Loosely Synchronized Rule-Based Planning for Multi-Agent Path Finding with Asynchronous Actions

Shuai Zhou[2*], **Shizhe Zhao**[1], **Zhongqiang Ren**[1,3†]

[1]UM-SJTU Joint Institute, Shanghai Jiao Tong University, China
[2]SHIEN-MING WU School of Intelligent Engineering, South China University of Technology, China
[3]Department of Automation, Shanghai Jiao Tong University, China
davidzhou718@gmail.com, {shizhe.zhao,zhongqiang.ren}@sjtu.edu.cn

Abstract

Multi-Agent Path Finding (MAPF) seeks collision-free paths for multiple agents from their respective starting locations to their respective goal locations while minimizing path costs. Although many MAPF algorithms were developed and can handle up to thousands of agents, they usually rely on the assumption that each action of the agent takes a time unit, and the actions of all agents are synchronized in a sense that the actions of agents start at the same discrete time step, which may limit their use in practice. Only a few algorithms were developed to address asynchronous actions, and they all lie on one end of the spectrum, focusing on finding optimal solutions with limited scalability. This paper develops new planners that lie on the other end of the spectrum, trading off solution quality for scalability, by finding an unbounded sub-optimal solution for many agents. Our method leverages both search methods (LSS) in handling asynchronous actions and rule-based planning methods (PIBT) for MAPF. We analyze the properties of our method and test it against several baselines with up to 1000 agents in various maps. Given a runtime limit, our method can handle an order of magnitude more agents than the baselines with about 25% longer makespan.

Code — https://github.com/rap-lab-org/public_LSRP
Extended version — https://arxiv.org/abs/2412.11678

1 Introduction

Multi-Agent Path Finding (MAPF) computes collision-free paths for multiple agents from their starting locations to destinations within a shared environment, while minimizing the path costs, which arises in applications such as warehouse logistics. Usually the environment is represented by a graph, where vertices represent the location that the agent can reach, and edges represent the transition between two locations. MAPF is NP-hard to solve to optimality (Yu and LaValle 2013), and a variety of MAPF planners were developed, ranging from optimal planners (Sharon et al. 2015; Wagner and Choset 2015), bounded sub-optimal planners (Barer et al. 2014; Li, Ruml, and Koenig 2021) to unbounded sub-optimal planners (Okumura et al. 2022; De Wilde, Ter Mors, and Witteveen 2013). These planners often rely on the assumption that each action of any agent takes the same duration, i.e., a time unit, and the actions of all agents are synchronized, in a sense that, the action of each agent starts at the same discrete time step. This assumption limits the application of MAPF planners, especially when the agent speeds are different or an agent has to vary its speed when going through different edges.

To get rid of this assumption on synchronous actions, MAPF variants such as Continuous-Time MAPF (Andreychuk et al. 2022), MAPF with Asynchronous Actions (Ren, Rathinam, and Choset 2021), MAPF$_R$ (Walker, Sturtevant, and Felner 2018) were proposed, and only a few algorithms were developed to solve these problems. Most of these algorithms lie on one end of the spectrum, finding optimal or bounded sub-optimal solutions at the cost of limited scalability as the number of agents grows. To name a few, Continuous-Time Conflict-Based Search (CCBS) (Andreychuk et al. 2022) extends the well-known Conflict-Based Search (CBS) to handle various action times and is able to find an optimal solution to the problem. Loosely Synchronized Search (LSS) (Ren, Rathinam, and Choset 2021) extends A* and M* (Wagner and Choset 2015) to handle the problem and can be combined with heuristic inflation to obtain bounded sub-optimal solutions. Although these algorithms can provide solution quality guarantees, they can handle only a small amount of agents (≤ 100) within a runtime limit of few minutes. Currently, we are not aware of any algorithm that can scale up to hundreds of agents with asynchronous actions. This paper seeks to develop new algorithms that lie on the other end of the spectrum, trading off completeness and solution optimality for scalability.

When all agents' actions are synchronous, the existing MAPF algorithms, such as rule-based planning (Erdmann and Lozano-Perez 1987; Luna and Bekris 2011; De Wilde, Ter Mors, and Witteveen 2013; Okumura et al. 2022), can readily scale up to thousands of agents by finding an unbounded sub-optimal solution. However, extending them to handle asynchronous actions introduce additional challenges: Rule-based planning usually relies on the notion of time step where all agents take actions and plans forward in a step-by-step fashion. When the actions of agents are of various duration, there is no notion of planning steps,

[*]Shuai Zhou conducted this research during his internship at UM-SJTU Joint Institute at Shanghai Jiao Tong University.
[†]Corresponding Author.

and one may have to plan multiple actions for a fast moving agent and only one action for a slow agent. In other words, an action with long duration of an agent may affect multiple subsequent actions of another agent, which thus complicates the interaction among the agents. To handle these challenges, we leverage the state space proposed in (Ren, Rathinam, and Choset 2021) where times are incorporated, and leverage Priority Inheritance with Backtracking (PIBT) (Okumura et al. 2022), a recent and fast rule-based planner, to this new state space. We introduce a cache mechanism to loosely synchronize the actions of agents during the search, when these actions have different, yet close, starting times. We therefore name our approach Loosely Synchronized Rule-based Planning (LSRP).

We analyze the theoretic properties of LSRP and show that LSRP guarantees reachability in graphs when the graph satisfies certain conditions. For the experiments, we compare LSRP against several baselines including CCBS (Andreychuk et al. 2022) and prioritized planning in various maps from a MAPF benchmark (Stern et al. 2019), and the results show that LSRP can solve up to an order of magnitude more agents than existing methods with low runtime, despite about 25% longer makespan. Additionally, our asynchronous planning method produces better solutions, whose makespan ranges from 55% to 90% of those planned by ignoring the asynchronous actions.

