Multi-Agent Motion Planning For Differential Drive Robots Through Stationary State Search*

Jingtian Yan, Jiaoyang Li

Carnegie Mellon University
jingtianyan@cmu.edu, jiaoyangli@cmu.edu

Abstract

Multi-Agent Motion Planning (MAMP) finds various applications in fields such as traffic management, airport operations, and warehouse automation. In many of these environments, differential drive robots are commonly used. These robots have a kinodynamic model that allows only in-place rotation and movement along their current orientation, subject to speed and acceleration limits. However, existing Multi-Agent Path Finding (MAPF)-based methods often use simplified models for robot kinodynamics, which limits their practicality and realism. In this paper, we introduce a three-level framework called MASS to address these challenges. MASS combines MAPF-based methods with our proposed stationary state search planner to generate high-quality kinodynamically-feasible plans. We further extend MASS using an adaptive window mechanism to address the lifelong MAMP problem. Empirically, we tested our methods on the single-shot grid map domain and the lifelong warehouse domain. Our method shows up to 400% improvements in terms of throughput compared to existing methods.

1 Introduction

We study the Multi-Agent Motion Planning (MAMP) problem which aims to find collision-free kinodynamically feasible paths for a team of agents in a fully observable environment while minimizing their arrival time. This problem finds various real-world applications, including traffic management (Ho et al. 2019), airport operations (Li et al. 2019), and warehouse automation (Kou et al. 2019). Differential drive robots are widely used in many of these environments. These robots, often navigating on a grid map, can move forward along their orientation with bounded velocity and acceleration. They can only change their orientation through in-place rotation when at zero speed. Although much work has been done to address the MAMP problem, existing methods often either fail to account for the orientations of robots or overlook continuous dynamic constraints.

Multi-Agent Path Finding (MAPF) (Stern et al. 2019) methods are a promising solution that scales to hundreds of agents. However, they assume instantaneous movement and infinite acceleration capabilities, resulting in plans that are unrealistic for real-world execution (see Fig. 1 (a) for an example). To apply MAPF methods to MAMP, ADG (Hönig et al. 2019) post-processes the speed profiles of the MAPF plan to meet kinodynamic constraints while maintaining the passing orders of agents at each location. However, as shown in Fig. 1 (b), such methods can lead to long execution time as the initial MAPF plan (Fig. 1 (a)) overlooks kinodynamic constraints. SIPP-IP (Ali and Yakovlev 2023) extends MAPF methods to MAMP by searching with a fixed number of predefined actions with discretized speeds and accelerations. However, as shown in Fig. 1 (c), due to its discretized nature, the limited action choices can lead to long execution time or even failures in solving certain cases. Moreover, to account for the different choices of speeds and accelerations, SIPP-IP explores a high-dimensional state space which compromises its efficiency. The recent work PSB (Yan and Li 2024) avoids such discretization by combining search-based and optimization-based methods, but it does not consider the orientations of agents, making it hard to apply to differential drive robots, as shown in Fig. 1 (d).

In this work, we introduce **MA**PF-**S**SIPP-**S**PS (MASS),

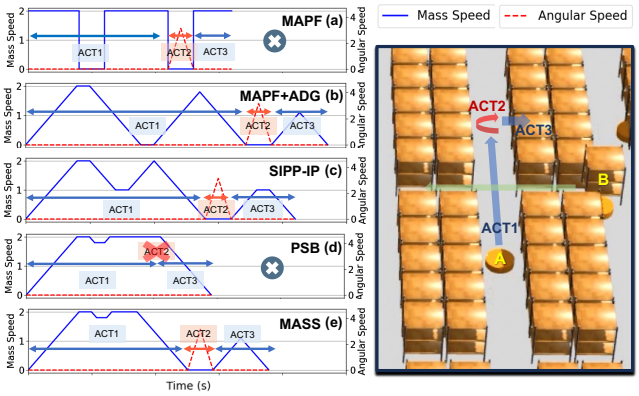

Figure 1: Speed profile of agent A (blue line: linear velocity, red line: angular velocity) generated by (a) MAPF, (b) MAPF+ADG, (c) SIPP-IP, (d) PSB, and (e) MASS. Agent A first moves upward (ACT1) while adjusting its speed to avoid collisions with agent B, performs an in-place rotation (ACT2), and then moves to the right (ACT3).

*The appendix of this paper is available at: https://doi.org/10.48550/arXiv.2412.13359

a framework to address the MAMP problem for differential drive robots. MASS uses a key observation that the plan for these robots always alternates between rotation and movement. Thus, the state at the transition between two actions (finish a movement to start rotation or the reverse) is critically important. Since those states must have zero speed, we refer to them as *stationary states*. Instead of searching at high-dimensional state space that models various speeds of robots, we focus our search on these stationary states and actions that connect them. In MASS, we use a MAPF-based planner at Level 1 to resolve collisions between agents. This level imposes temporal constraints on Level 2 and calls it to get a plan for each agent. We propose Stationary Safe Interval Path Planning (SSIPP) at Level 2 to search for a single-agent kinodynamically feasible plan. Compared to standard SIPP (Phillips and Likhachev 2011), SSIPP uses stationary node expansion to find neighboring stationary states and the actions needed to reach them. At Level 3, an optimization-based speed profile solver (SPS) is used to determine the speed profiles for these actions.

Our main contributions include: 1. We propose a framework called MASS, a three-level MAMP planner capable of finding collision-free plans for a large group of differential drive robots. 2. We evaluate MASS on the standard MAPF benchmark, showing significant improvement in terms of success rate, especially for large-scale maps. 3. We extend MASS to address the lifelong MAMP problem where agents are assigned new goals after they reach their current ones. We evaluate MASS in a high-fidelity automated warehouse simulator (shown in Fig. 7). MASS shows up to 400% improvement in terms of solution cost compared to a MAPF planner with a post-processing framework.

2 Background

In this section, we begin with a review of MAPF algorithms. After that, we go through the related work in MAMP.

