Speedup Techniques for Switchable Temporal Plan Graph Optimization

He Jiang, Muhan Lin, Jiaoyang Li

Carnegie Mellon University
{hejiangrivers, muhanlin, jiaoyangli}@cmu.edu

Abstract

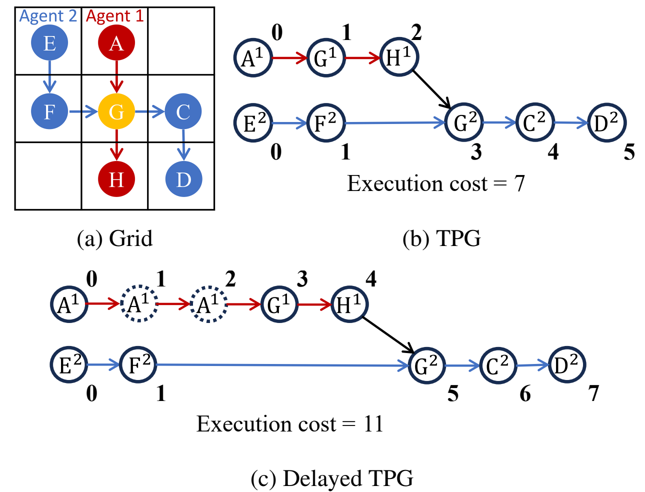
Multi-Agent Path Finding (MAPF) focuses on planning collision-free paths for multiple agents. However, during the execution of a MAPF plan, agents may encounter unexpected delays, which can lead to inefficiencies, deadlocks, or even collisions. To address these issues, the Switchable Temporal Plan Graph provides a framework for finding an acyclic Temporal Plan Graph with the minimum execution cost under delays, ensuring deadlock- and collision-free execution. Unfortunately, existing optimal algorithms, such as Mixed Integer Linear Programming and Graph-Based Switchable Edge Search (GSES), are often too slow for practical use. This paper introduces Improved GSES, which significantly accelerates GSES through four speedup techniques: stronger admissible heuristics, edge grouping, prioritized branching, and incremental implementation. Experiments conducted on four different map types with varying numbers of agents demonstrate that Improved GSES consistently achieves over twice the success rate of GSES and delivers up to a 30-fold speedup on instances where both methods successfully find solutions.

1 Introduction

Multi-Agent Path Finding (MAPF) (Stern et al. 2019) focuses on planning collision-free paths for multiple agents to navigate from their starting locations to destinations. However, during execution, unpredictable delays can arise due to mechanical differences, accidental events, or sim-to-real gaps. For instance, in autonomous vehicle systems, a pedestrian crossing might force a vehicle to stop, delaying its movements and potentially causing subsequent vehicles to halt as well. If such delays are not managed properly, they can lead to inefficiencies, deadlocks, or even collisions.

To address these issues, the Temporal Plan Graph (TPG) framework was first introduced to ensure deadlock- and collision-free execution (Hönig et al. 2016; Ma, Kumar, and Koenig 2017). TPG encodes and enforces the order in which agents visit the same location using a directed acyclic graph, where directed edges represent precedence. However, the strict precedence constraints often lead to unnecessary waiting. For example, as shown in Figure 1c, a delay by Agent 1 causes Agent 2 to wait unnecessarily at vertex F^2 because TPG enforces the original order of visiting G. In such cases,

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.



(a) Grid (b) TPG

(c) Delayed TPG

Figure 1: (a) shows an example of a MAPF problem. In the initial MAPF plan, Agent 1 moves from A to H (red arrows) and visits G first. Agent 2 moves from E to D (blue arrows) and visits G after Agent 1. (b) shows the TPG of the initial plan. Each row of vertices encodes an agent's path with superscripts $1, 2$ differentiating agents. The black arrow encodes the precedence that agent 2 can only visit G^2 after agent 1 arrives H^1. The bold number near a vertex v^i shows the earliest possible time for agent i to arrive at this vertex. The execution cost below the TPG shows the overall cost of these two agents. (c) shows the TPG with a 2-timestep delay at vertex A^1, which is encoded by two dashed vertices.

switching the visiting order of these two agents would allow Agent 2 to proceed first, avoiding unnecessary delays.

To capture this idea, Berndt et al. (2024) introduced the notion of a switchable edge that can be settled in one of two directions, each representing one possible visiting order of two agents at a location. Then, they proposed the Switchable Temporal Plan Graph (STPG), a superclass of TPG that contains a set of switchable edges. By gradually settling these switchable edges, STPG allows the search for a Temporal Plan Graph (TPG) that encodes optimal visiting orders to minimize total travel time under delays. However, both Berndt's original solution based on Mixed Integer Linear Programming (MILP) and the subsequent Graph-

Based Switchable Edge Search (GSES) algorithm (Feng et al. 2024) are too slow to scale to scenarios with just 100 agents in the experiments.

This work makes a significant advancement in tackling the scalability issue of STPG. Specifically, we improve GSES with a series of speedup techniques that leverage the underlying structure of STPG. By estimating the future cost induced by currently unsettled switchable edges, we propose stronger admissible heuristics. To reduce the size of the search tree, we introduce a novel algorithm that identifies all the maximal groups of switchable edges whose directions can be determined simultaneously (Berndt et al. 2024). Additionally, we introduce two straightforward yet highly effective engineering techniques—prioritized branching and incremental implementation—to enhance the efficiency of the search. We prove the correctness of these techniques and demonstrate through experiments that our final algorithm, Improved GSES (IGSES), significantly outperforms the baseline GSES and other approaches. For instance, IGSES consistently achieves more than double the success rates of GSES and shows a 10- to 30-fold speedup in average search time on instances solved by both algorithms.

2 Related Work

Approaches for handling delays in MAPF execution can be broadly categorized into offline and online methods.

Offline approaches consider delays during planning. Robust MAPF planning methods (Atzmon et al. 2018, 2020) generate robust plans under bounded or probabilistic delay assumptions. While these methods improve robustness, they tend to produce conservative solutions and require significantly more computation than standard MAPF algorithms.

Online approaches, on the other hand, react to delays as they occur. The simplest method is to replan paths for all agents upon a delay, but this is computationally expensive. TPG (Hönig et al. 2016; Ma, Kumar, and Koenig 2017) avoids heavy replanning by adhering to the original paths and precedence constraints, but it often results in unnecessary waits due to overly strict ordering. To address this problem, Berndt et al. (2024) introduce the STPG framework to optimize the precedence by MILP. Despite its generality, the MILP approach is too slow for large-scale problems. To improve the scalability, Feng et al. (2024) propose a dedicated search algorithm, GSES, to replace the MILP. However, GSES still suffers from inefficiencies due to the large search tree and redundant computation. In contrast, Kottinger et al. (2024) proposed an alternative formulation that solves the same problem as STPG by introducing delays to agents' original plans. It constructs a graph for each agent and then applies standard MAPF algorithms to search for solutions. We refer to the optimal version of their algorithm, which applies Conflict-Based Search (CBS) (Sharon et al. 2015), as CBS with Delays (CBS-D). MILP, GSES, and CBS-D are all optimal algorithms, and we will compare our method against them in terms of planning speed in the experiments.