2 Problem Definition

Let set $I = \{1, 2, \dots, N\}$ denote a set of N agents. All agents move in a workspace represented as a finite graph $G = (V, E)$, where the vertex set V represents all possible locations of agents and the edge set $E \subseteq V \times V$ denotes the set of all the possible actions that can move an agent between a pair of vertices in V. An edge between $u, v \in V$ is denoted as $(u, v) \in E$ and the cost of $e \in E$ is a finite positive real number $cost(e) \in \mathbb{R}^+$. Let $v_s^i, v_g^i \in V$ respectively denote the start and goal location of agent i.

All agents share a global clock and start moving from v_s^i at t = 0. Let $D(i, v_1, v_2) \in \mathbb{R}^+$ denote amount of time (i.e., duration) for agent i to go through edge (v_1, v_2). Note that different agents may have different duration when traversing the same edge, and the same agent may have different duration when traversing different edges.[1]

When agent i goes through an edge $(v_1, v_2) \in E$ between times $(t_1, t_1 + D(i, v_1, v_2))$, agent i is defined to occupy: (1) v_1 at $t = t_1$, (2) v_2 at $t = t_2$ and (3) both v_1,v_2 within the open interval $(t_1, t_1 + D(i, v_1, v_2))$. Two agents are in conflict if they occupy the same vertex at the same time. We refer to this definition of conflict as the *duration conflict* hereafter. Fig. 1 provides an illustration.

Let π^i denote a path from v_s^i to v_g^i via a sequence of vertices $v \in G$. Any two vertices v_k^i and v_{k+1}^i in π^i are either connected by edge $(v^i, v_{+1}^i) \in E$ or is a self-loop.

[1] As a special case, a self-loop $(v, v), v \in V$ indicates a wait-in-place action of an agent and its duration $D(i, v, v)$ is the amount of waiting time at vertex v, which can be any non-negative real number and is to be determined by the planner.

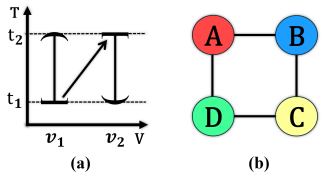

Figure 1: (a) shows the occupation of vertices when an agent traverses edge (v_1, v_2) between times (t_1, t_2) as shown by the black arrow. Round brackets represent open intervals, while square brackets represent closed intervals. The black vertical lines mean the vertices are occupied during the time range. (b) By duration conflict, the agents cannot move, while in conventional MAPF (Stern et al. 2019), the agents can move together clockwise or counter-clockwise.

Let $g(\pi^i(v_s^i, v_g^i))$ denote the cost of the path, which is defined as the sum of duration of edges along the path. Let $\pi = (\pi^1, \pi^2, \dots, \pi^n)$ represent a joint path of all agents, and its cost is the sum of individual path costs of all the agents, i.e., $g(\pi) = \sum_i g(\pi^i)$.

The goal of the Multi-Agent Path Finding with Asynchronous Actions (MAPF-AA) is to find a conflict-free joint path π connecting v_s^i, v_g^i for all agents $i \in I$, such that $g(\pi)$ reaches the minimum. This work seeks to develop algorithms that can quickly solve MAPF-AA instances with many agents by finding a (unbounded) sub-optimal solution.

Remark 1 *The same definition of occupation and conflict was introduced in (Ren, Rathinam, and Choset 2021), which generalizes the notion of following conflict and cycle conflict in (Stern et al. 2019), and is similar to the "mode" in (Okumura, Tamura, and Défago 2021). Conventional MAPF usually considers vertex and edge conflicts (Sharon et al. 2015; Okumura 2023), which differ from the duration conflict here. Another related problem definition is MAPF with duration conflict (denoted as MAPF-DC), which replaces the vertex and edge conflict in MAPF with duration conflict. MAPF-DC is a special case of MAPF-AA where the duration is the same constant number for any agent and any edge.*

3 Preliminaries

3.1 Priority Inheritance with Backtracking

Priority Inheritance with Backtracking (PIBT) plans the actions of the agents in a step-by-step manner until all agents reach their goals. PIBT assigns each agent a changing priority value. In each step, a planning function is called to plan the next action of the agents based on their current priorities. This planning function selects actions based on the individual shortest path to the goal of each agent, and actions toward a location closer to the goal are first selected. When two agents seek to occupy the same position, the higher-priority agent is able to take this location, and pushes the lower-priority agent to another less desired location. This function is applied recursively, where the pushed agent is planned next and inherits the priority of the pushing agent. When all agents' actions are planned for the current time step, PIBT starts a new iteration to plan the next time step.

PIBT guarantees that the agent with the highest priority eventually reaches its goal, at which it becomes the lowest priority agent. Therefore, each agent becomes the highest priority agent at least once and is able to reach its goal at some time step. PIBT requires that for each vertex $v \in G$, there is a cycle in G containing v, so that PIBT can plan all agents to their goals. Otherwise, PIBT is incomplete, i.e., PIBT may not be able to find a feasible solution even if the instance is solvable. PIBT runs fast and can scale to many agents for MAPF. Our LSRP leverages the idea of PIBT to handle a large number of agents.

3.2 Loosely Synchronized Search

Loosely Synchronized Search (LSS) extends A* and M*-based approaches to solve MAPF-AA by introducing new search states that include both the locations and the action times (i.e., as timestamps) of the agents. Similarly to A*, LSS iteratively selects states from an open list, expands the states to generate new states, prunes states that are in-conflict or less promising, and adds remaining states to open for future expansion, until a conflict-free joint path for all agents is found from the starts to the goals. To expand a state, LSS only considers the agent(s) $i \in I$ with the smallest timestamps and plan its actions, which increases the timestamp of agent i. In a future iteration, other agents $j \neq i$ will be planned if the timestamp of j becomes the smallest. Planning all agents together may lead to a large branching factor and LSS leverage M* to remedy this issue. LSS is complete and finds an optimal solution for MAPF-AA but can only handle a relatively small amount of agents. Our LSRP leverages the state definition and expansion in LSS to handle asynchronous actions.

