P3GASUS: Pre-Planned Path Execution Graphs for Multi-Agent Systems at Ultra-Large Scale

Tanishq Duhan[1], Chengyang He[1], and Guillaume Sartoretti[1]

Abstract—Executing pre-planned paths in multi-agent systems is challenging, as a lack of synchronization can lead to collisions or live-/deadlocks, while enforcing strict synchronization may cause a widespread team delay in reaching goals. An Action Dependency Graph (ADG) solves this problem by identifying and synchronizing only the necessary robots during execution by post-processing all agents' paths into a static directed graph with actions as nodes and edges representing the execution precedence order between actions. However, ADG's creation phase currently requires an exhaustive search of the action space that inflates both computation and communication ($O(R^2T^2)$, where R is the number of robots and T is the maximum path length). This makes ADG the bottleneck for current state-of-the-art MAPF planners, which can solve for up to 10000 agents, and lifelong MAPF, which needs constant replanning during execution. Moreover, this biquadratic scaling also limits the extension of ADG to continuous space scenarios, where high-frequency updates typically effectively result in long path lengths. Addressing these limitations, in this work, we propose three improved execution graphs to cater to different needs and scenarios: SAGE, which adds edges based on the sequence in which robots visit a position; MAGE, which takes the execution graph of SAGE as input and prunes its redundant edges, reducing communication burden during execution; and FORTED which combines both approaches with a reduced complexity of $O(RT)$, making it the overall best in discrete scenarios, trading-off scalability to continuous space scenarios. All three methods achieve speedups of 300-8000 folds over ADG, with MAGE and FORTED reducing communication overhead by more than 14 times. By integrating these methods with a framework for robust distributed path execution for both discrete and continuous scenarios, we introduce P3GASUS, an end-to-end method for pre-planned path execution in multi-agent systems. Finally, we validate the effectiveness of P3GASUS in discrete and continuous multi-agent scenarios using hybrid (real and virtual) teams with up to 3000 robots.

Index Terms—Distributed Robot Systems, Path Planning for Multiple Mobile Robots, Software Architecture for Robotics

Fig. 1: The P3GASUS framework for executing pre-calculated paths for ultra-large robot teams (http://github.com/marmotlab/P3GASUS).

I. INTRODUCTION

In multi-agent systems, reliably executing pre-planned paths on real robot swarms remains a significant challenge due to discrepancies between the planning models and the robots' actual behavior. Even in a simple scenario where agents must sequentially cross an intersection, minor control inaccuracies
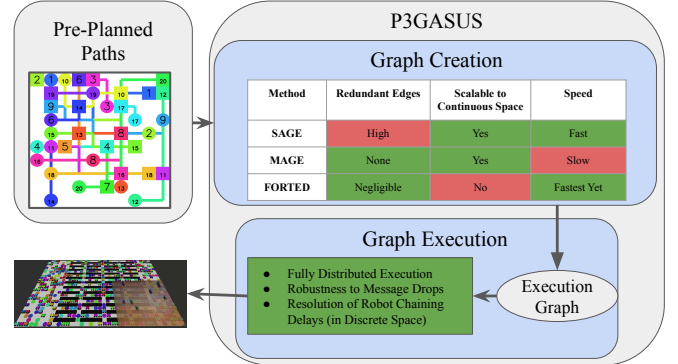
can build up and cause individual agent delays [1]. If not properly addressed, these delays can cascade, leading to collisions, live- or deadlocks among agents, or widespread team delays.

One approach to address this challenge is to translate pre-computed paths into an executable graph, such as the Action Dependency Graph (ADG) proposed by Hönig et al. [2]. ADG preserves the order in which agents cross path intersection points by identifying any critical action one agent must complete before another agent can proceed with its actions. These inter-robot action dependencies are then represented as edges in the ADG, effectively capturing the sequential relationships between actions. Constraining execution according to the ADG ensures the scheduling of interdependent actions, regardless of the execution order of independent actions. The main advantage of ADG is that it pre-calculates the execution graph, which remains unchanged throughout the execution process on robots. As a result, it allows fully parallel, entirely distributed, real-time autonomous execution by robots, where each robot can operate independently, using global communication to convey the completion of inter-robot-dependent actions.

However, creating an ADG remains computationally intensive with a time complexity of $O(R^2T^2)$, where R is the number of robots and T is the maximum of all agent path lengths. This is because ADG relies on a brute-force graph search to identify inter-robot dependencies, overlooking the simple yet crucial insight that dependencies only arise where robot paths intersect. As a result, ADG becomes impractical for the deployment of current state-of-the-art one-shot and lifelong Multi-Agent Path Finding (MAPF) planners, such as LNS2 [3], LaCAM [4], and SILLM [5], which can plan paths for up to 10,000 agents. Moreover, the quadratic path length term (T^2) also limits its applicability to environments with longer paths or continuous motion planning scenarios. The

last major drawback of ADG is the presence of redundant edges, which are edges that can be implicitly inferred through transitive relationships within the graph. Although removing these redundant edges does not affect the overall execution of the plan, it can substantially reduce the communication overhead between robots. These issues may be negligible for small-scale solutions involving a few dozen robots, but they become significant when dealing with even a few hundred, let alone thousands of robots, where computation time and communication overheads can become substantial.

In this paper, we present P3GASUS, an end-to-end framework for executing pre-calculated paths on robots (Figure 1). P3GASUS consists of two steps: converting pre-calculated paths into executable graphs and executing these graphs. To address the first step, we propose three methods, namely: **S**peedy **A**ction **G**raph for **E**xecution (**SAGE**), **M**inimal **A**ction **G**raph for **E**xecution (**MAGE**), and **F**ast **O(RT)** graph for **E**xecution - **D**iscrete (**FORTED**). Each method is designed to cater to different needs and scenarios. Targeting computation efficiency, SAGE stores the positions visited by the agents and adds an edge to the graph when another robot visits the same position in the future. As a result, SAGE generates graphs fully identical to ADG in several orders of magnitude less time. MAGE takes the execution graphs generated by SAGE as input and refines them by eliminating redundant edges that are implicitly represented through transitive relationships within the graph. This refinement leads to a substantial reduction in communication during execution, but at the cost of increased computation time compared to SAGE, though it still achieves a speedup of several hundred times over ADG. While SAGE and MAGE offer versatility by supporting both discrete and continuous scenarios, FORTED trades off this flexibility for the best overall performance in discrete space instances. By exploiting key structural properties unique to discrete MAPF execution graphs, it integrates graph creation and refinement into a single, efficient step, with an $O(RT)$ complexity, dramatically improving upon the $O(R^2T^2)$ time complexity of the original ADG. In practice, this allows FORTED to calculate an execution graph for 10,000 agents, with a path length of 500 each, in *73.79* seconds. In contrast, ADG exceeds the same amount of time for fewer than *160* agents.

