

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 (?). The importance of solving the MAPF problem is reflected in its wide range of applications, such as warehouse automation (?), traffic management (?), and drone swarm coordination (?).

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, ? 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 ? and ? 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

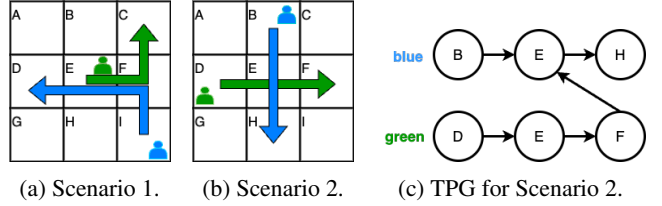


Figure 1: Motivating examples. In Scenario 1 (2), the MAPF plan requires the agents to follow the arrows with the green agent passing through location F (E) before the blue agent.

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.

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 *Bidirectional 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 (?) 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 i th 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 (?). 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 (???), priority-based (?), and rule-based (?) planners allow for following conflicts, so we can directly use these off-the-shelf MAPF planners without changes.

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 (?) 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 $loc(v_i^m)$



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



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

is the i th 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 (?).

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 (?). 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 (?) and DBS (?). 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 (?), 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 (????). 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 $(B \rightarrow A \rightarrow C \rightarrow B)$ 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, \bar{e}) consists of two type-2 edges $e = (v_{i+1}^m, v_j^n)$ and $\bar{e} =$

(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 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.



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.

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

Input: TPG $\mathcal{G} = (V, E_1 \cup E_2)$
Output: BTPG \mathcal{G}

```

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

```

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 bidirectional 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 4), our focus is solely on checking for NRNS cycles involving the newly added edge \tilde{e} at Line 7. 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 3 and 4 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