3.3 Other Related Approaches

Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011) is a single-agent graph search algorithm that can find a path from start to goal with the minimum arrival time among dynamic obstacles along known trajectories. SIPP can be used together with priority-based planning to handle MAPF-AA. Specifically, each agent is assigned with a unique priority, and the agents are planned from the highest priority to the lowest using SIPP, where the planned agents are treated as dynamic obstacles. This priority-based method is used as a baseline in our experiments.

Additionally, CCBS (Andreychuk et al. 2022) is a two-level search algorithm that can be used to handle MAPF-AA. CCBS is similar to CBS (Sharon et al. 2015) for MAPF. The high-level search detects conflicts between any pair of agents, and resolves conflicts by generating constraints that forbid an agent from using certain vertices within certain time ranges. The low-level search uses SIPP to plan a single-agent path subject to the constraints added by the high-level. CCBS iteratively detects conflicts on the high-level and resolves conflicts using the low-level search until no conflict is detect along the paths. CCBS is guaranteed to find an optimal solution if the given problem instance is solvable. In practice, CCBS can handle tens of agents within a few minutes (Andreychuk et al. 2022). This paper uses CCBS as another baseline in the experiments.

Algorithm 1: LSRP, LSRP-SWAP

Input: $graph\ \mathcal{G}, starts\ \{v_s^1, \dots, v_s^n\}, goals\ \{v_g^1, \dots, v_g^n\}$
Output: $paths\ \{\pi^1, \dots, \pi^n\}$

$T \leftarrow \{0\}$; $S_T \leftarrow \{s_0\}$; $\Phi \leftarrow \{\}$
$\epsilon_0 \leftarrow$ INITPRIORITY()
$\epsilon \leftarrow \epsilon_0$
while $T \neq \emptyset$ **do**
  $s_{prev} \leftarrow S_T.back()$
  if $\forall i \in I, s_{prev}^i.v = v_g^i$ **then**
    return POSTPROCESS(S_T)
  for $i \in I$ **do**
    if $s_{prev}^i.v = v_g^i$ **then** $\epsilon^i \leftarrow \epsilon_0^i$
    else $\epsilon^i \leftarrow \epsilon^i + 1$
  $t_{min} \leftarrow T.pop()$
  $I_{curr} \leftarrow$ EXTRACTAGENTS(I, t_{min}, s_{prev})
  if $T \neq \emptyset$ **then**
    $t_{next} \leftarrow T.top()$
  else
    $t_{next} \leftarrow t_{min} + \min_{i \in I, e \in E} D(i, e)$
  $s_{next} \leftarrow$ GET_SNEXT(Φ, I_{curr}, s_{prev})
  for $i \in I_{curr}$ in descending order of ϵ^i **do**
    if $s_{next}^i = \emptyset$ **then**
      ASY-PUSH($i, \{\}, t_{min}, t_{next}$, False)
      (or ASY-PUSH-SWAP($i, \{\}, t_{min}, t_{next}$, False))
  $S_T.append(s_{next})$
  Add $s_{prev}^i.t_v$ for $i \in I$ to T
return failure

4 Method

4.1 Notation and State Definition

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}) = G \times G \times \cdots \times G$ denote the joint graph, the Cartesian product of N copies of G, where $v \in \mathcal{V}$ represents a joint vertex, and $e \in \mathcal{E}$ represents a joint edge that connects a pair of joint vertices. The joint vertices corresponding to the start and goal vertices of all the agents are $v_s = (v_s^1, v_s^2, \cdots, v_s^n)$ and $v_g = (v_g^1, v_g^2, \cdots, v_g^n)$ respectively. A joint search state (Ren, Rathinam, and Choset 2021) is $s = (s^1, s^2, \dots, s^n)$, where s^i is the individual state of agent i, which consists of four components: (1) $p \in V$, a (parent) vertex in G, from which the agent i begin its action; (2) $v \in V$, a vertex in G, at which the agent i arrives; (3) t_p, the timestamp of p, representing the departure from p; (4) t_v, the timestamp of v, representing the arrival time at v.

An individual state $s^i = (p, v, t_p, t_v)$ describe the location occupied by agent i within time interval $[t_p, t_s]$ with a pair of vertices (p, v). Intuitively, an individual state is also an action of agent i, where i moves from vertex p to v between timestamps t_p and t_v. For the initial state s_0, we define $p = v = v_s^i$ and $t_p = t_v = 0, \forall i \in I$. Let $\epsilon = (\epsilon^1, \epsilon^2, \cdots, \epsilon^N)$ denote the priorities of the agents, which are positive real numbers in $[0, 1]$. In this paper, we use the dot operator (.) to retrieve the element inside the variable (e.g. $s^i.t_v$ denote the arrival time t_v of agent i in the individual state s^i).

4.2 Algorithm Overview

LSRP is shown in Alg. 1. Let T denote a list of timestamps, and LSRP initializes T as a list containing only 0, the start-

ing timestamp of all agents. Let S_T denote a list of joint states, which initially contains only the initial joint state. Let Φ denote a dictionary where the keys are the timestamps and the values are joint states. Φ is used to cache the planned actions of the agents and will be explained later. Then, LSRP initializes the priorities of the agents (Line 2), which can be set in different ways (such as using random numbers).

LSRP plans the actions of the agents iteratively until all agents reach their goals. LSRP searches in a depth-first fashion. In each iteration, LSRP retrieves the most recent joint state that was added to S_T, and denote it as s_{prev} (Line 5). If all agents have reached their goals in s_{prev}, S_T now stores a list of joint states that brings all agents from v_s to v_g. LSRP thus builds a joint path out of S_T, which is returned as the solution (Line 7), and then terminates. Otherwise, LSRP resets the priorities of the agents that have reached their goals, and increase the priority by one for agents that have not reached their goals yet (Lines 9-10).

