Multi-Agent Path Execution with Uncertainty

Yihao Liu[1], **Xueyan Tang**[1], **Wentong Cai**[1], **Jingning Li**[2]

[1] School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2] NCS Pte Ltd, Singapore
yihao002@e.ntu.edu.sg, asxytang@ntu.edu.sg, aswtcai@ntu.edu.sg, jingning.li@ncs.com.sg

Abstract

In real-world multi-agent applications, unexpected conditions can break the assumptions made in path planning and degrade the effectiveness of path execution. This paper studies robust and effective execution of multi-agent path plans under uncertainty. To guarantee conflict-freeness and deadlock-freeness, we define a feasibility problem to check whether the remaining portion of a path plan can be successfully executed. We prove that the problem is NP-complete and propose a feasibility test algorithm. We further develop algorithms to coordinate the agents online and have as many of them as possible moving concurrently to maximize the effectiveness of execution. We experimentally demonstrate the path execution effectiveness and computational efficiency of our algorithms.

Introduction

In many multi-agent applications such as automated warehouse management and security patrolling, agents carry out tasks by moving from one location to another concurrently. To avoid collisions, we need to carefully plan the paths for the agents and coordinate the execution of the plan. While path planning (often known as Multi-Agent Path Finding) has been extensively studied (Felner et al. 2017), designing execution policies for planned paths has received less attention. Robust and effective execution is not only critical but also challenging because plan execution is often accompanied by uncertainties due to unforeseen circumstances.

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

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

Related Work

Multi-Agent Path Finding (MAPF) is a classical problem to find a conflict-free plan for a group of agents to move from their start locations to goal locations with cost objectives to be optimized (Stern et al. 2019). Some recent MAPF works have attempted to consider uncertainty in plan execution.

The k-robust MAPF problem (Atzmon et al. 2020b) allows each agent to be delayed for up to k timesteps. The MAPF with Time Uncertainty problem (Shahar et al. 2021) assumes upper and lower bounds on the timesteps taken to move between adjacent locations. The limitations of these works are that the plan may become invalid when the assumed delay bounds are violated and may be far from optimal in the cost objective if the bounds are loose. The MAPF with Uncertainty problem (Wagner and Choset 2017) and the p-robust MAPF problem (Atzmon et al. 2020a) pursue plans that can be executed without conflicts with probabilistic guarantees. They rely on the knowledge of the probability that an agent will be delayed when moving between adjacent locations, which is often hard to obtain in practice. The MAPF with Delay Probabilities problem (Ma, Kumar, and Koenig 2017) assumes the same knowledge and aims to find a plan to minimize the expected makespan. In this work, a plan execution policy called MCP is designed to avoid conflicts by using a directed graph to capture precedence dependencies among agents, which originates from the Temporal Plan Graph (TPG) proposed by (Hönig et al. 2016). These graph concepts are extended to the Action Dependency Graph (ADG) by (Hönig et al. 2019). Sticking to a static ADG/TPG, however, can lead to ineffective execution when facing unexpected delays. Some recent studies (Berndt et al. 2020; Coskun, O'Kane, and Valtorta 2021; Paul, Feng, and Li 2023) optimize the ADG/TPG by switching precedence dependencies during plan execution via mixed-integer programming or A* search etc. These approaches generally have high computational overheads because they strive for

a new plan optimizing the cost assuming no further delay, which may not be necessary if further delays may occur in the future. Our study is motivated by the above works. We develop efficient heuristics to keep plan execution effective online without assuming any prior knowledge on the delays that may arise in the execution. Recently, the Bidirectional Temporal Plan Graph (BTPG) was introduced by (Su, Veerapaneni, and Li 2024) to facilitate switching precedence dependencies during plan execution. We shall contrast and compare our solution with BTPG in the experiment section.

We also note that our multi-agent path execution problem has similarity to the job-shop scheduling problem (Mascis and Pacciarelli 2002) and the flow control problem in store-and-forward networks (Arbib, Italiano, and Panconesi 1990). We shall discuss the differences in relevant sections.

