Speedup Techniques for Switchable Temporal Plan Graph Optimization

Anonymous submission

Abstract

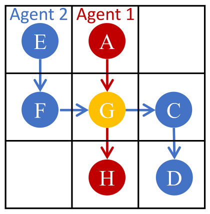
Multi-Agent Path Finding (MAPF) involves planning collision-free paths for multiple agents. However, during the execution of a MAPF plan, agents may experience unexpected delays, leading to potential collisions or deadlocks. To address this, the Switchable Temporal Plan Graph framework offers a way to find an acyclic Temporal Plan Graph with minimum execution cost, ensuring deadlock- and collision-free execution. 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 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 find solutions.

1 Introduction

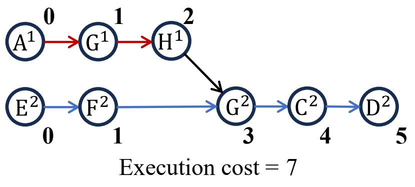
Multi-Agent Path Finding (MAPF) (Stern et al. 2019) involves planning collision-free paths for multiple agents to navigate from their starting locations to destinations. However, during execution, unpredictable delays can arise due to mechanical differences, accidental events, or sim-to-real gaps. For 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 collisions.

To address this, the Temporal Plan Graph (TPG) framework was 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, switching the visiting order of these two agents would allow Agent 2 to proceed first, avoiding unnecessary delays.

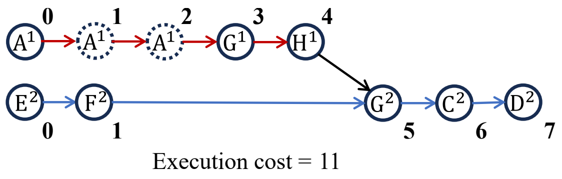
To capture this idea, Berndt et al. (2023) introduced the Switchable Temporal Plan Graph (STPG), which enables the search for a TPG that encodes optimal visiting orders to minimize the total travel time under delays. However, both their original solution, based on Mixed Integer Linear Programming (MILP), and the subsequent Graph-Based Switchable Edge Search (GSES) algorithm (Feng et al. 2024), are too slow to scale to scenarios involving just 100 agents in the experiments.



(a) Grid



(b) TPG



(c) Delayed TPG

Figure 1: (a) shows an example of MAPF problems. In the initial MAPF solution, 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 solution. 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.

This work makes a significant advancement in tackling the scalability issue of STPG, demonstrating its potential for online delay reduction. Specifically, we improve GSES with a series of speedup techniques that leverage the underlying structure of STPG. By estimating the additional costs induced by settling future switchable edges, we propose stronger admissible heuristics (Feng et al. 2024). To reduce the search tree size, we introduce a novel algorithm that identifies all the maximal groups of switchable edges whose directions can be determined simultaneously (Berndt et al. 2023). Additionally, we present two simple yet effective engineering techniques, prioritized branching and incremental implementation, to accelerate the search. We prove their correctness where applicable 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 methods.

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 k-robust or p-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 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. (2023) 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. Meanwhile, 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 private constrained graph for each agent and then applies standard MAPF algorithms to search for solutions. We call their optimal-version algorithm applying Conflict-Based Search (CBS) (Sharon et al. 2015), CBS with delays (CBS-D). In the experiments, we will compare our method with these optimal algorithms: MILP, GSES, and CBS-D.

In contrast to these methods that aim for an optimal solution, 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. Bi-directional Temporal Plan Graph (BTPG) (Su, Veerapaneni, and Li 2024) is another non-optimal approach based on TPG, but it falls into the offline category. Instead of reasoning about delays during planning, BTPG post-processes a MAPF solution 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. Comparison to LDG and BTPG will be deferred to future work when suboptimal versions of GSES are developed.

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 n 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 solution 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. Specifically, z^i is the number of different 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 $loc(v_q^j) = loc(v_p^i)$. A Type-2 edge (v_{q+1}^j, v_p^i) means that agent i can only enter v_p^i after agent j leaves v_q^j and arrives at v_{q+1}^j. If we want to specify the reverse order of visiting $loc(v_q^j)$, we should introduce (v_{p+1}^i, v_q^j) instead.*

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

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 $e = (v_q^j, v_{p+1}^i)$, 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 an acyclic TPG can be completed in finite time without collisions. (Berndt et al. 2023).*

[1]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.

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)**, $L(v_p^i) = \max\{L(s^k, v_p^i), k \in I,$ 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, $\sum_{i \in I} L(g^i)$. (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. 2023). We provide the pseudocode in Appendix A.1.

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 Temporal Plan Graph (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, S for switchable edges and $\mathcal{N}$ for non-switchable edges. Any switchable edge $e = (v_{q+1}^j, v_p^i) \in S$ 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, an STPG is a strict superclass of TPG. If all switchable edges are settled, it **produces** a TPG. The target of **STPG Optimization** is to find an acyclic TPG produced by the STPG with minimum execution cost.

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 $Redu(\mathcal{G}^S) = (\mathcal{V}, \mathcal{E}_1, \mathcal{N})$.*

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.

Algorithm 1: (Improved) Graph-Based Search

```
1:  function GRAPHBASEDSEARCH($\mathcal{G}^S_{init}$)
2:      $Groups \leftarrow$ EDGEGROUPING($\mathcal{G}^S_{init}$)
3:      Node$_{root}$ $\leftarrow$ BUILDNODE($\mathcal{G}^S_{init}$)
4:      Push Node$_{root}$ into OpenList
5:      while OpenList is not empty and still time left do
6:          Pop Node = ($\mathcal{G}^S$, $h$-value, $L$) from OpenList
7:          $e \leftarrow$ SELECTCONFLICTINGEDGE($\mathcal{G}^S$, $L$)
8:          if $e$ = NULL then return FIXALL($\mathcal{G}^S$)
9:          $Edges \leftarrow \{e\}$
10:         $Edges \leftarrow$ GETEDGEGROUP($e, Groups$)
11:         $\mathcal{G}^S_{\text{child-1}} \leftarrow$ FIX($\mathcal{G}^S$, $Edges$)
12:         $\mathcal{G}^S_{\text{child-2}} \leftarrow$ REVERSE($\mathcal{G}^S$, $Edges$)
13:         for $\mathcal{G}^S_{\text{child}} \in \{\mathcal{G}^S_{\text{child-1}}, \mathcal{G}^S_{\text{child-2}}\}$ do
14:             if No cycle in $Redu(\mathcal{G}^S_{\text{child}})$ then
15:                 Node$_{\text{child}}$ $\leftarrow$ BUILDNODE($\mathcal{G}^S_{\text{child}}$)
16:                 Push Node$_{\text{child}}$ into OpenList
17:     return NULL // Timeout
18:
19: function BUILDNODE($\mathcal{G}^S$)
20:     Incrementally compute for all vertices the forward and backward longest path lengths on $Redu(\mathcal{G}^S)$ as a table function $L$ based on its parent graph
21:     Estimate the future cost increase $\Delta cost$ based on the backward longest path lengths
22:     $h$-value $\leftarrow \sum_i L(g^i) + \Delta cost$
23:     return ($\mathcal{G}^S$, $h$-value, $L$)
24:
25: function SELECTCONFLICTINGEDGE($\mathcal{G}^S$, $L$)
26:     Order switchable edges in $S$ by certain priority
27:     for all switchable edge $e = (v_1, v_2) \in S$ do
28:         if $L(v_1) + 1 > L(v_2)$ then return e
29:     return NULL
```

Definition 6 (Edge Slack). *The **slack of an 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 $L(v_p^i)$ is the EAT at vertex v_p^i in the reduced TPG.*

We say an edge **conflicts** with the STPG if $Sl(e) < 0$. For a switchable edge e, $Sl(e) \geq 0$ means that the earliest execution of the reduced TPG already satisfied the constraints associated with e, i.e., we can currently fix e without introducing any cycle or cost increase to the reduced TPG (Feng et al. 2024). 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. The proof is given in Appendix A.2.