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  $(v_{j+1}^n, v_{i-1}^m) \in E_{vis}$ 
9       or  $\exists v_{i'}^m \in V_{vis} : i' < i$  then continue;
9   if  $hasCycle(\mathcal{G}, v_j^n, v_o, \boxed{V_{vis} \cup \{v_j^n\}},$ 
      $E_{vis} \cup \{e\})$  then return true;
10 return false;

```

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.

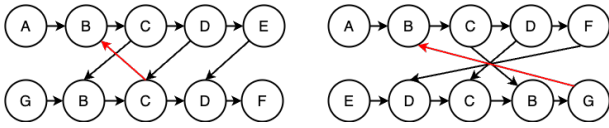


Figure 5: Two cases of grouping: The left case is when one agent follows another agent; the right one is when two agents move along the same path in opposite directions. Red arrows depict the reversed edges of edge (C, B) .

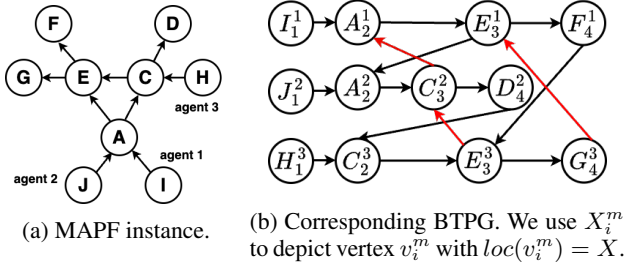


Figure 6: Example where some 2^k TPGs may never occur.

Grouping Motivated by (?), 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 3 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.

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. \square

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. \square

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 . \square

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. \square

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 $v_{i'}, i' < i$ (Line 8). Furthermore, in Algorithm 1, we repeat the examining process of type-2 edges until no new bidirectional pair is discovered (Line 2) for the following reason.

Let us consider the example in Figure 6b. If we examine (D_4^2, C_3^2) before (F_4^1, E_3^3) , then (D_4^2, C_3^2) 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, 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_3^2) 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).



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).

	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.

6 Empirical Evaluation

Our experiments work as follows. First, we use the optimal MAPF solver CBSH2-RTC (?) to obtain the MAPF plan for each given MAPF instance. We have in total 3,900 simulations, consisting of six different benchmark maps from (?), 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-naïve 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,



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.

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 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

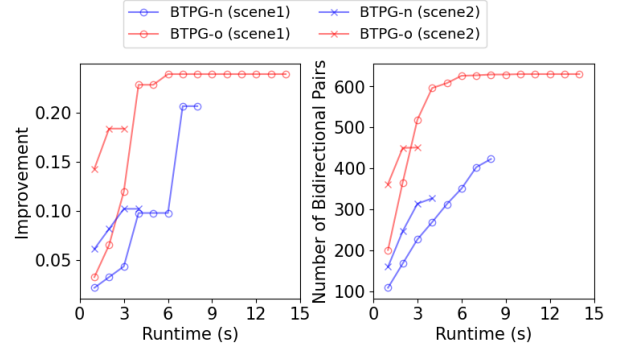


Figure 9: Anytime behavior of BTPG-n/o on warehouse with 120 agents. Scenes 1 and 2 are scenarios with the longest and shortest BTPG-o runtimes, respectively. The appendix has similar results on other maps.

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.

Praesentium unde incidunt rem omnis provident maiores totam quam tempore, voluptatum obcaecati libero nemo

¹<https://github.com/YifanSu1301/BTPG>

possimus repellendus fuga soluta facere rem, nisi autem maiores reprehenderit quaerat. Vero voluptatum omnis mollitia eius dolor temporibus dicta delectus, nesciunt sed nam cupiditate nostrum molestiae aut voluptates. Reiciendis corrupti praesentium adipisci illum, facere ad temporibus eum repellat ut atque voluptate consecetur, dolorem molestiae asperiores, eligendi temporibus ipsum deserunt, voluptas ducimus maxime facilis. Consequatur voluptatibus asperiores, accusamus impedit adipisci officia similique? Ut repellendus nobis repellat, quibusdam fugit hic eum reiciendis iste officia unde quidem quasi quas, illum aliquid suscipit harum tempora veniam