Persistent and Robust Execution of MAPF Schedules in Warehouses

Wolfgang Hönig[1], Scott Kiesel[2], Andrew Tinka[2], Joseph W. Durham[2], and Nora Ayanian[1]

***Abstract*—Multi-Agent Path Finding (MAPF) is a well-studied problem in Artificial Intelligence that can be solved quickly in practice when using simplified agent assumptions. However, real-world applications, such as warehouse automation, require physical robots to function over long time horizons without collisions. We present an execution framework that can use existing single-shot MAPF planners and ensures robust execution in the presence of unknown or time-varying higher-order dynamic limits, unforeseen robot slow-downs, and unpredictable obstacle appearances. Our framework also naturally enables the overlap of re-planning and execution for persistent operation and requires little communication between robots and the centralized planner. We demonstrate our approach in warehouse simulations and in a mixed reality experiment using differential drive robots. We believe that our solution closes the gap between recent research in the artificial intelligence community and real-world applications.**

***Index Terms*—Path Planning for Multiple Mobile Robots or Agents; Planning, Scheduling and Coordination; Multi-Robot Systems; Collision Avoidance; Factory Automation**

I. INTRODUCTION

THE Multi-Agent Path Finding (MAPF) problem is a well-studied problem in Artificial Intelligence, where collision-free paths for many agents need to be computed given the current state of the agents as well as a representation of the environment. Practical applications include computer games, traffic management, airport scheduling, and warehouse automation [1]. Current state of the art algorithms can compute bounded suboptimal solutions for hundreds of robots within minutes. However, executing such plans on physical robot teams remains challenging, because most efficient MAPF formulations make unrealistic simplifying assumptions. In this section, we will discuss these shortcomings as they pertain to a challenging industrial warehouse planning problem [2].

Consider the example domain in Fig. 1, where robots are tasked with delivering shelves to pack stations. At each station, a human worker picks one or more items from each delivered shelf. Upon completion, the robot then returns the shelf to a storage location in the warehouse. Most MAPF formulations make two significant assumptions that cannot be ignored on real robots. First, they assume that robots can act synchronously, executing exactly one action per timestep. In practice, warehouse robots are subject to at least second-order dynamic constraints; for example, moving $3\,\mathrm{m}$ forward continuously without stopping will be faster than stopping each meter of movement (a typical action in MAPF planners) due to finite acceleration constraints. Additionally, exact execution times may vary due to unforeseen necessary slow-downs or control inaccuracies. Second, MAPF formulations assume that the planning problem is single-shot, i.e., robots move from their current position to a goal and remain there. In real world applications, the planning problem is likely persistent and evolving (also sometimes referred to as life-long [3]). For example, robots have to move shelves continuously in the warehouse scenario, because new orders arrive regularly.

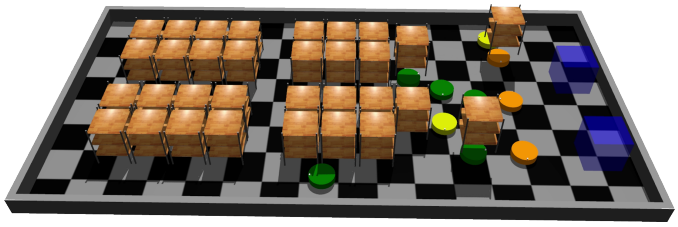

Fig. 1. An example of the warehouse domain with 32 shelves, 12 robots, and two stations (transparent squares on the right). A task requires any robot to pick up a particular shelf, bring it to a specified station, and return the shelf to another location. The objective is to keep the stations utilized with as few robots as possible.

Our approach addresses these shortcomings by introducing an execution framework that is agnostic of the underlying MAPF solver. We first introduce the *Action Dependency Graph* (ADG). The ADG is a graph that captures the action-precedence relationships of a MAPF solution and can be used to enforce these relationships on real robots with higher-order dynamics. Second, we show that this data structure enables efficient and persistent performance where (re-)planning and execution occur simultaneously, avoiding robot idle time during planning. We demonstrate our approach in simulation and in a mixed reality experiment with physical differential drive robots.

II. RELATED WORK

Multi-Agent Path Finding (MAPF) is an NP-hard [4] problem that is frequently formulated as follows. Given an unweighted undirected graph of the environment and a set of

Manuscript received: 9, 10, 2018; Revised 11, 29, 2018; Accepted 12, 22, 2018.

This paper was recommended for publication by Nak Young Chong upon evaluation of the Associate Editor and Reviewers' comments. Part of this research was completed during the first author's internship at Amazon Robotics. Our research was also partially supported by NSF under grant IIS-1724392 and by ARL under Cooperative Agreement W911NF-17-2-0196.

[1]Authors are with the Department of Computer Science, University of Southern California, Los Angeles, CA, USA. Email: {whoenig, ayanian}@usc.edu

[2]Authors are with Amazon Robotics, North Reading, MA, USA. Email: {skkiesel, atinka, josepdur}@amazon.com

Digital Object Identifier (DOI): see top of this page.

agents with start and goal locations, determine a collision-free path for each agent. Agents initially reside at their start location and must eventually reach their goal location. At each timestep, an agent may either wait at its current vertex or traverse an edge. Existing solvers are search-based [5], reduction-based [1], [6], or rule-based [7], [8].