Different from these optimal methods, Liu et al. (2024) propose a non-optimal heuristic approach called Location Dependency Graph (LDG), which reduces waits online using a formulation similar to STPG. Bidirectional TPG (BTPG) (Su, Veerapaneni, and Li 2024) is another non-optimal approach based on TPG, but it falls into the offline category. BTPG post-processes a MAPF plan and produces an extended TPG with special bidirectional edges. These edges enable agents to switch visiting orders at certain locations in a first-come-first-served manner during execution.

3 Background

In this section, we provide the necessary definitions and background for STPG optimization. For more details, please refer to the GSES paper (Feng et al. 2024).

3.1 Preliminaries: MAPF, TPG, and STPG

Definition 1 (MAPF). *MAPF problem aims at finding collision-free paths for a team of agents indexed by $i \in \mathcal{I}$ on a given graph, where each agent i has a start location s^i and a goal location g^i. At each timestep, an agent can move to an adjacent location or wait at its current location. We disallow two types of conflicts like previous works (Feng et al. 2024):*

1. ***Vertex conflict**: two agents take the same location at the same timestep.*
2. ***Following conflict**: one agent enters a location occupied by another agent at the previous timestep.*

Definition 2 (TPG). *A Temporal Plan Graph (TPG) is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}_1, \mathcal{E}_2)$ that encodes a MAPF plan by recording its precedence of visiting locations.*

The vertex set $\mathcal{V} = \{v_p^i : p \in [0, z^i], i \in \mathcal{I}\}$ records all locations that must be visited sequentially by each agent i.[1] *Specifically, z^i is the number of locations for agent i, and each v_p^i is associated with a specific location, $loc(v_p^i)$.*

*The edge set $\mathcal{E}_1$ contains all **Type-1** edges, which encode the precedence between an agent's two consecutive vertices. A Type-1 edge (v_p^i, v_{p+1}^i) must be introduced for each pair of v_p^i and v_{p+1}^i to ensure that v_p^i must be visited before v_{p+1}^i.*

*The edge set $\mathcal{E}_2$ contains all **Type-2** edges, which specify the order of two agents visiting the same location. Exactly one Type-2 edge must be introduced for each pair of v_q^j and $v_p^i, i \neq j$, with $loc(v_q^j) = loc(v_p^i)$. We can introduce the Type-2 edge (v_{q+1}^j, v_p^i) to encode that agent i can only enter v_p^i after agent j leaves v_q^j and arrives at v_{q+1}^j, or introduce its **reversed edge** (v_{p+1}^i, v_q^j) to specify that agent j can only enter v_q^j after agent i leaves v_p^i and arrives at v_{p+1}^i.*

For example, in Figure 1b, the red and blue arrows are Type-1 edges, and the black arrow is a Type-2 edge.

When agent i moves along an edge e from v_p^i to v_{p+1}^i and suffers from a t-timestep delay, we insert t vertices along e to encode this delay (e.g., Figure 1c) so that each edge in the figures always takes 1 timestep for an agent to move.[2]

A TPG is executed in the following way. At each timestep, an agent i can move from v_p^i to v_{p+1}^i if for every edge $e =$

[1] Following the previous work, we merge consecutive vertices that represent the same location into one.

[2] In the implementation, we actually define edge costs to encode delays compactly. However, for easy understanding, we explain our algorithm by inserting extra vertices in this paper.

(v_q^j, v_{p+1}^i), agent j has visited v_q^j. The execution of a TPG is completed when all the agents arrive at their goals.

Theorem 1. *The execution of a TPG can be completed without collisions in finite time if and only if it is acyclic. (Berndt et al. 2024).*

We define **the execution cost of an acyclic TPG** as the minimum sum of travel time of all agents to complete the execution of the TPG. It can be obtained by simulation, but the following theorem provides a faster way to compute it.

Definition 3 (EAT)**.** *The **earliest arrival time (EAT)** at a vertex v_p^i is the earliest possible timestep that agent i can arrive at v_p^i in an execution of TPG.*

Theorem 2. *The EAT at a vertex v_p^i is equal to the **forward longest path length (FLPL)**, $L(v_p^i) = \max\{L(s^k, v_p^i), k \in \mathcal{I}$, if s^k is connected to $v_p^i\}$, where $L(s^k, v_p^i)$ is the longest path length from agent k's start, s^k, to v_p^i. The execution cost of an acyclic TPG is equal to the sum of all agents' EATs at their goals, $\sum_{i \in \mathcal{I}} L(g^i)$. (Feng et al. 2024).*

Since an acyclic TPG is a directed acyclic graph, we can obtain the longest path length of all vertices by first topological sort and then dynamic programming in linear-time complexity $O(|\mathcal{V}| + |\mathcal{E}_1| + |\mathcal{E}_2|)$ (Berndt et al. 2024). We provide the pseudocode in Appendix A.1.[3] For example, in Figure 1c, we annotate the earliest arrival time at each vertex near the circles. The EAT of G^2 can be computed by $\max\{L(H^1) + 1, L(F^2) + 1\} = \max\{5, 2\} = 5$. Because the EAT of two goals H^1 and D^2 are 4 and 7, the execution cost of this TPG is $4 + 7 = 11$.

Definition 4 (STPG)**.** *Given a TPG $\mathcal{G} = (\mathcal{V}, \mathcal{E}_1, \mathcal{E}_2)$, a Switchable TPG (STPG) $\mathcal{G}^S = (\mathcal{V}, \mathcal{E}_1, (\mathcal{S}, \mathcal{N}))$ partitions Type-2 edges $\mathcal{E}_2$ into two disjoint edge sets, $\mathcal{S}$ for switchable edges and $\mathcal{N}$ for non-switchable edges. Any switchable edge $e = (v_{q+1}^j, v_p^i) \in \mathcal{S}$ can be **settled** to a non-switchable edge by one of the following two operations:*

1. ***Fix** e: removes e from $\mathcal{S}$ and adds it to $\mathcal{N}$.*
2. ***Reverse** e: removes e from $\mathcal{S}$ and adds its **reversed edge** $Re(e) = (v_{p+1}^i, v_q^j)$ to $\mathcal{N}$.*

*For convenience, we also define the **direction of an edge** $e = (v_{q+1}^j, v_p^i)$ as from j to i and its reverse direction as from i to j, where i, j are the indices of agents.*

Clearly, STPG is a superclass of TPG. If all switchable edges are settled, it **produces** a TPG. The goal of **STPG Optimization** is to find an acyclic TPG with the minimum execution cost among all those produced by the STPG.

3.2 The Baseline Algorithm: GSES

We now introduce our baseline algorithm, Graph-based Switchable Edge Search (GSES) (Feng et al. 2024). We start with some useful definitions.

