A Real-Time Rescheduling Algorithm for Multi-robot Plan Execution

Primary Keywords: *Multi-Agent Planning*

Abstract

One area of research in Multi-Agent Path Finding (MAPF) is to determine how re-planning can be efficiently achieved in the case of agents being delayed during execution. One option is to determine a new wait ordering to find the most optimal new solution that can be produced by re-ordering the wait order. We propose to use a Switchable Temporal Plan Graph and a heuristic search algorithm to approach finding a new optimal wait order. We prove the admissibility of our algorithm and experiment with its efficiency in a variety of conditions by measuring re-planning speed in different maps, with varying numbers of agents and randomized scenarios for agents' start and goal locations. Our algorithm shows a fast runtime in all experimental setups.

Introduction

Multi-Agent Path Finding (MAPF) is the problem of finding a collision-free plan that navigates a team of agents from their start locations to goal locations. There are various versions of this problem with different constraints and assumptions, such as whether or not agents may follow behind each other in a path, whether wait times for agents should allow for various properties of the robots, and whether agents can switch locations traveling along the same path and so on (Stern et al. 2019). Applications of this problem are relevant in a variety of planning and scheduling tasks, such as intralogistics (Wurman, D'Andrea, and Mountz 2007), aircraft towing (Morris et al. 2016), and video games (Ma et al. 2017). However, the MAPF problem has been shown to be NP-hard to solve optimally (Yu and LaValle 2013). Thus, optimizing the speed with which optimal solutions can be determined has become an active area of research.

During the execution of the plan, agents may have unexpected delays due to mechanical differences, unexpected events, and so on. To handle such delays, existing research proposes using a Temporal Plan Graph(Hönig et al. 2016), which records the precedence relationships of a MAPF solution, to execute the plan with robustness, where each agent in TPG only moves to the next vertex recorded in its plan if the related precedence relationships are satisfied. However, execution with TPG introduces a large amount of waits into the execution results due to the knock-on effect in the precedence relationship.

Problem Definition

We first introduce the formal definition of MAPF.

Definition 1 (MAPF)**.** Multi-Agent Path Finding (MAPF) is an optimization problem of finding collision-free paths for a team of agents $\mathcal{A}$ on a given map. Each agent has a unique start location and a unique goal location. Time is discretized into unit-size steps. In each timestep, agents can move to an adjacent location or wait at the current location. A path specifies the actions of an agent at each timestep from its start location to its goal location. We say two agents $i, j \in \mathcal{A}$ collide if either of the following happens:

1. i and j are at the same location at the same timestep.
2. i leaves a location l at a timestep t and j enters the same location l at the timestep t.

A MAPF solution is a set of collision-free paths, one for each agent in $\mathcal{A}$.

Remark 1. The above definition of collision coincides with that in the setting of *k-robust plan* (Chen et al. 2021) with k set to 1. Thus our replanning algorithm will operate on an 1-robust MAPF solution.

For our discussion, we will stick to the following format for a MAPF solution, though our algorithm is not dependent on the specific format of the MAPF solution.

Assumption 1. A MAPF solution takes the form of a set of paths $\mathcal{P} = \{p_i : i \in \mathcal{A}\}$. Each path p_i is an ordered sequence of location-timestep tuples $(l_0^i, t_0^i) \rightarrow (l_1^i, t_1^i) \rightarrow \cdots \rightarrow (l_{zi}^i, t_{zi}^i)$ with the following properties:

- $t_0^i = 0$. l_0^i is the start location of agent i and l_{zi}^i is its goal location.
- Each tuple (l_k^i, t_k^i) in p_i for $k > 0$ indicates that agent i is planned to perform a move action into the location l_k^i at timestep t_k^i. So t_{zi}^i records the time when agent i reaches its goal, i.e. the travel time of i.
- We require a temporal ordering of the sequence: $t_k^i < t_s^i$ for all $0 \leq k < s \leq zi$.

These properties together force all consecutive pairs of location l_k^i and l_{k+1}^i to be adjacent on the map. A wait action is implicitly defined between two consecutive tuples (l_k^i, t_k^i) and (l_{k+1}^i, t_{k+1}^i): if $t_{k+1}^i - t_k^i = \Delta$, then agent i is planned to wait at l_k^i for $\Delta - 1$ timesteps before moving to l_{k+1}^i.

We also formalize some phrases related to MAPF: A MAPF solution is **optimal** if it minimizes the sum of travel time for all agents, i.e., $\sum_{i \in \mathcal{A}} t^i_{zi}$. And agents are said to be **executing** a MAPF solution if they act as specified in their paths.

In this paper, we consider the replanning problem when agents are subject to a delay during execution. We model this as Problem 1, parameterized by a delay probability p and a delay length d.

Problem 1 ((p, Δ)-Delay)**.** Given a MAPF solution, at any timestep t during the execution, with probability p a delay happens, in which: Let $\mathcal{B}$ denote the set of agents that have already reached their goals. Then a uniformly random agent $i \in (\mathcal{A} \setminus \mathcal{B})$ is forced to wait at its current location from timestep t to $t + (\Delta - 1)$.

When such a delay happens, the delayed agent might block the paths of other undelayed agents and thus hinder their execution. One naive fix is that once a delay happens, we re-run a MAPF solver with this delay constraint to produce a new solution. However, as discussed in the introduction, this approach is usually expensive. Instead, we propose a fast replanning algorithm that lets agents stick to their original location-wise paths but with different move-or-wait sequences.

Temporal Plan Graph

Roughly, our algorithm optimizes the *orderings* for multiple agents that are planned to visit the same location. This is achieved using a graph-based abstraction called Temporal Plan Graph.

Definition 2 (TPG, (Hönig et al. 2016))**.** A Temporal Plan Graph (TPG) is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that represents the precedence relationships of a MAPF solution. Given a MAPF solution $\mathcal{P}$, the corresponding TPG is defined as follows:

The set of vertices is $\mathcal{V} = \{v^i_k : i \in \mathcal{A}, k \in [0, zi]\}$, where each vertex v^i_k corresponds to the k^{th} tuple in the path p_i. The set of edges $\mathcal{E}$ is partitioned into two types of edges $\mathcal{E}_1$ and $\mathcal{E}_2$.