However, even with an execution graph, existing methods still encounter real-world issues like computation delays, message drops, and robot chaining, which become especially relevant at large scale. To tackle this, the second stage of P3GASUS focuses on executing the generated graphs using a distributed framework that works for both discrete and continuous scenarios. Finally, we validate P3GASUS across discrete and continuous multi-agent tasks with hybrid real–virtual teams, demonstrating efficient, scalable path execution for warehouses, airports/ports, and last-mile delivery.

II. BACKGROUND AND RELATED WORK

A. Multi Agent Path Finding (MAPF)

In the standard Multi-Agent Path Finding (MAPF) problem [6], a team of robots must navigate from their initial positions to predefined goal locations within a known environment,

typically represented as an undirected graph. The problem assumes that time and space are discretized such that each robot occupies exactly one cell at a time step and that all robots can synchronously and perfectly move to an adjacent cell in any direction. Despite these simplifying assumptions, MAPF remains an NP-hard problem to solve, even for suboptimal solutions. Nevertheless, state-of-the-art MAPF solvers have demonstrated the ability to handle large-scale instances, with some capable of solving problems involving over 10,000 agents [3], [4], [5]. Although the simplifying assumptions of MAPF enable the development of scalable and high-quality solutions, they make the output paths difficult to execute.

B. Execution of a MAPF Plan

To execute MAPF paths, one approach is to adopt the assumptions of MAPF and synchronize action execution, where an agent can only proceed with its action once all previous actions by all agents have been completed: a method referred to as a "Fully Synchronized Policy" (FSP) in MAPF-DP [7] and "ALLSTOP" in RMTRACK [8]. However, this approach has two significant drawbacks: it is inefficient, and fails to guarantee collision avoidance even with perfect synchronization. For instance, if two agents are moving perpendicular to each other, they may still collide at their corners if the planner's assumed cell size is less than $\sqrt{2}$ times the robot size. This requirement for a larger cell size leads to a lower-resolution environmental map, which can render a previously solvable scenario unsolvable.

To solve these problems of ALLSTOP, the authors of RMTRACK [8] and MAPF-POST [9] independently proposed two very different approaches with similar ideas: instead of stopping all agents, stop only the necessary ones. RMTRACK achieved this by defining the concept of homotopy in robot coordination space and proposed a control scheme to ensure that a robot can only start moving to its next waypoint if the path is clear of all collisions in coordination space. In contrast, MAPF-POST proposed a Temporal Plan Graph (TPG), which is used to construct a Simple Temporal Network (STN) that schedules actions based on their precedence order and the robots' speed, defining the earliest possible start time for each.

As a successor to MAPF-POST, ADG [2] introduces two significant improvements. First, it represents the Temporal Planning Graph (TPG) in terms of actions instead of positions (hence the name Action Dependency Graph), allowing a more direct modeling of dependencies between agent actions. Secondly, the ADG is utilized directly for execution, eliminating the need to construct and update an STN at each time step. This leads to a substantial reduction in communication burden compared to MAPF-POST, as agents only need to exchange information upon completing an action rather than continuously broadcasting their positions. This enables the possibility of distributed execution, where agents can be given their portion of the ADG and execute the planned solution robustly and persistently, via global communications.

Recently, a non-peer-reviewed paper [10] proposed ideas similar to those in this work, namely the *Candidate Partitioning (CP)* and *Sparse Candidate Partitioning (SCP)* approaches, which resemble our proposed SAGE and FORTED

frameworks, respectively. However, as we later demonstrate, the methods introduced in this paper are algorithmically more efficient. Another study [11] proposed an $O(RT)$ method to refine MAPF paths by removing redundant wait actions. While effective for reducing collective idle times, its objective is limited to path refinement and does not directly extend to execution-graph construction.

In parallel, a new class of approaches has emerged that builds on ADG by enabling selective edge reordering to improve flexibility during execution, as shown in BTPG [12] and SADG [13]. These methods preserve the overall solution validity while accommodating agent delays, thereby minimizing the need for replanning. By relaxing the strict temporal constraints of the original ADG, they provide a more adaptable and robust framework for path execution. Nonetheless, since these methods are fundamentally derived from ADG, they inherit its core limitations of long computation times and redundant communication overheads. The same constraints also extend to the recently proposed MAPF simulator [14], which relies on ADG as its base structure for constructing execution graphs.

C. Path Planning in Continuous Space

In recent years, the research community has acknowledged the limitations of the traditional Multi-Agent Path Finding (MAPF) problem, prompting a growing interest in addressing multi-agent path planning with relaxed constraints on discrete space and time. This shift has led to the development of new problem formulations, including Multi-Agent Motion Planning (MAMP) [15], [16], Multi-Agent Self Driving [17], and Multi-Agent Trajectory Optimization [18], among others [19]. These formulations treat multi-agent path planning as a continuous optimization problem in space and time, either planning the trajectories of all agents or generating motion signals, such as velocity and acceleration, to guide them to their goals. The use of an execution graph in these methods can facilitate sim-to-real validation for both planned trajectories and motion outputs, as position-based execution graphs are more reliable and robust in handling disturbances when executing pre-calculated plans on robots, even for short windows, compared to relying on recorded motion signals.

A recent work, APEX-MR [20], also explored the concept of using TPG/ADG with multi-robot arms based on their motion plans.