Definition 5 (Reduced TPG)**.** *The reduced TPG of an STPG $\mathcal{G}^S = (\mathcal{V}, \mathcal{E}_1, (\mathcal{S}, \mathcal{N}))$ is the TPG that omits all switchable edges, denoted as $Redu(\mathcal{G}^S) = (\mathcal{V}, \mathcal{E}_1, \mathcal{N})$.*

[3]The appendix is available in the extended version of the paper at: https://arxiv.org/abs/2412.15908.

Algorithm 1: Improved GSES

```
1:  function IMPROVEDGSES(G^S_init)
2:      Groups ← EDGEGROUPING(G^S_init)
3:      Node_root ← BUILDNODE(G^S_init)
4:      Push Node_root into OpenList
5:      while OpenList is not empty and still time left do
6:          Pop Node = (G^S, h-value, L) from OpenList
7:          e ← SELECTCONFLICTINGEDGE(G^S, L)
8:          if e = NULL then return FIXALL(G^S)
9:          Edges ← {e}
10:         Edges ← GETEDGEGROUP(e, Groups)
11:         G^S_child-1 ← FIX(G^S, Edges)
12:         G^S_child-2 ← REVERSE(G^S, Edges)
13:         for G^S_child ∈ {G^S_child-1, G^S_child-2} do
14:             if no cycle in Redu(G^S_child) then
15:                 Node_child ← BUILDNODE(G^S_child)
16:                 Push Node_child into OpenList
17:     return NULL // Timeout
18:
19: function BUILDNODE(G^S)
20:     Incrementally compute for all vertices the forward
        and backward longest path lengths on Redu(G^S) as
        a function L based on its parent node
21:     Estimate the future cost increase Δcost based on the
        backward longest path lengths
22:     h-value ← Σ_i L(g^i)+Δcost
23:     return (G^S, h-value, L)
24:
25: function SELECTCONFLICTINGEDGE(G^S, L)
26:     Order all switchable edges in S by certain priority
27:     for all switchable edge e ∈ S do
28:         if Sl(e) < 0 then return e
29:     return NULL
```

Then, we define the **execution cost of an STPG** to be the execution cost of its reduced TPG. Clearly, it provides a lower bound for the execution cost of any acyclic TPG that can be produced from this STPG because the execution cost will only increase with more switchable edges settled.

Definition 6 (Edge Slack)**.** *The **slack of a switchable edge** $e = (v_{q+1}^j, v_p^i)$ in an STPG is defined as $Sl(e) = L(v_p^i) - L(v_{q+1}^j) - 1$, where $L(v)$ is the EAT at vertex v in the reduced TPG.*

We say a switchable edge **conflicts** with the STPG if $Sl(e) < 0$. For example, in the Figure 2a, the switchable edge (H^1, G^2) conflicts with the STPG because its slack $Sl(H^1, G^2) = L(G^2) - L(H^1) - 1 = 2 - 4 - 1 = -3 < 0$. For a switchable edge e, $Sl(e) \geq 0$ means that the earliest execution of the reduced TPG already satisfied the constraints imposed by e, i.e., we can fix e without introducing any cycle or cost increase to the reduced TPG. Therefore, if all remaining switchable edges do not conflict, we can fix them all and obtain an acyclic TPG with the same execution cost as the current reduced TPG (Feng et al. 2024). The proof is given in Appendix A.2. This result can help the GSES algorithm, introduced next, stop the search earlier.

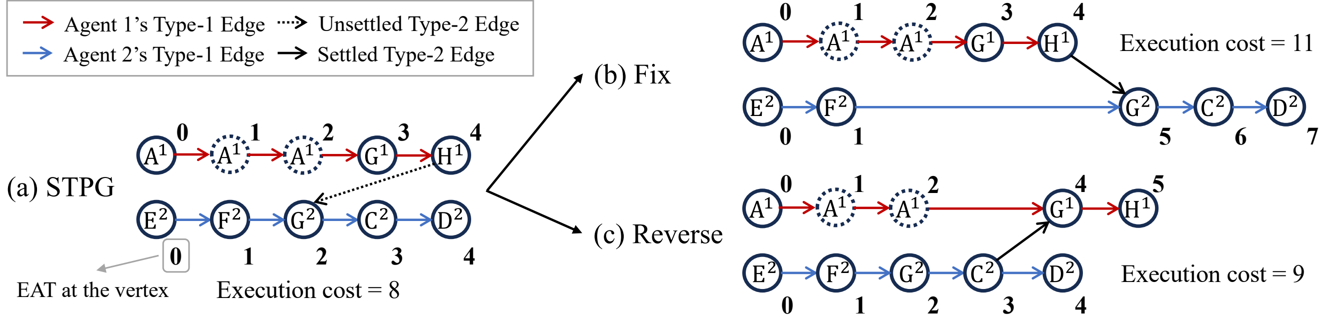


Figure 2: (a) is the STPG built from the TPG in Figure 1c. The only switchable edge (H^1, G^2) is dashed. It can be fixed to edge (H^1, G^2) as in (b) or reversed to edge (C^2, G^1) as in (c). The different choices result in different execution costs. Notably, when computing the execution cost of STPG (defined in Section 3.2), the dashed switchable edge will be ignored.

Algorithm 1 shows the pseudocode of GSES,[4] a best-first search algorithm with each search node comprising three parts: an STPG, its execution cost, and the EATs of all vertices in its reduced TPG (Line 23). The execution cost is used as the h-value in the best-first search (The g-value is always 0), and the EATs are used to find conflicting edges.

GSES starts with a root node containing the initial STPG, which is constructed from the TPG of the initial MAPF plan. Almost all Type-2 edges in the TPG can be turned into switchable edges except those pointing to goal vertices and those whose reversed edges point to start vertices (Berndt et al. 2024). For example, the STPG built from the TPG in Figure 1c is illustrated in Figure 2a.

When GSES expands a search node, it tries to find a conflicting edge (Line 7). If there is none, GSES terminates the search and returns a solution by fixing all the remaining switchable edges (Line 8). Otherwise, GSES branches the search tree based on the conflicting edge. Two child STPGs are generated: one fixes the edge (Line 11), and another one reverses it (Line 12). Before generating the child nodes and pushing them into the priority queue, GSES detects cycles in their reduced TPGs and discards cyclic ones (Line 14). The longest path lengths (Line 20) and the h-value (Line 22) are computed for acyclic ones to build child nodes.

4 Speedup Techniques

This section describes our methods to speed up the search in Algorithm 1, including constructing stronger admissible heuristics by estimating the future cost (Section 4.1), finding switchable edges that could be grouped and branched together (Section 4.2), different ways to prioritize switchable edges for branching (Section 4.3), and incremental implementation of computing longest path lengths (Section 4.4).

4.1 Stronger Admissible Heuristics

In the Line 22 of Algorithm 1, we use the execution cost of the current STPG as the admissible h-value in the GSES. However, this cost is an estimation based on the reduced TPG with currently settled edges.

[4]The red text belongs to IGSES and can be ignored for now.