- **Type 1 edges** connect two vertices corresponding to consecutive tuples for the same agent. The set of Type 1 edges is $\mathcal{E}_1 = \{(v^i_k, v^i_{k+1}) : \forall i \in \mathcal{A}, k \in [0, zi)\}$.
- **Type 2 edges** connect two vertices of distinct agents, as long as they correspond to tuples containing the same location. The set of Type 2 edges is

$$\mathcal{E}_2 = \{(v^j_{s+1}, v^i_k) : \forall i \neq j \in \mathcal{A}, s \in [0, zj), k \in [0, zi] \text{ satisfying } l^j_s = l^i_k \text{ and } t^j_s < t^i_k\}$$

Remark 2. Note that in the above description, we define Type 2 edge as (v^j_{s+1}, v^i_k) instead of (v^j_s, v^i_k) in order to avoid the second type of collision in Definition 1

Executing a TPG

A TPG contains sufficient information for agents' execution. Procedure 1 describes how to execute a TPG in detail, which includes two helper functions $\text{INIT}_{\text{EXEC}}$ and $\text{STEP}_{\text{EXEC}}$ and a main function EXEC, along with two marks "satisfied" and "unsatisfied" for vertices.

Procedure 1: Execute ($\mathcal{G} = (\mathcal{V}, \mathcal{E})$)
Lines highlighted in blue are activated to compute the cost of a TPG, and can be omitted for the mere purpose of execution.

```
1  Define a counter cost;
2  Function INIT_EXEC(G)
3      cost ← 0;
4      Mark all vertices in V_0 = {v_0^i : i ∈ A} as satisfied;
5      Mark all remaining v ∈ (V \ V_0) as unsatisfied;
6  Function STEP_EXEC(G, i)
7      if ∀k : v_k^i satisfied then
8          return NULL;
9      cost ← cost + 1;
10     v ← v_k^i : v_k^i unsatisfied and ∀k' < k, v_k'^i satisfied;
11     forall (u, v) ∈ E do
12         if u unsatisfied then
13             return NULL
14     return v
15 Function EXEC(G)
16     INIT_EXEC(G);
17     while there exists unsatisfied vertex in V do
18         Define a set S ← ∅;
19         forall agent i ∈ A do
20             Add STEP_EXEC(G, i) into S;
21         forall v ∈ S do
22             if v ≠ NULL then
23                 Mark v as satisfied;
24     return cost;
```

Intuitively, marking a vertex as satisfied corresponds to moving an agent to the corresponding location, and we do so if and only if all in-neighbors of this vertex have already been satisfied. The execution terminates when all vertices are satisfied, i.e. all agents have reached their goals. We now formally state and prove some properties of a TPG.

Definition 3 (Deadlock)**.** When executing a TPG as Procedure 1, we say a *deadlock* is encountered iff in an iteration of the while-loop [line 17], the set $\mathcal{S}$ contains only NULL on line 21 yet there exists unsatisfied vertex in $\mathcal{V}$.

Lemma 1. *Executing $\mathcal{G}$ encounters a deadlock if and only if there exists a cycle in $\mathcal{G}$.*

Proof. Assuming that a deadlock is encountered in the t^{th} iteration of the loop. Let $\mathcal{V}'$ denote that set of all vertices that are unsatisfied, which is non-empty by the definition of deadlock. Assume towards contradiction that $\mathcal{G}$ is acyclic, then there exists a topological ordering (i.e. a linear ordering of its vertices such that for every directed edge (u, v), u

comes before v in the ordering) of $\mathcal{V}'$. In this case, $\mathcal{S}$ must contain the first vertex v_{first} in the ordering of $\mathcal{V}'$, because it satisfies both conditions in $\text{STEP}_{\text{EXEC}}$:

1. $v_{\text{first}} = \arg\min_k(v_k^i$: agent $i \in A$, v_k^i is unsatisfied), and
2. For all $(u, v_{\text{first}}) \in \mathcal{E}$, u is satisfied.

This contradicts the deadlock condition that $\mathcal{S} = \{\text{NULL}\}$. □

Lemma 1 shows a correspondence between a cycle and a deadlock that holds for all TPG. Next, we focus on properties that are specific to a TPG that is constructed from a MAPF solution.

Proposition 2 (Collision-Free)**.** *Let $\mathcal{G}$ be a TPG constructed from a MAPF solution as in Definition 2. Assuming $\mathcal{G}$ is executed as in Procedure 1 and an agent i is moved to its k^{th} location l_k^i at timestep t iff vertex v_k^i is satisfied in the t^{th} iteration of the while-loop on line 17, any two agents i, j never collide.*

Proof. Assume towards contradiction that i and j collide because they are at the same location at the same timestep t, i.e. after the t^{th} iteration, $v_k^i = \arg\max_k(v_k^i : v_k^i$ satisfied), $v_s^j = \arg\max_s(v_s^j : v_s^j$ satisfied), and $l_k^i = l_s^j$. Either edge (v_{k+1}^i, v_s^j) or (v_{s+1}^j, v_k^i) should be in $\mathcal{E}$. But $(v_{k+1}^i, v_s^j) \notin \mathcal{E}$ as otherwise v_s^j cannot be satisfied since v_{k+1}^i is unsatisfied; similarly $(v_{s+1}^j, v_k^i) \notin \mathcal{E}$ as otherwise v_k^i cannot be satisfied since v_{s+1}^j is unsatisfied. This shows a contradiction.

If they collide because i leaves a location at a timestep t, and j enters the same location at timestep t, then $l_{k-1}^i = l_s^j$ and vertices v_k^i and v_s^j are satisfied exactly in the t^{th} iteration of the loop. However, this is impossible since either (v_k^i, v_s^j) or (v_{s+1}^j, v_{k-1}^i) is in $\mathcal{E}$, but the out-going vertex in neither of these edges are satisfied before the t^{th} iteration. □

One can also derive the following interesting corollary from the proof of Proposition 2, which is going to be useful later for our algorithm.

Corollary 3. *Let $\mathcal{G}$ be a collision-free TPG. If we replace an arbitrary Type 2 edge (v_{s+1}^j, v_k^i) in it with (v_{k+1}^i, v_s^j), the TPG remains to be collision-free.*

Proof. This is observed from the fact that the proof of Proposition 2 argues the non-existence of two pairs of edges $(v_{k+1}^i, v_s^j) - (v_{s+1}^j, v_k^i)$ and $(v_k^i, v_k^j s) - (v_{s+1}^j, v_{k-1}^i)$. Since in both pairs, the two edges are equal upon the replacement, the proof remains exactly the same after we perform an arbitrary replacement. □

Next, we relate the *cost* of a TPG with the sum of travel timesteps of a MAPF solution.

