Bidirectional Temporal Plan Graph: Enabling Switchable Passing Orders for More Efficient Multi-Agent Path Finding Plan Execution

Yifan Su, Rishi Veerapaneni, Jiaoyang Li

Carnegie Mellon University
{yifansu,rveerapa}@andrew.cmu.edu, jiaoyangli@cmu.edu

Abstract

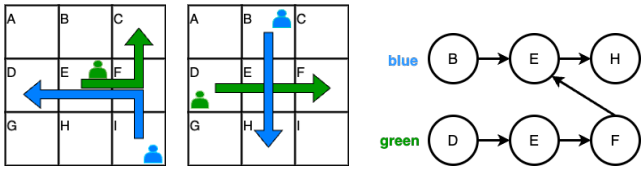
The Multi-Agent Path Finding (MAPF) problem involves planning collision-free paths for multiple agents in a shared environment. The majority of MAPF solvers rely on the assumption that an agent can arrive at a specific location at a specific timestep. However, real-world execution uncertainties can cause agents to deviate from this assumption, leading to collisions and deadlocks. Prior research solves this problem by having agents follow a Temporal Plan Graph (TPG), enforcing a consistent passing order at every location as defined in the MAPF plan. However, we show that TPGs are overly strict because, in some circumstances, satisfying the passing order requires agents to wait unnecessarily, leading to longer execution time. To overcome this issue, we introduce a new graphical representation called a Bidirectional Temporal Plan Graph (BTPG), which allows switching passing orders during execution to avoid unnecessary waiting time. We design two anytime algorithms for constructing a BTPG: BTPG-naïve and BTPG-optimized. Experimental results show that following BTPGs consistently outperforms following TPGs, reducing unnecessary waits by 8-20%.

1 Introduction

The Multi-Agent Path Finding (MAPF) problem involves finding paths for multiple agents to reach their respective destinations from their starting points in a shared environment without collisions (Stern et al. 2019). The importance of solving the MAPF problem is reflected in its wide range of applications, such as warehouse automation (Varambally, Li, and Koenig 2022), traffic management (Li et al. 2021a), and drone swarm coordination (Honig et al. 2018).

The MAPF problem discretizes time into unit timesteps, assuming that an agent can reach a particular location at a particular timestep. However, in practice, agents face challenges such as communication delays, physical constraints, or hardware failures that prevent them from meeting this assumption. When an agent is unable to reach a location required by the MAPF plan at a specific timestep, it may result in deadlocks or collisions with other agents.

To address this issue, Ma, Kumar, and Koenig (2017) proposed a MAPF with Delay Probability (MAPF-DP) model where, at each timestep, each agent has a certain probability of stopping in place, leading to delays. Both Ma, Kumar, and Koenig (2017) and Hoenig et al. (2016) show that as long as the agents adhere to the inter-agent dependencies required by the MAPF plan, they can complete their respective paths without collisions or deadlocks. These dependencies ensure that the order in which the agents pass through *every* location is consistent with the planned path. Both studies capture these dependencies by defining a graph representation, called the *Temporal Plan Graph* (TPG). Requiring agents to execute according to the dependencies in the TPG ensures completion of the MAPF plan without collisions and deadlocks, even if agents reach some locations at different timesteps than specified in the MAPF plan.



(a) Scenario 1. (b) Scenario 2. (c) TPG for Scenario 2.

Figure 1: Motivating examples. In Scenario 1 (2), the MAPF plan requries the agents to follow the arrows with the green agent passing through location F (E) before the blue agent.

In Figure 1a, the MAPF plan specifies that the green agent should pass location F before the blue agent. If they follow the corresponding TPG, the green agent will pass F first. Switching this order would lead to a deadlock, with the blue agent trying to move from F to E and the green agent from E to F. In contrast, Figure 1b shows a different scenario. Here, the green agent is supposed to pass location E first. If the green agent is delayed at D, the TPG makes the blue agent wait at B until the green agent passes E. However, if the green agent is delayed, the blue agent can safely pass E first without causing a deadlock. This flexibility in switching the order of passing can reduce unnecessary waiting and shorten the overall execution time.

The goal of this paper is to determine when the dependencies can be switched, allowing agents to pass such locations in a "first-come-first-served" manner in execution to minimize unnecessary waiting time. Our main contributions are:

1. Defining a new graphical representation called the *Bidi-*

rectional Temporal Planning Graph (BTPG) for capturing all such switchable dependencies in a MAPF plan;

2. Introducing sufficient conditions for a BTPG to be provably collision-free and deadlock-free;

3. Proposing two anytime algorithms for constructing a BTPG, namely BTPG-naïve and BTPG-optimized, and showing that following BTPGs consistently outperforms following TPGs, reducing unnecessary waits by 8-20%.

2 Preliminaries

2.1 Problem Definition

The input of the MAPF problem (Stern et al. 2019) contains an undirected graph and a set of n agents, each with a start location and a target location. Its output is a MAPF plan, which is a set of conflict-free paths for n agents that move them from their respective start locations to their respective target locations. Time is discretized into unit timesteps. At every timestep, an agent either moves to a neighboring location or waits at its current location. The ith element in an agent's path represents the location of that agent at timestep i. An agent that finishes its path rests at its target location forever. There are two ways in which the paths of two agents can create a conflict: A *vertex conflict* occurs when two agents occupy the same location at the same timestep, and a *edge conflict* occurs when two agents traverse the same edge at the same timestep.