Intuitively, we can obtain extra information from the unsettled switchable edges. For example, in Figure 2a, the execution cost of the STPG is 8. The switchable edge (H^1, G^2) is not settled and, thus, ignored in the computation. But we know eventually, this edge will be fixed or reversed. If we fix it (Figure 2b), then the EAT of G^2 will increase from 2 to 5 because now it must be visited after H^1, the EAT of which is 4. In this way, we can infer that the EAT of vertex D^2 will be postponed to 7, leading to an increase of 3 in the overall execution cost. If we reverse the edge, we will get a similar estimation of the future cost increase, which is 1 in Figure 2c. Then we know the overall cost will increase by at least $\min\{3, 1\} = 1$ in the future for the STPG in Figure 2a.

Before the formal reasoning, we introduce another useful concept, vertex slack, for easier understanding later.

Definition 7 (Vertex Slack)**.** *The **slack of a vertex** v_p^i to an agent j's goal location g^j in an STPG is defined as $Sl(v_p^i, g^j) = L(g^j) - L(v_p^i) - L(v_p^i, g^j)$, if v_p^i is connected to g^j in the reduced TPG. $L(v)$ is the EAT at vertex v, which is also the forward longest path length. $L(v_p^i, g^j)$ means the longest path length from v_p^i to g^j and we call it the **backward longest path length (BLPL)**.*

This slack term measures the maximum amount that we can increase the EAT at v_p^i (i.e., $L(v_p^i)$) without increasing the agent j's EAT at its goal (i.e., $L(g^j)$). In other words, $L(v_p^i) + Sl(v_p^i, g^j)$ is the latest time that agent i can reach v_p^i without increasing agent j's execution time. In Figure 2a, the EAT of G^2 is $L(G^2) = 2$, the longest path length from G^2 to D^2 is $L(G^2, D^2) = 2$ and the EAT of D^2 is $L(D^2) = 4$. Then the slack of G^2 is $Sl(G^2, D^2) = 4 - 2 - 2 = 0$. It means that there is no slack, and any increase in the EAT of G^2 will be reflected in the EAT of D^2. Thus, in Figure 2b, where $L(G^2)$ increases by 3 , $L(D^2)$ also increases by 3.

It is worth mentioning that $L(v_p^i, g^j)$ cannot be obtained during the computation of $L(v_p^i)$ and should be computed again by topological sort and dynamic programming in the backward direction (Line 20 of Algorithm 1). Unfortunately, in this computation, we need to maintain the backward longest path lengths from v_p^i to each goal location g^j separately because the increase in a vertex v_p^i's EAT may

influence other agents' EATs at their goals, regarding the graph structure. Thus, it takes $O(|I|(|\mathcal{V}| + |\mathcal{N}|))$ time for each STPG to compute $L(v_p^i, g^j)$, while $L(v_p^i)$ only takes $O(|\mathcal{V}|+|\mathcal{N}|)$. The pseudocode is provided in Appendix A.1.

We use the subscript d to represent a descendant search tree node. For example, $L_d(v_1, v_2)$ is the longest path length from v_1 to v_2 in a descendant search tree node d. A simple fact is that $L_d(v_1, v_2) \geq L(v_1, v_2)$, namely, the longest path length is non-decreasing from an ancestor to a descendant.

Now, we start our reasoning by assuming that we fix a switchable edge $e = (v_{q+1}^j, v_p^i)$ and add it to the reduced TPG of a descendant search tree node, then we can deduce the increase in the EAT of v_p^i. Specifically, $L_d(v_p^i) \geq L_d(v_{q+1}^j) + 1 \geq L(v_{q+1}^j) + 1$. The first inequality holds because of the precedence implied by the new edge. The second inequality exploits the monotonicity of the longest path length. Thus, the increase of $L(v_p^i)$ is $\Delta L(v_p^i) \triangleq L_d(v_p^i) - L(v_p^i) \geq L(v_{q+1}^j) + 1 - L(v_p^i) = -Sl(e)$. Namely, the EAT at v_p^i will be increased by at least the negative edge slack.

Next, we consider the increase in the EAT of agent m's goal g^m, if vertex v_p^i is connected to g^m. $L_d(g^m) \geq L_d(v_p^i) + L_d(v_p^i, g^m) \geq (L(v_p^i) + \Delta L(v_p^i)) + L(v_p^i, g^m) = \Delta L(v_p^i) + (L(v_p^i) + L(v_p^i, g^m)) = \Delta L(v_p^i) + (L(g^m) - Sl(v_p^i, g^m))$. The first inequality is based on the fact that the longest path must be no shorter than any path passing a specific vertex. The second inequality is based on the definition of $\Delta L(v_p^i)$ and the monotonicity of the longest path length. The last equality is obtained by plugging in the definition of vertex slack. So, we get $L_d(g^m) \geq \Delta L(v_p^i) + (L(g^m) - Sl(v_p^i, g^m))$. Then, we can move $L(g^m)$ to the left side and obtain the increase $\Delta L(g^m) \triangleq L_d(g^m) - L(g^m) \geq \Delta L(v_p^i) - Sl(v_p^i, g^m)$. Namely, if the increase in EAT at v_p^i is larger than the vertex slack at v_p^i, the EAT at g^m will increase by at least their difference. Since we obtain $\Delta L(v_p^i) \geq -Sl(e)$ earlier, we have $\Delta L(g^m) \geq -Sl(e) - Sl(v_p^i, g^m)$.[5]

Similarly, if we reverse the edge $e = (v_{q+1}^j, v_p^i)$ and add $Re(e) = (v_{p+1}^i, v_q^j)$ to the STPG of a descendant search tree node, we can deduce a similar result for the EAT of agent n's goal g^n, $\Delta L(g^n) \geq -Sl(Re(e)) - Sl(v_q^j, g^n)$, if v_q^j is connected to agent n's goal vertex g^n.

Since we must either fix or reverse to settle a switchable edge e eventually, we can get a conservative estimation of the future joint increase in $L(g^m) + L(g^n)$, $\Delta L(g^m) + \Delta L(g^n) \geq \Delta cost(g^m, g^n, e) \triangleq \min\{-Sl(e) - Sl(v_p^i, g^m), -Sl(Re(e)) - Sl(v_q^j, g^n)\}$. Further, if there are multiple such edges, we can take the maximum of all $\Delta cost(g^m, g^n, e)$. Namely, $\Delta L(g^m) + \Delta L(g^n) \geq \Delta cost(g^m, g^n) \triangleq \max_e\{cost(g^m, g^n, e)\}$.

In summary, we can now get a conservative estimation of the future increase in pairwise joint costs. Then, we can apply Weighted Pairwise Dependency Graph (Li et al. 2019) to obtain a lower-bound estimation of the overall future cost increase for all agents. Specifically, we build a weighted fully-connected undirected graph $G_D = (V_D, E_D, W_D)$, where each vertex $u_i \in V_D$ represents an agent, E_D are edges and W_D are edges' weights. The weight of an edge $(u_i, u_j) \in E_D$ is defined as the pairwise cost increase, $\Delta cost(g^i, g^j)$. Our target is to assign a cost increase x_i to each vertex u_i such that $x_i + x_j \geq \Delta cost(g^i, g^j)$ and the overall cost increase $\sum_i x_i$ is minimized. This formulation turns the original problem into an Edge-Weighted Minimum Vertex Cover (EWMVC) problem, an extension of the NP-hard Minimum Vertex Cover problem. Therefore, we choose a fast greedy matching algorithm implemented in the paper (Li et al. 2019) with worst-case $O(|V_D|^3)$ complexity to get an underestimation of the minimum overall cost increase, denoted as $\Delta cost$. The details of the matching algorithm are given in Appendix A.3. Since the future increase in the overall cost must be no less than the non-negative term, $\Delta cost$, combined with the original h-value, $\sum_i L(g^i)$, we get a stronger admissible heuristic value, $\sum_i L(g^i) + \Delta cost$.