Proposition 4. *Let $\mathcal{G}$ be a TPG constructed from a MAPF solution $\mathcal{P}$, the cost of $\mathcal{G}$ is no greater than the sum of travel time for agents following $\mathcal{P}$.*

We rely on the following lemma to prove Proposition 4.

Lemma 5. *If an arbitrary agent i is planned to move to location l_k^i at a time $t = t_k^i$ in $\mathcal{P}$, then vertex v_k^i is either satisfied or can be satisfied in the t^{th} iteration of the while-loop.*

Proof (of Lemma 5). We induct on t. When $t = 0$, this holds by the functionality of $\text{INIT}_{\text{EXEC}}$. For $t > 0$, we consider the non-trivial case that the index k of v_k^i is non-zero. For the Type 1 edge of v_k^i, since $t_{k-1}^i \leq t_k^i - 1 = t - 1$, v_{k-1}^i is satisfied by an inductive hypothesis. Let v_s^j be an arbitrary Type 2 in-neighbor of v_k^i, by construction of Type 2 edges, $t_{s-1}^j < t_k^i$, so $t_{s-1}^j \leq t - 1$. This shows that all in-neighbors of v_k^i must be satisfied after the $(t-1)^{\text{th}}$ iteration, thus v_k^i can be satisfied in the t^{th} iteration if it has not been satisfied yet. □

Proof (of Proposition 4). Lemma 5 shows that if $\mathcal{P}$ plans an agent i to enter its goal location at time t_{zi}^i, then all vertices of i are satisfied after the $(t_{zi}^i)^{\text{th}}$ iteration, i.e. agent i contributes to *cost* by at most t_{zi}^i units. Therefore $cost \leq \sum_{i \in \mathcal{A}} t_{zi}^i$, which is the sum of travel times of all agents in $\mathcal{P}$. □

Proposition 4 gives an immediate corollary that $\mathcal{G}$ is deadlock-free.

Corollary 6 (Deadlock-Free)**.** *If a TPG $\mathcal{G}$ is constructed from a MAPF solution $\mathcal{P}$, then it is deadlock-free.*

Proof. If $\mathcal{G}$ contains a deadlock, then its execution would enter the while-loop for infinitely many iterations, and in each iteration, *cost* strictly increases. Thus $cost = \infty$. Yet the sum of travel time of $\mathcal{P}$ is always finite, contradicting Proposition 4. □

Switchable TPG

TPG is a handy representation for precedence relationships. However, a standard TPG constructed as in Definition 2 is fixed and bound to a given set of paths. In contrast, our optimization algorithm will use the following extended notion of TPG, which enables flexible modifications of precedence relationships.

Definition 4 (Switchable TPG)**.** Given a TPG $\mathcal{G}_0$, let $\mathcal{E}_1$ denote its set of Type 1 edges and $\mathcal{E}_2$ denote its set of Type 2 edges, such that $\mathcal{G}_0 = (\mathcal{V}, \mathcal{E}_1, \mathcal{E}_2)$. A switchable variant of $\mathcal{G}_0$ is $\mathcal{G} = (\mathcal{V}, \mathcal{E}_1, (\mathcal{S}_{\mathcal{E}2}, \mathcal{N}_{\mathcal{E}2}))$, which partitions $\mathcal{E}_2$ into two disjoint subsets $\mathcal{S}_{\mathcal{E}2}$ (switchable Type 2 edges) and $\mathcal{N}_{\mathcal{E}2}$ (non-switchable Type 2 edges), and allows two operations with respect to any switchable edge $(v_{s+1}^j, v_k^i) \in \mathcal{S}_{\mathcal{E}2}$:

- $fix(v_{s+1}^j, v_k^i)$ removes this edge from $\mathcal{S}_{\mathcal{E}2}$ and add the same edge into $\mathcal{N}_{\mathcal{E}2}$. i.e., this operation fixes a switchable edge to be non-switchable.
- $reverse(v_{s+1}^j, v_k^i)$ removes this edge from $\mathcal{S}_{\mathcal{E}2}$ and add (v_{k+1}^i, v_s^j) into $\mathcal{N}_{\mathcal{E}2}$. i.e., this operation switches the precedence relation and then fixes it to be non-switchable.

Definition 4 defines a strict superclass of Definition 2. For clarity, we may refer to a TPG satisfying Definition 2 as a

non-switchable or a *standard* TPG. A switchable TPG degenerates into a standard TPG if $\mathcal{S}_{\mathcal{E}2}$ is empty, in which case its cost can be determined using Procedure 1. We say a switchable TPG $\mathcal{G}$ *produces* a standard TPG $\mathcal{G}'$ if $\mathcal{G}'$ can be generated through a sequence of fix or $reverse$ operations on $\mathcal{G}$. Note that by Corollary 3, all $\mathcal{G}'$ producible from $\mathcal{G}$ are simultaneously deadlock-free or not.

Then the roadmap of our replanning algorithm is:

(1) when a delay happens, construct a switchable TPG $\mathcal{G}$ corresponding to the current states of agents, and then

(2) produces a standard TPG $\mathcal{G}'$ from $\mathcal{G}$ with the minimum possible cost (among all $\mathcal{G}$-producible $\mathcal{G}'$), such that it represents an optimal ordering of precedence relationships, upon sticking to the original location-wise paths.

We end this section by specifying the construction of a switchable TPG corresponding to a delay situation.