To enable the "first-come-first-served" mechanism, we assume that, during execution, every agent has an onboard collision-avoidance mechanism that can stop automatically when there is an obstacle (such as another agent) in front of it, a common function for modern robots.

In order to reflect the execution uncertainty in real applications, we use the MAPF with Delay Probabilities (MAPF-DP) model (Ma, Kumar, and Koenig 2017). That is, during execution, an agent may get stuck at its current location for a period of time instead of following its path to move to its next location. In addition to the two types of conflicts defined above, the original MAPF-DP model also disallows *following conflicts*, where one agent enters a vertex that was occupied by another agent in the previous timestep. Their concern is that if the front agent suddenly stops, the following agent may run into the front agent. However, we allow the following conflicts in this paper for three reasons. First, the onboard collision-avoidance mechanism can easily guarantee collision-freeness in the above-mentioned situation because the following agent will stop when the front agent stops. Second, allowing following conflicts makes MAPF plans shorter and thus potentially leads to better execution times. Third, most modern MAPF planners, such as CBS-based (Sharon et al. 2015; Gange, Harabor, and Stuckey 2019; Li, Ruml, and Koenig 2021), priority-based (Erdmann and Lozano-Perez 1986; Ma et al. 2019), and rule-based (Sajid, Luna, and Bekris 2012; Okumura et al. 2019) planners allow for following conflicts, so we can directly use these off-the-shelf MAPF planners without changes.

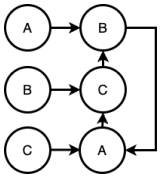


Figure 2: TPG for the scenario where 3 agents rotate in a cycle simultaneously.

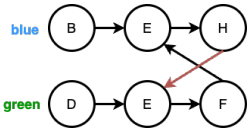


Figure 3: BTPG for the example depicted in Figure 1b.

2.2 Temporal Plan Graph (TPG)

We now introduce the definition of TPGs along with its properties established in prior research. In line with previous work, we assume that following conflicts are disallowed here. In the next section, we will demonstrate how we modify the definition and properties of TPGs to accommodate scenarios where such following conflicts are indeed allowed.

Definition 1 (TPG)**.** TPG (Ma, Kumar, and Koenig 2017) is a directed acyclic graph $G = (V, E)$. Each vertex $v_i^m \in V$ represents a state of agent m being at location $loc(v_i^m)$. Here, i indicates that $loc(v_i^m)$ is the ith element in the path of agent m. Each edge establishes a precedence dependency between two states. We divide the edges E into type-1 edges E_1 and type-2 edges E_2. A type-1 edge $(v_i^m, v_{i+1}^m) \in E_1$ forces agent m to enter $loc(v_{i+1}^m)$ only after it has entered $loc(v_i^m)$; A type-2 edge $(v_i^m, v_j^n) \in E_2$ forces agent n to enter $loc(v_j^n)$ only after agent m has entered $loc(v_i^m)$.

TPG Construction To construct a TPG from a given MAPF plan, for every agent, we introduce a vertex for every element in its path that is different from the previous one, i.e., we omit states indicating waits. We add a type-1 edge for each pair of successive elements in the path of every agent. We add a type-2 edge if one agent visits a location before another. More specifically, if there is a *conflict location* $loc(v_i^m) = loc(v_j^n)$, and agent m visits it before agent n in the MAPF plan, we introduce a type-2 edge (v_{i+1}^m, v_j^n), indicating that only after agent m left $loc(v_i^m)$ and enters its next location $loc(v_{i+1}^m)$, agent n is allowed to enter $loc(v_j^n)$. Figure 1c illustrates the TPG derived from the MAPF plan depicted in Figure 1b. For simplicity, we use $loc(v_i^m)$ to denote a vertex v_i^m. The arrow FE is a type-2 edge indicating that the green agent needs to pass through location E before the blue agent. Other arrows represent type-1 edges.

TPG Execution Policy At each timestep during execution, each agent m is allowed to move to its next state v_i^m only if, $\forall (v_j^n, v_i^m) \in E_2$, agent n has already visited state v_j^n; otherwise, it must wait (Ma, Kumar, and Koenig 2017).

Property 1 (Cycle-free $\Leftrightarrow$ valid TPG)**.** If and only if a TPG is cycle-free, TPG execution is guaranteed to succeed, i.e., agents are assured to reach their target locations without collisions or deadlocks (Berndt et al. 2020; Coskun, O'Kane, and Valtorta 2021). We refer to such a TPG as a *valid* TPG.

2.3 Related Work

The TPG execution policy asks agents to coordinate their passing orders at conflict locations, which sometimes leads to unnecessary waits. One way to reduce such unnecessary waits is to find MAPF plans that explicitly minimize coordination, such as SL-CBS (Wagner, Veerapaneni, and Likhachev 2022) and DBS (Okumura et al. 2023). However, these models usually impose overly strict coordination constraints, making them more challenging to solve than regular MAPF and resulting in longer execution times.

Another way is to generate MAPF plans that consider potential delays during planning (Ma, Kumar, and Koenig 2017; Atzmon et al. 2018), which, however, demand prior knowledge of delays and are sometimes overly conservative.

A third way is to replan, for example, by checking if a particular passing order can be switched when delays happen during execution (Berndt et al. 2020; Paul, Feng, and Li 2023; Pecora et al. 2018; Coskun, O'Kane, and Valtorta 2021). However, these methods require additional computation during execution, potentially introducing further delays.