III. PROBLEM STATEMENT

In our problem formulation, a set of robots $R = \{r_0, r_1, ..., r_n\}$ have to get from their current positions $S = \{s_0, s_1, ..., s_n\}$ to pre-defined destinations $D = \{d_0, d_1, ..., d_n\}$. We assume the existence of a planner that can solve this multi-robot path planning problem, providing a standard joint set of actions A for all n robots in the environment. Our objective is to use the joint set of actions A to construct a graph that effectively captures and maintains the sequence of agents executing these actions while ensuring the validity and feasibility of A throughout the execution process.

In the joint action space $A = \{a_0, a_1, ..., a_n\}$, the set of actions for a robot r_i is denoted as $a_i = \{a_i^0, a_i^1, ...a_i^T\}$, and each action a_i^t is characterized by three attributes:

- $start(a_{\mathbf{start},i}^t)$: The position of the robot where the action starts.
- $goal(a_{\mathbf{goal},i}^t)$: The target position of the robot after action execution.
- $time(a_i^{\mathbf{t}})$: The planned start time associated with the action's operation.

We consider A to be a solution of the MAPF problem if and only if $\forall r_i \in R, a_{start,i}^0 \equiv s_i \wedge a_{goal,i}^t \equiv d_i$, and there are no edge or vertex collisions, i.e. $\forall r_i, \forall r_j, \nexists a_{start,i}^t = a_{start,j}^t$, and $\forall r_i, \forall r_j, \nexists a_{start,i}^t = a_{goal,j}^t$ & $a_{start,j}^t = a_{goal,j}^t$

In the discrete case, we introduce an additional assumption that the solved joint set A does not contain cycle conflicts [6].

IV. GRAPH STRUCTURE

The final execution graph G is a directed acyclic graph, which has actions as nodes and the sequence of execution as edges. If there exists an edge from action a_i^t to action $a_j^{t'}$, G ensures that $a_j^{t'}$ happens only after a_i^t has been completed, hence a_i^t is called a **dependency of** action $a_j^{t'}$, or conversely $a_j^{t'}$ is **dependent on** a_i^t.

There exist three types of edge dependencies in G. $Type1$ are intra-robot dependencies, enforcing the fact that an action cannot be executed by a robot until all its previous actions have been executed in sequence. This means that to ensure continuity,

If a_r^{t-1} is a *Type1* dependency of a_r^t,

$$\text{then } a_{goal,r}^{t-1} = a_{start,r}^t \quad (1)$$

$Type2$ are inter-robot dependencies needed to conserve the precedence order of the original solution. For two robots r_1 and r_2,

If $\Big($distance$(a_{start,r_1}^{t_1}, a_{goal,r_2}^{t_2}) < \theta$ (for continuous case)

or $a_{start,r_1}^{t_1} = a_{goal,r_2}^{t_2}$ (for discrete case)$\Big)$ and $t_1 \leq t_2$,

$$\text{then } a_{r_1}^{t_1} \text{ is a *Type2* dependency of } a_{r_2}^{t_2}. \quad (2)$$

The parameter θ denotes the minimum allowable distance between robots to prevent collisions. Although θ can be theoretically calculated from the physical dimensions and dynamics of the robots, in practice, we determine it empirically by iterating through all time steps in the set A and identifying the minimum distance between any two robots at all time steps.

The validity of Equation 2 stems from the fact that r_2 is trying to occupy the position currently held by r_1. This means that $a_{r_1}^{t_1}$ must be completed first, vacating the position, before r_2 can execute $a_{r_2}^{t_2}$ to reach that position.

In the continuous case, we assume that all positions along each agent's path are sufficiently dense in both space and time, ensuring that all potential collision scenarios are effectively captured by Equation 2. Sparse sampling conditions are beyond the scope of this work, but are briefly discussed in Section VIII.

$Type3$ is a special type of $Type2$ edge dependency that arises in discrete space in cases where two robots follow

---

Algorithm 1 Pseudo-code for SAGE

1: **Input:** Joint Set of Actions, A
2: **Output:** Constructed graph G_{SAGE}
3: $goalStore \leftarrow$ list // For continuous spaces
4: $M \leftarrow$ hashMap() // For discrete spaces
5: /* Create the skeleton of G_{SAGE} with $Type1$ edges and store goal positions*/
6: **for** each action $a_i^t \in A$ **do**
7:      Add node a_i^t to G_{SAGE}
8:      **if** $t \geq 1$ **then**
9:          Add edge(a_i^{t-1}, a_i^t) to G_{SAGE}
10:      **end if**
11:      **if** continuous space **then**
12:          Add a_i^t to $goalStore$
13:      **else**
14:          Add a_i^t to $M[a_{goal,i}^t]$
15:      **end if**
16: **end for**
17: **if** continuous space **then**
18:      /* create KDT from stored goal positions */
19:      $KDT \leftarrow KDTree(goalStore)$
20: **end if**
21: /* Add all $Type2$ dependencies */
22: $taskQueue \leftarrow$ all $a_i^t \in A$, in ascending order of t
23: **while** $taskQueue$ is not empty **do**
24:      **for** all actions a_i^t with the same time t **do**
25:          **if** continuous space **then**
26:              $allPossibleDepend \leftarrow KDT.query(A_{start,i}^t, \theta)$
27:          **else**
28:              $allPossibleDepend \leftarrow M[a_{start,i}^t]$
29:          **end if**
30:          **for** each action $a_j^{t'} \in allPossibleDepend$, in ascending order of time t **do**
31:              **if** $t < t'$ AND there is no edge from a_i^t to robot j **then**
32:                  Add edge($a_i^t, a_j^{t'}$) to G_{SAGE}
33:              **end if**
34:          **end for**
35:      **end for**
36:      **for** all actions a_i^t with the same time t **do**
37:          Remove a_i^t from $taskQueue$ and $M[a_{start,i}^t]$
38:      **end for**
39: **end while**
40: **return** G_{SAGE}

---

each other and is only used to handle such cases separately during execution to avoid unnecessary delays. This distinction in execution is explained further in Section VII-A.