Then, LSRP extracts the next planning timestamp t_{min} from T, which is the minimum timestamp in T, and extracts the subset of agents $I_{curr} \subseteq I$ so that for each agent $i \in I$, its arrival timestamp t_v^i in s_{prev} is equal to t_{min} (Line 12). Intuitively, the agents in I_{curr} needs to determine their next actions at time t_{min}. Here, t_{next} is assigned to be the next planning timestamp in T if T is not empty. Otherwise (T is empty), t_{next} is set to be t_{min} plus the a small amount of time (Lines 13-16). To generate the next joint state based on s_{prev}, LSRP first checks if the actions of agents in I_{curr} have been planned and cached in Φ. If so, these cached actions are used and the corresponding individual states are generated and added to s_{next} (Line 17). For agents without cached actions (Line 19), LSRP invokes a procedure ASY-PUSH to plan the next action for that agent.

Finally, the generated next joint state s_{next} is appended to the end of S_T, and the arrival timestamp $s_{prev}^i.t_v$ of each agent $i \in I$ is added to T for future planning.

4.3 Recursive Asynchronous Push

ASY-PUSH takes the following inputs: i, the agent to be planned; ban, a list of vertices that agent i is banned from moving to; t, the current timestamp; t_{next}, the next timestamp; and bp, a boolean value indicating if agent i is being pushed away by other agents, which stands for "be pushed".

At the beginning, ASY-PUSH identifies all adjacent vertices C that agent i can reach from its current vertex in G. These vertices in C are sorted based on its distance to the agent's goal (Line 1-2, Alg.2). Then, for each of these vertices from the closest to the furthest, ASY-PUSH checks whether the agent can move to that vertex without running into conflicts with other agents, based on the occupancy status of that vertex (Line 7-27, Alg.2). ASY-PUSH stops as soon as a valid vertex (i.e., a vertex that is unoccupied or can be made unoccupied through the push operation) is found. Specifically, between lines 7-27 in Alg.2, ASY-PUSH may run into one of the following three cases:

Case 1 Occupied (Line 8): The procedure OCCUPIED returns true if either one of the following three conditions hold. (1) Vertex v is inside ban, which means agent i cannot be pushed to v. (2) v is occupied by another agent i' and i' either has been planned or i' is not in I_{curr}. (3) bp is true (which indicates that agent i is to be pushed away) and v is the vertex currently occupied by agent i (i.e., $v = s_{prev}^i.v$). When OCCUPIED returns true, agent i cannot move to v. Then, ASY-PUSH ends the current iteration of the for-loop, and checks the next vertex in C.

Case 2 Pushable (Line 9-18): ASY-PUSH invokes PUSH-REQUIRED to find if vertex v is occupied by another agent k that satisfy the following two conditions: (1) $k \in I_{curr}$ and (2) the action of k has not yet been planned (Line 9). If no such a k is found, ASY-PUSH goes to the "Unoccupied" case as explained next. If such a k is found, the current vertex v^i occupied by agent i is added to the list ban so that agent k will not try to push agent i in a future recursive call of ASY-PUSH, which can thus prevent cyclic push in the recursive ASY-PUSH calls. Then, a recursive

Algorithm 2: ASY-PUSH, ASY-PUSH-SWAP

Input: i, ban, t, t_{next}, bp
Notation: $v^i \leftarrow s_{prev}^i.v$

1: $C \leftarrow \text{Neigh}(v^i) \cup \{v^i\}$
2: sort C in increasing order of $\text{dist}(u, v_g^i)$ where $u \in C$
3: $j \leftarrow$ SWAP-REQUIRED-POSSIBLE$(i, C[0])$
4: **if** $j \neq \emptyset$ **then** $C.reverse()$
5: **if** $\epsilon^i > \epsilon^n (n \neq i) \forall n \in I$ **then**
6: $\quad C.move(v^i, 1)$ ▷ Move v^i to second
7: **for** $v \in C$ **do**
8: $\quad$ **if** OCCUPIED(v, s_{next}, ban, bp) **then continue**
9: $\quad k \leftarrow$ PUSH-REQUIRED()
10: $\quad$ **if** $k \neq \emptyset$ **then**
11: $\quad\quad ban.append(v^i)$
12: $\quad\quad t_{wait}^i \leftarrow$ ASY-PUSH(k, ban, t, t_{next}, True)
13: $\quad\quad$ **if** $t_{wait}^i = \emptyset$ **then continue**
14: $\quad\quad$ WAITANDMOVE$(i, v, t_{wait}^i, s_{next}, \Phi)$
15: $\quad\quad t_{move}^i \leftarrow t_{wait}^i + D(i, v^i, v)$
16: $\quad\quad$ **if** $(!bp) \wedge v = C[0] \wedge j \neq \emptyset \wedge s_{next}^j = \emptyset$ **then**
17: $\quad\quad\quad$ WAITANDMOVE$(j, v^i, t_{move}^i, s_{next}, \Phi)$
18: $\quad\quad$ **return** t_{move}^i
19: $\quad$ **if** $v = v^i$ **then**
20: $\quad\quad s_{next}^i \leftarrow state(v^i, v, t, t_{next})$
21: $\quad\quad$ **return**
22: $\quad t_{move}^i \leftarrow t + D(i, v^i, v)$
23: $\quad s_{next}^i \leftarrow state(v^i, v, t, t_{move}^i)$
24: $\quad$ **if** $(!bp) \wedge v = C[0] \wedge j \neq \emptyset \wedge s_{next}^j = \emptyset$ **then**
25: $\quad\quad$ WAITANDMOVE$(j, v^i, t_{move}^i, s_{next}, \Phi)$
26: $\quad$ **return** t_{move}^i
27: **return** $\emptyset$