Construction 1. Given a MAPF solution $\mathcal{P}$, we execute it by running Procedure 1 on its corresponding standard TPG $\mathcal{G}_0$ as constructed in Definition 2. Assume that at timestep t during the execution, agent i is delayed for Δ timesteps. We construct a switchable TPG $\mathcal{G}$ for this situation as follows:

1. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}_1, \mathcal{E}_2)$ be a copy of the standard TPG $\mathcal{G}_0$.
2. Convert it to a switchable TPG$(\mathcal{V}, \mathcal{E}_1, (\mathcal{S}_{\mathcal{E}2}, \mathcal{N}_{\mathcal{E}2}))$ by defining $\mathcal{N}_{\mathcal{E}2} = \{(u, v) : \text{ either } u \text{ or } v \text{ is marked as satisfied in the execution of } \mathcal{G}_0\}$ and $\mathcal{S}_{\mathcal{E}2} = (\mathcal{E}_2 \setminus \mathcal{N}_{\mathcal{E}2})$.
3. Let v_k^i be the next unsatisfied vertex of i, i.e. $v_k^i \leftarrow \text{STEP}_{\text{EXEC}}(\mathcal{G}_0, i)$ from Procedure 1. Create Δ-many new dummy vertices: $\mathcal{V}_{\text{new}} = \{v_1, v_2, \cdots, v_\Delta\}$ and $(\Delta + 1)$-many new edges:

   $\mathcal{E}_{\text{new}} = \{(v_{k-1}^i, v_1), (v_1, v_2), \cdots, (v_{\Delta-1}, v_\Delta), (v_\Delta, v_k^i)\}$.

   Modify $\mathcal{G}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_{\text{new}}$ and $\mathcal{E}_1 \leftarrow (\mathcal{E}_1 \cup \mathcal{E}_{\text{new}}) \setminus \{(v_{k-1}^i, v_k^i)\}$.

When executing this TPG, we interpret satisfying a dummy vertex as agent i waiting at location l_{k-1}^i that corresponds to vertex v_{k-1}^i.

Theorem 7. *Let $\mathcal{G}$ be a switchable TPG constructed as in Construction* 1*, there always exists a finite-cost, collision-free standard TPG that can be produced from $\mathcal{G}$.*

Proof. One naive solution $\mathcal{G}_{\text{naive}}$ is produced by fixing all switchable edges in $\mathcal{G}$. $\mathcal{G}_{\text{naive}}$ has a finite cost (i.e. is deadlock-free) because by Corollary 6, an initial TPG $\mathcal{G}_0$ constructed from a MAPF solution is deadlock-free. And by Lemma 1, a TPG has a deadlock iff it contains a cycle, so it suffices to argue that step 2 and 3 in Construction 1 does not introduce a new cycle. This holds since step 2 has no effect once we fix all switchable edges. Step 3 behaves as expanding a pre-existing edge (v_{k-1}^i, v_k^i) into a line of connecting edges. If a cycle exists in $\mathcal{G}_{\text{naive}}$, it either involves edges in this line or not. In the latter case, this cycle would exist exactly in $\mathcal{G}_0$, which is impossible. In the former case, the entire line has to be contained in this cycle, in which case (v_{k-1}^i, v_k^i) along with the remaining component of this cycle would form a cycle in $\mathcal{G}_0$, which is impossible.

Executing $\mathcal{G}_{\text{naive}}$ is collision-free because the exact same proof of Proposition 2 shows that two agents cannot collide if none of them is the delayed agent i or if the most-recently satisfied vertex of i is not a dummy vertex. So we may assume without loss of generality that agent i and j collide when i has already entered location l_{k-1}^i. However, such a collision is impossible since any vertex v_s^j for $j \neq i$ corresponding to the same location cannot be satisfied before v_k^i is satisfied. □

Algorithm

We now describe our algorithm in a top-down modular manner, starting with a high-level heuristic search framework in Algorithm 2. We abuse the operators fix and $reverse$ to take a switchable TPG along with a switchable edge as inputs and return a new TPG after the operation.

Algorithm 2: Replanning
HEURISTIC, TERMINATE, CYCLEDETECTION, and BRANCH are modules that will be specified later. $\mathcal{X}$ denotes some auxiliary information accompanying a TPG, whose format is defined by the set of modules.

Input: TPG $\mathcal{G}_{\text{root}} = (\mathcal{V}, \mathcal{E}_1, (\mathcal{S}_{\mathcal{E}2}, \mathcal{N}_{\mathcal{E}2}))$
Output: TPG $\mathcal{G}_{\text{result}}$

Initialize an empty priority queue $\mathcal{Q}$;
$h_{\text{root}} \leftarrow \text{HEURISTIC}(\mathcal{G}_{\text{root}}, \mathcal{X}_{init})$;
$\mathcal{Q}.push((\mathcal{G}_{\text{root}}, \mathcal{X}_{init}), 0, h_{\text{root}})$;
while $\mathcal{Q}$ is not empty **do**
  $((\mathcal{G}, \mathcal{X}), g, h) \leftarrow \mathcal{Q}.pop()$;
  $(g', \mathcal{X}', (v_k^i, v_s^j)) \leftarrow \text{BRANCH}(\mathcal{G}, \mathcal{X})$;
  if TERMINATE$(\mathcal{G}, \mathcal{X}')$ **then**
    fix all edges in $\mathcal{S}_{\mathcal{E}2}$ of $\mathcal{G}$;
    return $\mathcal{G}$;
  $\mathcal{G}_{\text{f}} \leftarrow fix(\mathcal{G}', (v_k^i, v_s^j))$;
  if not CYCLEDETECTION$(\mathcal{G}_{\text{f}}, (v_k^i, v_s^j))$ **then**
    $h_f \leftarrow \text{HEURISTIC}(\mathcal{G}_{\text{f}}, \mathcal{X}')$;
    $\mathcal{Q}.push((\mathcal{G}_{\text{f}}, \mathcal{X}'), g + g', h_f)$;
  $\mathcal{G}_{\text{r}} \leftarrow reverse(\mathcal{G}', (v_k^i, v_s^j))$;
  if not CYCLEDETECTION$(\mathcal{G}_{\text{r}}, (v_{s+1}^j, v_k^i))$ **then**
    $h_f \leftarrow \text{HEURISTIC}(\mathcal{G}_{\text{f}}, \mathcal{X}')$;
    $\mathcal{Q}.push(\mathcal{G}_{\text{r}}, \mathcal{X}'), g + g', h_r)$;
throw exception "No solution found";

Before analyzing Algorithm 2, we recall the notion of cost of a standard TPG and define a similar notion of *partial cost* for a switchable TPG as the cost of its *reduced* standard TPG, which contains only its non-switchable edges.

