       **if** \( i \in \theta_f.agents \cup \theta_t.agents \) **then continue**
28.       **if** \( \theta_f.agents \cap \theta_t.agents \neq \emptyset \) **then continue**
29.       \( \theta \leftarrow \{\text{agents: } \theta_f.agents + i + \theta_t.agents \text{ indexes: } \theta_f.indexes + t + \theta_t.indexes\} \)
30.       **if** ISDEADLOCK(\( \theta \)) **then return** \( \theta \)
31.     REGISTER(\( \theta \))
32. **return** NONE
Three cases of creating new fragments by extending existing fragments.

Fig. 7.

Fig. 8. Configuration of Table I.

**TABLE I**

**EXAMPLE OF DETECTING POTENTIAL CYCLIC DEADLOCKS**

| induction | key fragments | new fragments |
|-----------|---------------|---------------|
| {π₁}     | u             | (1, {u})      |
|           | v             | (1, {v})      |
| {π₁, π₂} | u             | (1, {u, v})   |
|           | v             | (2, {v})      |
|           | x             | (2, {x})      |
| {π₁, π₂, π₃} | u         | (1, {u, v, x}) |
|           | v             | (2, {v, x})   |
|           | x             | (3, {x})      |

Table I provides an example update of Θ_f using Algorithm 1.

The time complexity does not contradict the NP-completeness in detecting potential deadlocks (Lemma 8).

**Observation 20 (Space and Time Complexity):** Algorithm 1 requires \( \Omega(2^{2|A|}) \) both for space and time complexity in the worst case.

**Proof:** Consider the example in Fig. 9. In any solution, the number of fragments starting from \( u \) becomes \( \Omega(2^{2|A|}) \), implying the statement.

Although Algorithm 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 show two algorithms that use Algorithm 1 as a subprocedure to solve OTIMAPP.

### B. Prioritized Planning (PP)

Prioritized planning [17], [25] is neither complete nor optimal; however, it is computationally inexpensive. Hence, it is a popular approach to MAPF. It plans paths sequentially while avoiding collisions with the previously planned paths. Instead of interagent collisions, solvers for OTIMAPP must consider potential cyclic deadlocks.

Algorithm 2 is prioritized planning for OTIMAPP, referred to as PP. When planning a single-agent path, PP avoids using 1) the goals of other agents and 2) edges that cause 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 Algorithm 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 Theorem 17.

The 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. 10. One solution involves the repetition of the PP with random priorities until the problem is solved. Let call this PP⁺. However, finding good orders can be challenging.
Algorithm 3: DBS: Deadlock-based Search.

Input: an OTIMAPP instance
Output: a solution \{\pi_1, \ldots, \pi_N\} or FAILURE

1. \( R.\text{constraints} \leftarrow \emptyset \)
2. \( R.\text{paths} \leftarrow \) find paths with “no use of other goals”
3. \( \text{insert } R \text{ to OPEN} \)  \( \triangleright \) OPEN: priority queue
4. \textbf{while} OPEN \( \neq \emptyset \) \textbf{do}
5. \( N \leftarrow \text{OPEN}.\text{pop}() \)
6. \( C \leftarrow \text{get constraints of } N \text{ using Algorithm 1} \)
7. \textbf{if} \( C = \emptyset \) then \( \text{return } N.\text{paths} \)
8. \textbf{for} \( (i, u, v) \in C \) \textbf{do}
9. \( N' \leftarrow \{\text{constraints} : N.\text{constraints} + (i, u, v), \)
   \( \text{paths} : N.\text{paths}\} \)
10. \textbf{if} \( \pi_i \text{ in } N'.\text{paths} \) to follow \( N'.\text{constraints} \)
11. \textbf{if} \( \pi_i \) is found then \( \text{insert } N' \text{ to OPEN} \)
12. \textbf{return} FAILURE

because there are \(|A|!\) patterns. This motivates us to develop a search-based solver as described in the following section.

C. Deadlock-Based Search

We present DBS to solve OTIMAPP, based on a popular search-based MAPF solver called conflict-based search (CBS) [18]. 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 set 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 3 describes the high-level search of DBS. Each node in the high-level search contains \textit{constraints}, a list of tuples comprising one agent and two vertices (representing “from vertex” and “to vertex”), and \textit{paths} as a solution candidate. The root node has no constraints [Line 1]. Its paths are computed following “no use of other goals” in Theorem 17 [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 \textit{FAILURE} when OPEN becomes empty [Line 12]. We provide several complementary details below.