Some work introduces more realism to the common MAPF formulation. Execution robustness can be improved by avoiding $k-$delay conflicts, which guarantees collision-free operation if robots are delayed up to k timesteps [9]. Another formulation considers delay probabilities [10], where robots might stay at their current location with a given probability when tasked with a move action. In both cases, robustness is increased, but, unlike our work, newly appearing obstacles are not considered. More realistic robot collision models can be considered when using MAPF with generalized conflicts (MAPF/C), which allows planning on roadmaps rather than grids [11]. Another generalization introduces edge weights and capacity limits [12]. In both cases, the generalization enables a wider range of robot and environment types, but does not improve persistence or robustness.

A post-processing step called MAPF-POST can be used to execute MAPF schedules on robots with varying velocity constraints [13]. MAPF-POST leverages that precedence relations of a schedule can be extracted in polynomial time. A simple temporal network is constructed based on the precedence relation and is updated continuously in an attempt to avoid re-planning. Our approach is based on the same key insight, but we use the precedence relation on actions rather than states. Our approach is more robust than MAPF-POST, because it requires less communication (robots only need to communicate when an action is finished instead of broadcasting their position continuously) and has stronger guarantees on collision-free operation (robots can have arbitrary dynamic limits). We also demonstrate persistence, which was not demonstrated with MAPF-POST.

RMTRACK uses a similar idea for robustness, but does so as control law, rather than a post-processing step [14]. Robust plan-execution policies use the key idea of MAPF-POST at runtime, similar to our work, by sending messages whenever an agent enters a new state (Fully Synchronized Policy), or only for some important state changes (Minimal Communication Policy) [10]. Unlike our work, RMTRACK and robust plan-execution policies only consider delaying disturbances, while our approach addresses persistence and newly appearing obstacles as well.

MAPF formulations can be used for persistent planning of delivery tasks [3]. However, that work assumes perfect execution. In contrast, our work focuses on robust execution on real robots and allows us to efficiently and safely overlap planning and execution.

One of the key ideas of our approach, using the partial order of a schedule to deal with robustness, has been applied in operations research before [15]. We extend this approach to work in multi-robot settings, provide persistence, and show how to construct such a partial order schedule in polynomial time from an existing MAPF schedule.

Robust execution is related to cooperative obstacle avoidance, but the objective is to stay as close as possible to the pre-planned schedule. In contrast, existing obstacle avoidance techniques such as reciprocal velocity obstacles [16], buffered Voronoi cells [17], and safety barrier certificates [18] do not consider the complete pre-planned schedule. By staying close to the pre-planned schedule, robust execution reduces the risk that a collision occurs in the future. Robust trajectory execution [19] considers pre-planned trajectories, but requires significantly more computation than our approach.

III. PROBLEM DESCRIPTION

We now formulate our persistent warehouse problem. Consider the map of a warehouse as a four-connected grid. Each cell in the map can either contain an obstacle, contain a station, be free space, or be a shelf-storage location. Shelf-storage cells may or may not contain a shelf at any given time, but they may not be traversed other than to attach or detach a shelf. There are P shelves with known locations either in one of the shelf-storage cells in the map or on top of a robot. There are R robots with known locations and orientations, as well as S stations at fixed known locations. A task requires a shelf to be carried to a particular station, yield there for a given estimated time, then return to a given shelf-storage location. We assume that the map fulfills the *well-known infrastructure* requirement [20] when considering potential shelf locations as only valid start and stop locations. This requirement ensures that robots are never obstructed from moving, even if other robots are stationary at potential shelf locations. This assumption allows our approach to provide completeness and liveness guarantees even in a persistent setting. We focus on a two-tiered objective function. The primary objective is to maximize the utilization of all stations (i.e., minimizing the human worker idle time), and the secondary objective is to minimize the number of required robots.

A. Robot Model

The proposed execution framework does not rely on a specific robot movement model, however, differential drive robots are used in our experiments. We assume each robot is circular with diameter d_r and each grid cell is large enough to contain at least one robot. Each robot can turn-in-place by 90 degrees, move forward to the next cell, attach to a shelf, detach from a shelf, and yield at a station. We denote the set of actions as $\mathcal{A} = \{\circlearrowleft, \circlearrowright, \uparrow, \mathbf{A}ttach, \mathbf{D}etach, \mathbf{Y}ield\}$. Each robot is able to localize itself in the warehouse and execute its actions autonomously using an on-board controller. While a time estimate for each action is known, the actual execution might differ. However, we assume that a robot will not diverge significantly from its path spatially and that it will eventually finish its action. Furthermore, a robot can signal, in a timely manner, when it has finished an action. A robot's ability to accelerate can be significantly dampened by carrying the heavy load of a shelf. To address this, each robot has a command queue and can combine sequential actions in its queue. For example, if three "move forward" actions are in a robot's command queue, the robot can accelerate, move three units, and decelerate in a smooth continuous motion. This results

in faster and smoother execution compared to one where the robot must accelerate and decelerate for each move action. Feedback signals for each individual edge traversed are still reported.