Lemma 8. *Let $\mathcal{G}_{switch}$ be a switchable TPG and $\mathcal{G}$ be an arbitrary standard TPG produced from $\mathcal{G}_{switch}$. The partial cost of $\mathcal{G}_{switch}$ is no greater than the cost of $\mathcal{G}$.*

Proof. Let $\mathcal{G}_{\text{red}}$ be the reduced standard TPG of $\mathcal{G}_{\text{switch}}$ that contains only its non-switchable edges. Consider running Procedure 1 on $\mathcal{G}$ and $\mathcal{G}_{\text{red}}$, respectively. Since an edge appears in $\mathcal{G}_{\text{red}}$ must also appear in $\mathcal{G}$, we can inductive show that in any call to $\text{STEP}_{\text{EXEC}}$, if a vertex v can be marked as

satisfied in $\mathcal{G}$, then it can be marked as satisfied in $\mathcal{G}_{\text{red}}$ as well. Therefore the total timestep to satisfy all vertices in $\mathcal{G}_{\text{red}}$ cannot exceed that in $\mathcal{G}$. □

We now state and prove the correctness of our Algorithm 2, under some reasonable assumptions on the modules.

Assumption 2. Let $\mathcal{G}_{\text{root}}$ be a switchable TPG constructed as in Construction 1. We assume the modules in Algorithm 2 satisfy the following conditions:

- CYCLEDETECTION($\mathcal{G}$, (u, v)) returns true iff $\mathcal{G}'$ contains a cycle involving edge (u, v), where $\mathcal{G}'$ is the reduced standard TPG of $\mathcal{G}$ containing only its non-switchable edges.
- Given a TPG $\mathcal{G}$:
  - HEURISTIC$(\mathcal{G}, \mathcal{X})$ computes a value h.
  - BRANCH$(\mathcal{G}, \mathcal{X})$ outputs a value g', an updated auxiliary information $\mathcal{X}'$, and an edge in $\mathcal{S}_{\mathcal{E}2}$ if it is non-empty or NULL otherwise.

  Whenever we push $((\mathcal{G}, \mathcal{X}), g, h)$ into $\mathcal{Q}$, $g + h$ is guaranteed to be the partial cost of $\mathcal{G}$.
- On line 7 of Algorithm 2, TERMINATE returns true iff the partial cost of $\mathcal{G}$ equals the cost of $\mathcal{G}'$, where $\mathcal{G}'$ is produced from $\mathcal{G}$ by fixing all switchable edges.

We make the following observations from Assumption 2:

1. Under the assumption on TERMINATE, when Algorithm 2 reaches line 10, $\mathcal{G}$ must contain switchable edges so (v_k^i, v_s^j) returned by BRANCH is not NULL. This ensures that fix and $reverse$ on line 10 and 14 are well-defined.
2. As long as $\mathcal{G}_{\text{root}}$ is acyclic, it holds inductively that CYCLEDETECTION($\mathcal{G}$, (u, v)) on line 11 and 15 returns true iff the reduced standard TPG $\mathcal{G}'$ contains *any* cycle. This is because in each iteration, we introduce one new edge into $\mathcal{G}'$, and any new cycle formed in this iteration has to contain this new edge.

Theorem 9. *Under Assumption 2, taking $\mathcal{G}_{root}$ as an input, Algorithm 2 outputs a collision-free standard TPG $\mathcal{G}_{result}$ with the minimum cost among all possible $\mathcal{G}_{root}$-producible standard TPG.*

Proof. Theorem 7 shows the existence of a solution (in particular, it is collision-free so by Corollary 3, any $\mathcal{G}_{\text{root}}$-producible standard TPG is collision-free.) We now prove the completeness and then admissibility of Algorithm 2.

Algorithm 2 always terminates because by assumption TERMINATE($\mathcal{G}$) returns true once $\mathcal{G}$ contains no switchable edge. There are only finitely many possible operation sequences from $\mathcal{G}_{\text{root}}$ to any standard TPG, each corresponds to an element that can possibly be added to $\mathcal{Q}$. Thus the algorithm must return a solution or report an exception when all possibilities are exhausted after a finite number of steps. We show the completeness using the following claim:

Claim 10. *Let $\mathcal{G}'$ be an arbitrary deadlock-free solution that can be produced from $\mathcal{G}_{root}$. At the beginning of each iteration of the while-loop on line 4, there exists some $\mathcal{G} \in \mathcal{Q}$ such that $\mathcal{G}'$ can be produced from $\mathcal{G}$.*

Proof (of Claim 10). This holds inductively: at the beginning of the first iteration, $\mathcal{G}_{\text{root}} \in \mathcal{Q}$. During any iteration, if some $\mathcal{G} \in \mathcal{Q}$ such that $\mathcal{G}$ can produce $\mathcal{G}'$ is popped on line 5, then one of the following must hold:

- $\mathcal{G} = \mathcal{G}'$: $\mathcal{G}$ contains no switchable edge, thus the algorithm terminates in this iteration and the inductive step holds vacuously.
- $\mathcal{G}_f$ produces $\mathcal{G}'$: since $\mathcal{G}'$ is acyclic, so is the reduced TPG of $\mathcal{G}_f$. Thus $\mathcal{G}_f$ won't be pruned by CYCLEDETECTION and is added into $\mathcal{Q}$.
- $\mathcal{G}_r$ produces $\mathcal{G}'$: this is symmetric to the previous case.

In any case, the claim remains true after this iteration. □

Claim 10 shows that Algorithm 2 is guaranteed to find a solution if one exists, otherwise the priority queue would remain to be non-empty and the algorithm cannot exit the loop to throw an exception.

Finally, if $\mathcal{G}_{\text{result}}$ is outputted, it must have the minimum cost. Assume towards contradiction that when $\mathcal{G}_{\text{result}}$ is returned, there exists another $\mathcal{G}_0$ in $\mathcal{Q}$ that can produce a standard $\mathcal{G}_{\text{better}}$ with cost smaller than $\mathcal{G}_{\text{result}}$. Yet this is impossible since Lemma 8 implies that such $\mathcal{G}_0$ must have a smaller $g + h$ value and thus would be popped from $\mathcal{Q}$ before $\mathcal{G}_{\text{result}}$. On the other hand, Claim 10 shows that the existence of such a $\mathcal{G}_0$ is the necessary condition for the existence of $\mathcal{G}_{\text{better}}$. Therefore $\mathcal{G}_{\text{result}}$ has the minimum cost.

□

Execution-based Modules

In this and the subsequent section, we describe two sets of modules and prove that they satisfy Assumption 2. We start with describing a set of "execution-based" modules in Modules 3, which largely ensembles Procedure 1.