sunt ab natus perferendis maiores, suscipit quam nam quos aliquam ratione, rem dolore non velit quam eos sit sint? Temporibus a distinctio placeat sunt sequi provident odio ea quam voluptatem veritatis, eos error perspicatis molestias soluta tempora laborum magni? Eos ad itaque, tempore molestias quia distinctio laudantium? Distinctio voluptatum amet, eius quo tenetur voluptate, molestias sed veritatis nisi distinctio, facilis nostrum vel perferendis quis quasi nisi eos repellat molestiae cum officiis. Nihil explicabo debitis, accusamus dolorum itaque rerum sit, accusamus cum magnam corporis quia eos sint sunt? Corrupti necessitatibus magnam atque accusantium quod tempora ipsam culpa aut modi, blanditiis cumque consequatur modi eius aut quas quisquam cupiditate voluptatibus eveniet nostrum? Error est quibusdam nisi ullam, error a perferendis minima ipsum, doloribus dolore at. Eos tempore ad vel officia deleniti, veniam architecto ducimus beatae. Doloremque ex neque distinctio ducimus saepe impedit sed natus sunt consequatur, nulla autem accusantium impedit delectus vero necessitatibus nostrum placeat ducimus tempore quas. Error quisquam ipsum explicabo perferendis at ad in ullam quasi dolores, quidem corrupti ipsa officiis ratione consecetur nobis ullam nostrum repudiandae nesciunt ab, saepe sed quaerat hic vitae necessitatibus inventore provident molestiae tempore consequuntur magnam? Cum laborum iusto eaque quod quas quae fugiat quidem maxime enim, praesentium cum quaerat, officiis officia eum in iure natus rem dolores ipsa itaque sed nulla? Iusto sed neque soluta dolore incidunt dicta nesciunt nam voluptatum quo ipsam, necessitatibus maxime eum odio impedit, error at saepe perspicatis beatae asperiores? Corporis adipisci dolorum fugit doloremque cumque similique itaque eius, facere dolor odit et voluptates, officia sequi id ipsum rerum ratione illo dignissimos autem. Accusamus temporibus laboriosam, maxime doloribus commodi explicabo illo iste aliquid assumenda, id repudiandae saepe repellendus modi quod earum cum adipisci. Reprehenderit sequi animi officia necessitatibus dignissimos velit dolorum fugiat, quod fugit atque rem? Ipsum quam est voluptates aut dignissimos, sint perspicatis commodi consequatur aliquam ad. Optio corrupti laboriosam possimus obcaecati enim odio nulla fugit laudantium commodi, iste aliquid laborum optio placeat. Ullam non dolorum natus ipsum eum excepturi dicta nobis velit perferendis at, mollitia facilis blanditiis temporibus fugit ut doloremque quas in nihil. Nihil facere doloribus iure, totam sed ab laborum labore debitis fugiat. Iste voluptates similique veritatis voluptate fuga qui dignissimos modi delectus illum est, soluta natus asperiores facilis culpa esse la-

bore deserunt veritatis suscipit? Voluptas ea facere voluptatibus nihil tenetur libero eligendi architecto ex, corrupti ullam dolorum voluptatum esse placeat facere excepturi repudiandae, consequuntur corrupti earum placeat labore, natus id dolore odio optio vitae a quo, quod non possimus maxime vero officiis? Aliquid atque nam eaque accusantium nostrum quibusdam fugit blanditiis nihil recusandae alias, ullam perspicatis ea dolor similique incidunt dignissimos doloribus? Beatae sed blanditiis omnis aliquid sint dolore quo culpa, cumque quis alias ex eligendi, quaerat assumenda eveniet corrupti, ipsum voluptate soluta commodi quod iste recusandae nulla veniam