Problem Statement

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

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

We consider two classical types of conflicts: node conflicts where two agents stay at the same node at the same timestep; edge conflicts where two agents move along the same edge in opposite directions at the same timestep. In addition, we also consider following conflicts where an agent goes towards a node occupied by another agent in the previous timestep (Stern et al. 2019), because owing to uncertainty, if the latter agent fails to move and the former agent succeeds in moving, they may collide at the node. When an agent executes a *wait* action, it occupies the current node where it stays. When an agent executes a *move* action, we consider it occupying both the current node and the adjacent node it is moving to, until the agent arrives at the adjacent node. In path execution, we impose the constraint that any two agents cannot occupy the same node concurrently. Such a constraint addresses all the aforesaid node, edge and following conflicts. We assume no violation of the constraint in the initial (resp. final) state, i.e., the start (resp. goal) locations of all agents are distinct. But it is possible for a start location s_i to be the same as a goal location g_j.

Given a path plan $\mathcal{P}$, we would like to coordinate the *move* and *wait* actions taken by agents during the execution to ensure that all agents eventually arrive at their goal locations. We use the number of agents taking *move* actions as a measure of path execution effectiveness and aim to maximize this number at any time, which intuitively would help optimize cost objectives such as makespan or sum-of-costs.

Path Plan Feasibility Problem

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

Feasibility and Location Dependency Graph

If there exist common nodes among different paths, to execute the path plan, we need to decide the order for the agents to occupy common nodes, which can be modeled by a location dependency graph.

We refer to each node $v_{i,j}$ on an agent a_i's path p_i as a *location state*.[1] The location dependency graph (LDG) is a directed graph $\mathcal{G}_{\text{LDG}} = (\mathcal{V}_{\text{LDG}}, \mathcal{E}_{\text{LDG}})$, where $\mathcal{V}_{\text{LDG}} = \{v_{i,j} \mid 1 \leqslant i \leqslant M, 1 \leqslant j \leqslant n_i\}$ is the set of all possible location states of the agents, and $\mathcal{E}_{\text{LDG}}$ specifies the precedence constraints for the agents to achieve their location states. $\mathcal{E}_{\text{LDG}}$ consists of four parts. The first part of $\mathcal{E}_{\text{LDG}}$ is based on the path of each individual agent and includes the edges $\{\langle v_{i,j}, v_{i,j+1}\rangle \mid 1 \leqslant i \leqslant M, 1 \leqslant j < n_i\}$, meaning that agent a_i cannot achieve the location state $v_{i,j+1}$ before $v_{i,j}$.

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

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

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

The last part of $\mathcal{E}_{\text{LDG}}$ is to resolve the order of two non-initial and non-final location states from different agents that share the same node in the graph $\mathcal{G}$. Recall that any two agents cannot occupy the same node simultaneously. Thus,

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

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

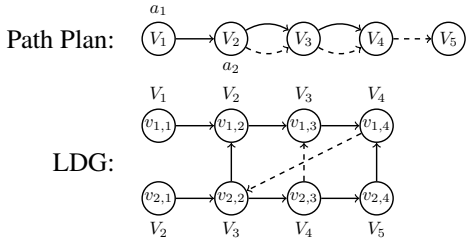

Figure 1: A feasible path plan and its LDG.

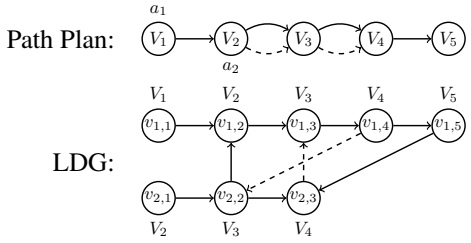

Figure 2: An infeasible path plan and its LDG.

for each pair of location states $v_{i,j}$ and $v_{i',j'}$ from two respective agents a_i and $a_{i'}$ sharing a common node, an edge $\langle v_{i,j+1}, v_{i',j'}\rangle$ can be formed if a_i achieves $v_{i,j}$ before $a_{i'}$ achieves $v_{i',j'}$, or an edge $\langle v_{i',j'+1}, v_{i,j}\rangle$ can be formed if $a_{i'}$ achieves $v_{i',j'}$ before a_i achieves $v_{i,j}$. We refer to these two edges as a pair of *unsettled edges*. To ensure conflict-free path execution, one of them must be chosen and added to the LDG. Note that by definition, different pairs of location states must have distinct unsettled edges. We refer to the LDG with predetermined edges only as a *partial* LDG, and refer to the LDG after choosing one edge from each pair of unsettled edges as a *complete* LDG. The following property is similar to what was stated in (Berndt et al. 2020; Coskun, O'Kane, and Valtorta 2021).