Proposition 11. *Module 3 satisfies Assumption 2.*

Proof. By the property of DFS, CYCLEDETECTION returns true iff there exists a cycle involving edge (u, v).

We show by induction that whenever we push $((\mathcal{G}, \mathcal{X}), g, h)$ into $\mathcal{Q}$ in Algorithm 2, $g + h$ equals the partial cost of $\mathcal{G}$.

- In the base case, on line 2 of Algorithm 2, HEURISTIC computes h_{root} by running exactly Procedure 1 on the reduced standard TPG of $\mathcal{G}_{\text{root}}$, so $0 + h_{\text{root}}$ is the partial cost of $0 + h_{\text{root}}$.
- Let $((\mathcal{G}, \mathcal{X}), g, h)$ be popped from $\mathcal{Q}$. Observe inductively that $\mathcal{X}$ records a state of execution of $\mathcal{G}$, and g records the execution *cost* up to that timestep. On line 6 of Algorithm 2, BRANCH continues execution from that state $\mathcal{X}$ until we reach the next switchable edge (as indicated by the modifications on line 13 of Modules 3). BRANCH outputs the execution cost g' from $\mathcal{X}$ to a current state $\mathcal{X}'$.
  Then line 10 or 14 of Algorithm 2 fix or $reverse$ this edge to get $\mathcal{G}_f$ or $\mathcal{G}_r$, and HEURISTIC on line 12 or 16 of Algorithm 2 continues executing the reduced TPG of $\mathcal{G}_f$ or $\mathcal{G}_r$ from $\mathcal{X}'$ till termination to get h. Thus $g + g' + h$ together sums to the total partial cost of $\mathcal{G}_f$ or $\mathcal{G}_r$.

Module 3: Execution-based Modules

1 **Function** CYCLEDETECTION$(\mathcal{G}, (u, v))$
2 Run DFS on $\mathcal{G}$ starting from vertex v;
3 **if** DFS encounters a cycle **then**
4 **return** true;
5 **return** false;

6 Auxillary information $\mathcal{X}$ is a map $\mathcal{X} : \mathcal{A} \to [0 : zi]$, which records the index of the most recently satisfied vertex for each agent;
7 Define $\mathcal{X}_{\text{init}}[i] = 0$ for all $i \in \mathcal{A}$;

8 **Function** TERMINATE$(\mathcal{G}, \mathcal{X})$
9 **if** $\forall i \in \mathcal{A}, \mathcal{X}[i] = zi$ **then**
10 **return** true;
11 **return** false;

12 **Function** BRANCH$(\mathcal{G}, \mathcal{X})$
13 Run Procedure 1 on $\mathcal{G}$ with the following modifications:
- Change line 4 of $\text{INIT}_{\text{EXEC}}$ to: Mark all vertices in $\mathcal{V}_0 = \{v_k^i : i \in \mathcal{A}, k \le \mathcal{X}[i]\}$ as satisfied;
- Change line 20 of EXEC to: **if** $v \leftarrow \text{STEP}_{\text{EXEC}}(\mathcal{G}, i) \neq$ NULL **and** $(\exists e \in \mathcal{S}_{\mathcal{E}2} : e := (v, u)$ or $(u, v))$ **then** **return** $(cost, \mathcal{X}' := \{i \mapsto \text{STEP}_{\text{EXEC}}(\mathcal{G}, i)\}, e)$; **else** Add v into $\mathcal{S}$;
- Change line 24 of EXEC to: **return** NULL;

  return the output of modified Procedure 1;

14 **Function** HEURISTIC$(\mathcal{G}, \mathcal{X})$
15 Define $\mathcal{G}'$ to be a reduced standard TPG containing only non-switchable edges of $\mathcal{G}$;
16 Run Procedure 1 on $\mathcal{G}'$ with the following modification:
- Change line 4 of $\text{INIT}_{\text{EXEC}}$ to: Mark all vertices in $\mathcal{V}_0 = \{v_k^i : i \in \mathcal{A}, k \le \mathcal{X}[i]\}$ as satisfied; **return** NULL;

  return the output of modified Procedure 1;

Finally, if on line 7 of Algorithm 2, TERMINATE returns true, then the last BRANCH outputs a goal state for all agents, i.e. the modified execution on line 13 of Modules 3 does not encounter any switchable edge. So it must have been executed on a standard TPG, whose partial cost is exactly its cost after vacuously fixing all its switchable edges. □

Therefore we show that Modules 3 provide an implementation for our algorithm framework.

Graph-based Modules

In this section, we introduce an alternative set of modules, which departs from the execution-based ideology of Modules 3 and focus on the graph properties of a TPG. We will see later in our experiment that this shift of focus significantly improves the efficiency of our algorithm. We start by presenting the following crucial theorem that provides a graph-based approach to compute the cost of a TPG.

Theorem 12. *Given a TPG, compute the longest path from vertex v_0^i to vertex v_{zi}^i for each $i \in \mathcal{A}$. Taking the sum of lengths of all such longest paths, this equals the cost of this TPG.*

Proof. To prove Theorem 12, we again refer back to Procedure 1. Fix a longest path from vertex v_0^i to vertex v_{zi}^i. We prove by induction that the distance from v_0^i to a vertex v_s^j on this longest path is equal to the number of iterations required in the while-loop (line 17) in Procedure 1 to satisfy v_k^i. In the base case, the distance from v_0^i to itself is indeed 0. In the inductive step, assuming v_s^j is satisfied in the $t-1^{\text{th}}$ iteration, and the longest path from v_0^i to v_s^j is $t-1$. Then the next vertex $v_{s'}^{j'}$ on the longest path is satisfied in the t^{th} iteration, because:

- #iterations $\ge t$ since v_s^j is a in-neighbor of $v_{s'}^{j'}$ which needs to be satisfied before $v_{s'}^{j'}$.
- #iterations $\le t$ since otherwise there must be another in-neighbor of $v_{s'}^{j'}$ that is not yet satisfied in the $t-1^{\text{th}}$ iteration, which is going to compose a longer path than the one we look at.