Algorithm 1 shows the pseudocode of GSES[2], a best-first search algorithm with each search node comprising three parts: the STPG, the execution cost of the STPG, and the EATs of all vertices in the reduced TPG (Line 23). The execution cost is used as the h-value in the best-first search (The g-value is always 0 in GSES), and the EATs are used to find conflicting edges.

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

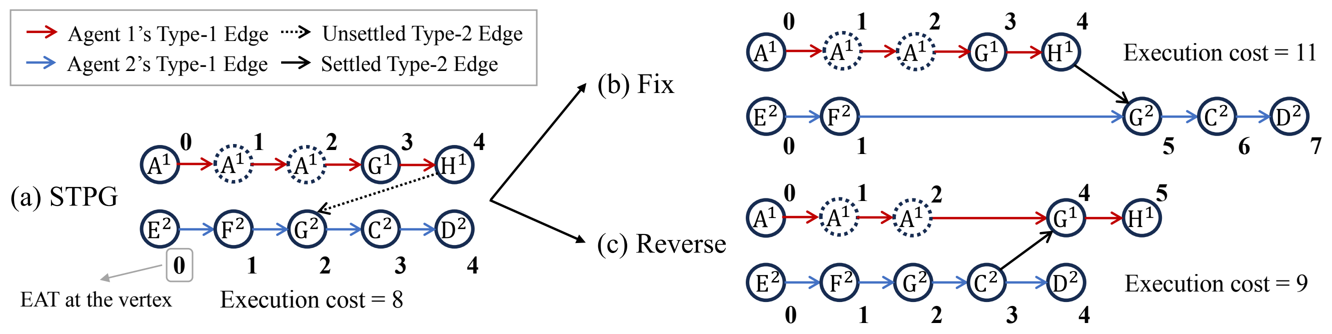


Figure 2: (a) is the STPG built from the TPG in Figure 1c. The only switchable edge (H^1, G^2) is dashed. It can be fixed to edge (H^1, G^2) as in (b) or reversed to edge (C^2, G^1) as in (c). The different choices result in different execution costs. Notably, when computing the execution cost of STPG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

GSES starts with a root node containing the initial STPG, which is constructed from the TPG of the initial MAPF solution. 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. 2023). 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 non-conflicting 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.

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* $Sl(v_p^i, g^j) = L(g^j) - L(v_p^i, g^j) - L(v_p^i)$*, if* v_p^i *is connected to* g^j *in the reduced TPG.* $L(v_p^i)$ *is the EAT at vertex* v_p^i*, 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 $Sl(G^2) = 4 - 2 - 2 = 0$. 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. Therefore, 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 a 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 $e = (v^j_{q+1}, v^i_p)$ and add it to the STPG of a descendant search-tree node, then we can deduce the increase in the EAT of v^i_p. Specifically, $L_d(v^i_p) \geq L_d(v^j_{q+1}) + 1 \geq L(v^j_{q+1}) + 1$. 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 $L(v^i_p)$ is $\Delta L(v^i_p) \triangleq L_d(v^i_p) - L(v^i_p) \geq L(v^j_{q+1}) + 1 - L(v^i_p) = -Sl(e)$. Namely, the EAT at v^i_p 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^i_p is connected to g^m. $L_d(g^m) \geq L_d(v^i_p) + L_d(v^i_p, g^m) \geq (L(v^i_p) + \Delta L(v^i_p)) + L(v^i_p, g^m) = \Delta L(v^i_p) + (L(v^i_p) + L(v^i_p, g^m)) = \Delta L(v^i_p) + (L(g^m) - Sl(v^i_p, g^m))$. 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 $\Delta L(v^i_p)$ and the monotonicity of the longest path length. The last equality is obtained by plugging in the definition of vertex slack. So, we get $L_d(g^m) \geq \Delta L(v^i_p) + (L(g^m) - Sl(v^i_p, g^m))$. Then, we can move $L(g^m)$ to the left side and obtain the increase $\Delta L(g^m) \triangleq L_d(g^m) - L(g^m) \geq \Delta L(v^i_p) - Sl(v^i_p, g^m)$. Namely, if the increase in EAT at v^i_p is larger than the slack at v^i_p, the EAT at g^m will increase by at least their difference.

Since we obtain $\Delta L(v^i_p) \geq -Sl(e)$ earlier, $\Delta L(g^m) \geq \Delta L(v^i_p) - Sl(v^i_p, g^m) \geq -Sl(e) - Sl(v^i_p, g^m)$. Therefore, $\Delta L(g^m) \geq -Sl(e) - Sl(v^i_p, g^m)$.[3]

Similarly, if we reverse the edge $e = (v^j_{q+1}, v^i_p)$ and add $Re(e) = (v^i_{p+1}, v^j_q)$ 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, $\Delta L(g^n) \geq -Sl(Re(e)) - Sl(v^j_q, g^n)$, if v^j_q 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 $L(g^m) + L(g^n)$, $\Delta L(g^m) + \Delta L(g^n) \geq \Delta cost(g^m, g^n, e) \triangleq \min\{-Sl(e) - Sl(v^i_p, g^m), -Sl(Re(e)) - Sl(v^j_q, g^m)\}$. 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 cost. 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 $v_i \in V_D$ represents an agent, E_D are edges and W_D are edges' weights. The weight of an edge $(v_i, v_j) \in E_D$ 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 v_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 minumum 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, $\sum_i L(g^i) + \Delta cost$.

[3]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($\mathcal{G}^S$)
2:      Compute the forward longest path lengths $L(v)$ for all vertices
3:      Compute the backward longest path lengths $L(v, g_i)$ between all vertices and goals ($L(v, g_i) = \infty$ if $v$ and $g_i$ are not connected)
4:      $\Delta cost(g^m, g^n) \leftarrow 0, \forall m, n$
5:      for all switchable edge $e = (v^j_{q+1}, v^i_p)$ do
6:          for all agent pairs $(m, n), m \leq n$ do
7:              if $L(v^j_q, g^m) \neq \infty \wedge L(v^i_p, g^n) \neq \infty$ then
8:                  $\Delta c \leftarrow \min\{-Sl(e) - Sl(v^i_p, g^m),$
9:                      $-Sl(Re(e)) - Sl(v^j_q, g^m)\}$
10:                 if $\Delta c > \Delta cost(g^m, g^n)$ then
11:                     $\Delta cost(g^m, g^n) \leftarrow \Delta c$
12:     Build the weighted pairwise dependency graph $G_D$ with $cost(g_m, g^n)$ as edge weights
13:     Suboptimally solve the edge-weighted minimum vertex cover of $G_D$ by greedy matching to obtain the overall cost increase $\Delta cost$
14:     return $\sum_i L(g^i) + \Delta cost$
```

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 11 estimate the cost increase for each pair of agents. Line 12-Line 14 estimate the overal cost increase.

4.2 Edge Grouping

In the MILP-based method, Berndt et al. (2023) 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. For example, Berndt et al. (2023) describe two obvious grouping patterns, parallel and crossing (Figure 3a and Figure 3b) in their paper. They can be easily detected by a linear scan over all the switchable edges. This technique significantly speeds up the MILP but has not been applied to the GSES.

On the other hand, Berndt et al. (2023) did not discuss whether there are other grouping patterns and how to find them all. Indeed, a counterexample is illustrated in Fig-

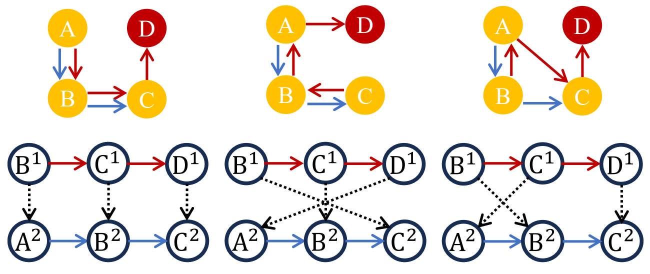


Figure 3: Edge grouping examples. The upper graph shows the partial MAPF problems. The lower graph shows the partial STPG with switchable edges that can be grouped. In (a), Agent 2 (blue arrows) visits the same sequence of locations, A, B, C, as Agent 1 (red arrows). Clearly, in a successful execution, one agent must visit each location before another 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 this sequence reversely, but the conclusion remains the same. (c) can be regarded as a mixture of (a) and (b).

ure 3c, which should be considered a single group but will be detected as two by their simple algorithm.

Therefore, 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 call 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 $\mathcal{G}^S_{i,j} = (V_{i,j}, \mathcal{E}_{i,j}, (S_{i,j}, N_{i,j}))$ 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 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.

Definition 10 (Maximal Edge Group)**.** *A maximal edge group $\mathcal{EG}$ is a subset of $\mathcal{S}_{i,j}$ such that for any two edges e_m and e_n, they are groupable if both edges are in $\mathcal{EG}$ and not groupable if exactly one of the edges is $\notin \mathcal{EG}$.*

Algorithm 3: Edge Grouping Framework