MAPF Algorithms MAPF methods have achieved remarkable progress in finding discrete collision-free paths for hundreds of agents. Most state-of-the-art MAPF methods, such as Conflict-Based Search (CBS) (Sharon et al. 2015; Andreychuk et al. 2022) and Priority-Based Search (PBS) (Ma et al. 2019), use a bi-level structure. At the high level, they resolve collisions among agents by introducing temporal obstacles into low-level single-agent solvers. These solvers then plan paths for individual agents trying to avoid those temporal obstacles. In our experiments, we test MASS with two MAPF algorithms, PP and PBS. Priority Planning (PP) (Erdmann and Lozano-Perez 1987) begins by assigning a total priority ordering to all agents requiring lower-priority agents to avoid collisions with higher-priority ones. Then, PP plans paths for each agent from high priority to low priority. During this process, agents treat the path from higher priority agents as temporal obstacles. PBS (Ma et al. 2019) searches for a good priority ordering that prevents collisions among agents. PBS explores a binary Priority Tree (PT) in a depth-first manner, where each PT node contains a set of partial priority orderings and corresponding paths. The root node starts with no priority orderings. When a collision between agents a_i and a_j is detected, the PT is expanded by creating two child nodes, each with an additional priority ordering $i \prec j$ or $j \prec i$, indicating a_i has higher or lower priority than a_j. In each child node, PBS uses a low-level planner to replan the paths based on the updated priority orderings. The search terminates when a PT node with collision-free paths is found.

MAMP Algorithms The first category of MAMP methods directly extends single-agent motion planners (Čáp et al. 2013). These methods combine the state space of individual agents into a collective joint space to perform single-agent motion planning. Since the dimension of this space increases exponentially in the number of agents, planning within the joint state space of agents presents scalability challenges. Another category of methods uses the MAPF methods to solve the MAMP problem. Some of them use discrete paths from MAPF planners to generate trajectories that meet kinodynamic constraints (Hönig et al. 2016; Zhang et al. 2021; Hönig et al. 2019). For instance, the Action Dependency Graph (ADG) (Hönig et al. 2019) generates speed profiles for each agent based on discrete MAPF plans. It post-processes the speed profiles of the MAPF plan to meet kinodynamic constraints while maintaining the passing orders of agents at each location by encoding the action-precedence relationships. The solution quality of these methods highly relies on the discrete paths from MAPF planners. However, as discussed in (Varambally, Li, and Koenig 2022), since the MAPF planners use an inaccurate kinodynamic model, their solution quality is often limited. Some other methods extend MAPF methods to consider robot kinodynamics during planning. These methods typically discretize the action space and use a graph-search-based method (Solis et al. 2021; Cohen et al. 2019; Ali and Yakovlev 2023). However, with their discretized nature, they consider a limited number of actions and thus fail to capture the full range of possible actions that agents could exhibit. Moreover, they also face scalability challenges as they search in a high-dimensional state space. To avoid such discretization, PSB (Yan and Li 2024) combines search-based and optimization-based methods to produce solutions with smooth speed profiles. However, PSB cannot handle the in-place rotation of agents, making it hard to apply to differential drive robots. The existing work closest to ours is the extended abstract by Kou et al. (2019). Instead of searching the high-dimensional state space, they suggest performing an A^* search over the states with speeds of zero and show promising preliminary results. Our MASS is inspired by this idea.

3 Problem Formulation

We define the MAMP problem for differential drive robots, referred to as MAMP$_D$, on an undirected graph $G = (V, E)$ with a set of M agents $\mathcal{R} = \{a_1, ..., a_M\}$. We adopt the grid model from the MAPF$_R$ problem (Walker, Sturtevant, and Felner 2018) and represent G as a four-neighbor grid map. Vertices in V represent grid cells in the map, with their locations the same as the center of each cell and shapes equal to the cell size. An edge $(v_i, v_j) \in E$ corresponds to possible transitions between v_i and v_j. We use a differential drive

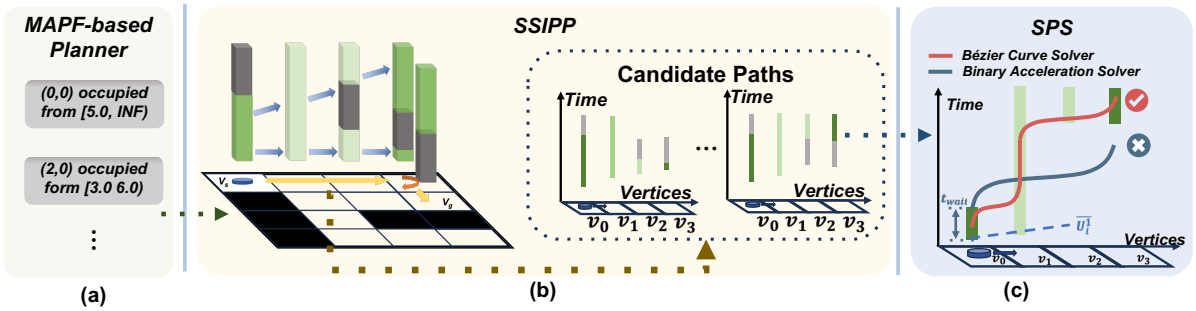

Figure 2: System overview. In (b), the green strips are safe intervals, the dark green strips are stationary safe intervals, and the gray boxes are temporal obstacles given by Level 1.

robot model with a specific shape. When at a vertex, an agent can have a discretized orientation $\theta \in \Theta$, where Θ is a finite set of possible orientations. We define the *state* of an agent as a collection of its vertex, orientation, and speed at a specific time. Each agent a_m initiates its movement from a specified *start (vertex)* $v_{s_m} \in V$ and start orientation $\theta_s \in \Theta$. Each agent a_m has a single designated *goal (vertex)* $v_{g_m} \in V$. All agents start simultaneously and remain at their respective goals after they finish. An agent can perform one of the following actions at each vertex with the action time being its duration:

Definition 1. *(Rotate) A rotate(θ_i, θ_j) lets an agent change its orientation from $\theta_i \in \Theta$ to $\theta_j \in \Theta$ on its current vertex. This action begins and ends with the agent at zero speed and follows a predefined angular velocity profile. The action time of $rotate(\theta_i, \theta_j)$ is no greater than the sum of the action time of $rotate(\theta_i, \theta_k)$ and $rotate(\theta_k, \theta_j)$ for all $\theta_k \in \Theta$.*