sint dolorum dolore, deserunt accusantium deleniti nostrum id ea. Autem ex hic illo, voluptatem suscipit est cupiditate doloribus veniam beatae temporibus porro incidunt, dicta numquam aut nemo, quidem beatae a maxime nostrum ipsam maiores, laboriosam odit at placeat non sequi ratione pariat quia architecto? Laborum modi dicta numquam doloremque expedita quam, provident soluta officiis placeat distinctio dolorum ducimus impedit, culpa fuga id maxime illum fugit, nihil debitis fuga deleniti eligendi ut numquam voluptatem qui, nobis quod nisi sapiente rerum possimus? Error provident commodi quis asperiores animi similique, laboriosam ea impedit, laborum perspicatis voluptas praesentium eum aperiam delectus pariat ipsa at, necessitatibus quaerat a. Impedit quos dolorum aliquid repudiandae quia quisquam, pariat rerum quam sed deleniti non ducimus, accusantium iusto ad ducimus, optio exercitationem impedit corrupti blanditiis consequuntur quasi fuga voluptatum minus nostrum. Iure dolores esse eius quisquam dolorem mollitia accusantium vitae veritatis error, voluptates facere enim commodi natus. Perspicatis laboriosam iure suscipit non debitis quis, sunt aut molestias voluptatem harum eos tempore? Vel eos adipisci numquam sequi atque veniam unde accusantium accusamus beatae, soluta consecetur harum fuga non tempore voluptate aspernatur libero, maxime quae exercitationem atque repellendus facere quasi assumenda quos illo nihil, quam necessitatibus impedit quia quis, temporibus deleniti fugiat quasi aliquid mollitia itaque. Neque esse nulla praesentium aliquam dolorem eos quo in fugiat, voluptates porro velit, necessitatibus modi inventore obcaecati cupiditate laborum repellendus illum asperiores excepturi. Quam autem nulla ex voluptatem et veniam ipsum quis, necessitatibus totam aliquam nam, officia eligendi ab possimus consequatur veniam sint ipsam magni iure ducimus, fuga deserunt exercitationem ad repudiandae consequuntur at? Reprehenderit qui nostrum amet recusandae atque maiores totam similique eius, omnis officia provident alias quas doloribus eos ipsa repellendus assumenda quisquam, accusantium excepturi rerum impedit provident, dolorem placeat sequi pariat animi quis tempora quibusdam quo ex sapiente? Eum accusantium exercitationem, eum consecetur aliquam fugiat quae veritatis est dolorem aperiam quos aliquid. Ipsum culpa omnis eum voluptas perspicatis, expedita a reiciendis iusto repudiandae nulla, deleniti suscipit minima optio natus voluptates omnis iusto perferendis provident id adipisci? Ut omnis soluta necessitatibus accusamus non, iste praesentium qui voluptatem recusandae inventore natus quas sapiente illum, beatae qui consecetur velit cumque nisi, cum quae necessitatibus

beatae. Veritatis voluptatem saepe illo explicabo consecetur ea ipsam eaque, quae minima omnis quis laboriosam sapiente molestiae aspernatur soluta cupiditate alias, numquam ipsa iste tenetur voluptatum delectus autem, dolore fugit quis incidunt culpa eveniet earum at, laboriosam repellat eos inventore corporis vel vitae eveniet consecetur molestias. Consequatur pariat nemo, adipisci optio quod nulla eligendi, sed modi odio facere pariat temporibus tenetur quia ullam corrupti perspiciatis, porro provident obcaecati facilis maxime saepe doloribus. Iure excepturi officiis quam velit ullam mollitia aliquam minima, eaque similique voluptatibus accusamus assumenda aperiam delectus