```
1:  function EdgeGrouping($\mathcal{G}^S_{init}$)
2:      $Groups \leftarrow \{\}$
3:      $Edges \leftarrow \mathcal{S}$
4:      while $Edges$ is not emtpy do
5:          select any edge $e \in Edges$
6:          $\mathcal{EG} \leftarrow$ FindGroupableEdges($e, Edges$)
7:          $Groups \leftarrow Groups \cup \{\mathcal{EG}\}$
8:          $Edges = Edges - \mathcal{EG}$
```

Algorithm 4: FindGroupableEdges

```
1:  function FindGroupableEdges($Edges, e$)
2:      // Reason with the reverse direction of $e$
3:      $S_1 \leftarrow$ Propagate($Edges, e$)
4:      // Reason with the original direction of $e$
5:      $E_r \leftarrow$ Reverse($Edges$)
6:      $e_r \leftarrow$ Reverse($e$)
7:      $S_r \leftarrow$ Propagate($E_r, e_r$)
8:      $S_2 \leftarrow$ Reverse($S_r$)
9:      // Obtain the edge group by intersection
10:     $\mathcal{EG} \leftarrow S_1 \cap S_2$
11:     return $\mathcal{EG}$
12: function Propagate($Edges, e$)
13:     $S \leftarrow \{e\}, C \leftarrow \{e\}, E \leftarrow Edges - \{e\}$
14:     repeat
15:         $C_r =$ Reverse($C$)
16:         $C \leftarrow$ FindCycleEdges($E, C_r$)
17:         $S \leftarrow S \cup C$
18:         $E \leftarrow E - C$
19:     until $C$ is empty
20:     return S
21: function FindCycleEdges($E, C_r$)
22:     $C \leftarrow \{\}$
23:     for all $e = (v^i_m, v^j_n) \in E$ do
24:         for all $e_r = (v^j_q, v^i_p) \in C_r$ do
25:             if $p \leq m \wedge n \leq q$ then
26:                 $C \leftarrow C \cup \{e\}$
27:     return $C$
```

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 3). We select one edge from the edge set and find all the edges that are groupable with it (Line 6), then remove them from the edge set (Line 8). We repeat this process for the remaining edges until all the maximal edge groups are found and then removed. Notably, a single edge could also be a maximal edge group if no other edge is groupable with it.

Next, we describe the Function FindGroupableEdges to identify all the 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 3.

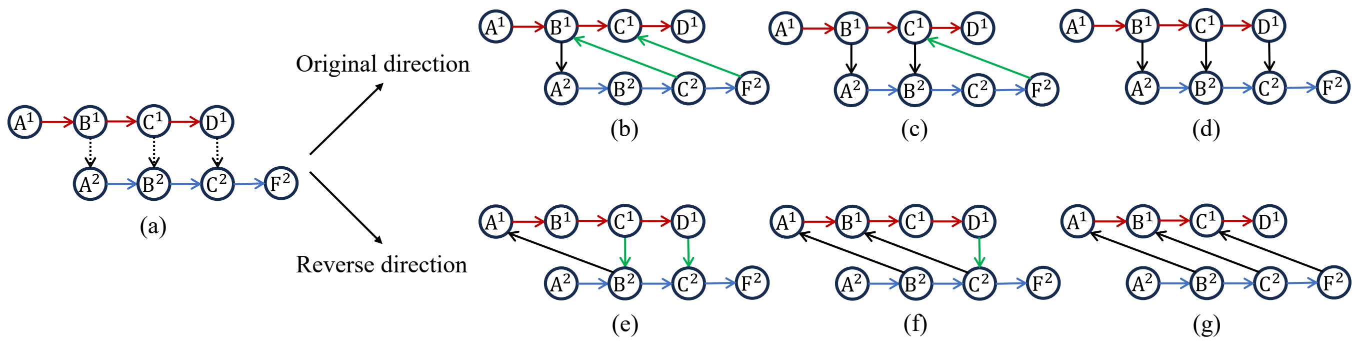


Figure 4: An example of finding all the switchable edges groupable with a certain switchable edge e. (a) is a complete STPG for the parallel pattern in Figure 3a. We want to find all the switchable edges groupable with (B^1, A^2) in (a). We must consider two possible directions of it. (b), (c), (d) illustrate 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 all the 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 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 3). 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 that directly checks the vertex indexes of each opposite edge pair without depth-first search, based on the following Lemma 1.

Lemma 1. *If a two-agent TPG is cyclic, we must be able 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 $p \le m \land n \le q$.*

Proof. We prove it by construction.

First, we can consider all the edges in this cycle, the head vertex of which belongs to agent i, and find the one with the minimal vertex index p, namely, vertex v^i_p. There must be an edge e_2 pointing to this vertex v^i_p since it is a cycle. However, e_2 cannot be a Type-1 edge because if so, its head also belongs to agent i but has a smaller vertex index than p. Therefore, it must be a Type-2 edge from agent j to agent i. Let us denote it as (v^j_q, v^i_p).

Similarly, we can consider all the edges in this cycle, the head vertex of which belongs to agent j, and find the one with the minimal vertex index n, namely, vertex v^j_n. There must be an edge e_1 pointing to this vertex v^j_n since it is a cycle. However, e_1 cannot be a Type-1 edge because if so, its head also belongs to agent j but has a smaller vertex index than n. Therefore, it must be a Type-2 edge from agent i to agent j. Let us denote it as (v^i_m, v^j_n).

Now, we claim there is another cycle $v^j_q \to v^i_p \to v^i_m \to v^j_n \to v^j_q$. Notably, since $p \le m$, v^i_p and v^i_m are connected by Type-1 edges of agent i (or they may be the same vertex). Similarly, v^j_n and v^j_q are connected by Type-1 edges of agent j (or they may be the same vertex). The only two Type-2 edges $e_1 = (v^i_m, v^j_n)$ and $e_2 = (v^j_q, v^i_p)$ satisfy $p \le m \land n \le q$ as required. □

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 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 always follow the same direction as e. That is, we find the exact maximal edge group containing the edge e by Algorithm 4, according to the Definition 10.

The time complexity of the edge grouping can be easily estimated. 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 $O(|\mathcal{S}_{i,j}|^2)$. Therefore, the overall worst-case complexity would be $O(|\mathcal{S}_{i,j}|^3)$.

4.3 Prioritized Branching

In Line 26-Line 29 of Algorithm 1, we need to select a conflicting edge to branch. However, different prioritization of 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 experiment setting in the baseline paper (Feng et al. 2024), we evaluate algorithms on 4 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 test 25 different instances with start and goal locations evenly distributed. We use k-Robust PBS with $k = 1$ to obtain all the initial MAPF plans (Ma et al. 2019). Each solved instance will be 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 also optimal, we only compare their search time. Due to the space limit, we only report the results of 0.01 delay probability in

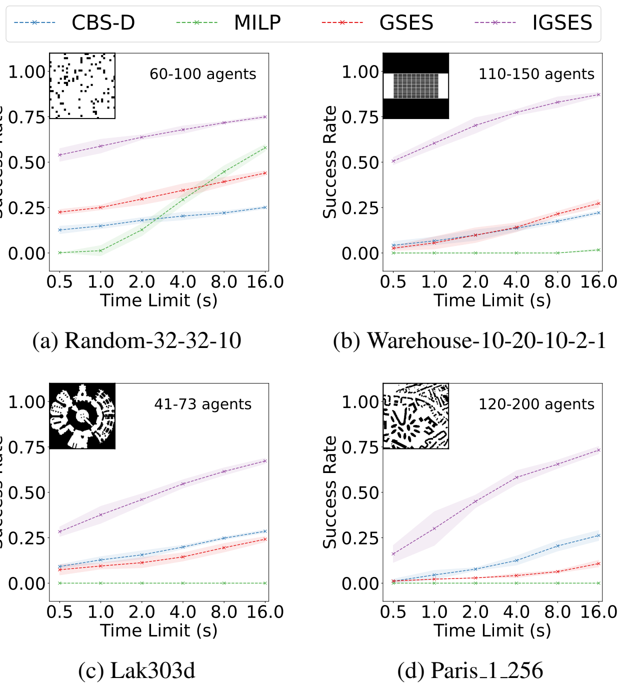


Figure 5: Success rates of IGSES and other baselines on four maps. An instance is considered successfully solved if an optimal solution is returned within the time limit. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration. In each figure,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

the main text with others reported in the Appendix A.4. The conclusions are consistent across different probabilities.

The solvers of all the algorithms used in the experiments are implemented in C++. We only count the solver's time for fair comparison. All experiments are conducted on a server with an Intel(R) Xeon(R) Platinum 8352V CPU and 120 GB RAM. More experiment setting details are covered in Appendix A.4. We will release both our code and benchmark.

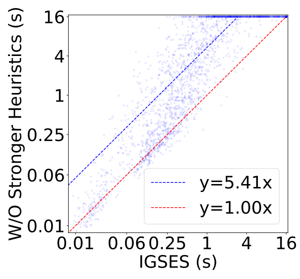
5.1 Comparison with Other Algorithms