\textbf{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, we use the descending order of the number of deadlocks with two agents, which is computed within a reasonable time.

\textbf{Line 6:} Let \(((i, j, k, \ldots, l), (t_i, t_j, t_k, \ldots, t_l))\) be the deadlock returned by Algorithm 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.

\textbf{Line 10:} Forces one path \(\pi_i\) 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.

\textbf{Theorem 21 (DBS):} DBS returns a solution when solutions satisfying Theorem 17 exist; otherwise, it returns \textit{FAILURE}.

\textbf{Proof:} Assume that there is a solution \(\pi = \{\pi_1, \ldots, \pi_N\}\) that satisfies the relaxed sufficient condition (Theorem 17). At each cycle [Line 4–Line 11], at least one node in \textit{OPEN} is consistent with \(\pi\), i.e., its constraints allow searching \(\pi\). This is derived from the following induction: 1) the initial node \(R\) is consistent with \(\pi\), and 2) the nodes generated from a consistent node with \(\pi\) 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 \(\pi\) (or another solution); otherwise, no such solutions exist.

\textbf{Example:} We describe an example of DBS using Fig. 10. 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.

\textbf{Optimization:} Although this article focuses on a feasibility problem, DBS can be adapted to optimization problems. The total and maximum path lengths in a solution can be defined as the objective functions. 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.

D. PP Versus DBS

DBS has the theoretical guarantee of finding solutions (Theorem 21) while PP does not. Indeed, there are instances solvable for DBS but unsolvable for PP even with any planning order. Fig. 11(a) shows such an example. Observe first that this instance has a solution satisfying the sufficient condition (Theorem 17), as shown in Fig. 11(b). DBS eventually returns it.

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...
Using the notation and just presents a minimum example.

\( (\text{path-c}) \) of agents (at red or cyan), reflecting avoiding cyclic deadlocks to agents 1 and 2. Then, it assigns a path shown in Fig. 11(c) 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. 11(c). 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 Definition 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. 11(c) to agent 3 (7 steps; cyan), reflecting avoiding cyclic deadlocks with agents 1 and 2. Now, agent 4 has no path without cyclic deadlocks.

Fig. 11(c). Then, 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. 11(c) 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. 11(c).

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. 11(c) 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. 11(c).

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 versus 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.

VII. RELAXATION OF FEASIBILITY

OTIMAPP is unfortunately computationally intractable (Theorems 6 and 7). Moreover, detecting potential cyclic deadlocks itself, which is a core of solving OTIMAPP, is computationally intractable (Lemma 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 22 (m-tolerant solution): A set of paths is an m-tolerant solution when

1) No reachable terminal deadlocks.
2) No reachable cyclic deadlocks with m agents or fewer.
3) Restricts cyclic deadlocks with fewer than m agents.

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 Section VIII-C). It should be noted that when a set 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 Algorithm 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 solvers in Section VI.

Unfortunately, the complexity of finding m-tolerant solutions remains intractable.

Theorem 23 (Complexity of m-Tolerant Solutions): Finding 2-tolerant solutions is NP-hard.

Proof: The proof of NP-hardness on undirected graphs (Theorem 7) already restricts cyclic deadlocks with three agents (i.e., 3-tolerant solutions). We further replace the clause constrainer as Fig. 12 to restrict 2-tolerant solutions. This new gadget cannot be used simultaneously by the three agents. Otherwise, agent \(c_2\) has a reachable cyclic deadlock with either \(c_1\) or \(c_3\) (at red colored edges), or if either \(c_1\) or \(c_3\) meet terminal deadlocks with \(c_2\). The translation is still in polynomial time.

VIII. EVALUATION

This section empirically demonstrates that OTIMAPP solutions are computable to a certain extent (see Section VIII-A),
and they are useful in adverse environments regarding timings (see Section VIII-B) through simulation experiments. We also present how n-tolerant solutions relax computational effort while incurring the risk of execution failure (see Section VIII-C), as well as OTIMAPP execution with mobile robots (see Section VIII-D). 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.