Definition 2. *(Move) A move(v_i, v_j) lets an agent move forward in its current orientation from v_i to v_j along a straight line segment $\phi_{i,j}$, which may include one or more vertices. This action begins and ends with the agent at zero speed and follows a speed profile $\ell_{i,j}(t \mid \phi_{i,j})$, denoted as the distance traveled by an agent as a function of time t along a given line segment $\phi_{i,j}$. For agent a_m, the speed profile of its move action is constrained by the following dynamic constraints:*

$$\underline{U_m^k} \leq \frac{d^k \ell_{i,j}(t \mid \phi_{i,j})}{dt^k} \leq \overline{U_m^k}, \forall k \in \{1, 2\} \tag{1}$$

$$\left.\frac{d\ell_{i,j}(t \mid \phi_{i,j})}{dt}\right|_{t=0,T_m} = \underline{U_m^1}, \tag{2}$$

where $\underline{U_m^k}$ and $\overline{U_m^k}$ represent the lower and upper bounds of speed (when $k = 1$) and acceleration (when $k = 2$), respectively, with the minimum speed being $\underline{U_m^1} = 0$. The planner needs to determine a speed profile (including action time) for each move action. We define a move action as dynamically feasible if its speed profile satisfies these constraints.

A *timed action* represents an action that starts at a specific time. A plan of a_m is a set of timed actions that move a_m from its start to its goal. An agent *reaches* a vertex iff the geometric centers of the vertex and the agent overlap. We define that an agent *occupies* a vertex v if its shape overlaps the shape of v. A *collision* occurs if two agents occupy the same v at overlapping time intervals. We use *arrival time* to indicate the time needed for an agent to reach its goal. Our task is to generate plans for all agents so that no collisions happen while minimizing the sum of their arrival time.

Lifelong MAMP$_D$ Compared to the single-shot MAMP$_D$ formulation, in the lifelong MAMP$_D$ model, there are two main differences: (D1) Each agent receives new goals assigned by an external task assigner during execution and must visit these assigned goals sequentially. (D2) Agents are not required to stay at their goals, instead, they must perform one of the three additional actions at each goal, namely attaching themselves to a shelf, detaching themselves from a shelf, or waiting at a station. Our task is to maximize the throughput (= average number of reached goal vertices in a certain time duration).

4 MASS

In this section, we begin with a system overview of our proposed method, MASS, followed by the specifics of the SSIPP used in Level 2. Next, we introduce a partial stationary expansion mechanism to improve its scalability. Then, we present the formulation for speed profile optimization and two example solvers. Finally, we discuss the techniques used to extend MASS to the lifelong MAMP$_D$ scenario.

4.1 System Overview

MAPF-based Planner At Level 1, we borrow the MAPF-based planner to resolve collisions between agents. Our framework is compatible with any MAPF solver employing a bi-level structure as discussed in related work. Empirically, we use PP and PBS as the Level-1 planner.

Stationary SIPP (SSIPP) The task of Level 2 is to find a plan for an agent with minimum arrival time while avoiding temporal obstacles (e.g., paths of higher priority agents from PP or PBS) given by Level 1. As shown in Fig. 2 (b), we first build a safe interval table $\mathcal{T}$ based on those temporal obstacles. This table associated each vertex of G with a set of *safe intervals*, which are time intervals not occupied by the temporal obstacles. Then, Level 2 performs an SSIPP search on $\mathcal{T}$ to find the neighbor stationary states along with the actions that lead to them, where the speed profile of these

actions is found by Level 3. We use a partial stationary expansion (PE) mechanism to further improve its scalability.

Speed Profile Solver (SPS) The task of Level 3 is to find a speed profile that travels within safe intervals given by Level 2, satisfies dynamic constraints, and achieves minimum action time. We introduce two solvers in this section: the Binary Acceleration Solver, an incomplete but fast method, and the Bézier-Curve Solver, a complete but slow method.

4.2 Stationary SIPP (SSIPP)

Given a safe interval table $\mathcal{T}$, the task of Level 2 is to find a collision-free plan for agent a_m while minimizing its arrival time. In our problem, agents move in continuous time with continuous dynamics, leading to an infinite number of possible states at each vertex. To address this, Level 2 employs SSIPP, which performs an A* search on $\mathcal{T}$. Compared to standard SIPP (Phillips and Likhachev 2011), SSIPP uses stationary node expansion to find stationary states and dynamically feasible actions connecting them, avoiding the need to explicitly search through the infinite state space. In the rest of this section, we omit subscript m for simplicity.

SSIPP Node The search node of SSIPP is defined as $n = \{v, \theta, \alpha, [lb, ub)\}$. $v \in V$ and $\theta \in \Theta$ are the vertex and orientation of the agent. α is the previous action that leads the agent to node n. $[lb, ub)$ is a stationary safe interval, a specific safe interval in which the agent can maintain a stationary state at v.

Main Algorithm Algorithm 1 shows the pseudo-code of SSIPP. We begin by initializing the root node with start vertex v_s, start orientation θ_s, and the first interval at v_s in $\mathcal{T}$ [Line 1]. Then we push the root node to an open list OPEN [Line 2]. The arrival time of the best plan p^* is initially set to infinity [Line 3]. We define the g-value of a node n as its lb-value, its h-value as the minimum time to move from its vertex to the goal, and its f-value as the sum of its g- and h-values, which is a lower bound on the arrival time of any plan that goes through n (i.e. stops at n within its time interval). At each iteration, we select the node n with the smallest f-value from OPEN [Line 5]. If the f-value of n is bigger than the arrival time of p^*, it indicates that p^* is the optimal plan. We terminate the search [Line 6] and return p^* [Line 11]. If n is at the goal with infinite $n.ub$, we check if its g-value is smaller than the arrival time of p^*. If true, we update p^* by backtracking all ancestor nodes of n [Line 7-8]. In either case, we proceed to the next iteration. For all other nodes, we use stationary node expansion to generate new neighbor nodes and push them to OPEN [Line 10]. This search proceeds until the optimal plan is found or OPEN is empty.