V. GRAPH CREATION

A. *Speedy Action Graph for Execution (SAGE)*

SAGE converts pre-planned paths into execution graphs identical to those generated by ADG, but at a significantly reduced computational cost. As illustrated in Equation 2, edge dependencies are added solely based on positions visited by a robot. Using this insight, SAGE reduces its computational time by distilling the search space for dependencies from the entire action space to a much smaller subset, as shown in Alg 1.

SAGE operates in two phases. First, it constructs a skeleton of the execution graph by adding all actions as nodes and all *Type1* edge dependencies between nodes belonging to the same robot while storing their goal positions. The idea behind storing goal positions is that for each action, M now contains all actions that satisfy the condition given by Equation 2. This step enables SAGE to reduce the search space for *Type2* edge dependencies of each action from the entire joint set of actions to only the actions that have the same goal position as the start position of that action. For discrete cases, it is sufficient to store all the actions in a position-based dictionary/HashMap. However, for continuous spaces, we employ K-Dimensional Trees (KDT) [21], a space-partitioning data structure that facilitates an efficient search of points within a specified distance θ from a given query point.

Algorithm 2 Pseudo-code for MAGE.

1: **Input:** Graph $G_{SAGE/FORTED}$, which may contain redundant edges
2: **Output:** Minimal graph G_{MAGE}
3: $G_{MAGE} \leftarrow G$
4: $DP \leftarrow$ 2-D matrix with dimensions ($R \times T, R \times T$)
5: **for** each robot's first node n **do**
6:      /* Use Algorithm 3 to trim the graph */
7:      reduceGraph(G_{MAGE}, n, DP)
8: **end for**
9: **return** G_{MAGE}

In the second phase, SAGE iterates over the joint set of actions in ascending time order, with any order being acceptable for actions that share the same time step, adding all *Type2* edge dependencies for each action. If an edge dependency already exists from an action to a robot, a prior action of the robot is already dependent on that action, so no new edge is needed, otherwise, a new edge is added.

The worst-case time complexity of SAGE in discrete space is $O(RT^2)$, where R is the number of robots and T is the number of time steps. In practice, however, the graph density is much lower than in the worst case, resulting in performance comparable to FORTED's $O(RT)$, as discussed in Sec VI-A.

While CP-ADG [10] is conceptually similar, SAGE removes agents from storage M (lines 36–38, Algorithm 1), requiring only $T(T+1)/2$ checks per position instead of T^2, compounding to $RT(T + 1)/2$ total checks instead of RT^2, yielding a consistent $2\times$ runtime improvement. Furthermore, SAGE verifies the existence of an edge before adding a *Type 2* edge (line 31), a step omitted in CP-ADG, resulting in CP-ADG having even more redundant edges than the original ADG.

In the continuous case, the time complexity for constructing the KD-tree is $O(RT \log(RT))$, where $n = R \cdot T$ is the number of points. The range query time complexity for one point in a 2-D KD tree is $O(\sqrt{RT} + m)$, where m is the number of found points. Since SAGE queries a total of RT points, the total time complexity is $O(RT(\sqrt{RT} + m))$.

B. *Minimal Action Graph for Execution (MAGE)*

MAGE takes an execution graph generated by SAGE or FORTED as input and refines it into a minimal communication graph. To understand how MAGE works, it is essential to define a redundant edge in the context of a directed execution graph G. An edge e^{ij} from node n^i to node n^j is considered redundant if, even after its removal, a path from n^i to n^j still exists in G. This means that the removal of e^{ij} does not affect the sequence of action execution, as the presence of an alternative path ensures that the action corresponding to n^i is still performed before n^j. While the concept of transitive edge reduction in execution graphs was first introduced in MAPF-DP [7], implementing it with existing transitive reduction algorithms is non-trivial. MAGE provides the first complete implementation for removing only transitive *Type2* edges while preserving *Type1* edges.

MAGE aims to remove all redundant edges from a given graph. As shown in algorithm 2, MAGE does it using the memoization method of Dynamic Programming. The process begins with the construction of a 2D matrix, DP, with dimensions $RT \times RT$, with R as the number of robots and T as the number of timesteps without wait actions. For two

---

Algorithm 3 `reduceGraph` Function for MAGE

1: **Input:** Incomplete Graph G_{MAGE}, node n, and matrix DP
2: **Output:** Populated array $DP[n]$, while reducing G_{MAGE}
3: /* (**OR**) is the bitwise OR operation */
4: **if** $DP[n][n] = 0$ **then**
5: $\quad DP[n][n] \leftarrow 1$
6: $\quad directEdges \leftarrow$ list of all nodes with a direct edge from n, in the ascending order of time
7: $\quad$ **if** $Type1$ dependency (n') exists in $directEdges$ **then**
8: $\quad\quad DP[n] \leftarrow DP[n]$ (**OR**) `reduceGraph`(G, n', DP)
9: $\quad\quad$ Remove n' from $directEdges$
10: $\quad$ **end if**
11: $\quad$ **for** each node n'' in $directEdges$ in order of ascending time **do**
12: $\quad\quad$ **if** $DP[n][n''] = 1$ **then**
13: $\quad\quad\quad$ Remove the direct edge from n to n'' in G
14: $\quad\quad$ **else**
15: $\quad\quad\quad DP[n] \leftarrow DP[n]$ (**OR**) `reduceGraph`(G, n'', DP)
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: **end if**
19: **return** $DP[n]$

---

nodes n^1 and n^2 in the graph G,

$$DP[n^1][n^2] = \begin{cases} 1 & \text{if there is a path from } n^1 \text{ to } n^2 \text{ in } G, \\ 0 & \text{else} \end{cases}$$

$DP[n^1]$ is a binary array that represents the reachability of all nodes in the graph G from a given node n_1. This binary array would require approximately 20 GB of memory for an instance of 4000 robots with a path length of 100 each. To mitigate this issue, it is possible to store the matrix on-disk and import only a few rows at a time.