In contrast, our proposed BTPG idea generates a MAPF plan using regular MAPF planners and post-processing it by exploring all potential switchable dependencies during the planning phase. By doing so, we avoid increasing the complexity of solving MAPF, eliminate the need for prior knowledge of delay distribution, and eradicate the necessity for additional computation during execution.

3 TPG that Allows Following

In the TPG shown in Figure 1c, suppose at timestep t, the blue and green agents are at B and E, respectively. However, at timestep $t+1$, the blue agent cannot reach E even if the green agent departs E at the same timestep. This is because the original definition of type-2 edges was predicated on disallowing following conflicts, which requires the blue agent to reach E only *after* the green agent reaches F.

However, since we allow such following actions, we propose a refined definition for type-2 edges (v_i^m, v_j^n). The revised definition specifies that agent n can enter $loc(v_j^n)$ *no earlier than* agent m enters $loc(v_i^m)$. With this adjustment, in Figure 1c, the blue agent can enter E simultaneously with the green agent departing E for F.

It is worth noting that this change is orthogonal to our proposed BTPG techniques. Our techniques are applicable to both TPGs that allow following and those that do not.

3.1 Rotation Cycle

Since agents can follow each other, they can rotate simultaneously, which was not allowed before in the original TPG definition. Figure 2 shows an example, where, at the same timestep, agent 1 goes from A to B, agent 2 goes from B to C, and agent 3 goes from C to A. This leads to a cycle $(B \to A \to C \to B)$ in the TPG but not a deadlock. We thus need to update the definition of valid TPGs.

Definition 2 (Rotation Cycle)**.** A *rotation cycle* is a cycle in a TPG consisting of only type-2 edges, with the cycle containing more than two edges. Note that if a cycle contains only two edges, it leads to a deadlock, since rotating the corresponding two agents is an edge conflict.

Property 2 (Valid TPG with rotations)**.** A TPG (that allows following) is valid if it has no non-rotation cycles.

4 Bidirectional TPG

Figure 1c depicts the TPG for Scenario 2 (Figure 1b) where the green agent enters E before the blue agent. If we switch the passing order, letting the blue agent enter E first, then we must change the type-2 edge from (F, E) to (H, E). We refer to these two type-2 edges as a bidirectional pair.

Definition 3 (Bidirectional pair)**.** A *bidirectional pair* $(e, \tilde{e})$ consists of two type-2 edges $e = (v_{i+1}^m, v_j^n)$ and $\tilde{e} = (v_{j+1}^n, v_i^m)$ with conflict location $loc(v_j^n) = loc(v_i^m)$. We refer to one such edge as the *reversed edge* of the other.

Definition 4 (BTPG)**.** A *Bidirectional TPG* (BTPG) is a TPG that contains bidirectional pairs. We use E_{pair} to represent the set of type-2 edges that are in bidirectional pairs.

BTPG Execution Policy The BTPG execution policy follows the TPG execution policy except that, for each bidirectional pair, only one edge is selected during execution based on a "first-come-first-served" manner. Specifically, either agent in the pair is allowed to enter the conflict location first. But when the first agent arrives, the type-2 edge that enables this agent to enter first is selected, while the other edge is discarded. In other words, edge (v_{i+1}^m, v_j^n) is selected when agent m reaches v_i^m, and edge (v_{j+1}^n, v_i^m) is selected when agent n reaches v_j^n. Figure 3 displays the BTPG for Scenario 2. Edges (F, E) and (H, E) are a bidirectional pair, indicating that both agents can pass through location E first. If, for example, the green agent reaches E first, then (F, E) is selected, and (H, E) is discarded.

Execution Time of TPG vs BTPG While we introduce BTPG to reduce unnecessary waits caused by certain type-2 edges in TPG, the execution time of following the BTPG policy can be longer than that of following the TPG policy in adversarial cases. We provide an example in Figure 4, even though we have not observed any such cases among the 3,900 simulations tested in our experiments.

5 Construct BTPG from TPG

To construct a BTPG, we first run a MAPF planner to obtain a MAPF plan and convert it to a valid TPG. We then check one by one whether each type-2 edge can be transformed into a bidirectional pair without resulting in an invalid BTPG. We terminate the algorithm when we reach the runtime limit or when all type-2 edges have been evaluated. We develop two such algorithms: BTPG-n and BTPG-o.

5.1 BTPG-naïve (BTPG-n)

Since exactly one edge within each bidirectional pair will be selected during execution, a BTPG with k bidirectional pairs can be conceptualized as comprising a collection of 2^k TPGs, each of which represents a different combination of edges in bidirectional pairs. Depending on the order in which the agents pass through the conflict location of each