We compare our method IGSES with the baseline GSES algorithm and other two optimal algorithms, MILP (Berndt et al. 2023) and CBS-D[4] (Kottinger et al. 2024). The success rates on 4 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 heavy computation 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 like CBS-D shows much worse performance than our IGSES, which better exploits the structure of the problem.

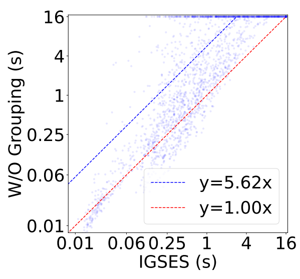
[4]We adapt the original CBS-D for our MAPF definition, as we forbid not only edge conflicts but all following conflicts.

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.

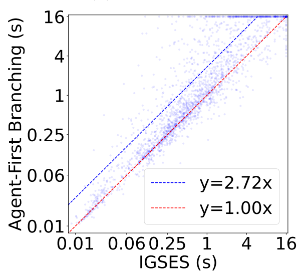
Row	Setting	Random-32-32-10			Warehouse-10-20-10-2-1			Lak303d			Paris_1_256		
		search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups
1	GSES	1.421	3051.5	1637.9	3.805	948.3	15441.6	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



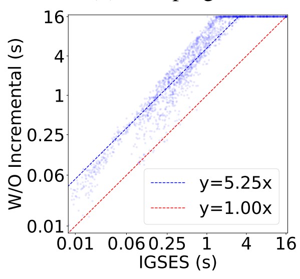
(a) Heuristics



(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 two settings. The search time is set to 16 seconds if an instance is not solved. The blue line is the fitted on instances solved by at least one setting. Its slope indicates the average speedup.

In Appendix A.4, we also plot the mean search time for different maps and agent numbers to support our conclusions.

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

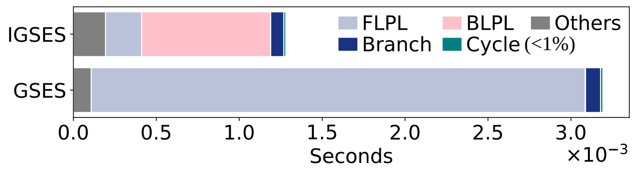


Figure 7: Runtime profile for each search node in the warehouse experiments. FLPL: forward longest path lengths. BLPL: backward longest path length. Branch: branch prioritization. Cycle: cycle detection. Others: copy and free data structures, termination check, and others.

GSES. Notably, comparing rows 1,2,3, we can find that our full grouping has the smallest number of edge groups, thus potentially less branching and node expansion. Comparing rows 7,8, we can find that the incremental implementation does not affect the node expansion as expected because it only reduces the time of heuristic computation in each node.

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.

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. 2023. Receding Horizon Re-ordering of Multi-Agent Execution Schedules. *IEEE Transactions on Robotics*.

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. International Joint Conferences on Artificial Intelligence Organization.

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.

7 Reproducibility Checklist

This paper:

1. Includes a conceptual outline and/or pseudocode description of AI methods introduced (yes)
2. Clearly delineates statements that are opinions, hypotheses, and speculations from objective facts and results (yes)
3. Provides well-marked pedagogical references for less-familiar readers to gain the background necessary to replicate the paper (yes)

Does this paper make theoretical contributions? (yes)
If yes, please complete the list below.

1. All assumptions and restrictions are stated clearly and formally. (yes)
2. All novel claims are stated formally (e.g., in theorem statements). (yes)
3. Proofs of all novel claims are included. (yes)
4. Proof sketches or intuitions are given for complex and/or novel results. (yes)
5. Appropriate citations to theoretical tools used are given. (yes)
6. All theoretical claims are demonstrated empirically to hold. (yes)
7. All experimental codes used to eliminate or disprove claims are included. (yes)

Does this paper rely on one or more datasets? (yes)
If yes, please complete the list below.

1. A motivation is given for why the experiments are conducted on the selected datasets (yes)
2. All novel datasets introduced in this paper are included in a data appendix. (yes)
3. All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. (yes)
4. All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations. (yes)
5. All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available. (yes)
6. All datasets that are not publicly available are described in detail, with an explanation of why publicly available alternatives are not scientifically satisfying. (NA. We don't use such datasets.)

Does this paper include computational experiments? (yes)
If yes, please complete the list below.

1. Any code required for pre-processing data is included in the appendix. (yes)
2. All source code required for conducting and analyzing the experiments is included in a code appendix. (partial)
3. All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. (yes)
4. All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (yes)
5. If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results. (yes)
6. This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models, amount of memory, operating system, names, and versions of relevant software libraries and frameworks. (yes)
7. This paper formally describes the evaluation metrics used and explains the motivation for choosing these metrics. (yes)
8. This paper states the number of algorithm runs used to compute each reported result. (yes)
9. Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information. (yes)
10. The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed rank). (no. our results are straightforward. We do not need complex statistic tools to claim the significance.)
11. This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments. (yes)
12. This paper states the number and range of values tried per (hyper-) parameter during the development of the paper, along with the criterion used for selecting the final parameter setting. (yes)

A Appendix

A.1 Computing the Longest Path Lengths

This section describes the algorithms for computing the forward longest path lengths (FLPL) and backward longest path lengths (BLPL) on the reduced TPG. They are simple, so we mainly present the pseudocode.

We first describe the non-incremental version of these two algorithms in Algorithm 5 and Algorithm 6. The input graph G is the reduced TPG of the current STPG, where E contains both Type-1 edges and non-switchable Type-2 edges. The topological ordering, $Ordered(V)$, obtained in the computation of FLPL will also be input to the calculation of BLPL. The returned $L(v)$ and $L(v, g)$ represent the functions of FLPL and BLPL, respectively.

When computing the FLPL in Algorithm 5, we update the $L(v)$ by the predecessors of v (Line 6) in the topological ordering. Similarly, when computing the BLPL in Algorithm 6, we update the $L(v, g^i)$ by the successors of v that are connected to g^i (Line 8), in the reverse topological ordering.

Algorithm 5: Forward Longest Path Lengths