Algorithm 3: WAITANDMOVE

Input: $i, v, t_{wait}^i, s_{prev}, s_{next}, \Phi$
Notation: $v^i \leftarrow s_{prev}^i.v$

1: $s_{next}^i \leftarrow state(v^i, v^i, t, t_{wait}^i)$
2: $t_{move}^i \leftarrow t_{wait}^i + D(i, v^i, v)$
3: Add $state(v^i, v, t_{wait}^i, t_{move}^i)$ to Φ
4: **return**

call of ASY-PUSH on agent k is invoked and the input argument bp is marked true, indicating that agent k is pushed by some other agent. bp is also used in SWAP-related procedures (Line 3-4,16-17 and 24-25), which will be explained later. This recursive call returns the timestamp when agent k finished its next action, and agent i has to wait till this timestamp, which is denoted as t^i_{wait}. Given t^i_{wait}, the procedure WAITANDMOVE is invoked to add both the wait action and the subsequent move action of agent i into Φ, the dictionary storing all cached actions. Then, the timestamp t^i_{move} when agent i reaches v is computed (Line 18) and returned.

Case 3 Unoccupied (Line 19-26): Vertex v is valid for agent i to move into and a successor individual state s^i_{next} for agent i is generated (Line 23). Then, ASY-PUSH returns the timestamp t^i_{move} when agent i arrives at v (Line 26).

Cache Future Actions During the search, Φ caches the planned actions of the agents, and is updated in Alg.3, which takes an agent $i \in I$, a vertex $v \in V$, the timestamp t^i_{wait} that agent i needs to wait before moving as the input. Alg.3 first generates the corresponding individual state s^i_{next} where agent i waits in place till t^i_{wait}(Line 1), and then calculates time t^i_{move} when agent i reaches v after the wait (Line 2). Finally, the future individual state corresponding to the move action of agent i from v^i to v between timestamps $[t^i_{wait}, t^i_{move}]$ is generated and stored in Φ.

Toy Example Fig. 2 shows an example in an undirected graph with three agents $I = \{1, 2, 3\}$ corresponding to the yellow, blue, and red circles. The duration for the agents to go through any edge is 1, 2, 3 respectively. S_0 shows the initial individual states. The initial priorities of the agents are set to $\{0.99, 0.66, 0.33\}$ respectively. At timestamp $t_{min} = 0$, I_{curr} includes all three agents. LSRP calls ASY-PUSH based on their priorities from the largest to the smallest.

LSRP calls ASY-PUSH with $i = 1$, and agent 1 attempts to move to vertex D as D is the closest to its goal. D is now occupied by agent 2, which leads to a recursive call on ASY-PUSH with $i = 2$ to check if agent 2 can be pushed away (Line 12, Alg.2). In this recursive call on $i = 2$, agent 2 tries to move to vertex B as B is the closest to its goal, which is occupied by agent 3, and another recursive call on ASY-PUSH with $i = 3$ is conducted. In this recursive call on ASY-PUSH with $i = 3$, agent 3 finds its goal vertex C is unoccupied, and $s^3_{next} = (B, C, 0, 3)$ is generated, which is shown in Fig. 2(b). Here, the returned timestamp is $t^3_{move} = 3$, which is the timestamp when agent 3 arrives vertex C. Then, the call on ASY-PUSH with $i = 2$ gets this returned timestamp and set it as $t^2_{wait} = 3$, the timestamp that agent 2 needs to wait till before moving to vertex D. An individual state $s^2_{next} = (D, D, 0, 3)$ is generated, which is s^2_{next} in Fig. 2(b), and another individual state $(D, B, 3, 5)$ is created and stored in Φ, which corresponds to the move action of agent 2 from D to B. Finally, the call on ASY-PUSH with $i = 1$ receives $t^1_{wait} = 5$, which is the timestamp that agent 1 has to wait till before moving to D. An individual state $s^1_{next} = (E, E, 0, 5)$ is generated, corresponding to the wait action of agent 1, which is s^1_{next} in Fig. 2(b). Then another individual state $(E, D, 5, 6)$ is also created and stored in Φ as the future move action of agent 1. Finally, all three

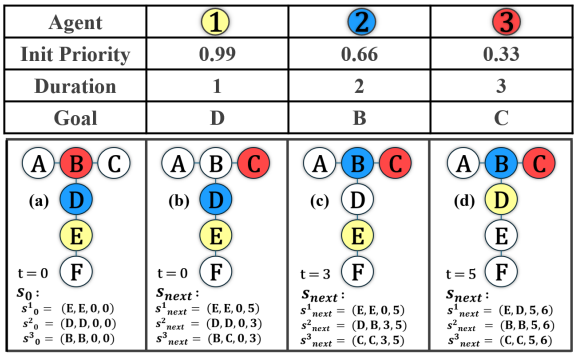

Figure 2: A toy example illustrating LSRP.

agents are planned and s_{next} is added to S_T. The arrival timestamps t_v in any individual state in s_{next} (i.e., {3,5}) are added to T for future planning.

In the next iteration of LSRP, the planning timestamp is $t_{min} = 3$ and I_{curr} is $\{2, 3\}$, as agents $i = 2, 3$ ends their previous actions at $t = 3$. To plan the next action of agents 2, 3, LSRP retrieves the move action from Φ and set $s^2_{next} = (D, B, 3, 5)$ for agent 2, which is s^2_{next} shown in Fig. 2(c). Agent 3 has no cached action in Φ and is planned by calling ASY-PUSH. Since agent 3 is at its goal, ASY-PUSH set agent 3 wait in vertex C until t_{next} by setting $s^3_{next} = (C, C, 3, 5)$ (Line 19-21,Alg.2). All agents in I_{curr} are now planned, and S_T,T are updated. LSRP ends this iteration and proceeds. The third planning timestamp is $t_{min} = 5$ and I_{curr} is $\{1, 2, 3\}$. LSRP plans in a similar way, and the movement of the agents is shown in Fig. 2(d). The last timestamp is $t_{min} = 6$. In this iteration, in s_{prev}, all agents have reached their goals and LSRP terminates.