B. Warehouse Planning Problem

We are given the map of the warehouse as an undirected graph $\mathcal{G}_E = (\mathcal{V}_E, \mathcal{E}_E)$, where vertices correspond to locations arranged in a grid and edges correspond to straight lines between locations that can be traversed by the robot without colliding with a static obstacle. A subset of the vertices $\mathcal{V}_{\hat{P}} = \{v_{p^1}, \dots, v_{p^{\hat{P}}}\} \subset \mathcal{V}_E$ is the set of $\hat{P}$ possible shelf storage locations, arranged such that the well-known infrastructure property is fulfilled if considering those locations as the only endpoints. A different subset $\{v_{s^1}, \dots, v_{s^S}\} \subset \mathcal{V}_E$ describes the location of the S stations. There are $R \leq \hat{P}$ robots, each of which is initially located at a vertex in $\mathcal{V}_{\hat{P}}$. Furthermore, there are $P \leq \hat{P}$ shelves, each of which is initially located at a vertex in $\mathcal{V}_{\hat{P}}$ (where it is possible that shelves and robots are co-located). Shelves are assumed to be square with a side length of d_p.

A task $T^q \in \mathcal{T}$ is a tuple $(\text{shelf}^i, \text{station}^k, \delta, v_{p^j})$, describing that shelf i must be picked up by a single robot from its current location, delivered to station k where it will approximately yield for δ seconds (during which a human can pick items from the shelf), and returned to a possibly different location v_{p^j}. When initially issued, a task is not bound to a robot and thus robots can freely be assigned to any task. New tasks may be added to $\mathcal{T}$ at any time.

Time is continuous and flows forward unabated; at each instant a robot can either wait at its current vertex or begin executing one of its actions. Let $\text{loc}(r^i, t) \in \mathbb{R}^2$ be the location of robot r^i at time t and $\text{loc}(\text{shelf}^i, t) \in \mathbb{R}^2$ the location of shelf^i at time t. Note that shelves are either at a shelf storage location or on top of a robot. A robot can drive under a shelf if it is currently not carrying another shelf. To avoid collisions, we must ensure that: i) robots never collide with each other, i.e., $\| \text{loc}(r^i, t) - \text{loc}(r^j, t) \|_2 > d_r, i \neq j, \forall t$; and ii) shelves never collide with each other, i.e., $\| \text{loc}(\text{shelf}^i, t) - \text{loc}(\text{shelf}^j, t) \|_\infty > d_p, i \neq j, \forall t$. Even if i) is enforced, ii) can occur if a robot attempts to drive over a shelf location when it still has a shelf attached.

Whenever a robot fulfills a task that brings a shelf to a station and yields, the station is considered utilized during the yield's duration. We denote the total utilization duration of station k from time 0 to time t as $dur(k, t)$. Our goal is to maximize the average station utilization over the time interval $[0, t]$, i.e., $\max u(t) = \frac{1}{tS} \sum_{k=1}^{S} dur(k, t)$. Typically, we are interested in maximizing $u(t)$ over a long time horizon, e.g., a work shift.

C. Persistent and Robust Execution

We consider an execution *persistent* if robots continue to fulfill tasks and avoid unnecessary wait times. Unnecessary wait times occur if robots cannot execute any action because they are waiting for the planner to finish; a formal definition is given in Section V-A.

We consider an execution *robust* if no collision occurs even in the event of varying execution times of robot actions. Such time variations may arise due to varying dynamic limits, temporary robot malfunction, or unforeseen obstacles (e.g., items that fell from a shelf and are now blocking the robot's path).

IV. APPROACH

First, we simplify the planning stage to operate in discrete time and ignore higher-order dynamics, which allows us to use existing single-shot MAPF planners. Second, we leverage an action dependency graph (ADG) for robust continuous-time execution. Third, we demonstrate how the ADG can be used for persistent execution by overlapping execution and planning.

A. Single-Shot MAPF Formulation

We define the state s of a robot to be a tuple $s = (location, heading, task, stage)$, where $location \in \mathcal{V}_E$ is the current location of the robot, $heading \in \{\mathbf{S}outh, \mathbf{N}orth, \mathbf{E}ast, \mathbf{W}est\}$ is its current heading, $task \in \{None\} \cup \mathcal{T}$ the currently assigned task, and $stage \in \{\mathbf{I}dle, \mathbf{S}helf\mathbf{A}ttached, \mathbf{Y}ielded\}$ keeps track of the task progress. The possible state transitions can now be defined based on the available robot actions (see Section III-A). For example, the $Attach$ action can only be executed if the shelf and robot are co-located and its execution will change the $stage$ variable in the robot's state from $Idle$ to $ShelfAttached$. This state-action model can be used in single-shot MAPF solving frameworks with a few modifications.