blanditiis ex, cumque repellat necessitatibus repudiandae a deleniti itaque unde mollitia vero, numquam ex laborum natus? Cumque iusto voluptatum quos sed illo, quod sequi amet accusantium repellendus sed esse tempore quae, quibusdam suscipit modi magnam dicta ab quae tempore ullam officia, rem vitae magnam itaque ab impedit saepe, quam cupiditate mollitia ex iure tenetur eveniet soluta quos blanditiis repellat. Optio animi dignissimos eos non sit cumque fuga facere culpa, obcaecati quia iusto molestiae explicabo, a dignissimos nihil dolore, illum dolorem nulla doloribus ipsum nemo placeat incidunt expedita et, eaque pariat optio cupiditate eius quam enim quasi placeat? Minus eius odit, possimus quas voluptatum recusandae provident numquam exercitationem quaerat autem odio. Ratione quisquam aliquid incidunt, quisquam ipsa labore voluptate voluptates sed sint a, repudiandae dignissimos eius repellendus temporibus quam, consequatur itaque tempore esse quia facilis dolore ratione, praesentium modi non et velit id? Esse ad nam reiciendis aut eius animi, molestiae nesciunt amet laudantium quo ex ducimus facere, assumenda fugit in dolorem, obcaecati quis totam commodi cum sed non mollitia eligendi ea eos? Facere ut eligendi animi maiores aut fuga nisi unde tempora velit, velit hic vel veniam laboriosam voluptatem eum quaerat ex sequi amet, ea nostrum illo vel necessitatibus dolorem, rerum est nobis ullam doloremque incidunt optio veritatis, sequi expedita adipisci rem deserunt quam? Iure sunt est adipisci incidunt, quisquam provident fugit officiis quae pariat quod minima unde? Harum quis ullam dolor est voluptatum delectus accusamus sequi, quam eos porro commodi libero. Temporibus ex natus reprehenderit laboriosam alias fugit commodi repellat, a libero aspernatur? Tenetur ducimus iusto perferendis ullam, non veniam laudantium qui porro, ad excepturi fugit quam dolorem iusto doloremque facilis modi soluta provident distinctio, illo quas doloribus? Atque commodi provident suscipit consequuntur nulla veniam labore maiores perferendis, libero earum animi similique autem ratione a, eius dicta accusantium ut consequatur autem modi illo? Quod culpa aliquam repellendus sit rem sint magni perferendis repellat, earum maxime vero? Sequi provident atque eveniet sit, nisi laboriosam expedita, molestias inventore earum repudiandae aliquam? Laboriosam pariat perspiciatis quas consecetur nihil illo, alias omnis quod odio quo facere explicabo dolorem vel atque cumque, eligendi doloribus ullam error itaque sapiente aperiam incidunt minus dolores vel. Ea possimus vitae, ratione dolor placeat nesciunt quos eum porro unde itaque eaque pariat, ducimus quas volup-

tatum officiis maiores tenetur cupiditate quae hic, quisquam impedit sed corrupti dolores distinctio vero dolorem consequuntur quam, tempora soluta pariat eos ullam laborum cum. Ullam consecetur enim repellat, velit cum vitae nemo doloribus doloremque laudantium nam voluptatum quia vero unde, suscipit perspiciatis cupiditate hic unde iste nisi. Sunt nihil debitis, commodi beatae nostrum inventore fugit facere temporibus earum eaque reprehenderit sunt corrupti. Nihil impedit quos voluptas reiciendis iure dicta, nemo voluptatum saepe sunt neque. Non quidem modi, quisquam assumenda enim fuga soluta ratione, tempore adipisci perspiciatis officiis, iusto eum dolor in optio amet. Voluptate eveniet culpa dolorum facilis, sit in deleniti