Theorem 1. *A path plan is feasible if and only if there exists an acyclic complete LDG.*

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

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

We remark that our LDG is conceptually similar to the alternative graph introduced for job-shop scheduling with blocking constraints (Mascis and Pacciarelli 2002). In job-shop scheduling, different jobs may have operations that must be processed on the same machine, but each machine can process only one operation at a time. In the alternative graph, fixed arcs (similar to our predetermined edges) are created to model the specified order of operations in each job, and alternative arcs (similar to our unsettled edges) are created to model possible precedence relations between operations of different jobs sharing the same machine. However, a job initially does not occupy any machine and after a job finishes, it releases the machine of the last operation. Thus, the alternative graph does not include arcs corresponding to the second and third parts of $\mathcal{E}_{\text{LDG}}$ in our LDG.

Hardness of Path Plan Feasibility

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

Theorem 2. *The path plan feasibility problem is NP-complete.*

Obviously, the path plan feasibility problem is in NP. We reduce the *LSAT* (linear *SAT*) problem to the path plan feasibility problem to prove Theorem 2. The classical *SAT* (Boolean satisfiability) problem is to determine whether a formula can be made *true* by assigning appropriate logical values (*true*, *false*) to its variables. If there are at most three variables in each clause, it is called a *3-SAT* problem, which is known to be NP-complete (Karp 1972). The *LSAT* problem is a subclass of the *3-SAT* problem, where each clause intersects at most one other clause. Moreover, if two clauses intersect, they have exactly one literal in common. Recently, it has been shown that the *LSAT* problem is also NP-complete (Arkin et al. 2018).

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

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

For each pair of variables x_i and x_j in the *LSAT* problem, we construct eight possible *auxiliary* agents a_{ij}, $a_{i\bar{j}}$, $a_{\bar{i}j}$, $a_{\bar{i}\bar{j}}$, a_{ji}, $a_{j\bar{i}}$, $a_{\bar{j}i}$, $a_{\bar{j}\bar{i}}$, and each agent is assigned a path of four nodes. The location states of these agents are shown in the upper and lower parts of Figure 3. The start and goal locations of the agents are all different. The pair of the second and third nodes in each path is one of the four combinations of a start location from $a_i/a_{\bar{i}}$ and a goal location from $a_j/a_{\bar{j}}$ or the four combinations of a start location from $a_j/a_{\bar{j}}$ and a goal location from $a_i/a_{\bar{i}}$. Since the paths of auxiliary agents include only the start and goal locations of primary agents,

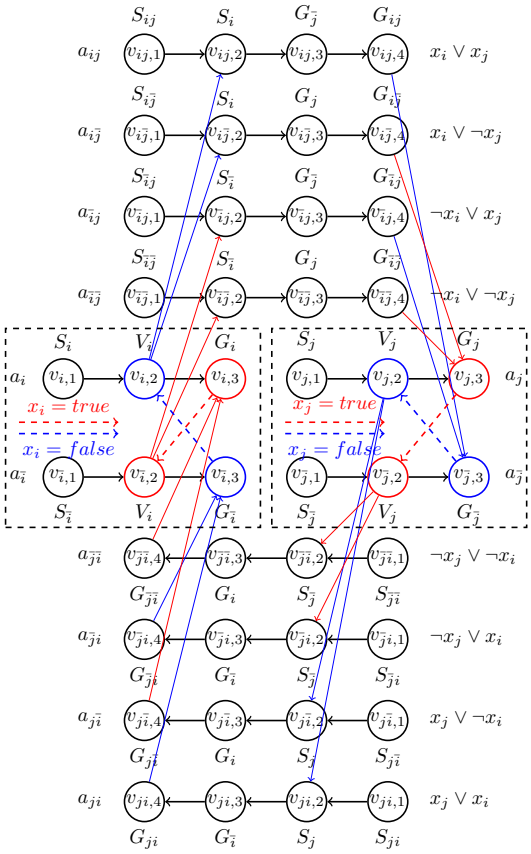

Figure 3: LDG of variables x_i and x_j.

only predetermined edges (solid lines) will be added to the LDG as shown in Figure 3.