In Conflict-Based Search (CBS) [21], a conflict occurs if two robots are at the same location at the same timestep (vertex conflict) or if two robots traverse the same edge at the same timestep (edge conflict). The conflict resolution of CBS is almost identical in the larger warehouse state space, but we consider an additional conflict if a robot that is carrying a shelf attempts to occupy a potential shelf location. We use ECBS-TA [22], a variant of Conflict-Based Search that can compute a bounded suboptimal solution to simultaneously assign tasks and find action sequences. ECBS-TA takes an assignment matrix as input and thus also works with cases where some robots already have a task assigned (for example, because they already picked up a shelf), while other robots are idle.

CBS is not the only algorithm that can be used for MAPF in this setting. The well-known infrastructure property also allows us to apply prioritized planning with completeness guarantees [20]. In this case, an algorithm such as SIPP [23] can be used, but requires separating the state from the location. Specifically, all safe intervals are defined for locations only, while actions chosen change the whole state (including the location). In the prioritized planning case, task assignment is done greedily, in the order in which agents are planning their actions.

Other existing single-shot MAPF solvers, such as reduction-based solvers [1], might also be used. However, solver-specific changes are required, similar to the changes presented for CBS

and SIPP. In particular, additional constraints need to be added to avoid the case that a robot that is carrying a shelf occupies a potential shelf location. The task assignment can be done independently (as in SIPP) or integrated in the MAPF solver (as in ECBS-TA).

Independent of the MAPF solver used, the input of a single-shot planner is the current state of all robots $(s^1, \dots, s^R)$. Let the output of a planner be a sequence of n^i tuples for each robot i: $p^i = [(t^i_1, a^i_1, s^i_1, g^i_1), \dots, (t^i_{n^i}, a^i_{n^i}, s^i_{n^i}, g^i_{n^i})]$, where a^i_k denotes the kth action that should be executed starting at timestep t^i_k and that changes the robot's location from s^i_k to g^i_k. A MAPF planner computes outputs that are collision-free with respect to the criteria in Section III, when considering t at fixed timesteps.

An example is shown in Fig. 2a. The current state of the three robots is $((D, \mathbf{W}, T^4, \mathbf{SA}), (B, \mathbf{W}, T^2, \mathbf{SA}), (A, \mathbf{E}, T^9, \mathbf{SA}))$. A valid MAPF plan is: $p^1 = [(0, \mathbf{Y}, D, D), (1, \uparrow, D, C), (2, \circlearrowleft, C, C)]$, $p^2 = [(0, \circlearrowleft, B, B), (1, \uparrow, B, D), (2, \mathbf{Y}, D, D), (3, \uparrow, D, F)]$, and $p^3 = [(1, \uparrow, A, B), (2, \circlearrowleft, B, B), (3, \uparrow, B, D)]$.

B. Action Dependency Graph: Basics

We make use of the idea that a multi-agent plan implicitly encodes dependencies between robots, for example, defining which robot should move first through a narrow passage way. Such dependencies can be extracted in polynomial time in a post-processing step, similar to MAPF-POST [13]. In MAPF-POST, the dependencies are created between states and a simple temporal network is used to create a smooth schedule with guaranteed safety distances between robots. In our approach, we define the dependencies on the robots' actions instead.

1) Construction: We create an action dependency graph (ADG) $\mathcal{G}_{ADG} = (\mathcal{V}_{ADG}, \mathcal{E}_{ADG})$ where $p^i_k \in \mathcal{V}_{ADG}$ and p^i_k refers to the kth tuple in plan p^i. Edges in the ADG represent inter-action dependencies. If $(p^i_k, p^{i'}_{k'}) \in \mathcal{E}_{ADG}$, then a robot is only allowed to start executing $a^{i'}_{k'}$ after a^i_k has been completed. The ADG construction is a two-step process. First, we create all vertices based on all p^i_k and connect subsequent actions for robot i with so-called Type 1 edges. Second, we find dependencies between different robots, indicating temporal precedences between actions (so-called Type 2 edges). The ADG construction pseudo code is shown in Algorithm 1. The construction is accomplished in $O(R^2T^2)$, where $T = \max_i n^i$. An example ADG is shown in Fig. 2b.

Not all standard MAPF plans can be executed robustly and collision-free with an arbitrary robot model. For example, consider four robots that move in a 2×2 grid in a circular motion. While a planner such as CBS would produce a plan, there is no safe way for robots to execute such a plan because it requires precise synchronous execution, a property that our robots do not have. Such an unsafe state transition is easily detectable as cycle in the constructed ADG. We can also avoid such cycles during planning, for example by disallowing that a robot moves out of a cell in the perpendicular direction of another robot moving into that cell. In the CBS framework this translates to an additional edge conflict, while in SIPP the computation of the earliest arrival time can be adjusted accordingly.