A. Stress Test

1) Setup: Each solver was tested with a timeout of 5 min on four-connected undirected grids picked up from [1] 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” clause of Theorem 17. However, unsolvable instances may still be included.

2) Result: Fig. 13 and 14 present the results. Since DBS detects unsolvable instances regarding the relaxed sufficient condition of Theorem 17, we 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.

1) 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.

2) PP frequently fails because of priority orders (e.g., Fig. 10), 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.

3) The bottleneck of each solver is the procedure for detecting potential cyclic deadlocks, an NP-hard problem (Lemma 8). This also led to similar success rates for PP* and DBS.

4) The bottom of Fig. 13 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.

5) 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.

6) As seen in runtime results of Fig. 13, 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. 14, 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 Section VI-D.

3) Solvability of OTIMAPP Versus 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 suboptimal MAPF algorithm called PIBT\(^{+}\) [26] 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.

B. Delay Tolerance

Next, we show 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) [4]. MAPF-DP emulates the imperfect execution of MAPF by introducing the possibility $p_i$ of unsuccessful moves to agent $i$ (remaining there).

1) 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. We tested the following two baselines.

1) MCPs [4] force agents to preserve the order relations of visiting one vertex in an offline MAPF plan at the runtime. The plan was obtained using CBS [27], a bounded suboptimal version of the CBS algorithm. The suboptimality was set to 1.05 to obtain plausible solutions in a short time.

2) Causal-PIBT [19] is an online time-independent planning method, that is, each agent repeats one-step planning and action adaptively to the surrounding current situations.

2) Result: Table II shows that the execution of OTIMAPP solutions outperforms the alternatives when there are delays in agents’ motions. This is because: 1) Unlike MCPs, OTIMAPP solutions are free from the temporal dependencies of offline plans in which one-agent delays are possibly fatal. 2) 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.

3) Discussion: Although finding OTIMAPP solutions is challenging, Table II 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 Section VIII-A. In association with this discussion of solvability, we additionally display the success rate of solving MAPF-DP by each approach in Table III. 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.

C. m-Tolerant Solutions

Recall that a set of paths is $m$-tolerant when there are no reachable cyclic deadlocks with $m$ or fewer agents. We next empirically evaluate how computational effort is relaxed by introducing $m$-tolerant solutions, as well as the risk of execution failure.

1) Setup: We used both PP$^+$ and DBS in the same experimental setting as in Section VIII-A. Further, for each successful planning outcome, we also simulated execution with randomly generated 100 execution schedules and then counted the number of executions that triggered actual deadlocks. An execution is regarded as a failure with deadlocks because several agents never reach their destinations eternally. In this way, we calculated the execution failure rate.

2) Result: Fig. 15 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., $\leq 6$). On the other hand, with a smaller $m$ (e.g., 2), 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.

D. Robot Demonstrations

Finally, we present two OTIMAPP execution demonstrations with centralized and decentralized mobile robots. The video is available online. Fig. 16 shows snapshots. The OTIMAPP solution was prepared using DBS.

In both cases, although the robots move without any synchronization procedures, they are ensured to eventually reach their goals owing to the nature of OTIMAPP. Moreover, for the latter, any actor has no methods to know the entire configuration at

---

**TABLE II**

| $|A| = 35$ | $\bar{p} = 0.2$ | $\bar{p} = 0.5$ | $\bar{p} = 0.8$ |
|-----------|----------------|----------------|----------------|
| MCPs+ECBS | 1015 (1004,1056) | 1422 (1640,1458) | 2551 (3507,2564) |
| Causal-PIBT | 956 (976,955) | 1238 (1225,1293) | 1841 (3141,1664) |
| OTIMAPP | 941 (911,951) | 1178 (1149,1189) | 1730 (1807,7352) |

| $\bar{p} = 0.5$ | $|A| = 20$ | $|A| = 40$ | $|A| = 60$ |
|----------------|-----------|-----------|-----------|
| MCPs+ECBS | 724 (711,736) | 1698 (1479,1316) | 2938 (291,284) |
| Causal-PIBT | 662 (631,671) | 1466 (1433,1479) | 2425 (2059,2446) |
| OTIMAPP | 639 (311,646) | 1395 (1333,1408) | 2328 (2331,2345) |

| $\bar{p} = 0.0$ | $|A| = 20$ | $|A| = 40$ | $|A| = 60$ |
|----------------|-----------|-----------|-----------|
| MCPs+ECBS | 449 (408,493) | 934 (851,893) | 1438 (1348,146) |
| Causal-PIBT | 472 (467,478) | 1042 (1033,1050) | 1725 (1713,1738) |
| OTIMAPP | 458 (452,464) | 993 (945,1001) | 1628 (1617,1639) |