MAGE uses the `reduceGraph` function (outlined in Algorithm 3) to distill the original graph G into its minimal form. This function works by removing redundant edges from the graph while also keeping track of the nodes that each edge visits. The process starts by checking if a node has already been visited. If it has, the function uses the pre-computed information to avoid redundant checks. If the node has not been visited before, the function makes a list of all the edges that start from that node, sorted in order of time. It then goes through each edge on the list, visiting its corresponding child node. If the edge is of *Type1*, it is considered essential and must be retained in the resulting graph. In this case, the function updates the set of nodes visited by the current node to include all nodes visited by the child node. If the edge is of *Type2*, the function checks whether the child node has already been visited. If it has, the edge is deemed redundant and is discarded. If not, the nodes visited by the child node are added to the current node's visited set. This process continues until all edges have been processed. The function then returns the updated set of nodes visited by the current node. By applying this procedure to the starting node of each robot, MAGE systematically explores the entire graph and removes all unnecessary edges, effectively constructing a minimal execution graph.

As MAGE takes a graph as input and returns a transitive reduction of the graph, MAGE's algorithm remains the same for both continuous and discrete cases and has a worst time complexity of $O(R^2T^2)$. In practice, however, the use of a memoized DP matrix drastically reduces the number of required operations, and MAGE empirically runs several orders of magnitude faster than ADG.
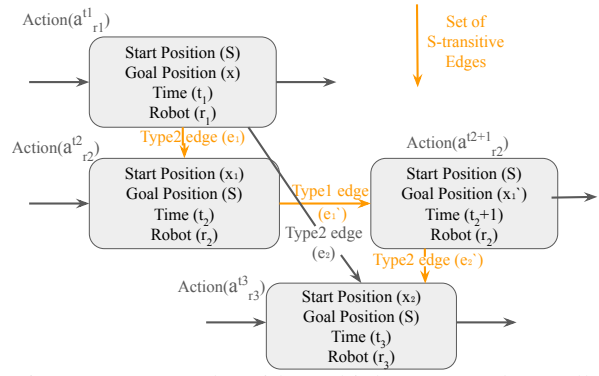

Fig. 2: For a node with multiple *Type2* edges, all but the earliest (e_1) are redundant due to S-transitive (orange) edges.

C. Fast *O(RT)* graph for Execution - Discrete (*FORTED*)

FORTED stands out as the top performer in discrete space due to its reliance on one key insight: the identification and removal of S-transitive redundancies. The concept of S-transitive redundancies is defined as follows:

Theorem 1. *Each node in the execution graph should have at most one Type2 edge. If the node has multiple Type2 edges, all except the chronological earliest are redundant.*

Proof: Consider a scenario, as shown in Figure 2, where action node $a_{r_1}^{t_1}$ is is a *Type2* dependency of action nodes $a_{r_2}^{t_2}$ and $a_{r_3}^{t_3}$ (with $t_1 < t_2 < t_3$) via edges e_1 and e_2, respectively.

From Equation 2, we deduce that $a_{start,r_1}^{t_1} = a_{goal,r_2}^{t_2} = a_{goal,r_3}^{t_3}$, implying that a robot r_1 occupies a position, say S, while robots r_2 and r_3 will arrive there in the future. Since $t_2 < t_3$, robot r_2 must vacate this position, ensuring that there is always a successor action node, $a_{r_2}^{t_2+1}$, connected to $a_{r_2}^{t_2}$ with a *Type1* dependent edge e_1', where $a_{goal,r_2}^{t_2} = a_{start,r_2}^{t_2+1}$ (from Equation 1). Now, to guarantee that robot r_2 vacates the position before robot r_3 arrives, the condition $t_2 + 1 \le t_3$ must hold. This leads to two key conditions:

$$a_{goal,r_2}^{t_2} = a_{start,r_2}^{t_2+1} = a_{goal,r_3}^{t_3} \quad \text{and} \quad t_2 + 1 \le t_3.$$

From these, Equation 2 implies the existence of a *Type2* edge e_2' from $a_{r_2}^{t_2+1}$ to $a_{r_3}^{t_3}$. Consequently, the ordered set of edges e_1, e_1', and e_2', named S-transitive edges, always ensures a path from $a_{r_1}^{t_1}$ to $a_{r_3}^{t_3}$, rendering edge e_2 a redundant edge. $\quad\square$

This insight, along with an optimized implementation, is presented in Algorithm 4. FORTED is similar to SAGE's algorithm for discrete space, but since only one *Type2* dependency per node is relevant, M can be simplified to be a dictionary that stores a single action instead of an array of actions.

After constructing this dictionary, the algorithm iterates through all joint actions in ascending time. For each action, the initial steps mirror those of SAGE's first phase: the action is added as a node, and corresponding *Type1* dependencies are established. The procedure then diverges from SAGE. When a position already exists in the dictionary, it indicates that another robot has previously occupied it, prompting the addition of a *Type2* edge between the stored and current action. Additionally, according to Theorem 1, since only one *Type2* edge per node is necessary, the stored action can be safely replaced with the new one. Once the iteration completes, the

---

Algorithm 4 Pseudo-code for FORTED

1: **Input:** Joint Set of Actions, A
2: **Output:** Constructed graph G_{FORTED}, free from s-transitive redundancy
3:   $M \leftarrow$ hashMap()
4: **for** each action $a_i^t \in A$, in ascending order of time t **do**
5:      /* Create the skeleton of the graph */
6:      Add node a_i^t to G_{FORTED}
7:      /* Add *Type1* edges if applicable */
8:      **if** $t \geq 1$ **then**
9:         Add edge(a_i^{t-1}, a_i^t) to G_{FORTED}
10:     **end if**
11:     /* Add *Type2* edges if applicable */
12:     **if** $a_{start,i}^t$ exists in M **then**
13:        Add edge($M[a_{start,i}^t], a_i^t$) to G_{FORTED}
14:     **end if**
15:     $M[a_{start,i}^t] \leftarrow a_i^t$
16: **end for**
17: /* Handle final actions which were not covered*/
18: **for** each action $a_i^{T_{final}} \in A$ **do**
19:     **if** $a_{goal,i}^{T_{final}}$ exists in M **then**
20:        Add edge($M[a_{goal,i}^{T_{final}}], a_i^{T_{final}}$) to G_{FORTED}
21:     **end if**
22: **end for**
23: **return** G_{FORTED}

---

algorithm addresses the remaining edge cases: dependencies arising from goal positions at the final timestep, which were not previously considered due to the focus on start positions.