Stationary Node Expansion At each stationary state, we let the agent perform an action different from its previous action; otherwise, two identical actions can be combined into one. Accordingly, stationary node expansion includes two types: move expansion and rotate expansion. Rotate expansion finds all neighbor nodes reachable through rotation, while move expansion does the same for movement.

Rotate Expansion: Since the orientation is discretized, during rotation expansion, we apply all the possible predefined rotation speed profiles (i.e., rotate $90°$, $-90°$, and $180°$) to generate the neighbor nodes of a given node $n = \{v, \theta, a, [lb, ub)\}$. Specifically, we create a new neighbor node $n' = \{v, \theta', rotate, [lb', ub)\}$ for each possible orientation $\theta' \in \Theta$, with lb' being the sum of lb and the rotation time. n' is discarded if $lb' \geq ub$.

Move Expansion: During move expansion, we first find all safe intervals in $\mathcal{T}$ at all vertices that may be reached through a move action from the current node, referred to as *reachable intervals*. Then, for each reachable interval, we treat it as a stationary safe interval and use Level 3 to find a speed profile to reach it. If a speed profile is found, we generate a new SSIPP node for this safe interval.

Concretely, if node n is the root node or its previous action is a rotate action, we first call `getMoveIntervals`

Algorithm 1: Stationary SIPP (SSIPP)

Input: start vertex v_s, start orientation θ_s, goal vertex v_g, safe interval table $\mathcal{T}$

1 $root\_n \leftarrow (v_s, \theta_s, none, \emptyset, \mathcal{T}[v_s][0])$
2 `pushToOPEN` $(root\_n)$
3 $p^*.arrival\_time \leftarrow \infty$
4 **while** $OPEN \neq \emptyset$ **do**
5 $n \leftarrow$ OPEN.pop()
6 **if** $n.f \geq p^*.arrival\_time$ **then return** p^*
7 **if** $n.v = v_g$ ***and*** $n.ub = \infty$ **then**
8 **if** $n.g < p^*.arrival\_time$ **then** $p^* \leftarrow getPlan(n)$
9 **continue**
10 `(partial) StationaryNodeExpansion`(n)
11 **return** "No solution found"
12 **Function** `stationNodeExpansion` (n)
13 **if** $n.\alpha \neq rotate$ **then** *// rotate expansion*
14 $\{n'_0, ..., n'_j\} \leftarrow$ `rotateExpansion`(n)
15 `pushToOPEN` $(n'_0, ..., n'_j)$
16 **if** $n.\alpha \neq move$ **then** *// move expansion*
17 $\mathbb{S} \leftarrow$ `getMoveIntervals`(n)
18 **for** $[lb, ub) \in \mathbb{S}$ **do**
19 `createNodeByMove` $(n, [lb, ub))$
20 **Function** `createNodeByMove` $(n, [lb, ub))$
21 $(\phi, S) \leftarrow$ `backTrack`$([lb, ub))$
22 $\ell(t) \leftarrow$ `getSpeedProfile`(ϕ, S)
23 **if** $\ell(t) \neq null$ **then**
24 $n' \leftarrow (vertex([lb, ub)), n.\theta, move, \emptyset,$ $[n.lb+\text{actionTime}(\ell), ub))$
25 `pushToOPEN` (n')

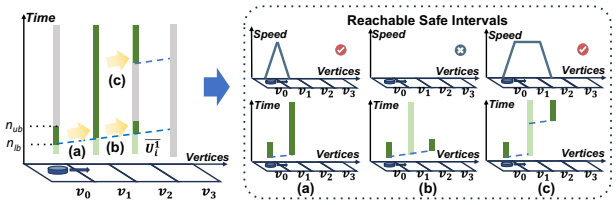

Figure 3: Illustration of the safe interval search process.

which uses a breadth-first search on safe intervals along the node's orientation to find all reachable intervals [Lines 16 and 17]. We use an example in Fig. 3 to illustrate this process. We begin by initializing the root interval using the interval of the current node and push it to a queue. During each iteration, we pop an interval from the queue and expand it by assuming the agent moves one vertex forward. In our case, we first expand the interval at v_0 denoted as $[lb_0, ub_0)$. As the agent moves from v_0 to v_1, a new interval $[lb_0 + t_{min}, \infty)$ is generated at v_1, where t_{min} is the minimum time required for this movement. Following (Yan and Li 2024), since dynamic constraints are considered in Level 3, we can use relaxed dynamic constraints to expedite this expansion process without compromising the guarantee of completeness. Specifically, we estimate t_{min} as the time the agent takes to move at maximum speed. For safe intervals at v_1 with a lower bound smaller than ub_0, which are the intervals that can be directly reached from $[lb_0, ub_0)$, we treat their overlap with $[lb_0 + t_{min}, \infty)$ as stationary safe intervals. We get the safe intervals shown in Fig. 3 (a) in our example and push it to a reachable interval set $\mathbb{S}$. In the next iteration, we continue to expand the safe intervals at v_1. This search process proceeds recursively until no stationary safe intervals can be found.

For each reachable interval $[lb, ub) \in \mathbb{S}$, we call `createNodeByMove` to generate a new SSIPP node [Line 19]. We backtrack to get all safe intervals $S = \{[lb_0, ub_0), ..., [lb, ub)\}$ along with its associated line segment ϕ [Line 21]. Then, we call Level 3 to find a speed profile $\ell(t)$ based on them [Line 22]. Using $\ell(t)$ found by Level 3, we generate a new node at the associated vertex of $[lb, ub)$ and push it to OPEN [Line 24-25].

Duplicate Detection We use a duplicate detection mechanism to eliminate redundant nodes during the search. Before inserting a new node n into OPEN, we check whether a node with the same vertex, orientation, and upper bound already exists in the open list or has been visited. If a duplicate node n' is found, we compare the lower bounds of n and n', and retain the node with the smaller lower bound. As shown in the appendix, we prove that this mechanism does not affect the completeness or the optimality of SSIPP.

Theorem 1 (Completeness and optimality of SSIPP)**.** *SSIPP is complete and returns the optimal solution if one exists when Level 3 is complete and optimal. Please refer to the appendix for detailed proof.*

4.3 Partial Stationary Expansion (PE)

During move expansion, we need to find the speed profiles for all reachable safe intervals. This branching factor can be very high, especially in large maps. To tackle this, we use a partial stationary expansion mechanism extended from (Goldenberg et al. 2014).