All settings used random-32-32-10. For each setting, we first picked up ten instances that OTIMAPP solutions were found by PP$^+$. For each instance and approach, we then performed 50 trials while changing the random seed. For all approaches, all execution trials succeeded. Thus, the results are means on 500 executions, accompanied with 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., $\bar{p} = 0$. 

**TABLE III**

| $\bar{p} = 0.5$ | $|A| = 20$ | $|A| = 40$ | $|A| = 60$ |
|----------------|-----------|-----------|-----------|
| MCPs+ECBS | 1.00 | 1.00 | 1.00 |
| Causal-PIBT | 1.00 | 1.00 | 1.00 |
| OTIMAPP (PP$^+$) | 1.00 | 0.80 | 0.44 |

For each $A$, we used 25 instances. They are the same as instances in Fig. 13. 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., 1250 executions).
Fig. 15. Results of \( m \)-tolerant solutions. The planning success rate (upper) is based on the same 25 identical instances as Fig. 13. The time limit was set to 5 min. The execution failure rate (lower) is based on 100 execution for each successful plan.

Fig. 16. OTIMAPP execution with ten robots in an \( 8 \times 8 \) grid. Colored arrows represent an OTIMAPP solution.

runtime, which cannot be addressed by conventional execution strategies. The implementation details are as follows.

1) Centralized Execution: We used toio robots (https://toio.io/). The robots evolve on a specific playmat and can be controlled by instructions of absolute coordinates. We created a virtual grid on the playmat and the robots followed the grid. Further, we informally confirmed that there is a nonnegligible action delay between robots when simultaneously sending instructions to several robots (e.g., ten robots, see the movie). Therefore, a one-shot execution—robots move alone without communication after the receipt of plans—will result in collisions and risk of execution failure. 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 asynchronously between robots while avoiding collisions.

2) Decentralized Execution: The AFADA platform [28] was used. It has an architecture comprising mobile robots that evolve over an active environment made of flat cells, each equipped with a computing unit. Adjacent cells communicate with each other via a serial interface. Further, cells form the environment in two ways: as a 2-D 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 acts; hence, the system is fully asynchronous, and no actor knows the entire configuration at runtime.

IX. RELATED WORK

A. Deadlock

A deadlock [29] 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 [30], [31]. 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 [30]. 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 [30]. OTIMAPP is based on prevention; 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 nondeadlock state from which reaching deadlocks is “inevitable” is referred to as unsafe [30]. 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 [32], that is, for each start-goal pair, a path exists that traverses no other starts and goals. The relaxed sufficient condition of OTIMAPP (Theorem 17) 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 [31] or a multirobot scheduling problem [33].

B. Path Planning for Multiple Robots

Path planning for multiple robots has been studied extensively. These approaches are typically categorized as reactive or deliberative approaches.

OTIMAPP essentially aims at removing the possibility of cyclic waiting, one of the four necessary conditions for deadlocks to happen [30].
In reactive approaches (e.g., [34], [35], [36], [37]), 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 shortightedness 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 is challenging. In contrast, OTIMAPP execution assumes only the mutual exclusion of locations. We consider this requirement 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 a multiagent path finding (MAPF) problem [1]. In a typical MAPF, the inputs are a graph and a set of start-goal pairs for agents. The objective is to find a set of “timed” paths because MAPF assumes that all agents act synchronously. Both optimal (e.g., [18], [38], [39]) and suboptimal (e.g., [25], [26], [40], [41], [42]) algorithms for MAPF have been extensively studied, 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 [7], [8], [9], [10], [43]. 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 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 multirobot 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 [4], [5], [6] or continuously synthesize deadlock-free scheduling [3], [33], [44], [45]. However, these approaches require runtime effort and additional infrastructure (e.g., steady networks and global monitoring systems) to continuously manage the status of all robots. In contrast, OTIMAPP 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 considered from Okumura et al. [19], which 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 [46] 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.