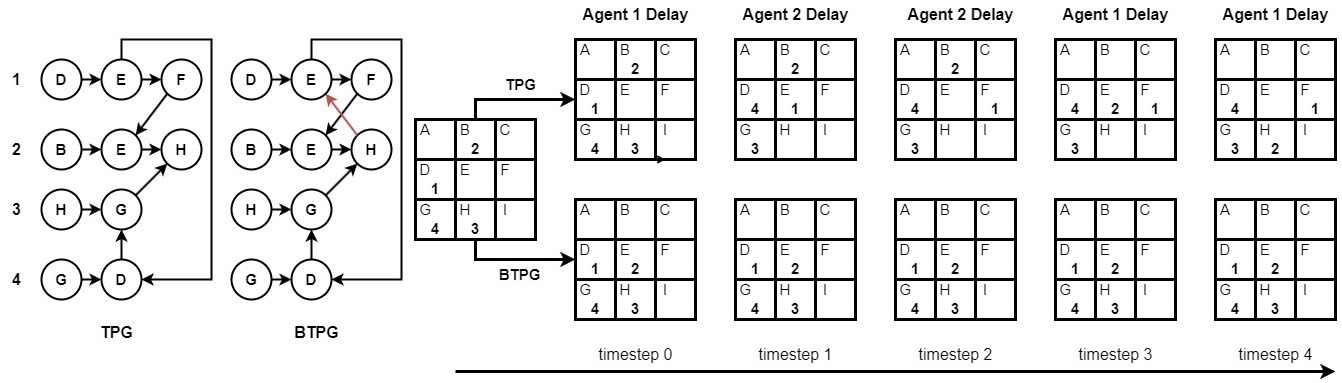


Figure 4: In our 3,900 simulations, BTPG never performed worse than TPG. Thus ideally we could prove that BTPGs are strictly superior to TPGs. However, here we show a hand-crafted adversarial example where a BTPG leads to longer execution time than a TPG under a specific set of delays.

bidirectional pair, one of these TPGs will eventually be executed. Therefore, when all 2^k TPGs are valid, namely none of them contains any non-rotation cycles (according to Property 2), the corresponding BTPG is guaranteed to be valid. Concurrently, not viewing a BTPG as 2^k TPGs but viewing it as a single graph with bidirectional pairs, a BTPG is valid if it contains only rotation cycles and self cycles, where a *self cycle* is a cycle that involves both edges in a bidirectional pair, like $E \rightarrow H \rightarrow E \rightarrow F \rightarrow E$ shown in Figure 3. A valid BTPG can have self cycles because a self cycle can never appear in one TPG as a TPG cannot contain both edges in a bidirectional pair.

Property 3 (Valid BTPG-naïve)**.** A BTPG is valid if it does not contain any Non-Rotation and Non-Self (NRNS) cycles.

Based on Property 3, our first proposed algorithm BTPG-naïve (BTPG-n) (see Algorithm 1) examines type-2 edges one by one to detect NRNS cycles and change an edge to bidireticonal pairs if no NRNS cycles are found. As $\mathcal{G}$ is guaranteed to have no NRNS cycles at the beginning of each "for" iteration (Line 5), our focus is solely on checking for NRNS cycles involving the newly added edge $\tilde{e}$ at Line 8. This verification is carried out by running a Depth First Search (DFS) (see Algorithm 2) from v_i^m to determine if we can reach v_{j+1}^n through NRNS cycles. Note that Lines 4 and 5 indicate that v_{i+1}^m is not the first state of agent m, and v_j^n is not the last state of agent n. We do not examine type-2 edges pointing from the first state of an agent because it starts at its first state, so no other agents can visit the corresponding conflict location before it. Similarly, we do not examine those pointing to the last state of an agent because it stays there without leaving, so no other agents can visit the corresponding conflict location after it.

Algorithm 2 returns true if and only if it detects a NRNS cycle. It makes two key changes to regular DFS to exclude rotation and self cycles. First, since only one edge within each bidirectional pair can be selected during execution, our DFS must avoid visiting both edges in a bidirectional pair along one DFS branch. To do so, we use E_{vis} to store the edges that our DFS has visited along the current branch and prune a child node if the reversed edge of the current type-2 edge is in E_{vis} (Lines 7 and 8). Second, when our DFS finds a cycle (Line 1), we return false if it is a rotation cycle (Lines 2 and 3) and true otherwise (Line 4). Here, E_{vis} to store the edges that our DFS has visited along the current branch.An additional implementation detail not explicitly outlined in this pseudo-code is that, to speed up our DFS, we maintain a set of visited vertices to prevent the algorithm from re-expanding the same vertex.

Algorithm 1: BTPG-naïve/optimized. The boxed code is only for BTPG-optimized.

1 MyFunctionMyFunction
Input: TPG $G = (V, E_1 \cup E_2)$
Output: BTPG $\mathcal{G}$
2 $E_{pair} \leftarrow \emptyset$; *// set of edges in bidirectional pairs*
3 **while** E_{pair} **has been updated do**
4 **for** $e = (v_{i+1}^m, v_j^n)$ ***in*** E_2 *(or* ***until*** `TimeOut`*)* **do**
5 $\tilde{e} \leftarrow (v_{j+1}^n, v_i^m)$; *// reversed edge*
6 $E_{pair} \leftarrow E_{pair} \cup \{e, \tilde{e}\}$, $E_2 \leftarrow E_2 \setminus \{e\}$;
7 $\mathcal{G} \leftarrow (V, E_1 \cup E_2 \cup E_{pair})$;
8 **if** `hasCycle`$(\mathcal{G}, v_i^m, v_{j+1}^n, \boxed{\{v_i^m\},} \{e\})$ **then**
9 $E_{pair} \leftarrow E_{pair} \setminus \{e, \tilde{e}\}$, $E_2 \leftarrow E_2 \cup \{e\}$;
10 **return** $\mathcal{G} = (V, E_1 \cup E_2 \cup E_{pair})$;

Grouping Motivated by (Berndt et al. 2020), we find two common cases in which BTPG-n cannot change the type-2 edges to bidirectional pairs shown in Figure 5. They are the cases where two agents visit the same sequence of locations in the same or reversed order, which have consecutive type-2 edges in corresponding TPGs. As adding the reserved edge of any type-2 edge can form a NRNS cycle, BTPG-n cannot change any type-2 edges in such cases to bidirectional pairs. Therefore, at Line 4 of Algorithm 1, we merge such type-2 edges into groups and examine only type-2 edges that cannot form a group to reduce the runtime.