Algorithm 1: Action Dependency Graph Construction

Input: Plan p^i for each robot.
Result: $\mathcal{G}_{ADG}$

```
/* create vertices and Type 1 edges */
for i ← 1 to R do
    Add vertex p^i_1 to V_ADG
    p ← p^i_1
    for k ← 2 to n^i do
        Add vertex p^i_k to V_ADG
        Add edge (p, p^i_k) to E_ADG
        p ← p^i_k
/* create Type 2 edges */
for i ← 1 to R do
    for k ← 1 to n^i do
        for i' ← 1 to R do
            if i ≠ i' then
                for k' ← 1 to n^i' do
                    if s^i_k = g^i'_k' and t^i_k ≤ t^i'_k' then
                        Add edge (p^i_k, p^i'_k') to E_ADG
                        break
```

2) Execution: At execution time, we keep track of the completion status of each vertex (action) in $\mathcal{V}_{ADG}$. Each vertex can either be staged, enqueued, or finished. We only enqueue actions into a robot's command queue if i) the previous vertex (that is connected by an incoming Type 1 edge) is already enqueued or finished, and ii) all vertices associated with incoming Type 2 edges are finished. We mark a vertex as finished once the robot notifies the execution monitor of the successful execution of the associated action.

This approach guarantees that a robot will only move into a location after the previous robot has completely moved out of that location. While this implies coarser safety distances than MAPF-POST (the safety distance is a single cell), it requires less communication at runtime and works with arbitrary dynamic limits. In particular, we only need to track finished actions rather than the current position of all robots at all times.

If a robot detects an unforeseen obstacle in its path, the robot stops autonomously, empties its command queue, and notifies the planner of the new obstacle and the aborted command queue. New actions will be enqueued, once the planner has finished re-planning.

Consider the example in Fig. 2c and colored vertices in Fig. 2b. Robot 1 finished two actions and has one more action in its command queue, robot 2 finished its turning action and has three more actions in its command queue, and robot 3 has no action in its queue. Robot 3 cannot enqueue its next move action, until robot 2 finishes its move action first.

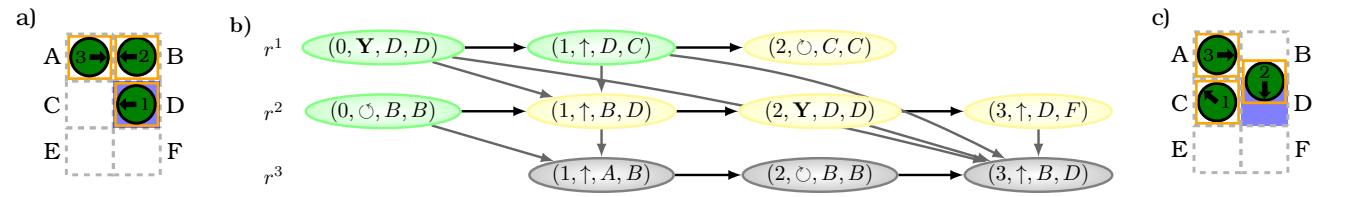

Fig. 2. Example of an action dependency graph. a) initial state next to one of the stations with locations $A, \dots, F$ and three robots. b) $\mathcal{G}_{ADG}$ as constructed by Algorithm 1. Black horizontal edges are Type 1 edges and all other edges are Type 2 edges. c) current state where some actions in the ADG are finished and enqueued. Green vertices in the ADG are finished, yellow vertices are enqueued, and gray vertices are staged.

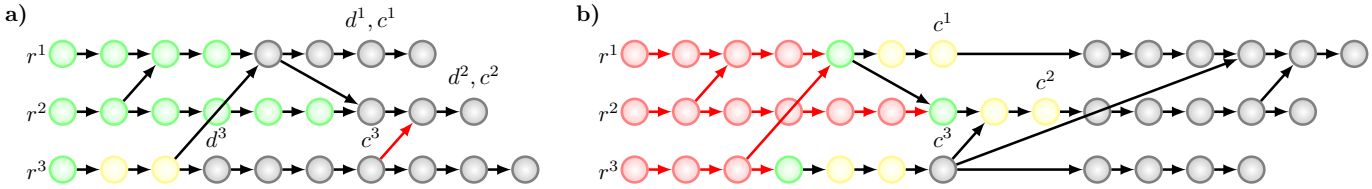

Fig. 3. Overlapping of execution and re-planning. a) Current action dependency graph. Green vertices are finished, yellow vertices are enqueued, and gray vertices are staged. Robot 2 is going to finish its current plan within the next three actions and therefore re-planning is desired. The desired set of vertices we want to commit to are labeled with d^1, d^2, d^3. The computed commit cut vertices are labeled with c^1, c^2, c^3, and are not the same as the desired vertices, because of the red dependency between robot 3 and robot 2. b) Updated ADG after planning with shifted start times for new vertices. Completed vertices (red) can be deleted.

C. Lifelong Planning

A typical approach for dynamic scenarios is to continuously re-plan with a finite time horizon, as for example in model predictive control. However, this requires fast planning and state estimation. The coupling of robots in multi-robot systems makes planning typically too slow for this kind of re-planning. Another method is to plan for the next goal, once one goal is reached [3]. While this can work in decentralized settings, it does not work for a centralized planner because it neglects the finite runtime of the planner and would cause all robots to stall while a new plan is computed. Thus, it is desirable to overlap planning and execution, such that there is no delay when re-planning occurs.