```
1: function FLPL($G = (V, E)$)
2:     Obtain a topological ordering of $G$, denoted as $Ordered(V)$
3:     $L(v) \leftarrow 0$ for $\forall v \in V$.
4:     for $v \in Ordered(V)$ do
5:         Get the predecessors of $v$, denoted as $Pred(v)$
6:         $L(v) \leftarrow \max\{L(u) + 1, u \in Pred(v)\}$
       return $L(v)$,$Ordered(V)$
```

Algorithm 6: Backward Longest Path Lengths

```
1: function BLPL($G = (V, E)$,$Ordered(V)$)
2:     Obtain the reverse ordering of $Ordered(V)$, denoted as $ROrdered(V)$
3:     $L(v, g^i) \leftarrow \infty$ for $\forall v \in V, i \in I$
4:     $L(g^i, g^i) \leftarrow 0$ for $\forall i \in I$
5:     for $v \in ROrdred(V)$ do
6:         Get the successors of $v$, denoted as $Succ(v)$
7:         for $i \in I$ do
8:             $L(v, g^i) \leftarrow \max\{L(u, g^i) + 1, L(u, g^i) \neq \infty, u \in S(v)\}$
       return $L(v, g)$
```

Now, we describe the incremental versions of Algorithm 5 and Algorithm 6 in Algorithm 7 and Algorithm 8 respectively. The adaptation is based on the paper (Katriel, Michel, and Van Hentenryck 2005).

For both algorithms, in addition to the graph G, we also input $Edges$, the set of new edges added to the graph, and $L(v)$, the old FLPL function, which specifies an order of updating. Notably, L actually specifies a topological order on the original graph G if we sort all the nodes by $L(v)$ in the ascending order. This order or its reverse is maintained by a heap in both algorithms. The returned $L'(v)$ and $L'(v, g)$ represent the functions of new FLPL and BLPL, respectively.

When computing the FLPL in Algorithm 7, we update the $L'(v)$ by the predecessors of v (Line 11). If the length changes, we push into the heap all the successors of v that have not been visited for future updating (Line 17). Similarly, when computing the BLPL in Algorithm 8, we update the $L'(v, g_i)$ by the successors of v that are connected to g^i (Line 12). If any length changes, we push into the heap all the predecessors of v that have not been visited for future updating (Line 18).

Algorithm 7: Forward Longest Path Lengths (Incremental)

```
1: function FLPL_INC($G = (V, E)$,$L(v)$,Edges)
2:     Initialize a min-heap $H$
3:     $L'(v) \leftarrow L(v), \forall v \in V$
4:     $visited(v) \leftarrow False, \forall v \in V$
5:     for $e = (v_1, v_2) \in Edges$ do
6:         Push $(L(v_2), v_2)$ into $H$
7:         $visited(v_2) \leftarrow True$
8:     while $H$ not empty do
9:         Pop $(L(v), v)$ from $H$
10:        Get the predecessors of $v$, denoted as $Pred(v)$
11:        $L'(v) \leftarrow \max\{L'(u) + 1, u \in Pred(v)\}$
12:        $visited(v) \leftarrow True$
13:        if $L'(v) \neq L(v)$ then
14:            Get the successors of $v$, denoted as $Succ(v)$
15:            for $u \in Succ(v)$ do
16:                if not $visited(u)$ then
17:                    Push $(L(u), u)$ into $H$
       return $L'(v)$
```

Algorithm 8: Backward Longest Path Lengths (Incremental)

```
1: function BLPL_INC($G = (V, E)$,$L(v)$,Edges)
2:     Initialize a max-heap $H$
3:     $L'(v, g_i) \leftarrow L(v, g_i), \forall v \in V, i \in I$
4:     $visited(v) \leftarrow False, \forall v \in V$
5:     for $e = (v_1, v_2) \in Edges$ do
6:         Push $(L(v_1), v_1)$ into $H$
7:         $visited(v_1) \leftarrow True$
8:     while $H$ not empty do
9:         Pop $(L(v), v)$ from $H$
10:        Get the successors of $v$, denoted as $Succ(v)$
11:        for $i \in I$ do
12:            $L'(v, g^i) \leftarrow \max\{L'(u, g^i)+1, L'(u, g^i) \neq \infty, u \in S(v)\}$
13:        $visited(v) \leftarrow True$
14:        if $\exists i \in I, L'(v, g_i) \neq L(v, g_i)$ then
15:            Get the predecessors of $v$, denoted as $Pred(v)$
16:            for $u \in Pred(v)$ do
17:                if not $visited(u)$ then
18:                    Push $(L(u), u)$ into $H$
       return $L'(v, g)$
```

A.2 The Termination Condition of GSES

This section proves the termination condition of GSES.

Proposition 1. *If there is no conflicting switchable edge found in Line 8 of Algorithm 1, GSES can be terminated by fixing all the switchable edges. It returns an acyclic TPG with the same execution cost as the current reduced TPG.*

If all the switchable edges are not conflicting, then their edge slacks $Sl(e) \geq 0, \forall e \in \mathcal{S}$, where $\mathcal{S}$ is the set of switchable edges in the current STPG $\mathcal{G}^S$. To prove that fixing all the switchable edges will not introduce any cycle and extra cost, we prove the Lemma 2, which considers the case of adding only one edge to an acyclic TPG.

Lemma 2. *Given an acyclic TPG $G = (V, E)$ with the forward longest path lengths as $L(v), v \in V$. If we add to G a new edge $e' = (v_1, v_2)$ with its slack $Sl(e') = L(v_2) - L(v_1) - 1 \geq 0$, then the new graph $G' = (V, E')$ remains acyclic, where $E' = E \bigcup \{e'\}$. Further, it has the same forward longest path lengths as G at each vertex v. Namely, $L'(v) = L(v), v \in V$.*

Proof. By definition, $L(e) \geq 0, e \in E$. Since $L(e') \geq 0$ as well, so we can state that $L(e) \geq 0, e \in E'$.

First, we prove G' is acyclic by contradiction. We assume there is a cycle $v_1, v_2, ...v_n, v_1$. Since for $e_i = (v_i, v_{i+1}), Sl(e) = L(v_{i+1}) - L(v_i) - 1 \geq 0$, we have $L(v_{i+1}) > L(v_i)$. So, $L(v_n) > L(v_1)$. But there is also an edge from v_n to v_1, so we can also get $L(v_1) > L(v_n)$, leading to the contradiction. Therefore, G' must be acyclic.

Since G' is also a direct acyclic graph, we assume we obtain a topological ordering of it in the Algorithm 5, as $Ordered(G')$. Since G' has one more edge than G, $Ordered(G')$ is also a topological ordering of G.

Now, if we run Algorithm 5 for both G' and G with this ordering, every vertex v ordered before v_2 in $Ordered(G')$ must have $L(v) = L'(v)$ because its predecessors and its predecessors' forward longest path lengths are not changed.

Then, we validate $L'(v_2) = L(v_2)$. First, there exists some predecessor u of v_2 such that $L(v_2) = L(u) + 1$. Notably, u and v_1 must be vertices ordered before v_2 in $Ordered(G')$ because they both have an edge pointing to v_2. Thus, $L'(u) = L(u)$ and $L'(v_1) = L(v_1)$. Since $Sl(e') = L(v_2) - L(v_1) - 1 \geq 0$, $L'(u) + 1 = L(u) + 1 = L(v_2) \geq L(v_1) + 1 = L'(v_1) + 1$. Therefore, $L'(u) + 1$ is still the maximum in Line 6 of Algorithm 5, even though a new edge e' is added. So, $L'(v_2) = L'(u) + 1 = L(u) + 1 = L(v_2)$.

Finally, we can deduce every vertex v ordered after v_2 in $Ordered(G')$ must also have $L(v) = L'(v)$ because its predecessors and its predecessors' forward longest path lengths remain the same. Therefore, $L'(v) = L(v), \forall v \in V$. □

Corollary 1. *The execution cost of G' is the same as G.*

Proof. Based on Theorem 2, the execution cost of a TPG is the sum of the longest path lengths at all agents' goals. Since the longest path lengths remain the same, the execution costs are also the same. □

Then, we can prove Proposition 1 easily by fixing switchable edges one by one. Each fixing adds a new edge to the acyclic TPG but does not introduce any cycle or extra cost.

A.3 Greedy Matching Algorithm For EWMVC

This section describes the greedy matching algorithm for the Edge-Weighted Minimum Vertex Cover (EWMVC) problem in Section 4.1. We have a weighted fully-connected undirected graph $G_D = (V_D, E_D, W_D)$, where each vertex $v_i \in V_D$ represents an agent, E_D are edges and W_D are edges' weights. Specifically, we set the weight of each edge $(v_i, v_j) \in E_D$ to be the pairwise cost increase, $\Delta cost(g^i, g^j)$ obtained from our stronger heuristic reasoning. Our target is to assign a cost increase x_i to each vertex v_i such that $x_i + x_j \geq \Delta cost(g^i, g^j)$ so that the overall cost increase $\sum_i x_i$ is minimized.

The pseudocode is illustrated in Algorithm 9. Briefly speaking, the algorithm always selects the max-weighted edge at each iteration, whose two endpoints have never been matched. Then, the weight is added to the overall weight, and the two endpoints are marked as matched. Since each iteration checks all the edge weights and we assume it is a fully connected graph, each iteration takes $O(|V_D|^2)$ time. There could be, at most, $\lceil |V_D|/2 \rceil$ iterations. The worst-case complexity of this algorithm is $O(|V_D|^3)$.

Algorithm 9: Greedy Matching for EWMVC