PE Node This node extends the SSIPP node by $n = \{v, \theta, \alpha, \mathcal{F}, [lb, ub)\}$, where *Reachable interval list* $\mathcal{F}$ is a list that contains all the reachable safe intervals. The intervals in $\mathcal{F}$ are sorted in ascending order of their p-value (= lb plus h-value at its associated vertex), which is an underestimate of the arrival time through this interval.

Algorithm 2: Partial Stationary Expansion

```
1  Function partialStationaryNodeExpansion(n)
2      if n.F = ∅ then        // expand node n for the first time
3          if n.α ≠ rotate then
4              {n'_0, ..., n'_j} ← rotationExpansion(n)
5              pushToOPEN (n'_0, ..., n'_j)
6          if n.α ≠ move then
7              n.F ←getMoveIntervals(n)
8              Sort intervals in n.F by their p-values
9      if n.F ≠ ∅ then              // generate one child node
10         createNodeByMove (n, n.F.pop())
11     if n.F ≠ ∅ then                    // reinsert node n
12         n.h ← n.F.top().p − n.g
13         pushToOPEN(n)
```

Partial Stationary Expansion In partial stationary expansion, instead of finding the speed profiles for all reachable intervals at once, we only generate the node based on the reachable interval that is most promising. As shown in Algorithm 2, if n is expanded for the first time, we first check the type of its previous action $n.\alpha$. If $n.\alpha$ is not *rotate*, we do rotate expansion to retrieve its neighbor nodes [Line 4-5]. If $n.\alpha$ is not *move*, instead of performing move expansion, we only retrieve all the reachable intervals for $n.\mathcal{F}$ [Line 7]. Then, if $n.\mathcal{F}$ is not empty, we pop the reachable interval with the smallest p-value in $n.\mathcal{F}$ and generate a neighbor node based on it [Line 10]. Finally, if $n.\mathcal{F}$ remains non-empty, we update the heuristic value of n using the smallest p-value in $n.\mathcal{F}$ and reinsert n into OPEN [Line 12-13].

Theorem 2 (Completeness and optimality of SSIPP with PE)**.** *The partial expansion mechanism preserves the completeness and optimality of SSIPP. Detailed proof is provided in the appendix.*

4.4 Speed Profile Solver (SPS)

Given the line segment $\phi_{i,j}$ and safe intervals S from Level 2, SPS aims to find a speed profile $\ell_{i,j}(t)$ with the shortest action time that satisfies both the dynamic constraints shown in Eqs. (1) and (2) and temporal constraints introduced by S (i.e., the agent remains within the safe interval while passing a vertex). This section introduces two SPS as examples. Notably, MASS is adaptable to other solvers, as long as it meets the specified constraints.

Binary Acceleration Solver (BAS) We adopt BAS from (Kou et al. 2019). This solver assumes that the agent begins by waiting at the first vertex v_i of the line segment $\phi_{i,j}$ for a duration of t_{wait}. Then, it moves with its maximum acceleration until reaching its maximum speed, moves at this speed for a duration of t_{move}, and finally decelerates with its maximum deceleration to stop at v_j. However, when the length of $\phi_{i,j}$ is small, the speed profile forms a triangle shape, where the agent accelerates to a lower peak speed and then decelerates to stop at v_j. t_{move} can be computed based on the length of $\phi_{i,j}$. Our task is to get the t_{wait} that minimizes action time while ensuring that $\ell_{i,j}(t)$ satisfies

Algorithm 3: Pseudocode for Windowed-SSIPP

Input: earliest start time t_e, goal list $\mathcal{G}$ and safe interval table $\mathcal{T}$

1 $root\_n \leftarrow (v_s, \theta_s, none, \emptyset, \mathfrak{g} = \mathcal{G}[0], [t_e, \mathcal{T}[v_s][0].ub))$
2 pushToOPEN ($root\_n$)
3 $n^*.f_{win} \leftarrow \infty$
4 **while** *OPEN* $\neq \emptyset$ **do**
5 $n \leftarrow$ OPEN.pop()
6 **if** $n.f_{win} > n^*.f_{win}$ **then return** $getPlan(n^*)$
7 **if** $n.t_{ub} = \infty$ ***and*** $n^*.f_{win} > n.f_{win}$ **then** $n^* \leftarrow n$
8 **if** $n.v = n.\mathfrak{g}$ ***and*** $n.lb + actionTime(\mathcal{G}[n.l].\alpha) < n.ub$ **then**
9 $n' \leftarrow (n.v, n.\theta, n.\mathfrak{g}.\alpha, \emptyset, \mathcal{G}.next(\mathfrak{g}), [n.lb + actionTime(\mathcal{G}[n.l].\alpha), n.ub))$
10 pushToOPEN (n')
11 **if** $n.t_{lb} < t_w$ **then**
12 partialStationaryNodeExpansion (n)
13 **return** "No solution found"

the temporal constraints introduced by S. This problem can be formulated as a Linear Programming (LP) problem. We borrow Fig. 2 (c) as a counterexample to show BAS is incomplete. In this case, a valid speed profile exists where the agent waits at v_1. However, BAS fails to find this solution because it can only decelerate upon reaching the goal.

Bézier-curve Solver (BCS) We borrow BCS from (Yan and Li 2024). BCS models the speed profile using a scaled Bézier curve, which can approximate any continuous function within its feasible range with sufficient control points. BCS encodes the temporal and dynamic constraints as an LP problem and then uses binary search to determine the optimal action time by solving this LP problem recursively. As shown in the paper, given any ϵ, BCP can find a speed profile ϵ-close to the optimal solution with a sufficient number of control points if one exists and returns failure otherwise.