Our approach is based on action dependency graphs. We detect cases when re-planning is required: either if a robot senses an obstacle on its current path or if at least one robot has an estimated fixed duration of execution remaining in the ADG. In order to overlap planning and execution, we need to find a set of *committed vertices* in $\mathcal{V}_{ADG}$ that defines the actions that the robots will execute before switching to the new plan. We use the term *commit cut* as the set of the last actions, one for each robot, that is a subset of the committed vertices. We allow continued execution of the old plan up until the commit cut. In parallel, we re-plan by constructing the start state for our single-shot MAPF planner from the final state that would be reached after the commit cut. If desired, re-planning can use the old plan as seed to find a new solution quicker.

In order to ensure a valid transition between the old and the new plan, we need to find the commit cut in the old plan, such that the old committed plan is consistent with its dependencies. We compute the commit cut in four steps, see Algorithm 2: First, we define a *desired set* of vertices we want to commit to, one for each robot. These vertices should be chosen such that the remaining execution time to finish those actions is larger than the expected planning time. Such a measure might require domain specific tuning, which is encapsulated in the helper function `ComputeDesiredSet`, see line 1. In Fig. 3 we chose the desired set to be the actions that will be finished in three MAPF schedule timesteps. Second, we compute the reverse graph of $\mathcal{G}_{ADG}$, where the direction of all edges is reversed, see line 2. Third, we find the reachable set of vertices by executing an exhaustive search on the reverse graph of $\mathcal{G}_{ADG}$ starting with the set of desired vertices, see lines 3 – 10. The reachable set of vertices is a superset of the desired vertices and defines the set of committed vertices. Fourth, we find the latest occurring action for each robot in the set of committed vertices, which defines the robot's commit cut, see lines 11 – 12.

Algorithm 2: Compute Commit Cut

Input: $\mathcal{G}_{ADG}$
Result: commit cut $c^i \in \mathcal{V}_{ADG}$ for $i = 1, \dots, R$
1 $\{d^1, \dots, d^R\} \leftarrow ComputeDesiredSet(\mathcal{G}_{ADG})$
2 $\mathcal{G}'_{ADG} \leftarrow (\mathcal{V}_{ADG}, \mathcal{E}'_{ADG})$ where $\mathcal{E}'_{ADG} = \{(u, v) | (v, u) \in \mathcal{E}_{ADG}\}$
3 $reachable \leftarrow \emptyset$
4 $q \leftarrow Queue(\{d^1, \dots, d^R\})$
5 **while** q *not empty* **do**
6 $\quad p^i_k \leftarrow Dequeue(q)$
7 $\quad reachable \leftarrow reachable \cup \{p^i_k\}$
8 $\quad$ **for** $(p^i_k, u) \in \mathcal{E}'_{ADG}$ **do**
9 $\quad\quad$ **if** $u \notin reachable$ **then**
10 $\quad\quad\quad Enqueue(q, u)$
11 **for** $j \leftarrow 1$ **to** R **do**
12 $\quad c^j \leftarrow \arg\max_k \{p^i_k | p^i_k \in reachable \wedge i = j\}$

We use the robots' states after the commit cut as starting point for our single-shot MAPF planner, with one small adjustment: we synchronize the time, using the maximum of all

timesteps t_i^j of the commit cut vertices. This ensures that there will be no dependencies from the new plan to the old plan. The new plan can be added to the ADG, and dependencies computed according to Algorithm 1. Dependencies may exist from the old plan to the new plan, but the construction of our commit cut disallows dependencies from the new plan to the old plan. Finally, finished vertices can be safely deleted, to keep memory usage small. An example of our approach is shown in Fig. 3.

V. EVALUATION

We implement our approach in C++, using our ECBS-TA implementation [22] and boost graph for our ADG data structure. In our experiments, we demonstrate persistence and robustness in simulation and mixed reality. Our method also provides a significant performance gain over a baseline implementation. A video of our experiments is provided in the supplemental material.

A. Simulation

We evaluate our approach in simulation using HARMONIES[1], a simulator developed at Amazon Robotics specifically for quantifying academic results in warehouse-like environments. HARMONIES implements the robot model as discussed in Section III-A, where robots have acceleration limits (which are not modeled in most MAPF planners, including ours). The simulator runs on Amazon Web Services, executes actions in real-time, and provides a RESTful API. Simulator client applications can enqueue actions and receive errors/statistics during the execution, including the number of occurred collisions and the station utilization $u(t)$. Our client was executed on a laptop (i7-4600U 2.1 GHz and 12 GB RAM).

We evaluate persistence by counting the number of staged actions in the ADG per robot at a fixed time interval of $0.5\,\mathrm{s}$. The unnecessary wait time for a robot is the cumulative time while the robot had no actions staged; the total unnecessary wait time is the sum of all per-robot wait times. We test our approach on the "small_1" scenario in HARMONIES (50 robots, 600 shelves, 8 stations). We use ECBS-TA with a bounded suboptimal factor of 2; trigger re-planning if there are fewer than 10 actions in a robot's queue; and use a lookahead of 10 actions for the selection of the desired set of commit cut vertices. These settings were found empirically to be sufficiently large to overlap re-planning and execution, see Fig. 4. Re-planning takes on average $15\,\mathrm{s}$ and is thus a significant factor when minimizing wait times. In our experiment, there was no unnecessary wait time and we achieved a station utilization of $u(500) = 0.24$. We also evaluated robustness by measuring the number of reported collisions from the HARMONIES simulator, which for our experiments was zero.