4.4 Relationship to PIBT and Causal PIBT

LSRP differs from PIBT in the following three aspects: **First**, PIBT solves MAPF where vertex and edge conflicts are considered. When one seeks to use PIBT-like approach to solve MAPF-DC (with duration conflict), the wait action may need to be considered in a similar way as LSRP does. Specifically, when two agents $i, j \in I_{curr}$ compete for the same vertex v, the higher-priority agent starts to move to v until the lower-priority agent is pushed away from v and reaches a less desired vertex u. Here, the wait and the move action of i are planned together and the move action is cached in Φ for future execution. In PIBT, agent i can move to v that is currently occupied by j as soon as j leaves v, and there is no need for agent i to wait and cache the move action. **Second**, different from MAPF-DC, MAPF-AA has various durations. As a result, in each iteration (with planning timestamp t), LSRP plans for agents whose arrival time is equal to t, instead of planning all agents as PIBT does. **Third**, LSRP also introduces a swap operation for MAPF-AA, which is demonstrated in Sec. 6.

Causal PIBT (Okumura, Tamura, and Défago 2021) extends PIBT to handle delays caused by imperfect execution of the planned path for MAPF. In the time-independent planning problem, due to the possible delays, the action duration

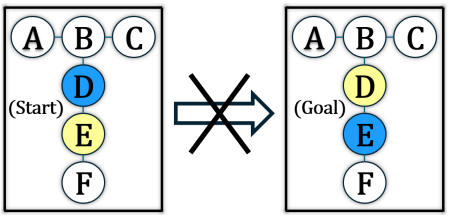

Figure 3: An example of LSRP never terminating with only ASY-PUSH. The blue and yellow agents will repeatedly push each other but will never successfully swap their locations to reach their goals.

is unknown until the action is finished. Causal PIBT thus plans "passively" by recording the dependency among the agents during execution using tree-like data structure. When one agent moves, Causal PIBT signals all the related agents based on their dependency. In MAPF-AA, the action duration is known, and LSRP thus plans "actively" by looking ahead into the future given the action duration of the agents, and caches the planned actions when needed. The fundamental ideas in LSRP and Causal PIBT are related and similar, but the algorithms are different since they are solving different problems.

5 Analysis

This section discusses the compromise on completeness for scalability in LSRP. We borrow two concepts from Causal-PIBT (Okumura, Tamura, and Défago 2021):

- *Strong termination*: there is time point where all agents are at their goals.
- *Weak termination*: all agents have reached their goals at least once.

LSRP only guarantees *weak termination* under conditions. In this section, the words 'reach goal' means that an agent reaches its goal location but may not stay there afterwards.

Given a graph G, a cycle $\{v_1, v_2, \ldots, v_\ell, v_1\}$ is a special path that starts and ends at the same vertex v_1. The length of a path (or a cycle) is the number of vertices in it. Given a graph G and N agents, if there exists a cycle of length $\geq N+1$ for all pairs of adjacent vertices in G, then we call this graph a *c-graph*. Intuitively, for a c-graph, LSRP guarantees that the agent with highest priority can push away any other agents that block its way to its goal, and reaches its goal within finite time. Once the agent reaches its goal, its priority is reset and thus becomes a small value, the priority of another agent becomes the highest and can move towards its goal. As a result, all agents are able to reach their goals at a certain time. Note that LSRP initializes all agents with a unique priority, and all agents' priorities are increased by one in each iteration of LSRP, unless the agent has reached its goal. Let $i_* \in I$ denote the agent with the highest priority when initializing LSRP. Let D_{max} denote the largest duration for any agent and any edge.

Theorem 1 *In a c-graph, in LSRP, when $i_* \in I_{curr}$, let v^* denote the nearest vertex from $v_g^{i_*}$ among all vertices in C, then i_* can reach v^* within time $N \cdot D_{max}$.*

Sketch Proof 1 *In each iteration, LSRP extracts the minimum timestamp t_{min} from T, and at the end of the iteration, the newly added timestamps to T must increase and be greater than t_{min}. As a result, t_{min} keeps increasing as LSRP iterates and all agents are planned. Now, consider the iterations of LSRP where agent $i_* \in I_{curr}$ and is planned. In* ASY-PUSH*, v^* is check at first. If no other agent (k) occupies v^*, then i_* reaches v^* with a duration that is no larger than D_{max}. Otherwise (i.e., another agent k occupies v^*), agent i_* seeks to push k to another vertices which may or may not be occupied by a third agent k'. Since the graph is a c-graph, there must be at least one unoccupied vertex in the cycle containing the $s_{prev}^{i_*}.v$, the current vertex occupied by i_*. In the worst case, all agents are inside this cycle and i_* has to push all other agents before i_* can reach v_*, which takes time at most $N \cdot D_{max}$, where the agent moves to its subsequent vertex in this cycle one after another.*

Let $diam(G)$ denote the diameter of G, the length of the longest path between any pair of vertices in G.