4.5 Lifelong MAMP$_D$

In this section, we extend MASS to address the lifelong MAMP$_D$ problem. Many works have been done to extend the single-shot MAPF problem to the lifelong scenario. In this work, we adapt the state-of-the-art method Rolling-Horizon Collision Resolution (RHCR) (Li et al. 2021) to MASS. RHCR decomposes the lifelong MAPF problem into a sequence of windowed MAPF instances. Specifically, it plans collision-free paths for t_w timesteps and replans paths once every t_h timesteps ($t_w \geq t_h$). However, in MASS, the actions can have arbitrary action time. As a result, we can no longer determine a fixed replanning window size t_w that guarantees that all agents have just completed their actions and arrived at vertices at time t_w. In this work, we incorporate an adaptive window mechanism that apply different window sizes for different agents.

Adaptive Window Similar to RHCR, we trigger replanning every t_h time duration to plan for the next episode. However, since planning for a fixed episode length t_w is not feasible, t_w serves only as the minimum size of the replanning window. We define actions that start before and finish after t as *t-crossing actions*. Within each episode, we plan a partial path for each agent, referred to as an *adaptive window plan*, which includes the t_w*-crossing action* and any preceding actions. During planning, we ensure these plans are collision-free between agents. Since t_h*-crossing actions* can have arbitrary length, the earliest start time for each agent at the next episode may vary. As agents must complete ongoing actions, for each agent, we determine its start location and its earliest start time for the next episode based on the final vertex and the completion time of its t_h*-crossing action* from the current episode. However, as shown in Fig. 4, this method introduces a new issue: Consider agents a_1 and a_2, where both their adaptive window plans at the current episode consist only of a t_w*-crossing action* (shown as the blue and yellow arrows in Fig. 4 (a)). As illustrated in Fig. 4 (b), these plans are collision-free in the current episode. However, in the next episode, when a_2 starts at B6, no feasible plan exists to avoid a collision. To resolve this, we always append an infinite waiting action at the end of each plan. In this case, using the same example, the adaptive window plan of a_1 will collide with a_2. Thus, the planner will replan to resolve such collisions. During planning, we use $n.f_{win} = \max(t_w, n.g) + n.h$ as an admissible f-value. This formulation discourages adaptive window plans that finish before t_w. The adaptive window mechanism finds the plan with the minimum cost to the goal by identifying the node with the minimum f_{win}-value and infinite upper bound. Since collisions outside the window are ignored, by using the heuristic that can accurately reflect movement without temporal obstacles, finding the node with the minimum f_{win}-value corresponds to identifying the plan with the minimum cost to the goal.

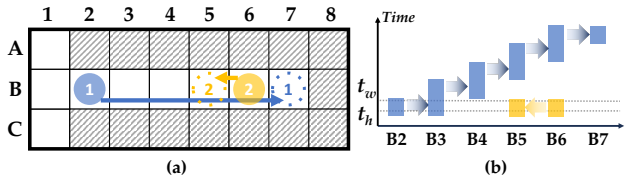

Figure 4: Limitation of directly applying RHCR to MASS. (a) Solid circles are the current vertices of agents and dashed circles are the expected start vertices for the next episode. (b) Time intervals occupied by agents at each vertex.

Main Method In every episode, for each agent a_m, we first update the start location v_{s_m}, the earliest start time t_{e_m}, and goal list $\mathcal{G}$. Here, the goal list stores the goal vertices in the order they should be visited. Then, we call MASS to find the plans for this episode. Here, Level 1 and Level 3 can be applied without any modification. As shown in Algorithm 3, we begin the search process of Level 2 by initializing the root node and pushing it to OPEN [Line 1-2]. During each iteration, we find the node n with the smallest f_{win}-value and remove it from OPEN [Line 5]. If the f_{win}-value of n is larger than the f_{win}-value of the optimal node n^* found so far, we terminate the search and return the plan [Line 6].

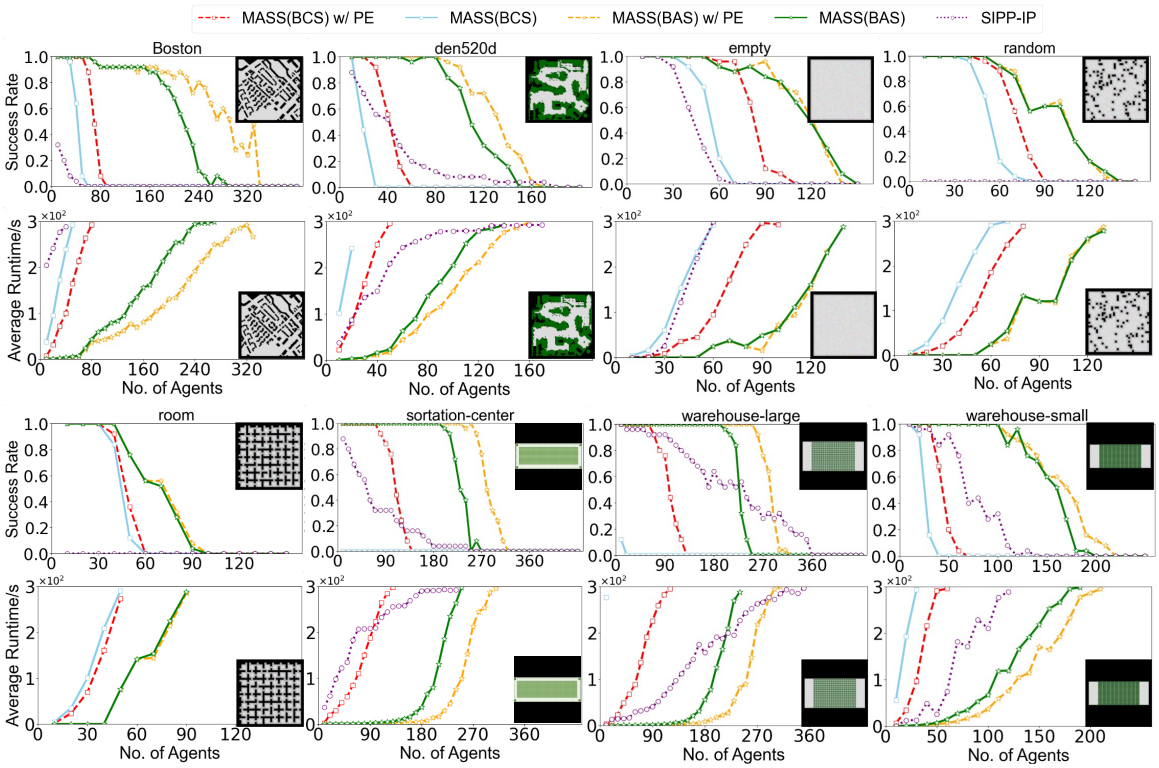

Figure 5: Success rate and average runtime across all maps. The success rate is the ratio of solved instances to all instances.