[5]Due to the monotonicity, we also have $\Delta L(g^m) = L_d(g^m) - L(g^m) \geq 0$. But we will omit this trivial condition for simplicity.

Algorithm 2: Compute Stronger Admissible Heuristics

```
1: function ComputeHeuristics(G^S)
2:     Compute the forward longest path lengths L(v) for all vertices
3:     Compute the backward longest path lengths L(v, g) between all vertices and goals (L(v, g) = ∞ if v and g are not connected)
4:     Δcost(g^m, g^n) ← 0, ∀m, n
5:     for all switchable edge e = (v^j_{q+1}, v^i_p) do
6:         for all agent pairs (m, n) do
7:             if L(v^i_p, g^m) ≠ ∞ ∧ L(v^j_q, g^n) ≠ ∞ then
8:                 Δc ← min{−Sl(e) − Sl(v^i_p, g^m),
9:                          −Sl(Re(e)) − Sl(v^j_q, g^n)}
10:                if Δc > Δcost(g^m, g^n) then
11:                    Δcost(g^m, g^n) ← Δc
12:                    Δcost(g^n, g^m) ← Δc
13:    Build the weighted pairwise dependency graph G_D with Δcost(g^m, g^n) as edge weights
14:    Suboptimally solve the edge-weighted minimum vertex cover of G_D by greedy matching to obtain the overall cost increase Δcost
15:    return Σ_i L(g^i) + Δcost
```

We summarize the computation of stronger admissible heuristics in Algorithm 2. Line 2-Line 3 compute the forward and backward longest path lengths. Line 4-Line 12 estimate the cost increase for each pair of agents. Line 13-Line 15 estimate the overall cost increase.

4.2 Edge Grouping

In the MILP-based method, Berndt et al. (2024) proposed a speedup method named dependency grouping, explicitly called edge grouping in this work. Edge grouping tries to find switchable edges whose directions can be decided together. If these edges do not follow the same direction, we can easily find a cycle within this group of edges. Berndt et al. (2024) describe two obvious grouping patterns, parallel and crossing (Figure 3a and Figure 3b), which can be easily

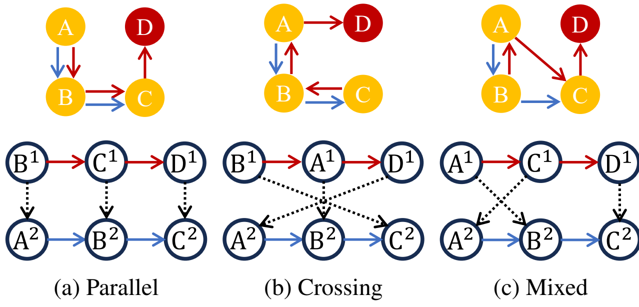


Figure 3: Edge grouping examples. The upper graphs show the partial MAPF trajectories. The lower graphs show the partial STPG with switchable edges that can be grouped. In (a), Agent 2 (blue arrows) visits the same sequence of locations, A, B, C, as Agent 1 (red arrows). In a successful execution, one agent must visit each location before the other agent. Namely, the visiting orders of each location must be the same. Therefore, these edges must always have the same direction and are groupable. In (b), Agent 2 traverses Agent 1's trajectory reversely, but the conclusion remains the same. (c) can be regarded as a mixture of (a) and (b).

detected by a linear scan over all switchable edges. However, they did not discuss whether there are other grouping patterns and how to find them all. Indeed, a counterexample is given in Figure 3c, which should be considered a single group but will be detected as two by their simple algorithm.

In this work, we devise an algorithm for finding all the maximal groups among switchable edges from agent i to agent j to reduce the search tree size. We name our method **full grouping** and the old one **simple grouping** to differentiate them. This algorithm will be called before the best-first search as a preprocessing step to the initial STPG (Line 2 of Algorithm 1) so that we consider an edge group rather than a single edge at each branch during the search (Line 10 of Algorithm 1). For the simplicity of the discussion, we assume the initial STPG has only switchable Type-2 edges since we can convert the non-switchable edges to switchable ones and re-settle their directions after the grouping. First, we formalize our definitions.

Definition 8 (Ordered-Pairwise Subgraph $\mathcal{G}^S_{i,j}$)**.** *The ordered-pairwise subgraph $\mathcal{G}^S_{i,j} = (\mathcal{V}_{i,j}, \mathcal{E}_{1\ i,j}, (\mathcal{S}_{i,j}, \mathcal{N}_{i,j}))$ for two agents i, j is a subgraph of an STPG $\mathcal{G}^S$ with only vertices of these two agents, their type-1 edges and all the Type-2 edges pointing from agent i to agent j.*

Definition 9 (Groupable)**.** *For a subgraph $\mathcal{G}^S_{i,j}$, two switchable edges $e_m, e_n \in \mathcal{S}_{i,j}$ are groupable, if, for all the two-agent acyclic TPGs that $\mathcal{G}^S_{i,j}$ can produce, either both e_m, e_n or both $Re(e_m), Re(e_n)$ are in them.*

Apparently, groupable is an equivalence relation that is reflexive, symmetric and transitive. It divides set $\mathcal{S}_{i,j}$ into a set of disjoint equivalence classes, which are called **maximal edge groups** in our setting. The following definition of the maximal edge group is a rephrase of the equivalence class defined by the groupable relation.

Algorithm 3: Edge Grouping

```
1: function EDGEGROUPING(G^S_{i,j})
2:     Groups ← {}, Edges ← S_{i,j}
3:     while Edges is not emtpy do
4:         select any edge e ∈ Edges
5:         EG ← FINDGROUPABLEEDGES(Edges, e)
6:         Groups ← Groups ∪ {EG}
7:         Edges = Edges − EG
8:     return Groups
```

Algorithm 4: FindGroupableEdges

```
1: function FINDGROUPABLEEDGES(Edges, e)
2:     // Reason with the reverse direction of e
3:     S_1 ← PROPAGATE(Edges, e)
4:     // Reason with the original direction of e
5:     E_r ← REVERSE(Edges)
6:     e_r ← REVERSE(e)
7:     S_r ← PROPAGATE(E_r, e_r)
8:     S_2 ← REVERSE(S_r)
9:     // Obtain the edge group by intersection
10:    EG ← S_1 ∩ S_2
11:    return EG
12: function PROPAGATE(Edges, e)
13:    S ← {e}, C ← {e}, E ← Edges − {e}
14:    repeat
15:        C_r = REVERSE(C)
16:        C ← FINDCYCLEEDGES(E, C_r)
17:        S ← S ∪ C
18:        E ← E − C
19:    until C is empty
20:    return S
21: function FINDCYCLEEDGES(E, C_r)
22:    C ← {}
23:    for all e = (v^i_m, v^j_n) ∈ E do
24:        for all e_r = (v^j_q, v^i_p) ∈ C_r do
25:            if p ≤ m ∧ n ≤ q then
26:                C ← C ∪ {e}
27:    return C
```