Theorem 2 *For a c-graph, LSRP (Alg. 1) returns a set of conflict-free paths such that for any agent $i \in I$ reaches its goal at a timestamp t with $t \leq diam(G) \cdot N^2 \cdot D_{max}$.*

Sketch Proof 2 *From Theorem 1, the agent with the highest priority arrives v^* within $N \cdot D_{max}$. So agent i arrives v_g^i within $diam(G) \cdot N \cdot D_{max}$. Once agent i reaches v_g^i, its priority is reset, which must be smaller than the priority of any other agents, and another agent j gains the highest priority and is able to reach its goal. This process continues until all agents in I have gained the highest priority at least once, and reached their goals. Thus, the total time for all agents to achieve their goals is within $diam(G) \cdot N^2 \cdot D_{max}$.*

6 Extension with Swap Operation

Since LSRP only guarantees weak termination under conditions as aforementioned, for MAPF-AA in general, there are instances where LSRP never terminates although the instance is solvable. Fig. 3 shows an example. Assume the yellow agent has higher initial priority than the blue agent. The yellow agent first pushes the blue agent until reaches goal D. Then, the blue agent has higher priority and pushes the yellow agent until reaches goal E. As a result, these two agents iteratively push each other and can never successfully swap their locations.

Similar issues also appear in PIBT, which is addressed by an additional swap operation (Okumura et al. 2022). Inspired by this, we develop ASY-PUSH-SWAP (Alg. 2). ASY-PUSH-SWAP takes the same input as ASY-PUSH, and invokes a procedure SWAP-POSSIBLE-REQUIRED to check if there exists an agent j that needs to swap location with agent i. If such a j exists, ASY-PUSH-SWAP sorts the successor vertices in C based on their distance to agent j's goal from the furthest to the nearest (Line 4, Alg. 1), and plans the actions of i and j differently: if agent i can move to the furthest vertex in C, and agent j is not planned, then j moves to i's current vertex after i vacates it (Line 16-17,24-25 in Alg. 2). We refer to this variant of LSRP with swap as LSRP-SWAP. Since LSRP-SWAP alters the order of vertices in C, the analysis in Theorem 1 based on the nearest

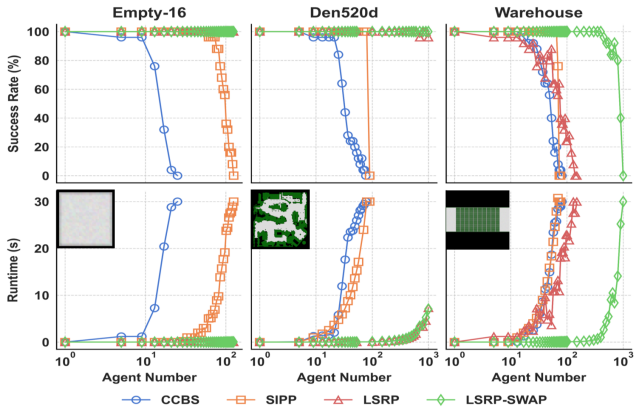

Figure 4: Success rate and runtime results

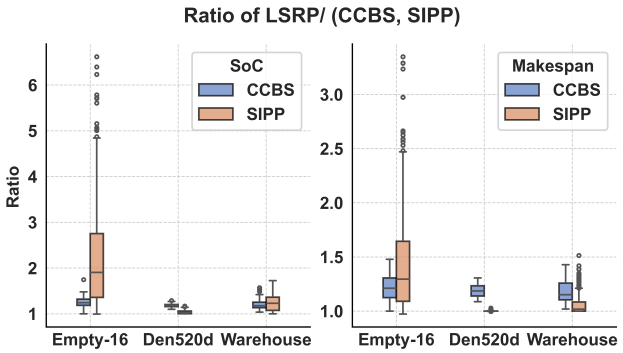

Figure 5: SoC and makespan ratios of LSRP

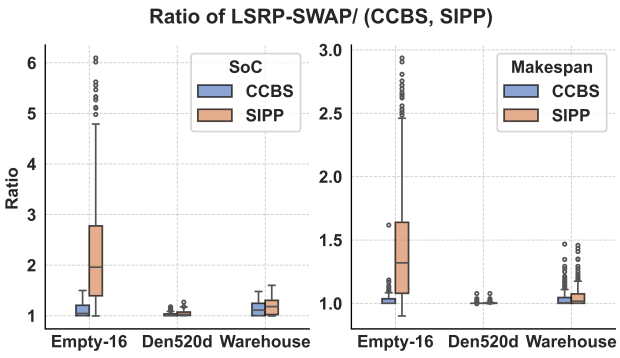

Figure 6: SoC and makespan ratios of LSRP-SWAP

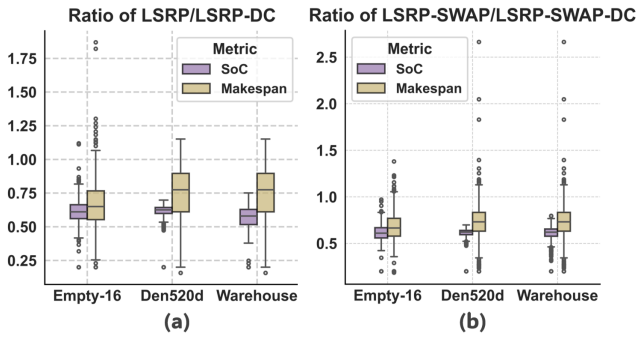

Figure 7: Solution costs comparison with and without considering asynchronous actions.

vertex v^* of an agent $i_* \in I_{curr}$ cannot be applied. Theorem 1 and 2 thus may not hold for LSRP-SWAP.

SWAP-REQUIRED-POSSIBLE is similar to the concept of the swap operation in PIBT (Okumura 2023). The main difference is that LSRP-SWAP must consider the duration conflict between agents rather than the vertex or edge conflict in PIBT. Appendix provides more detail of LSRP-SWAP.