As its name suggests, FORTED is an $O(RT)$ method for constructing execution graphs. Unlike SAGE, each node in FORTED can have only one *Type2* dependency, limiting the total number of edges to $O(RT)$. Moreover, the entire execution graph is constructed within a single loop, making FORTED more efficient than the SCP algorithm proposed in [10], which operates in $O(RT \log(RT))$ time despite employing a similar idea. However, FORTED attains its $O(RT)$ efficiency by exploiting S-transitive dependencies. As such dependencies cannot be guaranteed in continuous space, FORTED is only well-defined in discrete space.

VI. RESULTS

We ran experiments on an AMD Ryzen 7 5800X (8-core) using Python 3.11, with Ray-enabled parallelism to run concurrent tests uniformly across all baselines.

A. Discrete Space Instances

To ensure consistency, we report average results over 100 scenarios, with all methods, namely ADG, SAGE, MAGE, and FORTED, using identical paths. These paths are generated using the LaCAM planner [4] in environments with agent densities between 56% and 70%. The resulting paths served as input for ADG, SAGE, and FORTED, which produced execution graphs and associated processing times. For MAGE, we derived its execution graph from FORTED's output and calculated its total processing time as the sum of FORTED's processing time and MAGE's additional processing overhead.

Figure 3 represents the trends of the proposed methods, with the left axis and solid lines representing the time taken for computation and the right axis and dashed lines representing the communication burden. Since ADG's communication burden is equivalent to SAGE's, the dotted blue line for SAGE's communication also serves as a proxy for ADG.

Table I compares computation times across all four methods, showing that ADG requires $300\times$ to $8,000\times$ more time than

Number of Agents	Time taken to calculate graph (in seconds)			
	ADG	**SAGE**	**FORTED**	**MAGE**
100	20.28	0.03	**0.02**	0.11
200	215.48	0.13	**0.08**	0.77
300	824.93	0.28	**0.17**	2.54
400	1807.05	0.46	**0.27**	5.72
500	3730.34	0.71	**0.44**	11.56
...	...	...	...	...
10,000	306546236(est.)	193.54	**73.79**	N/A

TABLE I: Graph generation times of ADG [2] and our three proposed methods (SAGE, MAGE, and FORTED), with different team sizes. For ultra-large-scale systems with 10,000 agents, ADG's estimated computation time is approximately 9.72 years (extrapolated from measurements with up to 500 agents), whereas the proposed methods require only a few minutes, corresponding to a potential speedup of several million times.
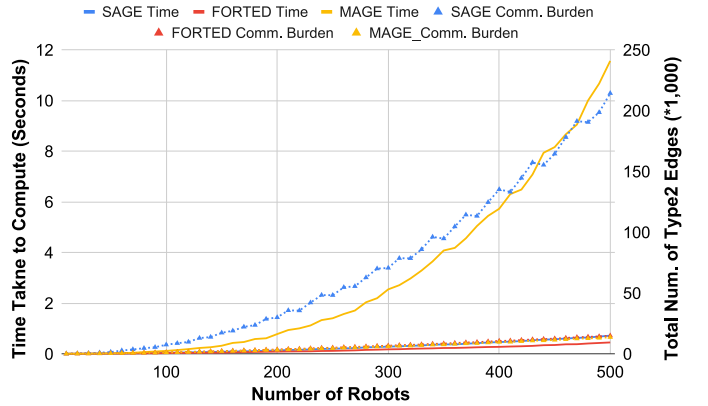
Fig. 3: The computational time (left axis, solid lines) and communication burden (right axis, dotted lines) of SAGE, MAGE, and FORTED. FORTED exhibits the fastest computation time, with a slightly higher communication burden than MAGE.

the proposed methods. This speedup stems from the exponential reduction in operations needed to construct execution graphs. For instance, in a scenario with 100 robots and paths of 24 steps, ADG requires approximately 4,319,078 operations, whereas MAGE (+FORTED) needs 17,653, SAGE 10,600, and FORTED just 6,400. These values are consistent with the speedup observed in Table I. Among the proposed methods, FORTED performs best overall, being 40% faster than SAGE while producing only 10% more edges than MAGE. Figure 3 also shows that in practice, SAGE takes only slightly more time than FORTED despite its theoretical $O(RT^2)$ complexity. Although MAGE scales as $O(R^2T^2)$, its use of a memoized DP matrix trades memory for speed, achieving over $300\times$ speedup relative to ADG and producing roughly $16\times$ fewer *Type2* edges for a 500-robot scenario, as reported in Table I.

B. Continuous Space Instances

We use CoPo [17] to solve randomly generated one-shot metadrive [22] intersection scenarios. Since CoPo is a motion planner, it generates motion commands that we convert to position commands by recording positions at a rate of X Hertz, as described in Section IV. In practice, we vary agent counts from 10 to 40 agents, limited by the number of agents that can fit in the intersection, and vary X from 10 to 100 Hertz. To get a baseline, we adopt the ADG/APEX-MR [20]