Definition 10 (Maximal Edge Group)**.** *A maximal edge group $\mathcal{EG}$ of an edge e is a subset of $\mathcal{S}_{i,j}$, which contains exactly all the edges groupable with e.*

Based on the property of the equivalence relation, the maximal edge group of an edge e must be unique. Thus, we apply a simple framework to find all the maximal edge groups in Algorithm 3. We initialize an edge set with all the switchable edges (Line 2). We select one edge from the edge set and find all the edges that are groupable with it (Line 5), then remove them from the edge set (Line 7). We repeat this process for the remaining edges until all the maximal edge groups are found and then removed.

Before discussing how to identify all groupable edges, we introduce a lemma that enables cycle detection in a two-agent TPG by examining only the vertex indexes of each opposite-direction edge pair.

Lemma 1. *If a two-agent TPG is cyclic, we must be able*

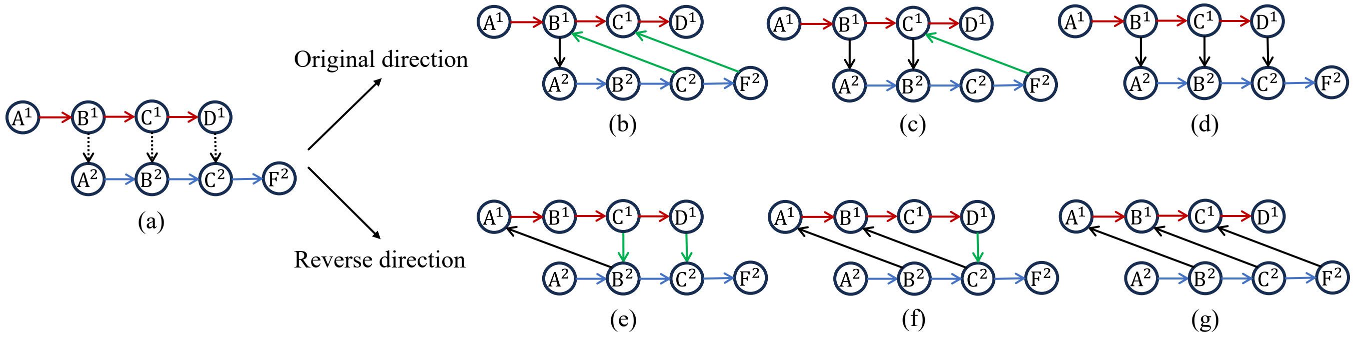


Figure 4: An example of finding all the switchable edges groupable with a certain switchable edge e. (a) is a complete STPG for the parallel pattern in Figure 3a. We want to find all the switchable edges groupable with (B^1, A^2) in (a). We must consider two possible directions of it. (b), (c), (d) illustrate the reasoning for the case we fix it. Specifically, in (b), we fix (B^1, A^2) (marked black) and temporarily reverse other switchable edges (marked green) to reason whether they must follow the same direction as (B^1, A^2) based on the constraint that there should be no cycle in the graph. Since there is a cycle $C^2 \to B^1 \to A^2$ in (b), we know (C^1, B^2) in (a) must be fixed instead of reversed. Then, we obtain (c). Similarly, there is a cycle $B^2 \to F^2 \to C^1$. So, (D^1, C^2) must also be fixed. Then, we obtain (d). As a result, we find both edges (C^1, B^2) and (D^1, C^2) need to follow the same direction as (B^1, A^2) if it is fixed. (e), (f), (g) illustrate the reasoning for the case that we reverse the switchable edge (B^1, A^2). We confirm that both edges (C^1, B^2) and (D^1, C^2) need to follow the same direction as (B^1, A^2) if it is reversed. Therefore, these three switchable edges must always select the same direction. Namely, they are groupable.

to find a cycle that contains exactly two Type-2 edges, one edge pointing from agent i to j and another edge pointing reversely. For these two edges $e_1 = (v^i_m, v^j_n)$ and $e_2 = (v^j_q, v^i_p)$, we must have $p \leq m \wedge n \leq q$.

Proof. Given an arbitrary cycle on the two-agent TPG, let v^i_p be the earliest vertex of agent i in the cycle (i.e., the vertex of agent i with the smallest p). The edge in the cycle that points to v^i_p must be from agent j. Denote this edge as (v^j_q, v^i_p). Similarly, let v^j_n be the earliest vertex of agent j in the cycle. The edge in the cycle that points to v^j_n must be from agent i. Denote this edge as (v^i_m, v^j_n). Since v^i_p and v^j_n are the earliest vertices of their respective agents, we know $p \leq m$ and $n \leq q$. Therefore, we find a cycle $v^j_q \to v^i_p \to \cdots \to v^i_m \to v^j_n \to \cdots \to v^j_q$, where the first "$\cdots$" denotes a sequence of Type-1 edges for agent i and the second "$\cdots$" denotes a sequence of Type-1 edges for agent j. This cycle meets the properties described by the lemma. □

Next, we describe the Function FINDGROUPABLEEDGES to identify all switchable edges that can be grouped with a specific edge e in Algorithm 4. We provide an intuitive example in Figure 4 with detailed reasoning in its caption for the parallel pattern mentioned in the Figure 3a.

To find all groupable edges, we must reason the two possible directions of e (Line 3 for the reverse direction and Line 5-Line 8 for the original direction). If e is reversed, we call Function PROPAGATE to find the set of edges that must follow the same direction as $Re(e)$. If e is fixed, because of the symmetry of the reasoning procedure, we need to reverse all the switchable edges first, then call PROPAGATE, and finally reverse the result back (Line 5-Line 8). This way, we can find another set of edges that must follow the same direction as e if e is fixed. The intersection of these two edge sets contains all the edges that must always select the same direction as e, namely all the groupable edges (Line 10).

Our core Function PROPAGATE tries to figure out all the edges in E that must also be reversed if we reverse the edge e. All the edges that must be reversed are recorded in S, the newly found ones are kept in C, and the remaining edges are maintained in E (Line 2). S and C start with only the edge e while E is initialized to all the edges but e. Line 15 reverses the direction of all newly found edges in C and Line 16 calls Function FINDCYCLEEDGES to find the edges in E that would form cycles with the reversed edges in C_r. Indeed, the returned edges from FINDCYCLEEDGES become the newly found edges that must be reversed as e. Otherwise, our TPG would be cyclic. The procedure of cycle detection is an efficient implementation based on the Lemma 1.