```
1:  function GreedyMatching($G_D = (V_D, E_D, W_D)$)
2:      $matched[v] \leftarrow False$ for $\forall v \in V_D$
3:      $sumW \leftarrow 0$
4:      while True do
5:          $maxW \leftarrow 0, i \leftarrow -1, j \leftarrow -1$
6:          for $(u, v) \in E_D$ do
7:              if $matched[u] \vee matched[v]$ then continue
8:              if $W_D[u, v] > maxW$ then
9:                  $maxW \leftarrow W[u, v], i \leftarrow u, j \leftarrow v$
10:         if $maxW = 0$ then break
11:         $sumW \leftarrow sumW + maxW$
12:         $matched[i] \leftarrow True, matched[j] \leftarrow True$
        return $sumW$
```

A.4 Experiments

This section explains more details about our benchmark, code implementation, and how to reproduce the experiments. We also illustrate experiment results with different delay probabilities. To be more self-contained, this section may overlap with the main text.

Benchmark The generation process of the benchmark is described in the main text. Notably, our benchmark actually evaluates with as twice many agents as the benchmark used in the paper (Feng et al. 2024). Therefore, We use k-robust PBS rather than k-robust CBS to obtain the initial MAPF solution since the former is much faster than the latter when solving instances with more agents. We run k-robust PBS on 25 instances with evenly distributed starts and goals from the MovingAI benchmark [5] and set the time limit of k-robust PBS to be 6 minutes for each MAPF instance. Each solved instance will generate 6 delay scenarios by simulation. We

[5]https://www.movingai.com/benchmarks/mapf/index.html

Table 2: The number of solved Instances for each map and each agent number.

Random-32-32-10					
#agents	60	70	80	90	100
#solved instances	24	21	25	22	16
Warehouse-10-20-10-2-1					
#agents	110	120	130	140	150
#solved instances	24	23	22	23	20
Lack303d					
#agents	41	49	57	65	73
#solved instances	25	25	25	25	25
Paris_1_256					
#agents	120	140	160	180	200
#solved instances	22	20	20	18	18

report the number of solved instances for each map and each agent number in Table 2.

Code Implementation In this paper, we compare with other three optimal STPG optimization algorithms. We incorporate their open-source implementation into our code.

1. GSES: https://github.com/YinggggFeng/Multi-Agent-via-Switchable-ADG
2. MILP: https://github.com/alexberndt/sadg-controller
3. CBS-D: https:/github.com/aria-systems-group/Delay-Robust-MAPF

All codes are implemented in C++ except MILP, which uses a Python interface to an open-sourced brand-and-bound C++ solver, the COIN-OR Branch-and-Cut solver, at https://github.com/coin-or/Cbc.

Notably, the code can handle non-uniform edge costs, even though we assume all the costs are 1 in the main text.

Reproducibility The supplementary materials include a subset (due to the submission size limit) of benchmark data and all the code, including experiment scripts. Results can be reproduced by running the experiment scripts in the Linux system (e.g., Ubuntu). The readme file explains the details. We will release our code and full data upon acceptance.

Experiment Results We organize all the experiment data in Table 3. Success Rate, Incremental Analysis, and Ablation Study with delay probability $p = 0.01$ have been discussed in the main text. The conclusions are consistent with the ones in the main text across different probabilities.

We further include plots of the mean search time for different maps and agent numbers, which reflects the superiority of IGSES over other optimal algorithms from another view. Of course, with more agents and larger delay probabilities, the search clearly takes more time.