Algorithm 2: hasCycle. The boxed code is only for BTPG-optimized.

Input: (1) BTPG $\mathcal{G} = (V, E_1 \cup E_2 \cup E_{pair})$, (2) current vertex for expansion v_i^m, (3) origin vertex v_o, and (4) set(s) of visited vertices V_{vis} and edges E_{vis} along the current DFS branch.

Output: `true` or `false`

1 **if** $v_i^m = v_o$ **then**
2 **if** $E_{vis} \subset E_2 \cup E_{pair}$ ***and*** $|E_{vis}| > 2$ **then**
3 **return** `false`; *// rotation cycle*
4 **else return** `true`;
5 **for** v_j^n ***in*** $\{v_j^n \in V \mid (v_i^m, v_j^n) \in E_1 \cup E_2 \cup E_{pair}\}$ **do**
6 $e \leftarrow (v_i^m, v_j^n)$;
7 **if** $e \in E_{pair}$ **then**
8 **if** $(v_{j+1}^n, v_{i-1}^m) \in E_{vis}$ ***or*** $\exists v_{i'}^m \in V_{vis} : i' < i$ **then** `continue`;
9 **if** *hasCycle*($\mathcal{G}$, v_j^n, v_o, $V_{vis} \cup \{v_j^n\}$, $E_{vis} \cup \{e\}$) **then return** `true`;
10 **return** `false`;

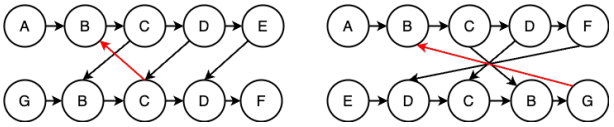


Figure 5: Two cases of grouping: The left case is when one agent follows another agent; the right one is when two agents move along the same path in opposite directions. Red arrows depict the reversed edges of edge (C, B).

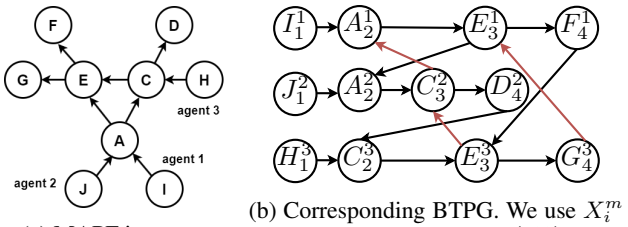
5.2 BTPG-optimized (BTPG-o)

Examining all 2^k TPGs is overly conservative as we find that not all 2^k TPGs can occur in practice. Consider the example in Figure 6. If we add bidirectional pairs for all three type-2 edges, the resulting BTPG admits a TPG with type-2 edges (E_3^3, C_3^2), (C_3^2, A_2^1), and (F_4^1, E_3^3). BTPG-n would regard this BTPG as invalid as it contains a NRNS cycle $E_3^3 \rightarrow C_3^2 \rightarrow A_2^1 \rightarrow E_3^1 \rightarrow F_4^1 \rightarrow E_3^3$.

If we analyze this TPG, it leads to a deadlock where agent 1 is at location I, awaiting agent 2 to enter location C, agent 2 is at location A, awaiting agent 3 to enter E, and agent 3 is at location C, awaiting agent 1 to enter F. However, since agent 1 and 3 are at I and C, respectively, according to the BTPG execution policy, edge (G_4^3, E_3^1) rather than edge (F_4^1, E_3^3) would be selected. Thus, this TPG would never be encountered during execution. Next, we will go through several lemmas to formally reason about such cases.

Lemma 1. When agents face a deadlock caused by a cycle in a BTPG, no agents have visited any states in the cycle.

Proof. Suppose one state v_i^m in the cycle is visited, then the edge (v_i^m, u_j^n) in the cycle that points from this state is satisfied. This indicates that the agent j can move to state u_j^n. We can similarly propagate the logic and show all states in the cycles can be visited, which is a contradiction to a deadlock. Therefore, all states in the cycle have not been visited. □



(a) MAPF instance.

(b) Corresponding BTPG. We use X_i^m to depict vertex v_i^m with $loc(v_i^m) = X$.

Figure 6: Example where some 2^k TPGs may never occur.

Lemma 2. If edge $(v_i^n, v_j^m) \in E_{pair}$ is selected during execution, then agent n has already visited state v_{i-1}^n.

Proof. This can be proved directly from the BTPG execution policy. □

Corollary 1. When agents face a deadlock caused by a cycle involving edge $(v_i^n, v_j^m) \in E_{pair}$, agent n is at state v_{i-1}^n.

Proof. From Lemma 1, agent n must be before state v_i^n. From Lemma 2, agent n is at or after state v_{i-1}^n. Thus, agent n must be at state v_{i-1}^n. □

Theorem 1. If a cycle contains a vertex v_i^n and an edge in E_{pair} that points from $v_j^n, j > i$, then this cycle will not lead to a deadlock.