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

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

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

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

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

Lemma 1. *If a variable assignment makes the LSAT formula evaluate to false, choosing the unsettled edges according to the assignments would give rise to a cycle in the complete LDG.*

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

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

Lemma 2. *If the choices of the unsettled edges lead to a cycle in the complete LDG, assigning the values to the variables according to the edge choices would make the LSAT formula evaluate to false.*

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

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

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

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

If the *LSAT* clause involved is a 2-literal clause (such as $x_i \vee x_j$), the cycle involves two auxiliary agents (such as a_{ij}, a_{ji}) and the primary agents for the two variables in the 2-literal clause (such as a_i, $a_{\bar{i}}$, a_j, $a_{\bar{j}}$). By the variable assignment corresponding to the unsettled edges chosen in the complete LDG, the 2-literal clause must evaluate to *false*.

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

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

Note that there is a one-to-one correspondence between the variable assignments and the complete LDGs. In addition, an *LSAT* formula is satisfiable if and only if there exists a satisfying assignment. By Theorem 1, a path plan is feasible if and only if there exists an acyclic complete LDG. Therefore, by Lemmas 1 and 2, the *LSAT* problem and the path plan feasibility problem constructed are equivalent.

We note that the NP-completeness result of our path plan feasibility problem is similar to that of the flow control problem in store-and-forward packet switching networks (Arbib, Italiano, and Panconesi 1990). However, there is an important difference. In the store-and-forward network, packets are removed from the network after reaching their destinations. This allows different packets to share the same destination. Indeed, the problem instances used for reductions in the NP-completeness proofs have packets sharing the same destination (Arbib, Italiano, and Panconesi 1990). In contrast, we assume that the goal locations of all agents are distinct because agents will stay at their goal locations after reaching them. Hence, the proofs of (Arbib, Italiano, and Panconesi 1990) are not directly applicable to our problem.

Feasibility Test

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

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

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

Algorithm 1: Feasibility test of path plan $\mathcal{P}$

```
1  E_0 ← unsettled edge pairs in P;
2  LDG_0 ← the initial partial LDG of P;
3  return FeasibilityTest(∅);
4
5  Function FeasibilityTest(E_settled):
6      LDG ← add settled edges in E_settled to LDG_0;
7      if |E_settled| = |E_0| then
8          return feasible (with complete LDG);
9      E_unsettled ← remove unsettled edge pairs that contain
         settled edges in E_settled from E_0;
10     if this is not the first FeasibilityTest call then
11         Select a pair of edges ⟨e, e'⟩ from E_unsettled;
12         if neither of e and e' forms a cycle in LDG then
13             Add e to E_settled;
14             if FeasibilityTest(E_settled) is feasible
                 then
15                 return feasible (with complete LDG);
16             Remove e from E_settled and add e' to E_settled;
17             return FeasibilityTest(E_settled);
18     foreach pair of unsettled edges ⟨e, e'⟩ in E_unsettled do
19         if e and e' both form a cycle in LDG then
20             return infeasible (together with the two
                 agents that e and e' involve);
21         else if one of e and e' forms a cycle in LDG then
22             ē ← the edge not forming a cycle in LDG;
23             Add ē to LDG and ē to E_settled;
24     return FeasibilityTest(E_settled);
```

when both calls return negative results, the function concludes that no acyclic complete LDG exists. If the unsettled edge pair $\langle e, e' \rangle$ selected falls in case (1) or (2), we scan all unsettled edge pairs again to find cases (1) and (2) (lines 18-23), and then make another recursive call (line 24).