If we cannot find any edges leading to cycles (Line 19), it implies that we find a settlement of all switchable edges leading to a TPG with no cycle. This TPG is a certificate that all the remaining edges can select the opposite of e's settled direction and, thus, are not groupable with e. Then, we can conclude that all the groupable edges must be a subset of S_1 in Line 3, similarly, of S_2 in Line 8, and consequently, of their intersection. On the other hand, according to the result of Function PROPAGATE, the edges in S_1 must select the same direction as e if e is reversed, and the edges in S_2 must select the same direction as e if e is fixed. Therefore, the edges in the intersection must follow the same direction as e. That is, we find the exact maximal edge group containing edge e by Algorithm 4, according to Definition 10.

Regarding the time complexity of the edge grouping, in the worst case, the while loop in Algorithm 3 needs to iterate over all edges and has a complexity of $O(|\mathcal{S}_{i,j}|)$. For each call to Function PROPAGATE, we at most need to check all pairs of edges, and it has a complexity of $O(|\mathcal{S}_{i,j}|^2)$. There-

fore, the overall worst-case complexity is $O(|\mathcal{S}_{i,j}|^3)$.

4.3 Prioritized Branching

In Line 26-Line 29 of Algorithm 1, we need to select a conflicting edge to branch. However, different prioritization in conflicting edge selection may influence the search speed. We experiment with the following four strategies:

1. **Random**: randomly select a conflicting edge.
2. **Earliest-First**: select the first conflicting edge $e = (v_p, v_q)$ with the smallest ordered-pair $(L(v_q), L(v_p))$.
3. **Agent-First**: select the first conflicting edge with the smallest agent index. This is the strategy used by GSES.
4. **Smallest-Edge-Slack-First**: select the first conflicting edge with the smallest edge slack $Sl(e)$.

In Section 5, we find that **Smallest-Edge-Slack-First** performs the best empirically. The intuition behind this design is that $Sl(e)$ reflects edge e's degree of conflict with the current STPG, and we want to address the most conflicting one first because it may influence the overall cost the most.

4.4 Incremental Implementation

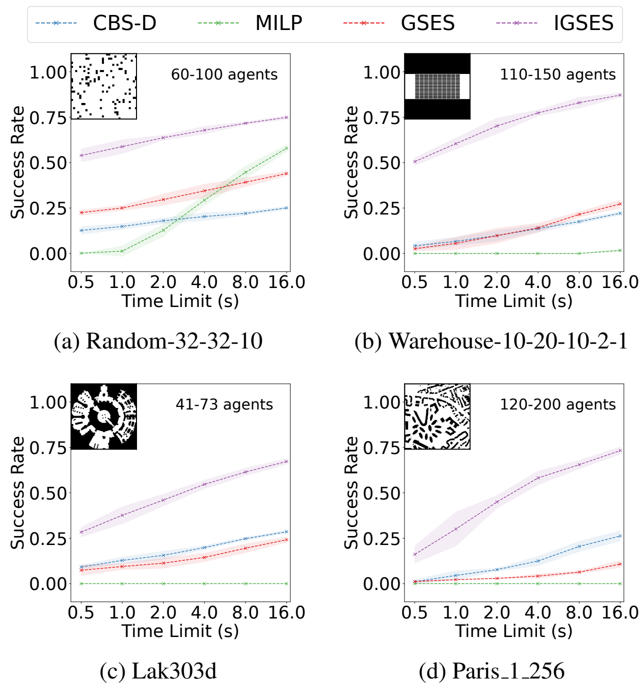
In GSES, the computation of the longest path lengths (Line 20 of Algorithm 1) takes the most time in a search node construction, as illustrated in Figure 7. But each time we build a child node, we only add one edge (group) to the reduced TPG of the parent node. Therefore, We directly apply an existing algorithm for computing the longest path lengths incrementally in a directed acyclic graph (Katriel, Michel, and Van Hentenryck 2005) to speed up the search.

5 Experiments

Following the setting in the baseline GSES paper (Feng et al. 2024), we evaluate algorithms on four maps from the MAPF benchmark (Stern et al. 2019), with 5 different numbers of agents per map. For each map and each number of agents, we generate 25 different instances with start and goal locations evenly distributed. We then run 1-robust PBS on each instance to obtain the initial MAPF plans. We add the 1-robust requirement (Atzmon et al. 2018) to PBS (Ma et al. 2019) because the original PBS does not resolve the following conflicts. Each solved instance is tested with 6 different delay scenarios. We obtain a scenario by simulating the initial MAPF plan until some delays happen. An agent will be independently delayed for 10-20 steps with a constant probability $p \in \{0.002, 0.01, 0.03\}$ at each step. We run 8 times for each scenario and set a time limit of 16 seconds for each run. Notably, since all the algorithms we compare with are optimal, we only focus on their search time. Due to the space limit, we only report the results of 0.01 delay probability in the main text with others reported in Appendix A.4. The conclusions are consistent across different probabilities.

All the algorithms in the experiments are implemented in C++.[6] All the experiments are conducted on a server with an Intel(R) Xeon(R) Platinum 8352V CPU and 120 GB RAM.

[6]MILP also uses a Python wrapper, so we count only its C++ solver's time for a fair comparison.



(a) Random-32-32-10
(b) Warehouse-10-20-10-2-1
(c) Lak303d
(d) Paris_1_256

Figure 5: Success rates of IGSES and other baselines on four maps. An instance is considered successfully solved if an optimal solution is returned within the time limit. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration. In each figure, the top-left corner shows the corresponding map, and the top-right corner shows the range of agent numbers.

Additional experimental details are covered in Appendix A.4. The code and benchmark are publicly available.[7]

5.1 Comparison with Other Algorithms

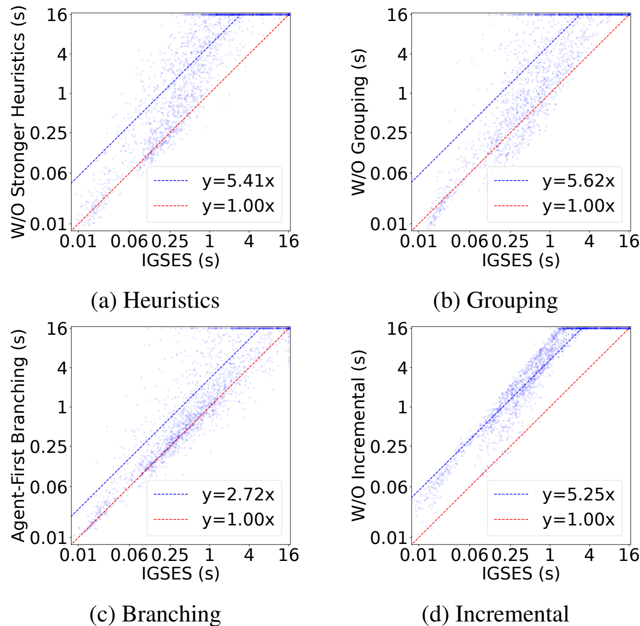
We compare our method IGSES with the baseline GSES algorithm and other two optimal algorithms, MILP (Berndt et al. 2024) and CBS-D[8] (Kottinger et al. 2024). The success rates on four maps with increasing sizes are shown in Figure 5. Compared to GSES, IGSES at least doubles the success rates in most cases. Due to the inefficiency of a general branch-and-bound solver, MILP can only solve instances on the small Random map. An interesting observation is that CBS-D is worse than GSES on small maps but generally becomes better than GSES as the map size increases. However, directly applying existing MAPF algorithms without tailored adaptations like CBS-D shows much worse performance than our IGSES, which better exploits the problem's structure. In Appendix A.4, we also plot the mean search time for different maps and agent numbers to support our conclusions.