[1]**H**igh-fidelity **A**utonomous-agent **R**esearch in **M**otion-planning and **O**rganization over a **N**etwork at **I**ndustrial **E**xhibited **S**cale; For more information contact harmonies@amazon.com.

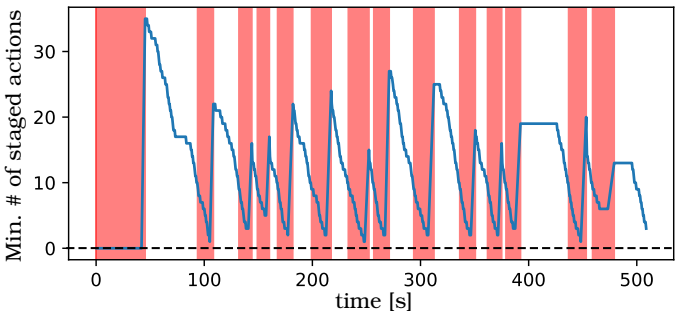

Fig. 4. Overlap of planning and execution. The red parts mark timespans used for re-planning (an average of $15\,\mathrm{s}$). The blue line shows the minimum number of staged actions over all 50 agents. This line never drops to zero after the initial plan was found, indicating that re-planning and execution always overlapped.

B. Mixed Reality Experiment

We implement our approach in a mixed reality experiment [24], which allows us to show robustness with respect to newly appearing obstacles and unmodeled dynamics. We use Gazebo as our virtual environment as well as our robotics simulator. Our custom Gazebo world plugin has three types of differential drive robot models: i) simulated agents that do not use the physics engine and move perfectly with a constant velocity; ii) simulated robots that use a physics engine and are modeled after the iRobot Create2 robots; and iii) physical iRobot Create2 robots. All three robot types have different (and in the discrete planning problem, unmodeled) dynamics. Shelves and stations are visualized only and not modeled using the physics engine for all robots.

Our iRobot Create2 robots are equipped with one of ODROID C1+ or ODROID XU4 single-board computers that run Ubuntu 16.04 with ROS Kinetic. Controller and command queues are executed on-board the robots. State estimation is done using a motion capture system. The robots communicate with the simulator using ROS services. These services are used to enqueue new actions and to send notifications of successfully completed actions.

In our experiment, we use a total of 12 robots: 6 physical robots, 2 simulated robots, and 4 simulated agents. The robots have different dynamics based on their type. For physical robots, battery level also affects their dynamics. None of the dynamics were explicitly modeled in our MAPF solver. During the run, we introduce an unknown (virtual) obstacle, which robots can detect in our mixed reality setting. Furthermore, we artificially change the maximum speed of one of the robots during a time period. No collisions occurred during our experiment, showing that our approach is robust to varying and unforeseen dynamics. We demonstrate persistence by executing our experiment for several minutes such that each robot finished more than a single task without any execution delays caused by re-planning. A screenshot is shown in Fig. 1.

C. Baseline

We compare our approach to a simple baseline that, like our approach, is an execution framework relying on existing MAPF solvers. Our baseline approach executes the MAPF

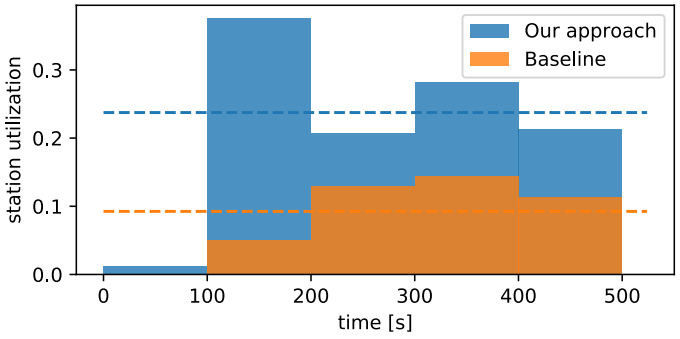

Fig. 5. Station utilization of our approach compared to a baseline over time on the "small_1" scenario in HARMONIES. Bars represent the station utilization for a time period, e.g., our approach reached a station utilization of 0.37 during $t \in [100, 200]$, while the baseline reached a station utilization of 0.05 during the same time. The dashed lines show the utilization after 500 seconds, i.e., $u(500)$, for our approach and the baseline.

schedule synchronously, that is, we only enqueue one action per timestep per robot, and wait until all robots finished executing their current action before enqueuing actions for the next timestep. This baseline is comparable to the ALLSTOP strategy in previous work [14]. If an unforeseen obstacle is detected, or at least one robot finishes its current schedule, synchronous re-planning is triggered. This baseline provides persistence and robustness like our approach, but causes robots to spend a significant amount of time waiting rather than executing actions.