Agents	X	Avg. Path Length	Time (seconds)			Number of Comms	
	(Hz)	(TimeSteps)	ADG	SAGE	MAGE	ADG/SAGE	MAGE
10	10	1563.19	4170.76	**17.23**	20.86	4561.07	**2557.88**
10	20	1359.58	2542.91	**9.49**	11.34	5983.57	**3260.20**
10	25	1549.13	2476.16	**7.16**	9.20	7352.48	**3912.82**
10	50	3846.52	15 500.17	**42.61**	49.22	15 862.63	**8405.68**
10	100	4995.63	26 621.78	**51.88**	61.87	28 743.00	**14045.72**
20	10	1373.44	5957.83	**10.95**	25.56	12 711.75	**4213.59**
20	20	2034.77	23 916.36	**41.74**	55.81	23 860.35	**7434.63**
20	25	1967.95	16 051.98	**24.23**	33.61	29 215.50	**8654.43**
20	50	3617.04	45 778.00	**51.02**	69.55	54 688.38	**16521.31**
20	100	6453.18	139 208.00	**123.39**	175.75	103 468.70	**31529.30**
30	10	2731.92	34 334.92	**46.16**	135.79	33 167.87	**7773.07**
30	20	3176.65	59 532.34	**67.07**	134.54	65 481.75	**12729.03**
30	25	3461.46	62 446.41	**62.64**	123.10	76 685.61	**15310.69**
30	50	5253.52	186 122.70	**147.61**	247.80	143 026.90	**27856.07**
30	100	8876.68	475 025.70	**268.48**	496.00	258 935.30	**52988.32**
40	10	1690.18	27 983.08	**33.43**	110.93	61 579.17	**9079.08**
40	20	2439.12	72 124.55	**60.57**	118.66	113 935.60	**16232.19**
40	25	3462.32	95 769.02	**71.15**	175.61	136 014.20	**20512.70**
40	50	5135.14	322 997.60	**201.07**	364.29	262 570.10	**37897.46**
40	100	7946.83	819 503.90	**382.89**	672.61	481 100.10	**71819.22**

TABLE II: Results in continuous scenarios with varying agents and decision rates show that SAGE is over $2,000\times$ faster than ADG for the 40-100 case. MAGE reduces communication by $7\times$ compared to ADG and SAGE, while being $1,200\times$ faster than ADG.

approach, which involves iterating over four nested loops. This baseline is then compared to both SAGE and MAGE. Similar to Section VI-A, the recorded MAGE's time is the sum of SAGE's time, in addition to MAGE's own computation time.

The comparison between the time taken and the communication burden for all methods is shown in Table II. The results are compared on the same 50 scenarios for a particular agent-X scenario, except when the time taken by ADG was more than 100,000 seconds, in which case only 20 scenarios are considered. The comparison shows that SAGE computes the execution graph a few hundred to a few thousand times faster than ADG, while MAGE refines the graph to contain fewer than 1/7 times as many edges.

VII. REAL-ROBOT EXPERIMENTS

This section details the practical considerations for running P3GASUS on large-scale multi-agent teams, with full experiment videos in the supplemental material. Building on ADG's execution scheme, we extend it to address challenges unique to executing pre-planned paths in large-scale discrete and continuous settings.

A. Discrete Space Instances

The execution starts when a central node takes the start position, the goal positions, and the environment map to generate a plan for all agents. The central node then uses this path to generate an execution graph for all agents. Relevant parts of this graph, like an individual robot's actions and inter-robot dependencies, are communicated to all robots. Since message drop may occur, we employ the popular computer network paradigm of message acknowledgments. Each message published waits for an acknowledgment, and if an acknowledgment is not received, the message is re-published after a given timeout. Only after the central node receives acknowledgments for all messages are robots allowed to start their action execution.

Robots have a look-ahead-like controller that enqueues all actions aligned with the current direction of motion, provided they are not blocked by inter-robot dependencies. This ensures a smooth robot motion without having to accelerate and decelerate after each action execution. If a robot encounters a change in its direction of motion in its plan, it completes all enqueued actions before enqueuing actions in the new direction. If an action has incoming *Type2* edges, it is only enqueued after all its dependency actions have been completed. Since this check depends on the number of *Type2* edges in the graph, edge reduction via MAGE or FORTED also improves centralized execution by reducing runtime checks. If a robot completes an action with outgoing *Type2* edges, it publishes a global message to all dependent robots, and parallely continues executing its enqueued actions while awaiting acknowledgments. Similar to the central node, the robot can retransmit a specific message for another robot if it does not receive an acknowledgment within a timeout period.

While this architecture helps prevent robot collisions by allowing robots to enqueue their actions only upon the full completion of their dependent action, it limits the ability of robots moving in the same direction to synchronize effectively. To elaborate, consider 100 robots moving one step forward in a straight line. Due to the *Type2* edges linking each robot's movement to that of the one ahead, each robot can only move after the preceding one has finished its move. As a result, a motion that should ideally be completed in a single timestep is now stretched over 100 timesteps. To address this inefficiency, we propose a new type of edge dependency, termed *Type3*, specifically designed to resolve such chaining issues. A *Type3* edge is a specialized form of a *Type2* dependency, occurring when two adjacent robots move in the same direction. In this scenario, P3GASUS switches to a MAPF-POST [9] inspired position broadcasting method. In the case of a *Type3* dependency, the leading robot continuously broadcasts its coordinates to the trailing robot via local communication, allowing the latter to follow closely, while maintaining a predefined safety distance. Since *Type3* dependencies are restricted to adjacent robots, an action can have at most one such edge and therefore only needs to transmit its coordinates to a single following robot. In longer chains, each robot sends its position data only to its immediate successor, creating a communication cascade down the line. Given the high frequency of these broadcasts, occasional message loss has a negligible impact on the chain's overall performance.

As illustrated in Figure 4, we evaluate the effectiveness of our architecture by deploying a pre-planned path on a mixed team of 292 virtual and 8 real robots within a warehouse environment. For precise localization, ground truth positions are used for the virtual robots, while the real robots rely on external positioning. By design, real and virtual robots have different velocities, and our execution framework effectively handles such motion inaccuracies and delays, enabling robust and collision-free execution. We have tested our framework for teams of up to 3000 robots in simulation, with the video included in the supplementary material.

B. Continuous Space Instances

In continuous space, the execution architecture retains a central node for graph computation while robots execute
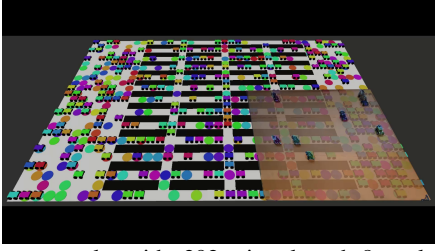
Fig. 4: An example with 292 virtual and 8 real robots in a discrete MAPF environment executing their paths using the proposed communication and execution scheme.
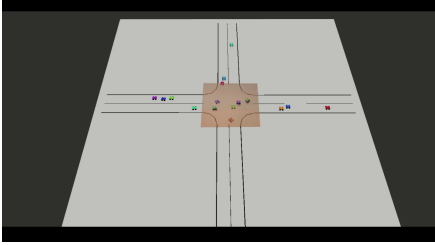

Fig. 5: An example of 8 real and 8 virtual robots in a continuous space instance, executing their preplanned paths using our proposed communication and execution scheme.

decentralized actions. Global communication via messages and acknowledgments remains unchanged, but local communication is unnecessary due to the absence of *Type3* edges.