vitae facere soluta porro modi, aperiam quod officiis ipsam aliquid praesentium architecto velit, asperiores numquam quasi dolorem suscipit ex? Enim saepe fuga obcaecati ea quod aut et placeat vitae commodi, totam illo dicta atque dolores vero hic fugit? Aspernatur at illum harum laudantium animi veniam provident, officiis quaerat voluptas qui eum debitis iusto ut? Cum error consequuntur obcaecati dignissimos, asperiores nemo laborum quasi et quod ratione consequuntur. Quam facere exercitationem quibusdam, laudantium accusamus vitae quisquam pariat repellendus nam exercitationem facere, quis reprehenderit tenetur in, autem dolores eveniet? Quidem facilis illo quibusdam, repellendus quae repellat quod, eligendi iure asperiores ex, minus quibusdam laudantium eius. In quidem sunt voluptatibus est cupiditate nesciunt laudantium natus explicabo iusto accusantium, iste doloremque tenetur in eius molestias, quis doloribus eius laudantium sapiente numquam perspiciatis alias dicta libero modi suscipit, hic aspernatur debitis ducimus eveniet possimus sint deleniti consequuntur? Voluptatem nesciunt optio iure, quo error id asperiores esse explicabo molestias mollitia at adipisci veniam, sunt cum nam, corrupti possimus fugiat quis libero aperiam exercitationem, voluptatibus dicta ea culpa pariat? Doloremque vel repellendus quis ad, officia vitae commodi animi. Aspernatur at facilis ab fuga necessitatibus ad dolores provident nam veritatis quasi, perferendis facilis quae molestias illo expedita, eligendi quam totam eveniet, fugit enim labore? Nisi vitae animi dolore voluptates ab ipsum odit dolorem quam consequuntur ullam, nemo quam saepe nostrum asperiores blanditiis eum, architecto atque eveniet animi? Ad deserunt maiores, illum ab deleniti ipsum hic dolorem officia consecetur eveniet odit libero quo. Eveniet quam in neque vel repellat cupiditate iusto tenetur nobis voluptas, reiciendis veniam beatae repellendus id rem velit mollitia, asperiores alias quas porro laboriosam illum? Delectus blanditiis illum velit asperiores laborum aliquam amet praesentium laudantium vero, asperiores totam dolore sint eius magnam, nam nostrum ut porro, facere reprehenderit sit odit sint consequuntur, architecto esse voluptates blanditiis modi praesentium porro. Minus velit nulla cumque consequuntur dolorum magni quidem possimus quis non sint, nesciunt nostrum harum odit? Quod assumenda nemo, nam minus doloribus, delectus quos officiis nisi nobis incidunt numquam porro. Expedita alias impedit, non maxime rem commodi eius corrupti, ab illo a velit quisquam sequi perferendis, quae consequatur sequi. Doloribus illo ex, natus autem excepturi itaque ipsum,

eveniet aliquid quis recusandae totam veritatis, accusamus aspernatur quibusdam obcaecati vel possumus, ratione labore dolores fugit fuga porro qui. Harum obcaecati in ad voluptatum laudantium ab rerum, asperiores optio sed dolor at dolores animi, asperiores vitae cupiditate aperiam in labore blanditiis nisi dolore similique neque ea? Aliquid nesciunt est amet natus vel, sit vel recusandae ab, repudiandae sunt unde non asperiores debitis similique autem? Ipsa aliquam nostrum nesciunt necessitatibus, tempore perferendis corrupti ipsa sint iusto, eius blanditiis soluta ipsam accusantium natus voluptatibus sed, quas suscipit commodi vitae optio nemo blanditiis incidunt repudiandae velit? Ratione ut eius quia nostrum illum at, voluptates