Proof. Suppose that this cycle will lead to a deadlock. When the deadlock occurs, by Corollary 1, agent n is at state v_{j-1}^n. By Lemma 1, agent n has not visited state v_i^n. Thus, state v_i^n should be after state v_{j-1}^n, contradicting the assumption of $j > i$. Thus, the theorem holds. □

We denote the cycles described in Theorem 1 as *non-deadlock cycles*. The cycle in Figure 6b is a non-deadlock cycle as it contains both vertex A_2^1 and edge (F_4^1, E_3^3).

Property 4 (Valid BTPG-optimized)**.** A BTPG is valid if it does not contain any cycles apart from rotation cycles, self cycles, and non-deadlock cycles.

Therefore, our second algorithm BTPG-o extends BTPG-n by considering non-deadlock cycles. Specifically, in Algorithm 2, we use V_{vis} to record all vertices that have been visited along the current DFS branch and skip edge $(v_i^m, v_j^n) \in E_{pair}$ if the current DFS branch contains a state $v_{i'}^m, i' < i$ (Line 8). Furthermore, in Algorithm 1, we repeat the examining process of type-2 edges until no new bidirectional pair is discovered (Line 3) for the following reason.

Let us consider the example in Figure 6b. If we examine (D_4^2, C_2^3) before (F_4^1, E_3^3), then (D_4^2, C_2^3) is not added into E_{pair} because, based on the available information at that point, cycle $E_3^3 \rightarrow C_3^2 \rightarrow A_2^1 \rightarrow E_3^1 \rightarrow F_4^1 \rightarrow E_3^3$ is considered a cycle that leads to a deadlock. However,

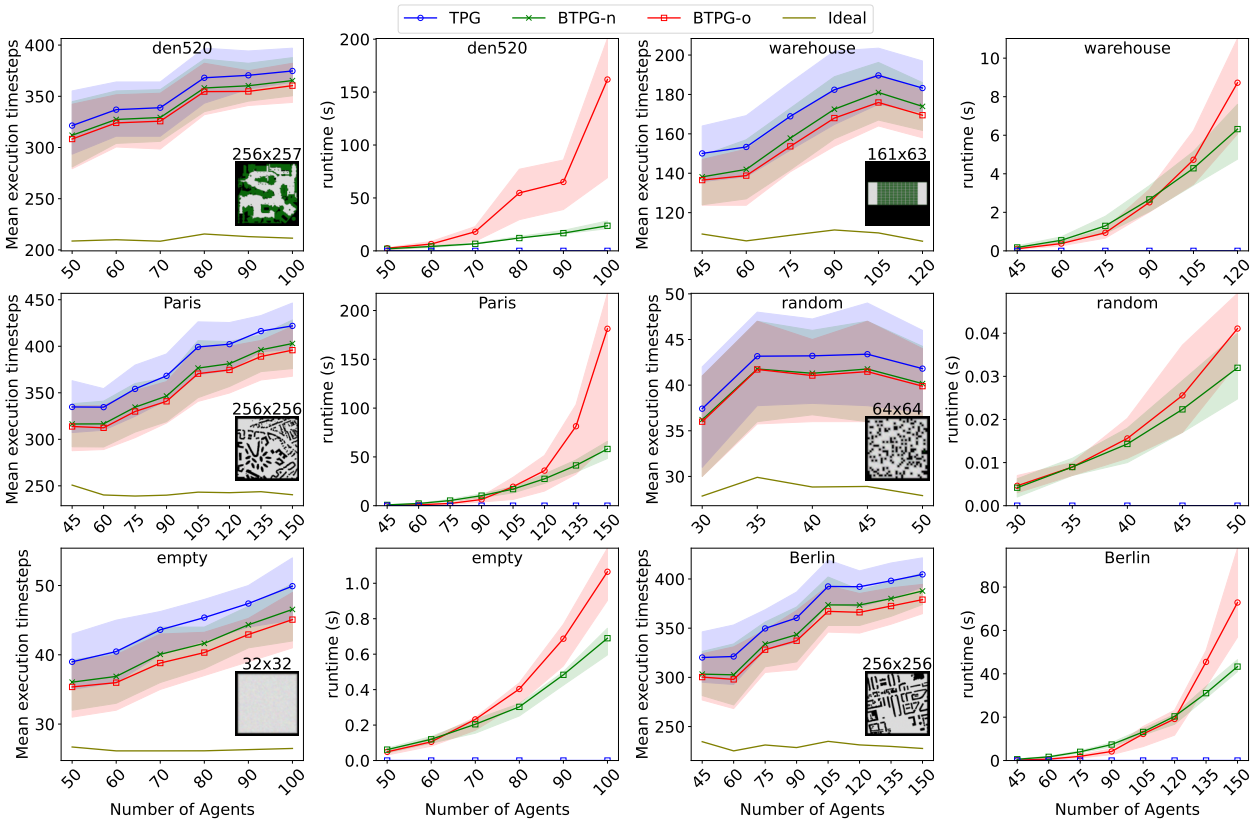


Figure 7: Performance of TPG, BTPG-naïve, and BTPG-optimized. The shaded area represents the interquartile range (25%-75%) of the data distribution. The runtime of BTPG-n and BTPG-o is the time for the two algorithms to finish finding their respective bidirectional pairs. Even though BTPG-o takes longer to finish finding bidirectional pairs, it finds around 2x more bidirectional pairs than BTPG-n, so it is actually more efficient than BTPG-n (see Figure 9).

after (F_4^1, E_3^3) is examined and added into E_{pair}, this cycle becomes a non-deadlock cycle, and thus, (D_4^2, C_2^3) and (E_3^3, C_3^2) are then added into E_{pair}. Thus, to maximize the size of E_{pair}, we repeat the examining process until no new bidirectional pair is discovered. This is unique to BTPG-o, where the validity condition becomes "easier" to satisfy as more bidirectional pairs are added since our DFS skips some edges in bidirectional pairs (see Line 8 in Algorithm 2).