For robot execution, the lookahead controller is modified to enqueue up to x future actions that are free of inter-robot dependencies. In practice, x is typically set between 3 and 5, striking a balance between ensuring smooth motion and minimizing deviation from the planned trajectory. The rest of the execution scheme remains consistent, with each robot publishing messages to its dependent peers upon completing a *Type2* action. As shown in Figure 5, we deployed a policy generated by CoPo on a hybrid team of 8 real and 8 virtual robots in an intersection environment to validate our framework.

VIII. CONCLUSION AND FUTURE WORK

In this work, we presented an end-to-end method for executing pre-planned paths on real robots, introducing three improved techniques to dramatically speed up the generation of execution graphs, along with a distributed execution framework for deploying these graphs on real robot teams. Our results demonstrate that the proposed graph construction methods are several orders of magnitude faster than the current state-of-the-art, while also significantly reducing the communication overhead.

Future work will aim to further relax certain assumptions to improve the robustness of our framework. In discrete space, this involves addressing cycle conflicts, while in continuous space, we plan to relax the dense spatial and temporal sampling assumption by interpolating additional points between action nodes to ensure adequate density even in sparse scenarios. Additionally, we aim to investigate AI-driven communication strategies to enable deployments in communication-denied environments.

REFERENCES

[1] J. Blumenkamp, S. Morad, J. Gielis, Q. Li, and A. Prorok, "A framework for real-world multi-robot systems running decentralized gnn-based policies," in *Proc. of ICRA 2022*, pp. 8772–8778.

[2] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Persistent and robust execution of mapf schedules in warehouses," *IEEE Robotics and Automation Letters*, vol. 4(2), pp. 1125–31, 2019.

[3] J. Li, Z. Chen, D. Harabor, P. J. Stuckey, and S. Koenig, "Mapf-lns2: Fast repairing for multi-agent path finding via large neighborhood search," in *Proc. of AAAI 2022*, vol. 36(9).

[4] K. Okumura, "LaCAM: Search-based algorithm for quick multi-agent pathfinding," in *Proc. of AAAI 2023*, vol. 37(10), pp. 10 256–10 265.

[5] H. Jiang, Y. Wang, R. Veerapaneni, T. Duhan, G. Sartoretti, and J. Li, "Deploying ten thousand robots: Scalable imitation learning for lifelong multi-agent path finding," in *Proc. of ICRA 2025*.

[6] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *Proc. of SoCS 2019*.

[7] H. Ma, T. S. Kumar, and S. Koenig, "Multi-agent path finding with delay probabilities," in *Proc. of AAAI 2017*, vol. 31(1).

[8] M. Čáp, J. Gregoire, and E. Frazzoli, "Provably safe and deadlock-free execution of multi-robot plans under delaying disturbances," in *Proc. of IROS 2016*, pp. 5113–5118.

[9] W. Hönig, T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig, "Summary: Multi-agent path finding with kinematic constraints," in *Proc. of IJCAI 2017*, pp. 4869–4873.

[10] J. Dunkel, "Streamlining the action dependency graph framework: Two key enhancements," *arXiv:2412.01277*, 2024.

[11] T. Guo and J. Yu, "Expected 1. x makespan-optimal multi-agent path finding on grid graphs in low polynomial time," *Journal of Artificial Intelligence Research*, vol. 81, pp. 443–479, 2024.

[12] Y. Su, R. Veerapaneni, and J. Li, "Bidirectional temporal plan graph: Enabling switchable passing orders for more efficient multi-agent path finding plan execution," *Proc. of AAAI 2024*, pp. 17 559–17 566.

[13] A. Berndt, N. Van Duijkeren, L. Palmieri, A. Kleiner, and T. Keviczky, "Receding horizon re-ordering of multi-agent execution schedules," *IEEE Transactions on Robotics*, vol. 40, pp. 1356–1372, 2023.

[14] J. Yan, Z. Li, W. Kang, K. Zheng, Y. Zhang, Z. Chen, Y. Zhang, D. Harabor, S. F. Smith, and J. Li, "Advancing mapf towards the real world: A scalable multi-agent realistic testbed (SMART)," *arXiv:2503.04798*, 2025.

[15] A. Tajbakhsh, L. T. Biegler, and A. M. Johnson, "Conflict-based model predictive control for scalable multi-robot motion planning," in *Proc. of ICRA 2024*, pp. 14 562–14 568.

[16] J. Yan and J. Li, "Multi-agent motion planning for differential drive robots through stationary state search," in *Proc. of the AAAI 2025*.

[17] Z. Peng, Q. Li, K. M. Hui, C. Liu, and B. Zhou, "Learning to simulate self-driven particles system with coordinated policy optimization," *Proc. of NeurIPS 2021*, vol. 34, pp. 10 784–10 797.

[18] Z. Williams, J. Chen, and N. Mehr, "Distributed potential ilqr: Scalable game-theoretic trajectory planning for multi-agent interactions," *arXiv:2303.04842*, 2023.

[19] S. Lin, A. Liu, J. Wang, and X. Kong, "A review of path-planning approaches for multiple mobile robots," *Machines*, vol. 10(9), 2022.

[20] P. Huang, R. Liu, S. Aggarwal, C. Liu, and J. Li, "APEX-MR: Multi-robot asynchronous planning and execution for cooperative assembly," *arXiv:2503.15836*, 2025.

[21] S. Maneewongvatana and D. M. Mount, "Analysis of approximate nearest neighbor searching with clustered point sets," *arXiv:cs/9901013*, 1999.

[22] Q. Li, Z. Peng, L. Feng, Q. Zhang, Z. Xue, and B. Zhou, "Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45(3), pp. 3461–3475, 2022.