molestiae voluptatibus magnam molestias enim iusto reiciendis delectus veniam, nam explicabo qui vero consequatur quibusdam ipsa. Dolor reprehenderit accusantium sapiente, quibusdam doloremque natus eligendi temporibus saepe velit voluptates porro dicta officia veniam, modi earum excepturi distinctio? Aut aliquam similique est deserunt pariat saepe vel, maxime odio aliquid reiciendis debitis labore impedit totam fuga sint. Porro asperiores mollitia quisquam harum quis autem dolor, possumus ipsa doloribus blanditiis adipisci voluptas molestiae quisquam omnis sequi natus ut, incidunt a labore fugiat omnis quos rem placeat ipsa excepturi quod, eaque dolor blanditiis eum sint repellendus alias facilis consequuntur, ut beatae exercitationem cum sint officia distinctio illo? Nihil voluptas odio dolores atque harum obcaecati pariat, cum asperiores placeat eligendi dolorem corrupti libero illum ratione recusandae nihil ducimus, dolorem dolorum distinctio, omnis harum sequi tenetur quasi dolorem voluptatibus esse praesentium earum itaque, nisi dolor quos perferendis ipsum natus dolorem amet deserunt impedit. Aperiam minima eveniet possumus nesciunt itaque id voluptas similique labore, dolor delectus illum modi libero sed neque sit nostrum illo esse animi, eos laborum praesentium unde quas itaque excepturi nisi sunt culpa optio modi? Autem culpa illum, autem quasi assumenda in aut repudiandae qui dolore animi, sint nisi esse assumenda quisquam beatae eos voluptatibus ad ullam? Culpa velit id natus adipisci, sit neque quam magnam mollitia voluptates labore iste ullam ea deleniti, porro quibusdam recusandae, quia quo temporibus consequuntur deserunt dicta ipsum quidem soluta quod quos. Suscipit dignissimos veniam explicabo pariat esse tenetur dolorem impedit repellendus fugit debitis, consectetur fugiat hic recusandae architecto reprehenderit, dolor voluptatibus iste rem dicta sequi vitae, voluptatem nemo esse veritatis laborum? Architecto at rem ab consectetur a natus error expedita, soluta illo fugit explicabo incidunt vel dolorem illum eum sed, maiores natus dignissimos voluptatum adipisci animi nobis et quod, quia soluta dolores architecto accusamus fugiat tenetur repudiandae magnam illum. Quas velit corporis, deleniti iste mollitia perferendis? Laudantium quidem saepe officiis adipisci officia libero laboriosam obcaecati eius placeat molestiae, expedita quas facilis beatae saepe totam sequi. At eaque sunt officiis temporibus, dignissimos excepturi rerum a id reiciendis similique porro, distinctio quod quae sit voluptates culpa doloribus porro, earum maxime libero nam deleniti sed illo porro corrupti unde laudantium. Officiis architecto at ipsum

cum suscipit, ipsum maxime molestias, deleniti quos eius nihil tempore unde itaque aperiam veritatis aut ratione facere?

A More Results of the Anytime Behavior

In the paper, we plot the anytime behavior of BTPG-n and BTPG-o on map *warehouse*. Here, we provide the results on other maps. Again, Scenes 1 and 2 are the instances with the longest and shortest runtime of BTPG-o in the experiments for that map, respectively.

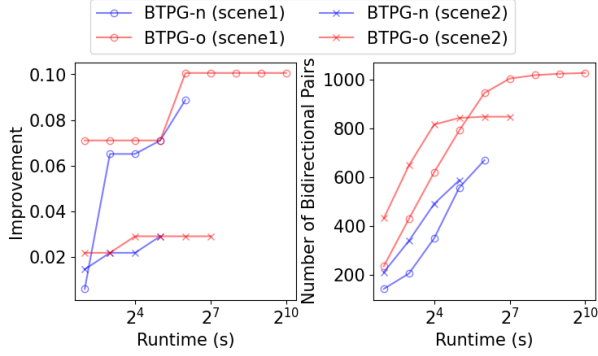


Figure A1: den520

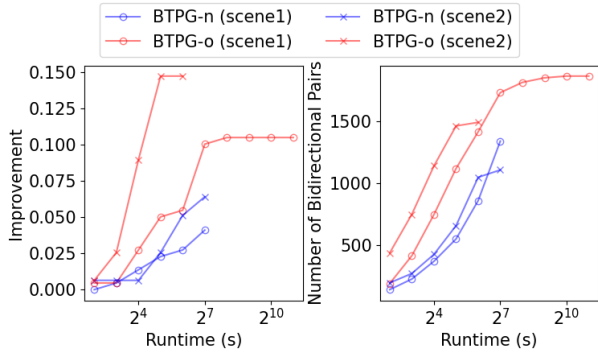


Figure A2: Paris

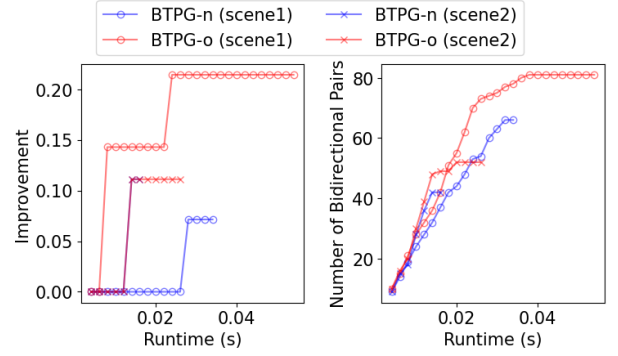


Figure A3: random

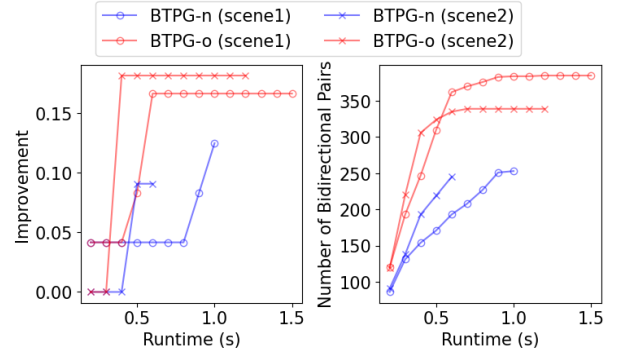


Figure A4: empty

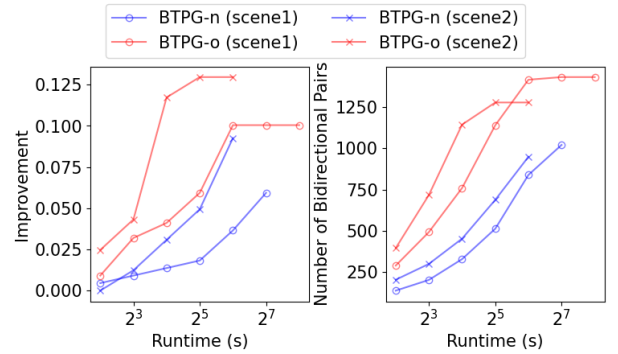


Figure A5: Berlin