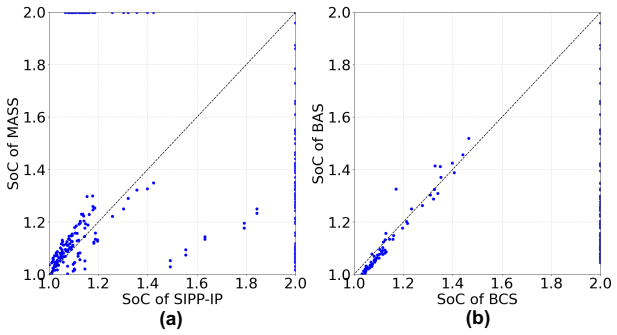

Figure 6: Relative SoC is the ratio of the total arrival time to the sum of individual agent arrival times without considering collisions. We use points at 2.0 to indicate unsolved instances. Figure (a) shows the relative SoC of SIPP-IP compared to MASS (both BAS and BCS). Figure (b) shows the relative SoC between MASS(BAS) and MASS(BCS).

Otherwise, in case $n.ub$ is infinite and n has a smaller f_{win}-value than n^*, we update n^* using n [Line 7]. If $n.v$ equals the goal vertex $n.\mathfrak{g}$ that n is trying to reach, we generate a new node n' that performs the required action $n.\mathfrak{g}.\alpha$ at vertex $n.\mathfrak{g}$ and set its goal using the next element in $\mathcal{G}$ [Lines 9 and 10]. Finally, if the lower bound of n is smaller than t_w, we do partial stationary expansion [Line 12]. If MASS is unable to find a solution within the given cutoff time, we reuse the adaptive window plan from the previous episode and continue its execution.

5 Empirical Evaluation

We implemented both our and baseline methods in C++. We conducted all experiments on an Ubuntu 20.04 machine equipped with an AMD 3990x processor and 188 GB of memory. Our code was executed using a single core for all computations. The source code for our method is publicly accessible at https://github.com/JingtianYan/MASS-AAAI.

5.1 Single-Shot MAMP$_D$

In this experiment, we use PBS as Level 1 and both BAS and BCS as Level 3 . We denote the resulting two variants as MASS(BAS) and MASS(BCS). They are further combined with the partial stationary expansion mechanism, denoted as MASS(BAS) w/ PE and MASS(BCS) w/ PE. We compare these methods with a straightforward extension of SIPP-IP (Ali and Yakovlev 2023). SIPP-IP is a state-of-the-art single-agent safe interval path planner designed to accommodate kinodynamic constraints and temporal obstacles, making it a suitable representation of motion-primitive-based methods. To adapt SIPP-IP for multi-agent scenarios, we replaced Level 2 and Level 3 in MASS with the SIPP-IP.

Simulation Setup We evaluated all methods on four-neighbor grid maps, including `empty` (empty-32-32, size: 32×32), `random` (random-32-32-10, size: 32×32), `room` (room-64-64-8, size: 64×64), `den520d` (den520d, size: 256×257), `Boston` (Boston_0_256, size: 256×256),

Runtime (s)		MASS(BCS) w/ PE	MASS(BAS) w/ PE
10 Agents	Total	34.17±24.3%	0.17±0.4%
	PBS	0.00±0.0%	0.03±0.0%
	SIPP	0.02±0.0%	0.13±0.3%
	SPS	34.16±24.3%	0.01±0.0%
150 Agents	Total	nan	170.50±114.3%
	PBS	nan	20.49±18.0%
	SIPP	nan	131.12±88.2%
	SPS	nan	18.88±13.5%

Table 1: Runtime breakdown of MASS on the `warehouse-small` map in seconds. PBS, SIPP, and SPS are the runtime of each component.

`warehouse-small` (warehouse-10-20-10-2-1, size: 161×63), `warehouse-large` (warehouse-20-40-10-2-2, size: 340×164) from the MovingAI benchmark (Stern et al. 2019), and the `sortation-center` map (size: 500×140) from the LMAPF Competition (Chan et al. 2024). For each map, we conducted experiments with a progressive increment in the number of agents, using the 25 "random scenarios" from the benchmark set. The agents were modeled as cycles with a diameter equal to the length of the grid cell. All agents adhered to the same kinodynamic constraints, where the speed is bounded by the range of $[0, 2]$ $cell/s$, while the acceleration is confined to $[-0.5, 0.5]$ $cell/s^2$.

Comparison As shown in Fig. 5, PE improves the success rate for MASS(BCS) on all maps. For MASS(BAS), it improves the success rate in large-scale maps, while maintaining comparable results on small-scale maps. This improvement is primarily because PE shows advantages when Level 3 is time-consuming (e.g., using BCS) or the branching factor during move expansion is high (e.g., on large maps). Compared to BCS, BAS demonstrates its advantage in terms of success rate. As shown in Fig. 6 (b), despite BCS being a complete and optimal method, BAS achieves a similar solution cost. We hypothesize this is due to the scalability limitations of BCS, as it can only handle less congested cases where BAS also provides near-optimal solutions. SIPP-IP has a low success rate in obstacle-rich maps due to its limited action choice. At the same time, as shown in Fig. 6 (a), it shows worse solution quality than MASS in certain cases.

Runtime As shown in Table 1, we include the runtime details of MASS with different SPS on `warehouse-10-20-10-2-1` map. The primary runtime bottleneck for MASS(BAS) lies in Level 2 (SIPP) for both the 10-agent and 150-agent cases. In contrast, MASS(BCS) is limited by Level 3 (SPS) in 10-agent scenarios and fails to scale to scenarios with a larger number of agents.