X. Conclusion

This article 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 article presented both theoretical and practical aspects, including the solution condition, computational complexities, solvers, and the relaxed solution concept.

Finally, we discuss interesting directions for the development and extension of the OTIMAPP.

1) 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. A recent study [47] introduced an online time-independent planning method for the unlabeled problem; however, offline planning remains missing.

2) Continuous spaces: We studied discretized environments but extending the work to continuous spaces has practical values. In this direction, definitions of potential/reachable deadlocks in continuous spaces should be elaborated like [48].

3) Enhancing each solver: This article presented two basic solvers based on MAPF studies, which is a very active research field. Using state-of-the-art MAPF techniques such as [26], [41], [42], powerful OTIMAPP algorithms are expected to be developed.

4) Applications to other multiagent planning domains: We believe that OTIMAPP can be leveraged for other resource allocation problems with mutual exclusion, e.g., distributed databases [15].

Appendix A

Proofs of Computational Complexity

Lemma (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. We show 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.

We now explain how to translate the 3-SAT formula to the OTIMAPP instance and the corresponding set of paths. Without loss of generality, we assume that all variables appear positively and negatively in the formula. Throughout the proof, we use the following example:

\[(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \land \neg x_3).\]
A. Construction of an OTIMAPP Instance and a Set of Paths

A literal agent is introduced for each literal in each clause. Here, \( c_k^j \) denotes a literal agent for the \( k \)th literal in the \( j \)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 Theorem 6; however, their intuitions are similar and we use the same names.

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 a true to \( x_i \), and a lower path, corresponding to a 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: \( c_1^2(x_2), c_2^2(x_2), \) and \( c_3^2(\neg x_2) \). In Fig. 17, the upper and lower paths are highlighted by bold lines. \( c_1^2 \) and \( c_2^2 \) are connected to the upper path while \( c_3^2 \) 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. 17, three edges are annotated with the agent’s name. \( c_2^2 \) is supposed to use the colored middle one. Note that we use multiple edges for simplicity. The gadget can be easily converted into a simple graph, as shown immediately after this proof.

As a result, all literal agents take six edges to reach their goals. This is visualized by colored edges in Figs. 17 and 18. The special agent \( z \) uses two edges to reach its goal, through \( \blacklozenge \) marks in the figure. We have finished the description of 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.

B. 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 \( z \).
The formula is \((x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \land \neg x_3)\). The assignment is \(x_1 = \text{true},\ x_2 = \text{true},\ \text{and } x_3 = \text{true}\). 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.

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, i.e., this deadlock is reachable and thus potential.

For example, consider a satisfiable assignment \(x_1 = \text{true},\ x_2 = \text{true},\ x_3 = \text{true}\). Initially, move assigned agents, \(c_1^1, c_2^1, c_2^2, c_3^1, c_3^1\) to vacation vertices in each variable decider (Fig. 19; Step 1). Next, move \(c_2^2, c_3^2,\) and \(c_1^3\) to the first vertices of each clause constrainer of \(C^1, C^2,\) and \(C^3\), respectively (Fig. 19; Step 2). Subsequently, move all unassigned agents, \(c_1^2, c_2^3, c_3^3\).
and $c_3^1$, one step (Fig. 19; Step 3). Consequently, there is a cyclic deadlock with $c_1^2, c_2^1, c_3^1, c_2^2, c_1^3, c_1^2$, and $z$, as annotated with bold lines in Fig. 19.

C. 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 $c_3^2$ and $c_2^2$ are involved in the deadlock at the variable decider (see Fig. 17), then assign $false$ to $z$. 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, we have established the claim that a potential cyclic deadlock exists if and only if the formula is satisfiable. Next, we claim 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 8, we used multiple edges in a gadget clause constrainer for the reduction from 3-SAT. Since OTIMAPP assumes a simple graph (i.e., no multiple edges), we complement how to convert it into a correct OTIMAPP instance. Fig. 20 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, we add intermediate vertices for each edge, which can potentially result in cyclic deadlocks. For each agent $c_k^j$, a new agent $c_k^j$ is introduced. The start point is an intermediate vertex, whereas the goal point is the original goal of $c_k^j$. We furthermore change a goal for $c_k^j$ to the start of $c_k^j$. Consider now replacing all old clause constrainers with new gadgets. The translation is performed in polynomial time. The remainder of the proof is straightforward, as from Lemma 8.