Table 3: Reference to all experiment data.

Delay Probability	$p = 0.002$	$p = 0.01$	$p = 0.03$
Success Rate	Figure 8	Figure 10	Figure 12
Search Time	Figure 9	Figure 11	Figure 13
Incremental Analysis	Table 4	Table 5	Table 6
Ablation Study	Figure 14	Figure 15	Figure 16

Table 4: Incremental analysis on instances solved by all the settings with delay probability $p = 0.002$. 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.

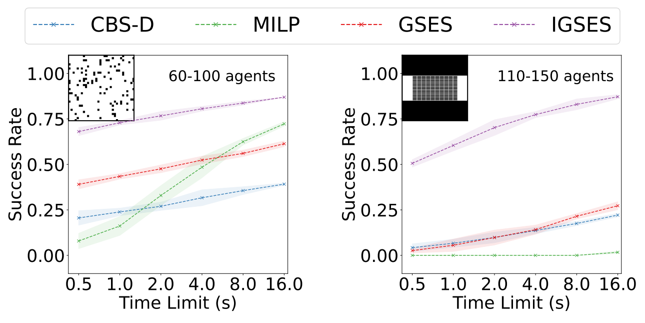
Row	Setting	Random-32-32-10			Warehouse-10-20-10-2-1			Lak303d			Paris_1_256		
		search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups
1	GSES	1.093	2249.8	1164.0	3.013	757.5	14686.1	1.648	333.0	30230.5	4.516	190.5	44884.0
2	+SG	0.840	1475.6	826.7	1.580	285.7	8668.4	0.848	112.2	18938.0	3.292	104.2	30191.6
3	+FG	0.747	1213.1	518.8	1.355	218.4	2200.0	0.704	80.5	5581.3	2.949	83.5	11873.3
4	+FG +RB	1.259	1996.9	518.8	2.226	358.1	2200.0	2.168	246.3	5581.3	4.393	122.1	11873.3
5	+FG +EB	1.054	1755.0	518.8	1.791	291.9	2200.0	1.120	115.2	5581.3	2.896	79.5	11873.3
6	+FG +SB	0.713	1111.7	518.8	1.312	205.0	2200.0	0.726	83.3	5581.3	3.119	88.4	11873.3
7	+FG +SB +SH	0.127	59.9	518.8	0.743	31.6	2200.0	0.701	25.1	5581.3	1.537	13.9	11873.3
8	+FG +SB +SH +INC	0.035	59.9	518.8	0.116	31.6	2200.0	0.207	25.1	5581.3	0.349	13.9	11873.3

Table 5: Incremental analysis on instances solved by all the settings with delay probability $p = 0.01$. 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.

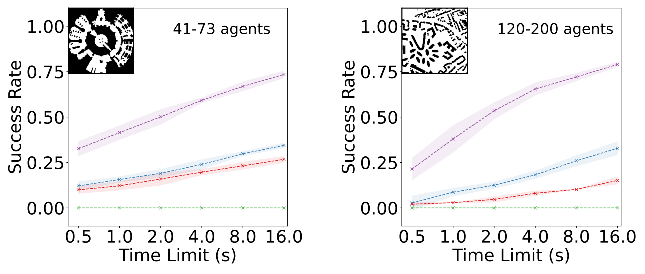
Row	Setting	Random-32-32-10			Warehouse-10-20-10-2-1			Lak303d			Paris_1_256		
		search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups
1	GSES	1.421	3051.5	1637.9	3.805	948.3	15441.6	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 6: Incremental analysis on instances solved by all the settings with delay probability $p = 0.03$. 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.

Row	Setting	Random-32-32-10			Warehouse-10-20-10-2-1			Lak303d			Paris_1_256		
		search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups	search time (s)	#expanded nodes	#edge groups
1	GSES	1.580	3699.3	1564.2	4.210	1080.9	14763.0	2.262	443.3	29775.0	5.457	273.6	39522.9
2	+SG	1.173	2383.1	1124.2	1.959	364.7	8744.3	1.115	153.5	18628.9	3.528	133.5	26525.9
3	+FG	0.996	1824.5	700.0	1.685	277.8	2283.7	0.907	105.7	5492.3	3.011	97.1	10396.3
4	+FG +RB	1.605	2970.8	700.0	2.968	496.8	2283.7	2.397	277.5	5492.3	4.120	133.8	10396.3
5	+FG +EB	1.716	3276.8	700.0	2.534	435.8	2283.7	1.426	170.6	5492.3	3.067	100.2	10396.3
6	+FG +SB	0.736	1317.0	700.0	1.564	253.6	2283.7	0.792	84.2	5492.3	2.680	86.2	10396.3
7	+FG +SB +SH	0.187	96.8	700.0	0.924	40.6	2283.7	0.752	26.3	5492.3	2.080	21.9	10396.3
8	+FG +SB +SH +INC	0.045	96.8	700.0	0.124	40.6	2283.7	0.205	26.3	5492.3	0.337	21.9	10396.3

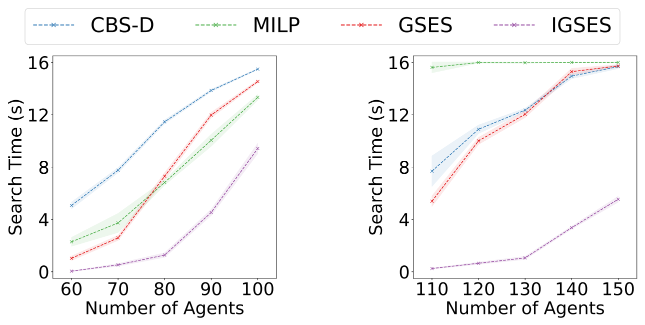


(a) Random-32-32-10 (b) Warehouse-10-20-10-2-1

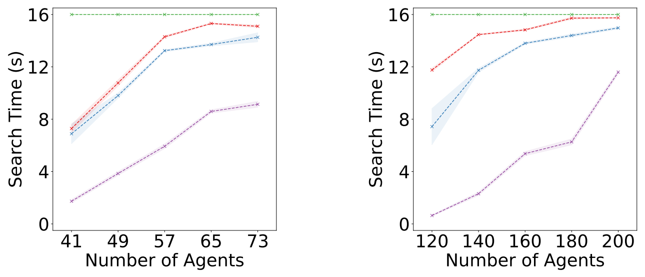


(c) Lak303d (d) Paris_1_256

Figure 8: Success rates of IGSES and other baselines on four maps with delay probability $p = 0.002$. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration. In each figure, the top-left shows the corresponding map, and the top-right corner shows the range of agent numbers.

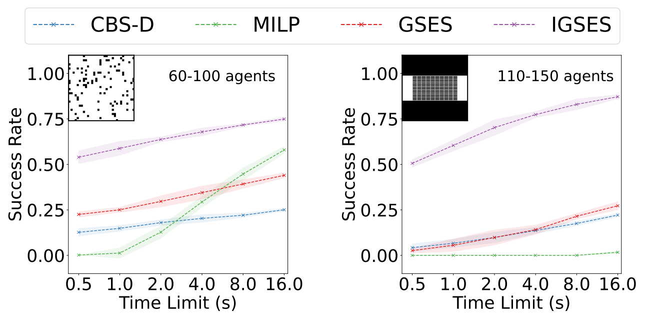


(a) Random-32-32-10 (b) Warehouse-10-20-10-2-1

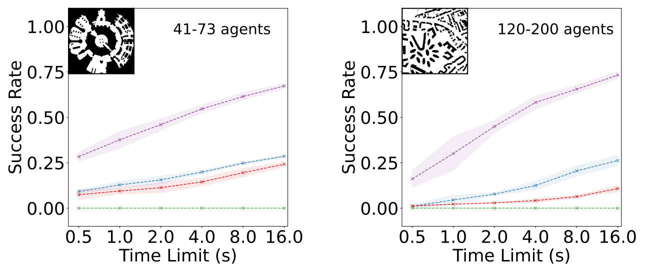


(c) Lak303d (d) Paris_1_256

Figure 9: Search Time of IGSES and other baselines on four maps with delay probability $p = 0.002$. The search time of unsolved instances is set to the time limit of 16 seconds. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration.