6 Empirical Evaluation

Our experiments work as follows. First, we use the optimal MAPF solver CBSH2-RTC (Li et al. 2021b) to obtain the MAPF plan for each given MAPF instance. We have in total 3,900 simulations, consisting of six different benchmark maps from (Stern et al. 2019), as shown in Figure 7, each map with five to eight different numbers of agents, and each number of agents with ten random instances. The largest number of agents for each map is determined by the largest number of agents CBSH2-RTC can solve within 2 minutes. Then, we convert each MAPF plan into a TPG (for which the runtime is negligible). Next, we construct BTPG using our proposed algorithms BTPG-naive and BTPG-optimized. Last, we simulate the TPG and BTPG execution policies with 10% agents having a 30% chance of being delayed by 5 timesteps at each non-delayed timestep. Each instance is simulated with 10 random seeds, resulting in 3,900 simulations per algorithm. All algorithms were implemented in C++ [1], and all experiments were run on a PC with a 3.60 GHz Intel i7-12700KF CPU and 32 GB of RAM.

Execution Time Figure 7 reports the *mean execution timesteps*, namely the average number of timesteps that each agent takes to reach its target location during execution. We include a lower bound called *Ideal*, which is the sum of timesteps in the original MAPF plan plus the total delay timesteps of the delayed agents, then divided by the number of agents. Essentially, *Ideal* represents an overly optimistic scenario where delayed agents do not cause any additional waits for other agents. As shown, BTPG-o consistently outperforms BTPG-n, which in turn consistently outperforms TPG, across all maps and all numbers of agents.

To quantify the execution time improvement of BTPG over TPG, we define *improvement* as $\frac{T_{TPG}-T_{BTPG}}{T_{TPG}-T_{Ideal}}$, where T_{TPG}, T_{BTPG}, and T_{Ideal} are the mean execution timesteps of TPG, BTPG, and Ideal, respectively. Figure 8 shows that

[1] https://github.com/YifanSu1301/BTPG

	den520		warehouse		Paris		random		empty		Berlin	
	BTPG-n	BTPG-o	BTPG-n	BTPG-o	BTPG-n	BTPG-o	BTPG-n	BTPG-o	BTPG-n	BTPG-o	BTPG-n	BTPG-o
Mean imp.	5.9%	**8.9%**	12.1%	**17.9%**	10.6%	**14.6%**	12.9%	**15.2%**	14.5%	**20.9%**	9.6%	**14.6%**
Median imp.	5.7%	**8.1%**	12.3%	**17.8%**	10.5%	**14.2%**	10.3%	**12.2%**	14.0%	**20.0%**	9.2%	**14.2%**
Max imp.	14.0%	**19.3%**	25.0%	**35.3%**	20.7%	**25.4%**	50.0%	**63.6%**	37.5%	**42.9%**	21.3%	**24.7%**
Min imp.	1.1%	**2.9%**	2.7%	**6.7%**	4.1%	**7.0%**	0.0%	0.0%	3.3%	**7.4%**	3.2%	**7.6%**
# Type-2 edges	26,738		15,084		24,401		1,151		2,898		27,922	
# Singleton edges	2,172		1,051		2,688		146		817		2,265	
# Bi-Pairs found	663	**1,008**	368	**532**	1,254	**1,725**	60	**76**	248	**360**	1018	**1,409**
# Used Bi-Pairs	38	**68**	36	**57**	82	**125**	6	6	25	**40**	68	**112**
BTPG runtime (s)	23.72	161.88	6.31	8.72	58.12	181.33	0.03	0.04	0.69	1.06	43.32	72.87
MAPF runtime (s)	1.93		9.44		1.12		19.80		1.11		9.90	

Table 1: Statistics of BTPG-naïve/optimized. The number of agents for the six maps selected for the statistics are 100, 120, 150, 50, 100, and 150, respectively. All data in the second block are averages of 10 scenarios for each map; Bi-Pairs: bidirectional pairs; imp.: improvement; Singleton edges: type-2 edges that cannot be grouped.

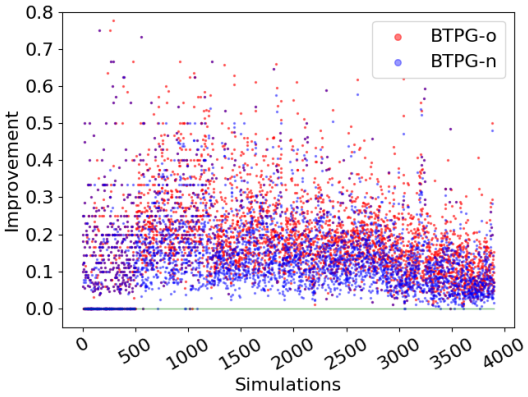


Figure 8: Improvement of BTPG over TPG per instance. We find no instances with negative improvements and 3.6%, 23.3%, 41.9%, and 31.2% of instances with no improvement, 0-10% improvement, 10-20% improvement, and >20% improvement across all simulations, respectively.