Acknowledgment

The authors would like to thank the anonymous reviewers for their many insightful comments, including our prior submissions. Keisuke Okumura would like to thank the support of the Yoshida Scholarship Foundation.

References

[1] R. Stern et al., “Multi-agent pathfinding: Definitions, variants, and benchmarks,” in Proc. Annu. Symp. Combinatorial Search, 2019, pp. 151–158.
[2] M. Ben-Ari and F. Mondada, Elements of Robotics. Berlin, Germany: Springer, 2017.
[3] M. Cáp, J. Gregoire, and E. Frazzoli, “Provably safe and deadlock-free execution of multi-robot plans under delaying disturbances,” in Proc. IEEE/RSJ Int. Conf. IntelI. Robots Syst., 2016, pp. 5113–5118.
[4] H. Ma, T. S. Kumar, and S. Koenig, “Multi-agent path finding with delay probabilities,” in Proc. AAAI Conf. Artif. Intell., 2017, pp. 3605–3612.
[5] W. Höning, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, “Persistent and robust execution of MAPF schedules in warehouses,” IEEE Robot. Autom. Lett., vol. 4, no. 2, pp. 1125–1131, Apr. 2019.
[6] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Bartak, and N.-F. Zhou, “Robust multi-agent path finding and executing,” J. Intell. Artif. Res., vol. 67, pp. 549–579, 2020.
[7] G. Wagner and H. Choset, “Path planning for multi agents under uncertainty,” in Proc. Int. Conf. Autom. Plan. Scheduling, vol. 27, pp. 577–585, 2017.
[8] M. Mansouri, B. Lacerda, N. Hawes, and F. Pecora, “Multi–robot planning under uncertain travel times and safety constraints,” in Proc. Int. Joint Conf. Artif. Intell., 2019, pp. 478–484.
[9] O. Pelzter, K. Brown, M. Schwager, M. J. Kochenderfer, and M. Sehr, “STT-CBS: A conflict-based search algorithm for multi-agent path finding with stochastic travel times,” 2020, arXiv:2004.08025.
[10] D. Atzmon, R. Stern, A. Felner, N. R. Sturtevant, and S. Koenig, “Probabilistic robust multi-agent path finding,” in Proc. Int. Conf. Autom. Plan. Scheduling, 2020, pp. 29–37.
[11] P. R. Wurman, R. D’Andrea, and M. Mountz, “Coordinating hundreds of cooperative, autonomous vehicles in warehouses,” AI Mag., vol. 29, no. 1, pp. 9–19, 2008.
[12] K. Dresner and P. Stone, “A multiagent approach to autonomous intersection management,” J. Artif. Intell. Res., vol. 31, pp. 591–656, 2008.
[13] X. Zhang et al., “Large-scale 3D printing by a team of mobile robots,” Autom. Construction, vol. 95, pp. 98–106, 2018.
[14] G. Tel, “Deadlock-free packet switching,” in Introduction Distributed Algorithms. Cambridge, U.K.: Cambridge Univ. Press, 2000, ch. 5.
[15] E. Knapp, “Deadlock detection in distributed databases,” ACM Comput. Surv., vol. 19, no. 4, pp. 303–328, 1987.
[16] D. M. Kornhauser, G. Miller, and P. Spirakis, “Coordinating pebble motion on graphs, the diameter of permutation groups, and applications,” Master’s thesis, Cambridge, MA, USA: MIT, 1984.
[17] M. Erdm and T. Lozano-Perez, “On multiple moving objects,” Algorithmica, vol. 2, pp. 477–521, 1987.
[18] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, “Conflict-based search for optimal multi-agent pathfinding,” Artif. Intell., vol. 219, pp. 40–66, 2015.
[19] K. Okumura, Y. Tamura, and X. Défago, “Time-independent planning for multi-robot moving agents,” in Proc. AAAI Conf. Artif. Intell., 2021, pp. 11299–11307.
[20] K. Okumura, F. Bonnet, Y. Tamura, and X. Défago, “Offline time-independent multi-agent path planning,” in Proc. Int. Joint Conf. Artif. Intell., 2022, pp. 4649–4656.
[21] K. Altisen, S. Devismes, S. Dubois, and F. Petit, “Introduction to distributed self-stabilizing algorithms,” Synth. Lectures Distrib. Comput. Theory, vol. 8, no. 1, pp. 1–165, 2019.
[22] B. Nebel, “On the computational complexity of multi-agent pathfinding on directed graphs,” in Proc. Int. Conf. Autom. Plan. Scheduling, 2020, pp. 212–216.
[23] D. Ratner and M. K. Warmuth, “Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable,” in Proc. AAAI Conf. Artif. Intell., 1986, pp. 168–172.
[24] J. Yu and S. M. LaValle, “Structure and intractability of optimal multi-robot path planning on graphs,” in Proc. AAAI Conf. Artif. Intell., 2013, pp. 1443–1449.