Theorem 3. *Algorithm 1 is sound and complete.*

Proof. Each recursive function call by lines 14 and 17 increases the number of settled edges $|\mathcal{E}_{\text{settled}}|$ by at least one. This is also true for each recursive function call by line 24 (except that in the first function call), because in order for lines 18-23 to be executed, the unsettled edge pair selected by line 11 must fall in case (1) or (2). The recursion terminates when $|\mathcal{E}_{\text{settled}}|$ reaches $|\mathcal{E}_0|$ (lines 7-8). Thus, the function always returns a result, so Algorithm 1 is complete.

We prove that Algorithm 1 is sound by backward induction on the function call. Note that the initial LDG generated by the input $\mathcal{E}_{\text{settled}}$ (line 6) is guaranteed to be acyclic in any function call.

1) Base case: when $|\mathcal{E}_{\text{settled}}| = |\mathcal{E}_0|$ (line 6), it means that there is no unsettled edge pair left. A positive result (feasible) is returned because the LDG is complete and acyclic.

2) Induction: we show that the function result is correct for an input $\mathcal{E}_{\text{settled}}$, given the hypothesis that the result is correct for any input $\mathcal{E}'_{\text{settled}}$ such that $|\mathcal{E}_{\text{settled}}| < |\mathcal{E}'_{\text{settled}}|$. We examine all return clauses in the function except line 8, which is covered in the base case. In line 20, selecting either edge from an unsettled edge pair forms a cycle in the LDG, so a negative result is returned. In line 24, all newly added settled edges by lines 21-23 have to be selected to avoid cycles, so the problems before and after adding these settled edges are equivalent. Since the recursive call is correct by the induction hypothesis, the returned result in line 24 is correct. In lines 13-17, the function joins the results of two recursive calls (correct by the induction hypothesis) with different selections of settled edges from the same unsettled edge pair. If both results are negative, a negative result is returned because selecting either edge cannot lead to an acyclic complete LDG. Otherwise, at least one acyclic complete LDG is returned so that the path plan is feasible. □

In our implementation, to check whether adding an unsettled edge $\langle v_{i,j}, v_{i',j'} \rangle$ will form a cycle in the LDG, we simply run a breadth-first search to see if there is a path from $v_{i',j'}$ to $v_{i,j}$ before the edge is added. The feasibility test has an exponential time complexity in the worst case since it may need to check complete LDGs exhaustively. In practice, however, the feasibility test normally runs fast, as shall be demonstrated in our experiments. This is because the algorithm fixes the choices for the edge pairs of cases (1) and (2) before examining those of case (3). Moreover, the test terminates once an acyclic complete LDG is found.

Coordinating Path Execution

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

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

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

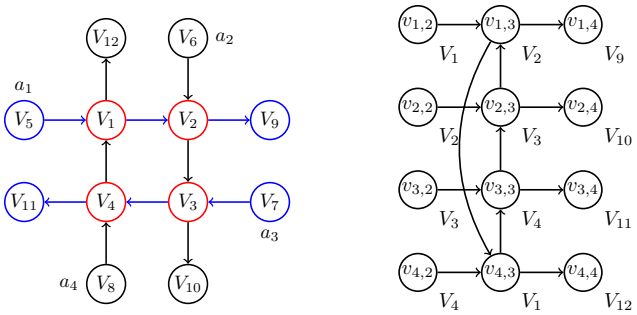

Figure 4: A path plan (left) and the LDG of the residual path plan if all four agents are unblocked initially (right).

guarantees that the residual path plan is feasible if any subset of the agents in $U \cup \Delta U$ arrive at their next nodes, because we can always wait until the remaining agents in $U \cup \Delta U$ arrive at their next nodes before unblocking any new agent.

For example, in Figure 4 (left), if all four agents are unblocked initially and arrive at the next nodes on their paths, none of them can move any further since a deadlock is formed. As shown in Figure 4 (right), there is a cycle in the (partial) LDG of the residual path plan, so it is infeasible. Thus, at most three agents can be unblocked initially.

To maximize the path execution effectiveness, the objective of our algorithm is to maximize $|\Delta U|$ (the number of agents to unblock) while meeting the above requirements. We develop a heuristic algorithm that first identifies the agents which can definitely be unblocked or must be kept blocked and then focuses on deciding the remaining agents in B. Algorithm 2 shows the details.