Therefore the cost computed by Procedure 1 which equals the sum of iterations for all agents to reach their goal vertex is exactly the sum of lengths of longest paths □

We adopt the following well-known algorithm to compute the longest paths:

- Compute a topological sort of all vertices in the TPG.
- Given a source vertex v_0, assign distance $dist(v_0) = 0$ and $dist(v) = -\infty$ for all $v \neq v_0$.
- For each vertex v in the topological order:
  - For each vertex u such that $(v, u) \in \mathcal{E}$: if $(dist(u) < dist(v) + 1)$, assign $dist(u) = dist(v) + 1$.

Using this idea, we specify the following set of graph-based modules.

Experiments

We design a series of experiments to evaluate the efficiency of our algorithms as well as the quality of the solutions they output. All runtime was measured on a General Purpose Linux Computer running Ubuntu 18.04 LTS with 4 Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz processors.

We consider the performance of the algorithms on 4 different maps from the MAPF benchmark suite (Stern et al. 2019), with 6 agent group sizes per map. The four maps are random-32-32-20, den312, warehouse-10-20-10-2-1, and empty-32-32, which we refer to as random, game,

Module 4: Graph-based Modules

Function CYCLEDETECTION($\mathcal{G}, (u, v)$)
  return CYCLEDETECTION($\mathcal{G}, (u, v)$) from Modules 3;

Auxillary information $\mathcal{X}$ is a map $\mathcal{X} : \mathcal{A} \to [0 : zi]$, which records the index of the most recently satisfied vertex for each agent;
Define $\mathcal{X}_{\text{init}}[i] = 0$ for all $i \in \mathcal{A}$;

Function TERMINATE($\mathcal{G}, \mathcal{X}$)
  if $\forall i \in \mathcal{A}, \mathcal{X}[i] = zi$ **then**
    return true;
  return false;

Function BRANCH($\mathcal{G}, \mathcal{X}$)
  Run Procedure 1 on $\mathcal{G}$ with the following modifications:
  - Change line 4 of $\text{INIT}_{\text{EXEC}}$ to:
    Mark all vertices in $\mathcal{V}_0 = \{v_k^i : i \in \mathcal{A}, k \le \mathcal{X}[i]\}$ as satisfied;
  - Change line 20 of EXEC to:
    if $v \leftarrow \text{STEP}_{\text{EXEC}}(\mathcal{G}, i) \neq \text{NULL}$ **and** $(\exists e \in \mathcal{S}_{\mathcal{E}2} : e := (v, u)$ or $(u, v))$ **then**
    return $(cost, \mathcal{X}' := \{i \mapsto \text{STEP}_{\text{EXEC}}(\mathcal{G}, i)\}, e)$;
    else Add v into $\mathcal{S}$;
  - Change line 24 of EXEC to:
    return NULL;
  return the output of modified Procedure 1;

Function HEURISTIC($\mathcal{G}, \mathcal{X}$)
  Define $\mathcal{G}'$ to be a reduced standard TPG containing only non-switchable edges of $\mathcal{G}$;
  Run Procedure 1 on $\mathcal{G}'$ with the following modification:
  - Change line 4 of $\text{INIT}_{\text{EXEC}}$ to:
    Mark all vertices in $\mathcal{V}_0 = \{v_k^i : i \in \mathcal{A}, k \le \mathcal{X}[i]\}$ as satisfied;
    return NULL;
  return the output of modified Procedure 1;

warehouse, and empty, respectively. An visual illustration of them can be found in Figure 1. Regarding each map and group size configuration, we run the algorithms on 5 different scenarios (start/goal locations) with 5 trials per scenario. We set a runtime limit of 90 seconds for each trial.

In the experiments, we simulate the movement of the agents following an optimal path initially planned by the bounded-suboptimal MAPF solver EECBS (Li, Ruml, and Koenig 2021) with the suboptimal bound set to 1.0. Each move during the simulation is subject to a constant probability of delay. When a delay happens, we pause the movement of the delayed agent for a fixed duration and then initiate the re-planning using the post-delay locations of the agents as their starts. In the following comparisons, the delay probability is set to 3%, and the delay duration is 20 steps.

Efficiency

In Figure 1a, we visualize the runtime of the three algorithms: SES, SES-C, and, SES-B. We denote the number of agents at the X-axis and the runtime of the algorithms in the unit of seconds at the Y-axis. The lines on the figures represent the average runtime of an algorithm on a configuration, and the bars represent the standard error across trials.

Figure 1a shows that under most configurations, SES runs 2 to 3 times faster than SES-C, and 3 to 4 times faster than SES-B. The decrease in runtime is mostly due to the cycle detection mechanism and the improved heuristic, manifested by Figure 2. Figure 2 presents the detailed runtime breakdown of SES-B, SES-C, and SES. We partition the runtime into the fragments spent on the heuristic, cycle-detection, and other operations.

Comparing the first two bars in Figure 2, we see that although SES-C is using the same heuristic as SES-B, the total amount of time spent by the heuristic decreases. This is due to the fact that the cycle detection mechanism prunes the deadlock nodes before we spend time running the heuristic on them. the improved heuristic in SES takes dramatically less time than the heuristic in SES-C. And comparing the last bar with the previous two, it is obvious that the improved heuristic in SES takes dramatically less time than the heuristic in SES-C or SES-B.

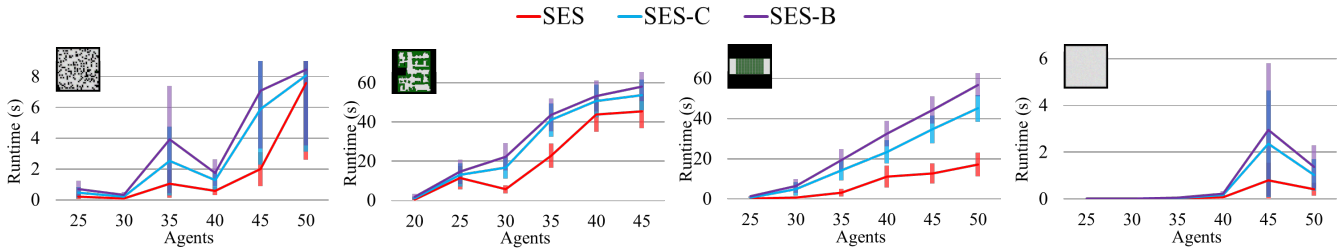
As stated in the experiment setup section, we set a runtime limit of 90 seconds when measuring the efficiency. When the algorithm terminates before the timeout, we denote the trial as a success. While the success rate of all three algorithms are close to 100% on the random, warehouse, and empty maps, we observe much higher failure rate on the more complicated game map. In Figure 3, we visualize the success rate of SES-B, SES-C, SES on the game map across 6 different agent counts. The success rate decreases as the agent count increases, and we see the largest gaps between the success rate of the three algorithms when running on 35 agents.