[25] D. Silver, “Cooperative pathfinding.” Proc. AAAI Conf. Artif. Intell. Interac. Digit. Entertainment, 2005, pp. 117–122.
[26] K. Okumura, M. Machida, X. Défago, and Y. Tamura, “Priority inheritance with backtracking for iterative multi-agent path finding.” Artif. Intell., vol. 310, 2022. Art. no. 103752.
[27] M. Barer, G. Sharon, R. Stern, and A. Felner, “Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem,” in Proc. Ann. Symp. Combinatorial Search, 2014, pp. 19–27.
[28] S. Kameyama, K. Okumura, Y. Tamura, and X. Défago, “Active modular environment for robot navigation,” in Proc. IEEE Int. Conf. Robot. Autom., 2021, pp. 8836–8842.
[29] E. G. Coffman, M. Elphick, and A. Shoshani, “System deadlocks,” ACM Comput. Surv., vol. 3, pp. 67–78, 1971.
[30] A. Silberschatz, P. B. Galvin, and G. Gagne, “Deadlocks,” in Operating System Concepts. Hoboken, NJ, USA: Wiley, 2006, ch. 7.
[31] M. P. Fanti and M. Zhou, “Deadlock control methods in automated manufacturing systems,” IEEE Trans. Syst., Man, Cybern.-Part A, Syst. Humans, vol. 34, no. 1, pp. 5–22, Jan. 2004.
[32] M. Čáp, P. Novák, A. Kleiner, and M. Selecký, “Prioritized planning algorithms for trajectory coordination of multiple mobile robots.” IEEE Trans. Autom. Sci. Eng., vol. 12, no. 3, pp. 835–849, Jul. 2015.
[33] A. Mannucci, L. Pallottino, and F. Fecora, “On provably safe and live multirobot coordination with online goal posting.” IEEE Trans. Robot., vol. 37, no. 6, pp. 1973–1991, Dec. 2021.
[34] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha, “Reciprocal n-body collision avoidance,” in Proc. Robot. Res., 14th Int. Symp. ISR, 2011, pp. 3–19.
[35] E. Lalish and K. A. Morgansen, “Distributed reactive collision avoidance,” Auton. Robots, vol. 32, pp. 207–226, 2012.
[36] B. Senbaslar, W. Höning, and N. Ayanian, “Robust trajectory execution for multi-robot teams using distributed real-time replanning,” in Proc. Int. Symp. Distrib. Robot. Syst., 2018, pp. 167–181.
[37] J. Park, D. Kim, G. C. Kim, D. Oh, and H. J. Kim, “Online distributed trajectory planning for quadrotor swarm with feasibility guarantee using linear safe corridor,” IEEE Robot. Autom. Lett., vol. 7, no. 2, pp. 4869–4876, Apr. 2022.
[38] G. Wagner and H. Choset, “Subdimensional expansion for multirobot path planning.” Artif. Intell., vol. 219, pp. 1–24, 2015.
[39] E. Lam, P. Le Bodic, D. D. Harabor, and P. J. Stuckey, “Branch-and-cut-and-price for multi-agent pathfinding,” Comput. Oper. Res., 2022, Art. no. 105809.
[40] B. De Wilde, A. W. Ter Mors, and C. Witteveen, “Push and rotate: Cooperative multi-agent path planning,” J. Artif. Intell. Res., 2014.
[41] J. Li, W. Ruml, and S. Koenig, “Eecbs: A bounded-suboptimal search algorithm for multi-agent path finding,” in Proc. AAAI Conf. Artif. Intell., 2021, pp. 12353–12362.
[42] K. Okumura, “LACAM: Search-based algorithm for quick multi-agent pathfinding,” in Proc. AAAI Conf. Artif. Intell., 2023.
[43] T. Shahar, S. Shekhar, D. Atzmon, A. Saffidine, B. Juba, and R. Stern, “Safe multi-agent pathfinding with time uncertainty,” J. Artif. Intell. Res., vol. 70, pp. 923–954, 2021.
[44] P. A. O’Donnell and T. Lozano-Pérez, “Deadlock-free and collision-free coordination of two robot manipulators,” in Proc. IEEE Int. Conf. Robot. Autom., 1989, pp. 484–489.
[45] A. Coskun and J. M. O’Kane, “Online plan repair in multi-robot coordination with disturbances,” in Proc. IEEE Int. Conf. Robot. Autom., 2019, pp. 3333–3339.
[46] N. Robertson and P. D. Seymour, “Disjoint paths–A survey,” SIAM J. Algbr. Discrete Methods, vol. 6, no. 2, pp. 300–305, 1985.
[47] K. Okumura and X. Défago, “Solving simultaneous target assignment and path planning efficiently with time-independent execution,” in Proc. Int. Conf. Autom. Plan. Scheduling, vol. 32, pp. 270–278, 2022.
[48] J. Gregoire, S. Bonnabel, and A. de La Fortelle, “Optimal cooperative motion planning for vehicles at intersections,” 2013, arXiv:1310.7729.