5.2 Lifelong MAMP$_D$

In this experiment, we use PP with random restart as Level 1 for MASS(BAS) and MASS(BCS), incorporating the partial expansion mechanism. We use PP w/ ADG (Varambally, Li, and Koenig 2022) to represent methods that combine MAPF with a robust execution framework. PP w/ ADG uses RHCR to decompose the lifelong MAPF problem into windowed MAPF instances, uses PP with SIPP for planning in each window, and uses ADG to execute the plans.

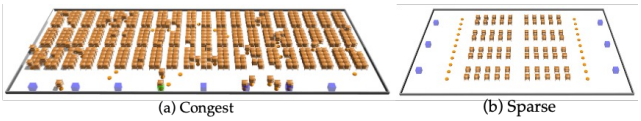

Figure 7: Simulation setup. (a) contains 8 stations, 600 shelves, and 50 agents (based on the iRobot Create 2). (b) contains 6 stations, 80 shelves, and 22 agents.

Env	t_w	**MASS(BAS)**	**MASS(BCS)**	**PP w/ ADG**
Sparse	20 s	**0.262±2.9%**	0.215±6.4%	0.142±8.4%
	25 s	**0.264±4.3%**	0.215±7.0%	0.152±8.3%
	30 s	**0.259±4.5%**	0.215±1.9%	0.149±6.8%
	40 s	**0.259±3.0%**	0.218±3.7%	0.148±7.1%
Congest	20 s	**0.373±1.4%**	0.294±4.4%	0.065±2.5%
	25 s	**0.380±3.4%**	0.300±3.4%	0.075±4.0%
	30 s	**0.371±4.1%**	0.294±13.4%	0.090±3.4%
	40 s	**0.372±6.7%**	0.300±6.4%	0.095±2.4%

Table 2: Throughput in *Congest* and *Sparse*.

Simulation Setup We borrow the simulation setup from (Hönig et al. 2019) to simulate a Kiva warehouse on the *Congest* map with 50 agents and *Sparse* map with 22 agents, using Amazon's HARMONIES simulator, as shown in Fig. 7. Each agent has a speed limit from $[0, 2]$ m/s and an acceleration limit from $[-0.5, 0.5]$ m/s. We run each method for 1,000 simulation time seconds and average the results over 7 runs.

Comparison We evaluate solution quality using the *throughput* (=average goals reached per second). As shown in Table 2, the throughput of MASS(BAS) and MASS(BCS) are significantly better than PP w/ ADG. This indicates that incorporating the kinodynamics of agents during the planning process can improve the solution quality. At the same time, MASS(BAS) achieved a slight improvement in throughput compared to MASS(BCS). This is attributed to the factor that BAS is able to explore more priority orderings within the given time window due to its shorter runtime.

6 Conclusion

This paper introduces MASS, a three-level multi-agent motion planning framework designed to tackle the MAMP problem for differential drive robots. MASS uses SSIPP to search the stationary state along with actions between them. We further add a partial stationary expansion mechanism to improve its scalability and extend MASS to the lifelong MAMP$_D$ domain. Empirically, MASS shows significant improvements in both scalability and solution quality compared to existing methods.

Acknowledgments

The research was supported by the National Science Foundation (NSF) under grant number #2328671 and a gift from Amazon. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

References

Ali, Z. A.; and Yakovlev, K. 2023. Safe Interval Path Planning with Kinodynamic Constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 12330–12337.

Andreychuk, A.; Yakovlev, K.; Surynek, P.; Atzmon, D.; and Stern, R. 2022. Multi-agent pathfinding with continuous time. *Artificial Intelligence*, 305: 103662.

Čáp, M.; Novák, P.; Vokrínek, J.; and Pěchouček, M. 2013. Multi-agent RRT: sampling-based cooperative pathfinding. In *Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems*, 1263–1264.

Chan, S.-H.; Chen, Z.; Guo, T.; Zhang, H.; Zhang, Y.; Harabor, D.; Koenig, S.; Wu, C.; and Yu, J. 2024. The League of Robot Runners Competition: Goals, Designs, and Implementation. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling – System Demonstrations Track*.

Cohen, L.; Uras, T.; Kumar, T. K. S.; and Koenig, S. 2019. Optimal and bounded-suboptimal multi-agent motion planning. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 44–51.

Erdmann, M.; and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica*, 2: 477–521.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, 50: 141–187.

Ho, F.; Salta, A.; Geraldes, R.; Goncalves, A.; Cavazza, M.; and Prendinger, H. 2019. Multi-Agent Path Finding for UAV Traffic Management. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, 131–139.

Hönig, W.; Kumar, T. K. S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-agent path finding with kinematic constraints. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26, 477–485.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J. W.; and Ayanian, N. 2019. Persistent and Robust Execution of MAPF Schedules in Warehouses. *IEEE Robotics and Automation Letters*, 4(2): 1125–1131.

Kou, N. M.; Peng, C.; Yan, X.; Yang, Z.; Liu, H.; Zhou, K.; Zhao, H.; Zhu, L.; and Xu, Y. 2019. Multi-agent path planning with non-constant velocity motion. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, 2069–2071.

Li, J.; Gong, M.; Liang, Z.; Liu, W.; Tong, Z.; Yi, L.; Morris, R.; Pasearanu, C.; and Koenig, S. 2019. Departure scheduling and taxiway path planning under uncertainty. In *Proceedings of the AIAA Aviation Forum*, 2930–2937.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 11272–11281.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 7643–7650.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 5628–5635.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66.

Solis, I.; Motes, J.; Sandström, R.; and Amato, N. M. 2021. Representation-Optimal Multi-Robot Motion Planning Using Conflict-Based Search. *IEEE Robotics and Automation Letters*, 6(3): 4608–4615.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, 151–159.

Varambally, S.; Li, J.; and Koenig, S. 2022. Which MAPF Model Works Best for Automated Warehousing? In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 190–198.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 534–540.

Yan, J.; and Li, J. 2024. Multi-Agent Motion Planning With Bézier Curve Optimization Under Kinodynamic Constraints. *IEEE Robotics and Automation Letters*, 9(3): 3021–3028.

Zhang, H.; Tiruviluamala, N.; Koenig, S.; and Kumar, T. S. 2021. Temporal reasoning with kinodynamic networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 415–425.