[7]https://github.com/DiligentPanda/STPG.git

[8]We adapt the original CBS-D for our MAPF definition, as we forbid not only edge conflicts but all following conflicts.

Row	Setting	Random-32-32-10 search time (s)	Random-32-32-10 #expanded nodes	Random-32-32-10 #edge groups	Warehouse-10-20-10-2-1 search time (s)	Warehouse-10-20-10-2-1 #expanded nodes	Warehouse-10-20-10-2-1 #edge groups	Lak303d search time (s)	Lak303d #expanded nodes	Lak303d #edge groups	Paris_1_256 search time (s)	Paris_1_256 #expanded nodes	Paris_1_256 #edge groups
1	GSES	1.421	3051.5	1637.9	3.805	948.3	15441.6	2.608	550.8	30320.6	5.221	253.1	40214.5
2	+SG	1.082	2025.7	1176.4	1.783	315.3	9134.2	1.242	185.7	19012.7	3.131	115.2	27065.5
3	+FG	0.945	1597.9	732.5	1.519	241.4	2367.0	1.021	130.8	5601.8	2.770	88.1	10706.9
4	+FG +RB	1.555	2495.2	732.5	2.685	447.4	2367.0	2.644	324.2	5601.8	3.553	114.5	10706.9
5	+FG +EB	1.185	2036.1	732.5	2.354	381.6	2367.0	1.481	172.9	5601.8	3.118	102.6	10706.9
6	+FG +SB	0.915	1480.5	732.5	1.472	229.5	2367.0	1.027	129.0	5601.8	2.472	78.7	10706.9
7	+FG +SB +SH	0.169	84.6	732.5	0.832	34.6	2367.0	0.871	33.1	5601.8	1.561	15.4	10706.9
8	+FG +SB +SH +INC	0.043	84.6	732.5	0.120	34.6	2367.0	0.221	33.1	5601.8	0.321	15.4	10706.9

Table 1: Incremental analysis on instances solved by all the settings. Rows 1,2,3 compare the effects of different grouping methods. GSES has no grouping. SG: simple grouping. FG: full grouping. Rows 3,4,5,6 compare the effects of different branching orders. Row 3 applies the default Agent-First branching in GSES. RB: Random branching. EB: Earliest-First branching. SB: Smallest-Edge-Slack-First branching. Rows 6,7 compare the effect of the stronger heuristics. SH: stronger heuristics. Rows 7,8 compare the effect of the incremental implementation. INC: incremental implementation. The last row is also our IGSES.



(a) Heuristics

(b) Grouping

(c) Branching

(d) Incremental

Figure 6: Ablation on four speedup techniques on all instances. In each figure, we compare IGSES to the ablated setting that replaces one of its techniques by the GSES's choice. Each point in the figure represents an instance with its x, y coordinates being the search time of the two settings. The search time is set to 16 seconds if an instance is not solved. The blue line is fitted on the instances solved by at least one setting. Its slope indicates the average speedup.

5.2 Ablation Studies

First, we provide an incremental analysis in Table 1. We begin with GSES in the first row, gradually compare different choices in each technique, and add the most effective one. Finally, we obtain IGSES in the last row, which achieves a 10- to 30-fold speedup and over a 90% reduction in node expansion on instances solved by all the settings, compared to GSES. Notably, comparing rows 1,2,3, we can find that our full grouping has the smallest number of edge groups, thus empirically less branching and node expansion. Comparing

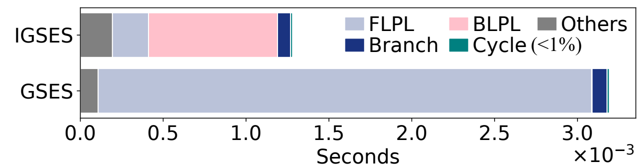


Figure 7: Runtime profile for each search node in the warehouse experiments. FLPL: forward longest path lengths. BLPL: backward longest path lengths. Branch: branch prioritization. Cycle: cycle detection. Others: copy and free data structures, termination check, and others.

rows 7,8, we can find that the incremental implementation does not affect the node expansion but significantly reduces the computation time as expected.

Further, we do an ablation study on the four techniques with all the instances, including unsolved ones, through pairwise comparison in Figure 6. All the techniques are critical to IGSES, as they all illustrate an obvious speedup.

Finally, we profile the average runtime on each node for GSES and IGSES with the warehouse experiment data in Figure 7. The computation of the longest path lengths takes the most runtime in both cases. IGSES's runtime on the forward longest path lengths is largely reduced mainly because of the incremental implementation. The backward longest path lengths take more time to compute than the forward, but IGSES's overall runtime is much smaller. The edge grouping time is not included here. Edge grouping can be computed only once before executing the MAPF plan and used whenever delays occur. It takes 0.037 seconds on average, which is negligible compared to the MAPF's planning time.

6 Conclusion and Future Work

In this paper, we study the STPG optimization problem. We analyze the weakness of the optimal GSES algorithm and propose four speedup techniques. Their effectiveness is validated by both theoretical proof and experimental data. To scale to larger instances, a potential future direction is to devise sub-optimal algorithms for the STPG optimization problem to trade-off between solution quality and speed.

Acknowledgments

The research was supported by the National Science Foundation (NSF) under grant number #2328671 and a gift from Amazon. 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.; Barták, R.; and Zhou, N.-F. 2018. Robust Multi-Agent Path Finding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 9, 2–9.

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

Berndt, A.; van Duijkeren, N.; Palmieri, L.; Kleiner, A.; and Keviczky, T. 2024. Receding Horizon Re-Ordering of Multi-Agent Execution Schedules. *IEEE Transactions on Robotics*, 40: 1356–1372.

Feng, Y.; Paul, A.; Chen, Z.; and Li, J. 2024. A Real-Time Rescheduling Algorithm for Multi-robot Plan Execution. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, 201–209.

Hönig, W.; Kumar, T.; 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.

Katriel, I.; Michel, L.; and Van Hentenryck, P. 2005. Maintaining Longest Paths Incrementally. *Constraints*, 10: 159–183.

Kottinger, J.; Geft, T.; Almagor, S.; Salzman, O.; and Lahijanian, M. 2024. Introducing Delays in Multi Agent Path Finding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, 37–45.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 442–449.

Liu, Y.; Tang, X.; Cai, W.; and Li, J. 2024. Multi-Agent Path Execution with Uncertainty. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, 64–72.

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. S.; and Koenig, S. 2017. Multi-Agent Path Finding with Delay Probabilities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31.

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.; et al. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 151–158.

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