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

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

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

Finally, we set $\Delta U = B \setminus B_K$ and examine whether all these remaining agents can be unblocked. We set the initial location states of all agents in $U \cup \Delta U$ to the next nodes on their respective paths and run the feasibility test. In our implementation, to improve efficiency, the feasibility test uses the previously found acyclic complete LDG (in the last run of Algorithm 2) as hints for choosing which edge to test first from an unsettled edge pair (line 13 of Algorithm 1). If the residual path plan is feasible, we can safely unblock the agents ΔU (lines 13-15). Otherwise, the feasibility test must return two agents involved in the cycles found in the

Algorithm 2: Heuristic approach to unblock agents

```
1  B_K ← ∅;
2  foreach agent a_i ∈ B do
3      v ← the next node on a_i's path;
4      if v is not occupied and does not appear in the residual
         path of any other agent then
5          U ← U ∪ {a_i};
6          B ← B \ {a_i};
7      if v is occupied by another agent then
8          B_K ← B_K ∪ {a_i};
9      if v is a_i's goal location and v appears in the residual
         path of another agent then
10         B_K ← B_K ∪ {a_i};
11 ΔU ← B \ B_K;
12 while ΔU ≠ ∅ do
13     P ← the residual paths of the agents A if all agents in
         U ∪ ΔU arrive at their next nodes;
14     if (P is feasible) then
15         break;
16     if an agent returned from feasibility test is in ΔU then
17         remove the agent from ΔU;
18     else
19         ΔU ← ∅;
20 if U ∪ ΔU = ∅ then
21     foreach agent a_i in B \ B_K do
22         P ← the residual paths of the agents A if all
             agents in U ∪ {a_i} arrive at their next nodes;
23         if (P is feasible) then
24             ΔU ← {a_i};
25             break;
26 return ΔU;
```

LDG (line 20 of Algorithm 1). Typically, between these two agents, at least one agent a comes from ΔU, since the residual path plan before unblocking ΔU was feasible. In this case, we remove a from ΔU to break cycles and check the feasibility of the residual path plan again. This process is repeated until the residual path plan becomes feasible (lines 12-17). If both agents returned from the feasibility test are not in ΔU, we simply set ΔU to an empty set (lines 18-19).

In the special case that both U and the resultant ΔU are empty, we need to pick at least one agent to unblock in order to proceed with the path execution. To do so, we examine each agent in $B \setminus B_K$ and check whether unblocking it alone leaves the residual path plan feasible (lines 21-25). Since the path plan is feasible, at least one agent can be unblocked.

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

Experimental Evaluation

We perform experiments on three types of maps: (1) 30 $\times$ 30 grid maps where 30% nodes are marked as obstacles at

random; (2) the room-32-32-4 map and (3) the warehouse-10-20-10-2-1 map from the Moving AI MAPF benchmark (Stern et al. 2019). The start and goal locations of the agents are randomly positioned. For each instance generated, we run Explicit Estimation CBS (EECBS) (Li, Ruml, and Koenig 2021) with the suboptimality factor set to 1.1 to obtain a near-optimal MAPF solution under the standard assumption that each *move* action takes one timestep to complete. EECBS is a state-of-the-art variant of Conflict-Based Search (CBS) (Sharon et al. 2015) that incorporates many techniques for speeding up CBS.