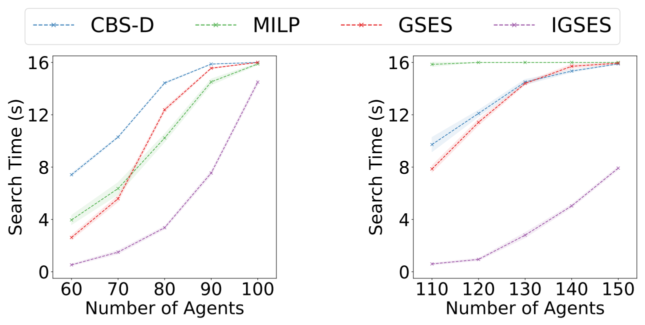


(a) Random-32-32-10 (b) Warehouse-10-20-10-2-1

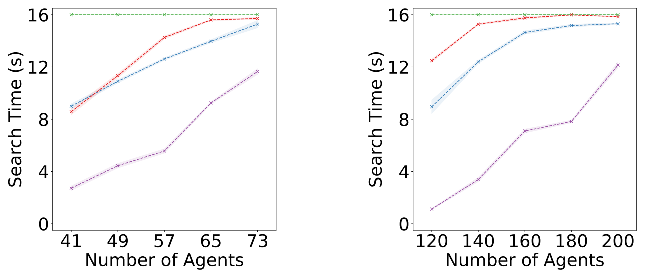


(c) Lak303d (d) Paris_1_256

Figure 10: Success rates of IGSES and other baselines on four maps with delay probability $p = 0.01$. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration. In each figure, the top-left shows the corresponding map, and the top-right corner shows the range of agent numbers.

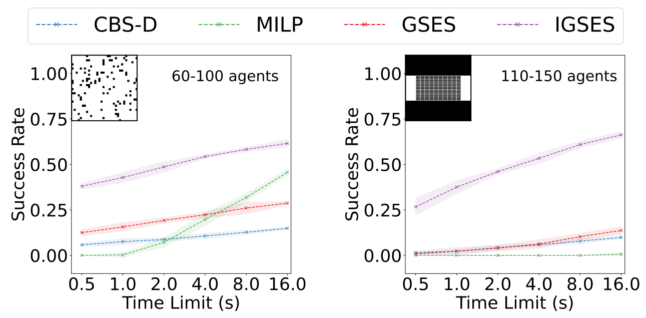


(a) Random-32-32-10 (b) Warehouse-10-20-10-2-1

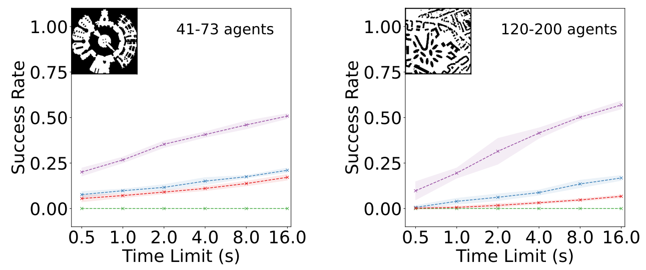


(c) Lak303d (d) Paris_1_256

Figure 11: Search Time of IGSES and other baselines on four maps with delay probability $p = 0.01$. The search time of unsolved instances is set to the time limit of 16 seconds. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration.

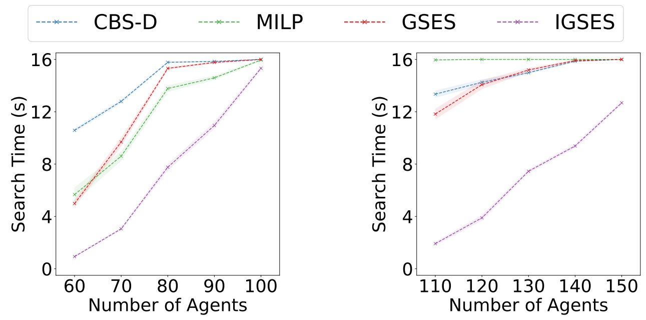


(a) Random-32-32-10 (b) Warehouse-10-20-10-2-1

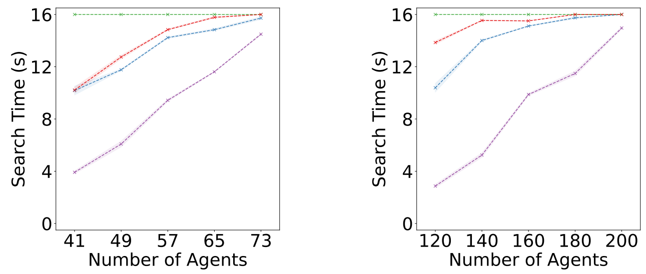


(c) Lak303d (d) Paris_1_256

Figure 12: Success rates of IGSES and other baselines on four maps with delay probability $p = 0.03$. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration. In each figure, the top-left shows the corresponding map, and the top-right corner shows the range of agent numbers.

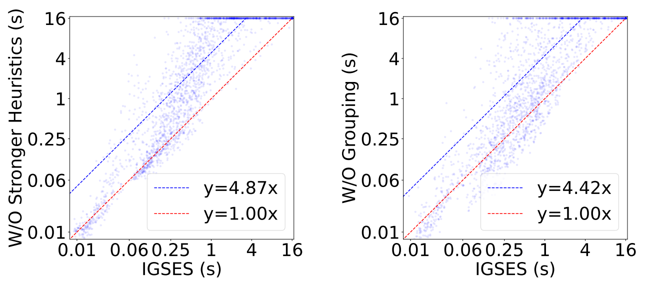


(a) Random-32-32-10 (b) Warehouse-10-20-10-2-1

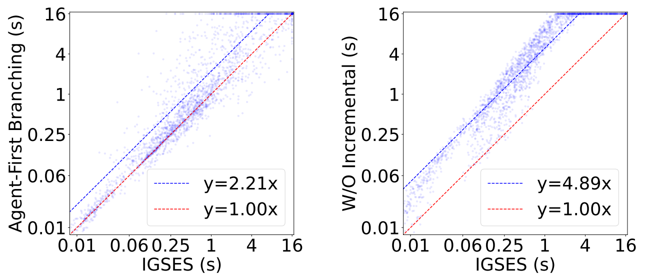


(c) Lak303d (d) Paris_1_256

Figure 13: Search Time of IGSES and other baselines on four maps with delay probability $p = 0.03$. The search time of unsolved instances is set to the time limit of 16 seconds. The shading areas indicate the standard deviations of different runs. They are multiplied by 10 for illustration.

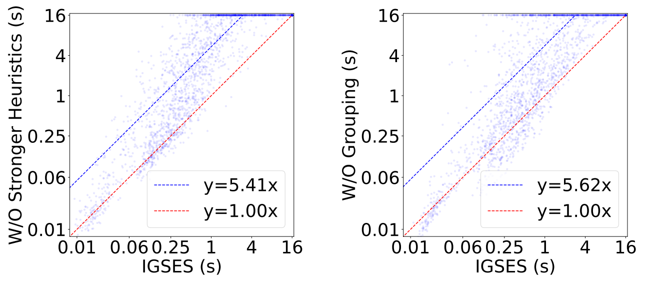


(a) Heuristics (b) Grouping

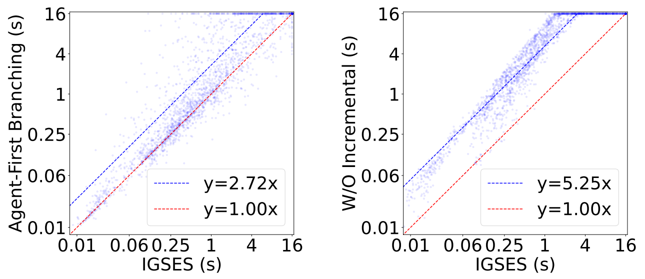


(c) Branching (d) Incremental

Figure 14: Ablation on four speedup techniques on all instances with delay probability $p = 0.002$. In each figure, we compare IGSES to the setting that replaces one of its techniques by the GSES's choice. Each point in the graph represents an instance. The search time of an unsolved instance is set to 16 seconds. The blue line is the fitted on instances solved by at least one setting. Its sloped indicates the average speedup.

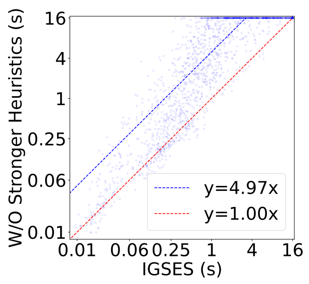


(a) Heuristics (b) Grouping

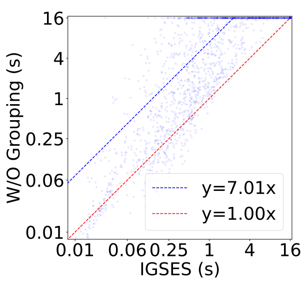


(c) Branching (d) Incremental

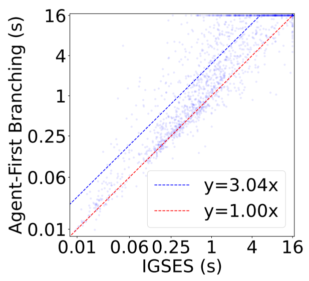
Figure 15: Ablation on four speedup techniques on all instances with delay probability $p = 0.01$. In each figure, we compare IGSES to the setting that replaces one of its techniques by the GSES's choice. Each point in the graph represents an instance. The search time of an unsolved instance is set to 16 seconds. The blue line is the fitted on instances solved by at least one setting. Its sloped indicates the average speedup.



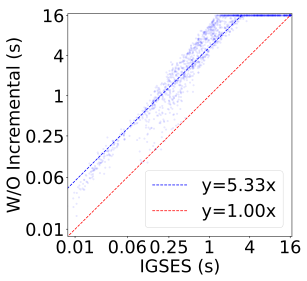
(a) Heuristics



(b) Grouping



(c) Branching



(d) Incremental

Figure 16: Ablation on four speedup techniques on all instances with delay probability $p = 0.03$. In each figure, we compare IGSES to the setting that replaces one of its techniques by the GSES's choice. Each point in the graph represents an instance. The search time of an unsolved instance is set to 16 seconds. The blue line is the fitted on instances solved by at least one setting. Its sloped indicates the average speedup.