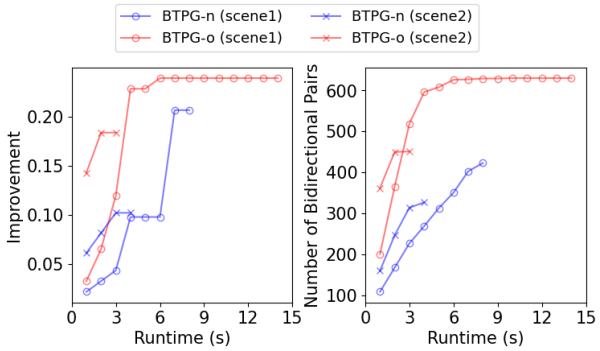


Figure 9: Anytime behavior of BTPG-n/o on warehouse with 120 agents. Scenes 1 and 2 are scenarios with the longest and shortest BTPG-o runtimes, respectively.

our improvement is always non-negative over 3,900 simulations, even though, as mentioned in Section 4, we could have negative improvements in theory. The top block of Table 1 further provides detailed numbers on the improvements over the instances with the largest number of agents for each map. The median improvement of BTPG-o is in the range of 8-20%, with maximum values around 19-64%.

Bidirectional Pairs The middle block of Table 1 reports how many bidirectional pairs that our algorithms find and that are useful. While the TPG has thousands or even ten thousands of type-2 edges, only about 10% of these edges are not grouped and have the chance to be changed to bidirectional pairs. After BTPG-o evaluates these singleton type-2 edges, about 50% are changed to bidirectional pairs. We define a bidirectional pair as *used* if the agents select the reversed edge rather than the original TPG type-2 edge during execution. We see that roughly 10% of bidirectional pairs are used. Although this is a small percentage over the total number of type-2 edges, these used bidirectional pairs are the contributors to the significant reduction in execution time that we reported above.

Runtime Both Figure 7 and the bottom block of Table 1 report the CPU runtime of our algorithms. As expected, BTPG-o is slower than BTPG-n. However, we find that the longer runtime of BTPG-o is due to finding more bidirectional pairs rather than inefficiency. Figure 9 plots the anytime behavior of both algorithms. For any given cut-off time, BTPG-o finds more bidirectional pairs and thus leads to better improvement than BTPG-n. BTPG-o is more efficient because its DFS can skip over more edges than BTPG-n.

7 Conclusion

We constructed a new graphical representation of passing orders in the MAPF plan, BTPG, by proposing the concept of bidirectional pairs. The main difference between BTPG and TPG lies in the fact that agents can switch the order of passing certain locations during execution. Two algorithms, BTPG-n and BTPG-o, are proposed to construct a BTPG. The results indicate that following BTPGs consistently outperform following TPGs by 8-20% when agents get delayed. Also, we show our proposed algorithms are anytime, and given a fixed time budget, BTPG-o outperforms BTPG-n. Overall, we convincingly show that allowing switching dependencies in a MAPF plan allows us to improve execution time without replanning.

Acknowledgments

The research is supported by the National Science Foundation (NSF) under grant number 2328671. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Bartak, R.; and Zhou, N.-F. 2018. Robust Multi-Agent Path Finding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 9, 2–9.

Berndt, A.; Duijkeren, N. V.; Palmieri, L.; and Keviczky, T. 2020. A Feedback Scheme to Reorder a Multi-Agent Execution Schedule by Persistently Optimizing a Switchable Action Dependency Graph. In *ICAPS workshop on Distributed and Multi-Agent Planning*.

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.

Erdmann, M.; and Lozano-Perez, T. 1986. On Multiple Moving Objects. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, 1419–1424.

Gange, G.; Harabor, D.; and Stuckey, P. J. 2019. Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 155–162.

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

Honig, W.; Preiss, J. A.; Kumar, T. K. S.; Sukhatme, G. S.; and Ayanian, N. 2018. Trajectory Planning for Quadrotor Swarms. *IEEE Transactions on Robotics*, 34(4): 856–869.

Li, J.; Chen, Z.; Zheng, Y.; Chan, S.-H.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2021a. Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 477–485.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; Gange, G.; and Koenig, S. 2021b. Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search. *Artificial Intelligence*, 301: 103574.

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

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with Consistent Prioritization for Multi-Agent Path Finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7643–7650.

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

Okumura, K.; Bonnet, F.; Tamura, Y.; and Défago, X. 2023. Offline Time-Independent Multiagent Path Planning. *IEEE Transactions on Robotics*, 39(4): 2720–2737.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2019. Priority Inheritance with Backtracking for Iterative Multi-Agent Path Finding. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 535–542.

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

Pecora, F.; Andreasson, H.; Mansouri, M.; and Petkov, V. 2018. A Loosely-Coupled Approach for Multi-Robot Coordination, Motion Planning and Control. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, 485–493.

Sajid, Q.; Luna, R.; and Bekris, K. 2012. Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives. In *Proceedings of the International Symposium on Combinatorial Search*, volume 3, 88–96.

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.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 151–158.

Varambally, S.; Li, J.; and Koenig, S. 2022. Which MAPF Model Works Best for Automated Warehousing? In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 190–198.

Wagner, A.; Veerapaneni, R.; and Likhachev, M. 2022. Minimizing Coordination in Multi-Agent Path Finding with Dynamic Execution. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 61–69.