Keisuke Okumura (Graduate Student Member, IEEE) received the B.Eng. and M.Eng. degrees in computer science from Tokyo Institute of Technology, Tokyo, Japan, in 2018 and 2020, respectively. He is currently working toward the Ph.D. degree in artificial intelligence and robotics with the Tokyo Institute of Technology. He also spent half a year as a Visiting Scholar with Sorbonne University, France, in 2022. His research interests include controlling multiple moving agents, and he has been actively publishing papers on this topic in prestigious journals and conferences such as AIJ, IJCAI, and AAAI. Mr. Okumura received the ICAPS 2022 Best Student Paper Award. For more details visit https://kei18.github.io/.

François Bonnet received the M.S. degree in computer science from the ENS Cachan at Rennes, Rennes, France, in 2006 and the Ph.D. degree in computer science from the University of Rennes 1, Rennes, France, in 2010. He is a Specially Appointed Associated Professor with Tokyo Tech since 2018. He worked with JAIST as a JSPS postdoctoral fellow until 2012 and then as an Assistant Professor until 2017. He spent one year as a Specially Appointed Assistant Professor with Osaka University. His research interests include theoretical distributed computing, discrete algorithms, and combinatorial game theory.

Yasumasa Tamura (Member, IEEE) received the master’s and Ph.D. degrees in computer science from Hokkaido University, Sapporo, Japan, in 2014 and 2015, respectively. He is an Assistant Professor with the School of Computing, Tokyo Institute of Technology, Japan. He worked with Hokkaido University as a JSPS Research Fellow, and also worked with Université Libre de Bruxelles, Belgium, as a visiting Researcher until 2017. His research interests include computational intelligence, swarm robotics, and soft computing.

Xavier Défago received the master’s and Ph.D. degrees in computer science from the Swiss Federal Institute of Technology, Lausanne, Switzerland, in 1995 and 2000, respectively. He is a Full Professor with Tokyo Institute of Technology since 2016. Before his current position at Tokyo Tech, he was a faculty member with the Japan Advanced Institute of Science and Technology. Meanwhile, he has also been a PRESTO Researcher for the Japan Science and Technology Agency, and an invited Researcher for CNRS (France) at Sorbonne University and at INRIA Sophia Antipolis. He is a member of the IFIP working group 10.4 on dependable computing and fault-tolerance. He served as a program chair of several international conferences on distributed systems and dependability (SRDS, ICCDS, DSN, SSS). His research interests include various aspects of dependable and secure distributed systems, robotics, cyber-physical systems, embedded systems, and programming languages.