Solution Quality Improvement

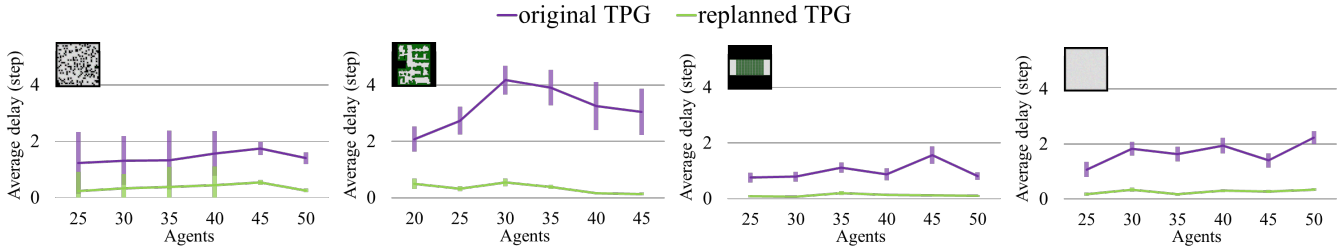
To measure the improvement of our re-planned solution in comparison to the original non-replanned solution, in Figure 1b. we plot the average delay of a solution defined by the following equation:

$$average\ delay = \frac{\sum_{i \in A} \{time(v_1^i, v_{-1}^i) - dist(v_1^i, v_{-1}^i)\}}{|A|}$$

where $time(v_1^i, v_{-1}^i)$ is the travel time of agent i moving from its current location to its goal location when replanning starts, and $dist(v_i^i, v_{-1}^i)$ is the travel distance from its current location to its goal location when replanning starts. Because all three algorithms we proposed are guaranteed to output the same optimal solution along the original path, we only compare one of them with the non-replanned solution. The average delays of the original solution and the re-planned solution are visualized in Figure 1b, denoted as



(a) Runtime in seconds.



(b) Average delay in timesteps.

Figure 1: Runtime and average delay on random, game, warehouse, and empty maps.

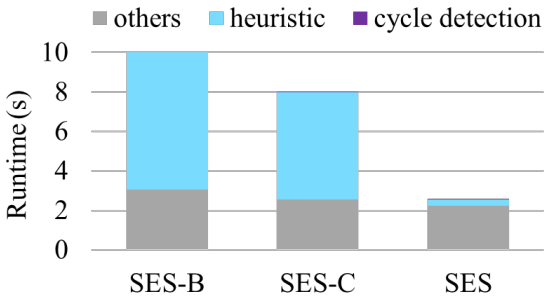


Figure 2: The breakdown of the average runtime in seconds for SES-B, SES-C, and SES, running on the random map with 50 agents, with delay probability = 3 and delay duration = 20. The runtimes of cycle detection (which can hardly been seen in the figure) are 0.000, 0.028, and 0.024 seconds respectively. Timeout trials are ignored for the purpose of this figure.

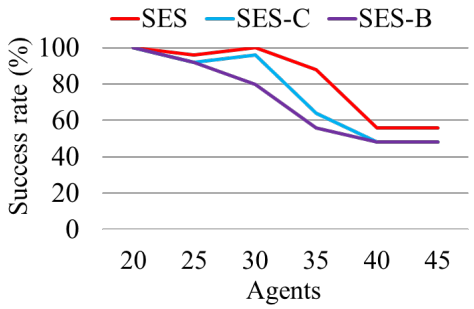


Figure 3: The success rate in percentage for SES-B, SES-C, and SES, running on the game map with delay probability = 3 and delay duration = 20.

	p = 3%	p = 10%	p = 25%
duration=3	7.58	15.38	15.51
duration=10	9.10	17.13	17.71
duration=20	7.57	21.76	21.87

Table 1: The runtime in seconds of SES running on the random map with 50 agents.

	p = 3%	p = 10%	p = 25%
duration=3	0.36	0.34	0.10
duration=10	4.27	3.80	4.62
duration=20	14.01	14.34	13.36

Table 2: The percentage of decrease of the solution cost replanned by SES in comparison to the original solution on the random map with 50 agents.

original TPG and replanned TPG, respectively. The lines on the figures represent the mean of the average delay on a configuration, and the bars represent the standard error across trials.

Figure 1b shows that while the average delay of the non-replanned solutions vary on different maps, the replanned solutions are almost always able to keep the average delay close to 0. In another word, following our replanned solution, the amount of timestep an agent have to wait between two moves is on average close to 0.

Delay Configurations

Lastly, in Table 1 and Table 2, we evaluate the performance of SES across different delay probabilities (p) and duration.

In general, the runtime increases as the delay probability increases. This might due to the fact that larger delay prob-

abilities are correlated with earlier replannings. In this case, agents are potentially further from their goals, thus leading to more switchable actions to explore in the search tree.

Another observation is that the replanned solutions reduce more costs as the delay duration increases. This might be a sign that our algorithm works most effectively in the situation when agents are subject to long delays.

References

Chen, Z.; Harabor, D. D.; Li, J.; and Stuckey, P. J. 2021. Symmetry Breaking for k-Robust Multi-Agent Path Finding. *ArXiv*, abs/2102.08689.

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 Coles, A. J.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, 477–485. AAAI Press.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: Bounded-Suboptimal Search for Multi-Agent Path Finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 12353–12362.

Ma, H.; Yang, J.; Cohen, L.; Kumar, T. K.; and Koenig, S. 2017. Feasibility study: Moving non-homogeneous teams in congested video game environments. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 270–272.

Morris, R.; Pasareanu, C. S.; Luckow, K. S.; Malik, W.; Ma, H.; Kumar, T. K. S.; and Koenig, S. 2016. Planning, Scheduling and Monitoring for Airport Surface Operations. In *AAAI Workshop on Planning for Hybrid Systems*, 608–614.

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 International Symposium on Combinatorial Search (SoCS)*, 151–159.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2007. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1752–1760.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1444–1449.