We compare our proposed method with four methods. The first method is a baseline method that follows the precedence dependencies among agents derived from the EECBS solution by applying the MCP execution policy (Ma, Kumar, and Koenig 2017). The second method is a replanning method that replans the paths of agents whenever any agent not yet reaching its goal location is delayed (by the pause described below). We again run EECBS with the suboptimality factor set to 1.1 for replanning. For fair comparison with our method, replanning does not know the length of the pause and it assumes that the pause will finish in the current timestep. Thus, if the pause lasts for multiple timesteps, replanning is carried out repeatedly. The third method is Causal-PIBT+ (Okumura, Tamura, and Défago 2021), which is a state-of-the-art online progressive planning method looking one step ahead to avoid conflicts and guarantee reachability (all agents will reach their goal locations but not necessarily be there at the same time). Causal-PIBT+ uses an offline plan as hints to compensate for short-sightedness. We use the aforesaid EECBS solution as hints for Causal-PIBT+. The fourth method is BTPG-optimized (Su, Veerapaneni, and Li 2024). BTPG identifies the precedence dependencies (unsettled edge pairs) that can be switched in any manner without introducing cycles in the dependency graph (LDG). That is, if there are n edge pairs identified, up to 2^n possible dependency graphs (complete LDGs) are required to be acyclic. The advantage is low overhead to switch dependencies during the execution as there is no need to check for cycles. But this approach may significantly restrict the set of switchable dependencies. In contrast, all dependencies can possibly be switched in our method (as long as one acyclic complete LDG is identified, even if switching dependencies in other ways create cycles).

We emulate uncertainty by pausing 10% randomly chosen agents every k timesteps. Each pause lasts for k timesteps. If an agent is executing a *wait* action when paused, it will stay at its current node for $k+1$ timesteps; if an agent is executing a *move* action when paused, it will arrive at the adjacent node after $k+1$ timesteps. For fair comparison, we pause the same subset of agents at the same timestep for all the algorithms. We measure the sum-of-costs, i.e., the total timesteps taken by all agents to arrive at their goal locations. Given a k setting, we run the path execution for 10 times with different random subsets of agents paused. Overall, there are 1000 runs for random grid maps (10 maps $\times$ 10 agent placements $\times$ 10 uncertainty emulations) and 500 runs for the room or warehouse map (50 agent placements $\times$ 10 uncertainty emulations). We present the average result

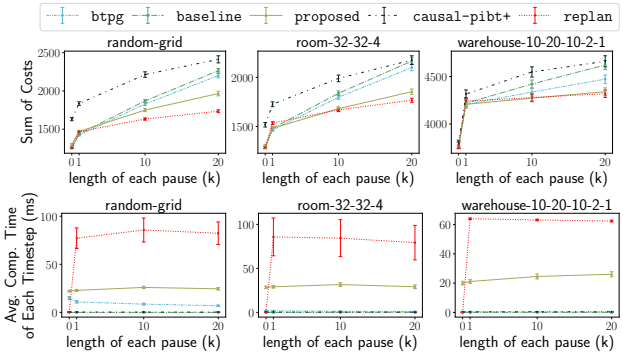

Figure 5: Comparison of proposed and other methods

for these runs together with the 95% confidence interval. We implement the algorithms in C++ and run the experiments on a machine with Intel Core i9-9820X 3.30GHz CPU and 64GB memory.

Figure 5 shows the results for 40 agents. As seen from the upper row of Figure 5, when $k = 0$ (no pause in path execution), the methods except Causal-PIBT+ have similar sum-of-costs. When $k > 0$, our method substantially reduces the sum-of-costs compared to the baseline method. The improvement generally increases with the length of each pause. The replanning method often produces lower sum-of-costs than our method. This is because replanning can also change the paths for agents to travel, while our method keeps the paths of agents unchanged. It can be seen that the sum-of-costs difference between our method and replanning is much smaller than that between our method and the baseline method. This implies that simply adjusting the order for agents to occupy common nodes can attain most of the benefits by replanning in optimization. The sum-of-costs of BTPG is usually between the baseline method and our method. This is because our method allows more switchable dependencies than BTPG. Causal-PIBT+ performs worse than the baseline method in many cases. Causal-PIBT+ is less successful in optimizing the sum-of-costs because it looks only a single step ahead in path planning.

The lower row of Figure 5 compares the average computational time per timestep in path execution. The baseline, BTPG and Causal-PIBT+ methods have fairly low computational times. The replanning method takes much higher computational time than our method. For our method, we observe that running Algorithm 2 on average requires less than 2 feasibility tests and running a feasibility test is much faster than replanning. From our experiments, the average computational time of our method is about 0.25s for 50 agents, 0.7s for 60 agents, and 2.5s for 70 agents on random grid maps.