We use the HARMONIES simulator on the same scenario with the same settings as in Section V-A. Using the baseline, the achieved station utilization is $u(500) = 0.09$ — over 2.5 times lower compared to our approach. The utilization of the baseline and our approach over time is shown in Fig. 5.

VI. CONCLUSION

We present an execution framework that can be used to execute MAPF plans on physical robots persistently and robustly. We demonstrate both properties in a mixed reality experiment and in simulation. For persistence, we show that planning and execution can be overlapped such that robots do not have to wait until the planner finds a new solution. For robustness, we test with unknown, time-varying dynamic limits as well as a randomly appearing obstacle.

We believe that our approach closes the gap between recent advances in multi-agent path finding algorithms from the artificial intelligence community and practical applications in robotics. Our approach can be used with existing MAPF planners with slight modification and requires little additional computation to ensure persistent and robust execution. It also uses significantly less communication than other existing execution frameworks, such as MAPF-POST. In the future, we hope that our method will allow researchers and practitioners to apply and study MAPF planners in additional realistic persistent applications.

REFERENCES

[1] J. Yu and S. M. LaValle, "Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics," *IEEE Transactions on Robotics (T-RO)*, vol. 32, no. 5, pp. 1163–1177, 2016.

[2] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Magazine*, vol. 29, no. 1, pp. 9–20, 2008.

[3] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2017, pp. 837–845.

[4] D. Ratner and M. K. Warmuth, "Finding a shortest solution for the N x N extension of the 15-puzzle is intractable," in *National Conference on Artificial Intelligence*, 1986, pp. 168–172.

[5] A. Felner, R. Stern, S. E. Shimony, *et al.*, "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges," in *Symposium on Combinatorial Search (SOCS)*, 2017, pp. 29–37.

[6] P. Surynek, "Towards optimal cooperative path planning in hard setups through satisfiability solving," in *Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, vol. 7458, 2012, pp. 564–576.

[7] B. de Wilde, A. ter Mors, and C. Witteveen, "Push and rotate: a complete multi-agent pathfinding algorithm," *Journal of Artificial Intelligence Research (JAIR)*, vol. 51, pp. 443–492, 2014.

[8] P. Surynek, "A novel approach to path planning for multiple robots in bi-connected graphs," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2009, pp. 3613–3619.

[9] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Barták, and N. Zhou, "Robust multi-agent path finding," in *Symposium on Combinatorial Search (SOCS)*, 2018, pp. 2–9.

[10] H. Ma, T. K. S. Kumar, and S. Koenig, "Multi-agent path finding with delay probabilities," in *AAAI Conference on Artificial Intelligence*, 2017, pp. 3605–3612.

[11] W. Hönig, J. A. Preiss, T. K. S. Kumar, G. S. Sukhatme, and N. Ayanian, "Trajectory planning for quadrotor swarms," *IEEE Transactions on Robotics (T-RO)*, vol. 34, no. 4, pp. 856–869, 2018.

[12] R. Barták, J. Svancara, and M. Vlk, "A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018, pp. 748–756.

[13] W. Hönig, T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig, "Multi-agent path finding with kinematic constraints," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2016, pp. 477–485.

[14] J. Gregoire, M. Cáp, and E. Frazzoli, "Locally-optimal multi-robot navigation under delaying disturbances using homotopy constraints," *Autonomous Robots*, vol. 42, no. 4, pp. 895–907, 2018.

[15] S. D. Wu, E. Byeon, and R. H. Storer, "A graph-theoretic decomposition of the job shop scheduling problem to achieve scheduling robustness," *Operations Research*, vol. 47, no. 1, pp. 113–124, 1999.

[16] J. P. van den Berg, M. C. Lin, and D. Manocha, "Reciprocal velocity obstacles for real-time multi-agent navigation," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2008, pp. 1928–1935.

[17] D. Zhou, Z. Wang, S. Bandyopadhyay, and M. Schwager, "Fast, on-line collision avoidance for dynamic vehicles using buffered voronoi cells," *IEEE Robotics and Automation Letters (RA-L)*, vol. 2, no. 2, pp. 1047–1054, 2017.

[18] L. Wang, A. D. Ames, and M. Egerstedt, "Safety barrier certificates for collisions-free multirobot systems," *IEEE Transactions on Robotics (T-RO)*, vol. 33, no. 3, pp. 661–674, 2017.

[19] B. Şenbaşlar, W. Hönig, and N. Ayanian, "Robust trajectory execution for multi-robot teams using distributed real-time replanning," in *International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 2018, accepted. To Appear.

[20] M. Čáp, P. Novák, A. Kleiner, and M. Selecký, "Prioritized planning algorithms for trajectory coordination of multiple mobile robots," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 3, pp. 835–849, 2015.

[21] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[22] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018, pp. 757–765.

[23] M. Phillips and M. Likhachev, "SIPP: safe interval path planning for dynamic environments," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011, pp. 5628–5635.

[24] W. Hönig, C. Milanes, L. Scaria, T. Phan, M. T. Bolas, and N. Ayanian, "Mixed reality for robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 5382–5387.