Conclusion

We have developed a generic framework for coping with arbitrary delays in the execution of a multi-agent path plan. The framework takes planned paths of agents as input and coordinates agent movements in an online fashion according to the evolving status to avoid conflicts and deadlocks. Experimental results show the advantages of our algorithms.

Acknowledgments

This study is supported under the RIE2020 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from Singapore Telecommunications Limited (Singtel), through Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU), and by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award MOE-T2EP20121-0005).

References

Arbib, C.; Italiano, G. F.; and Panconesi, A. 1990. Predicting Deadlock in Store-and-Forward Networks. *Networks*, 20(7): 861–881.

Arkin, E. M.; Banik, A.; Carmi, P.; Citovsky, G.; Katz, M. J.; Mitchell, J. S.; and Simakov, M. 2018. Selecting and Covering Colored Points. *Discrete Applied Mathematics*, 250: 75–86.

Atzmon, D.; Stern, R.; Felner, A.; Sturtevant, N. R.; and Koenig, S. 2020a. Probabilistic Robust Multi-Agent Path Finding. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 29–37.

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2020b. Robust Multi-Agent Path Finding and Executing. *Journal of Artificial Intelligence Research*, 67: 549–579.

Berndt, A.; Van Duijkeren, N.; Palmieri, L.; and Keviczky, T. 2020. A Feedback Scheme to Reorder a Multi-Agent Execution Schedule by Persistently Optimizing a Switchable Action Dependency Graph. *arXiv preprint arXiv:2010.05254*.

Coskun, A.; O'Kane, J.; and Valtorta, M. 2021. Deadlock-Free Online Plan Repair in Multi-robot Coordination with Disturbances. In *Proceedings of the International FLAIRS Conference*, volume 34.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N.; Wagner, G.; and Surynek, P. 2017. Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges. In *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS)*, 29–37.

Hönig, W.; Kumar, T. K. S.; Cohen, L.; Ma, H.; Xu, H.; ; Ayanian, N.; and Koenig, S. 2016. Multi-Agent Path Finding with Kinematic Constraints. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, 477–485.

Hönig, W.; Keisel, S.; Tinka, A.; Durham, J. W.; and Ayanian, N. 2019. Persistent and Robust Execution of MAPF Schedules in Warehouses. *IEEE Robotics and Automation Letters*, 4(2): 1125–1131.

Karp, R. M. 1972. Reducibility among Combinatorial Problems. In Miller, R. E.; Thatcher, J. W.; and Bohlinger, J. D., eds., *Complexity of Computer Computations*, 85–103. Boston, MA: Springer US. ISBN 978-1-4684-2001-2.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, 12353–12362. https://github.com/Jiaoyang-Li/EECBS.

Ma, H.; Kumar, T. S.; and Koenig, S. 2017. Multi-Agent Path Finding with Delay Probabilities. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, 3605–3612.

Mascis, A.; and Pacciarelli, D. 2002. Job-shop Scheduling with Blocking and No-wait Constraints. *European Journal of Operational Research*, 143(3): 498–517.

Okumura, K.; Tamura, Y.; and Défago, X. 2021. Time-Independent Planning for Multiple Moving Agents. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, 11299–11307. https://github.com/Kei18/time-independent-planning.

Paul, A.; Feng, Y.; and Li, J. 2023. A Fast Rescheduling Algorithm for Real-Time Multi-Robot Coordination [Extended Abstract]. In *Proceedings of the 16th Annual Symposium on Combinatorial Search (SoCS)*, 175–176.

Shahar, T.; Shekhar, S.; Atzmon, D.; Saffidine, A.; Juba, B.; and Stern, R. 2021. Safe Multi-Agent Pathfinding with Time Uncertainty. *Journal of Artificial Intelligence Research*, 70: 923–954.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence*, 219: 40–66.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS)*, 151–158.

Su, Y.; Veerapaneni, R.; and Li, J. 2024. Bidirectional Temporal Plan Graph: Enabling Switchable Passing Orders for More Efficient Multi-Agent Path Finding Plan Execution. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, 17559–17566. https://github.com/YifanSu1301/BTPG.

Wagner, G.; and Choset, H. 2017. Path Planning for Multiple Agents Under Uncertainty. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*, 577–585.