7 Experimental Results

Our experiments use three maps and the corresponding instances (starts and goals) from a MAPF benchmark (Stern et al. 2019). For each map, we run 25 instances with varying number of agents N, and we set a 30-seconds runtime limit for each instance. We make the grid maps four-connected, and each agent has a constant duration when going through any edge in the grid. This duration constant of agents vary from 1.0 to 5.0. We implement our LSRP and LSRP-SWAP in C++, and compare against two baselines. The first baseline is a modified CCBS (Andreychuk et al. 2022). The original CCBS implementation considers the shape of the agents and does not allow different agents to have different durations when going through the same edge. We modified this public implementation by using the duration conflict and allow different agents to have different duration. Note that the constraints remain "sound" (Andreychuk et al. 2022), and the solution obtained is optimal to MAPF-AA. The second baseline SIPP (see Sec. 3.3) adopts prioritized planning using SIPP (Phillips and Likhachev 2011) as the single-agent planner. It uses the same initial priority as LSRP and LSRP-SWAP. All tests use Intel i5 CPU with 16GB RAM.

7.1 Success Rates and Runtime

Fig. 4 shows the success rates and runtime of the algorithms. Overall, it is obvious that our LSRP and LSRP-SWAP can often handle more than an order of magnitude number of agents than the baselines within the time limit with much smaller runtime. In particular, LSRP-SWAP runs fastest and handles up to 1000 agents in our tests. In sparse maps (`Empty-16`,`Den520d`), LSRP and LSRP-SWAP both scale well with respect to N, and the reason is that these graphs have many cycles that make the push operation highly efficient when resolving conflicts between the agents. In a cluttered environment (`Warehouse`), LSRP has similar performance to SIPP while LSRP-SWAP outperforms both LSRP and SIPP, which shows the advantage of the swap operation in comparison with LSRP without swap.

7.2 Solution Quality

Now we examine the solution quality of proposed methods. We measure the solution quality using both sum of costs (SoC) and makespan. We compare the solution quality (SoC and makespan) of LSRP and LSRP-SWAP with baselines

(i.e., CCBS and SIPP) respectively. The ratios A/B used for comparison are computed based on the instances from the experiment in Sec. 7.1 that were successfully solved by both planners A and B, where A is LSRP or LSRP-SWAP, and B is CCBS or SIPP. The higher the ratio, the better the solution quality of the baseline B.

As shown in Fig. 5 and 6, the median ratios is about 4x in SoC, and 1.25x in makespan, which means LSRP and LSRP-SWAP find more expensive solutions than the baselines. It indicates LSRP and LSRP-SWAP achieve high scalability at the cost of solution quality, while the baselines usually find high quality solution with limited scalability. Note that SIPP solves more instances than CCBS, and the ratios for CCBS and SIPP are thus calculated based on different sets of instances. So SIPP sometimes has a higher ratio than the optimal planner CCBS on the map `Empty-16`.

7.3 Impact of Asynchronous Actions

This compares the solution costs of LSRP and LSRP-SWAP when asynchronous actions are considered versus when they are not. We use the same instances as in Sec. 7.1. When ignoring the asynchronous actions, the duration constant of all agents are set to 5.0, and the resulting MAPF-AA problem becomes MAPF-DC, where all agents have common planning timestamps and can be planned in a step-by-step manner. Note that even for MAPF-DC, due to the duration conflict, PIBT cannot be directly applied as discussed in Sec. 4.4. As shown in Fig. 7, by considering the asynchronous actions, the obtained solutions are usually 30% and sometimes up to 75% cheaper than the solutions that ignore the asynchronous actions. This result verifies the importance of considering asynchronous actions during planning, especially when agents have very different duration.

8 Conclusion and Future Work

This paper develops rule-based planners for MAPF-AA by leveraging both PIBT for MAPF and LSS to handle asynchronous actions. The experimental results verify their ability to achieve high scalability for up to a thousand agents in various maps, at the cost of solution quality. Future work includes developing an anytime planner that can further improve solution quality within the runtime limit for MAPF-AA. Specifically, LSRP can be potentially extended to an anytime version that is similar to LaCAM over PIBT (Okumura 2023). This extension would allow LSRP to iteratively optimize solution quality within the runtime limit.

Acknowledgements

This work was supported in part by the Natural Science Foundation of China under Grant 62403313.

References

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

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the international symposium on combinatorial Search*, volume 5, 19–27.

De Wilde, B.; Ter Mors, A. W.; and Witteveen, C. 2013. Push and rotate: cooperative multi-agent path planning. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 87–94.

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

Li, J.; Ruml, W.; and Koenig, S. 2021. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, 12353–12362.

Luna, R.; and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, volume 11, 294–300.

Okumura, K. 2023. Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI)*.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2022. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. *Artificial Intelligence*, 103752.

Okumura, K.; Tamura, Y.; and Défago, X. 2021. Time-Independent Planning for Multiple Moving Agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13): 11299–11307.

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

Ren, Z.; Rathinam, S.; and Choset, H. 2021. Loosely synchronized search for multi-agent path finding with asynchronous actions. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9714–9719. IEEE.

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

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 151–158.

Wagner, G.; and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artificial intelligence*, 219: 1–24.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In *IJCAI*, 534–540.

Yu, J.; and LaValle, S. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, 1443–1449.

Appendix

This section provides the details of LSRP-SWAP, which involves first detecting whether two agents need to swap their locations, and second planning the actions of the agents to achieve the swap. LSRP-SWAP does not consider the duration conflict between the agents in the detection process. When planning the actions of the agents to achieve the swap, the duration conflict is considered on Lines 17 and 25 in Alg. 2 to let the agents wait and then move when needed.

We introduce the following concepts, and some of them have been mentioned earlier in the main paper.

SWAP: SWAP is introduced in (Okumura 2023). It is a process where two agents exchange their current vertices in a conflict-free manner through a sequence of actions.

PUSH: When i applies a PUSH operation on j, j moves to another adjacent vertex that is not currently occupied by i, then i moves to j's current vertex.

PULL: This is the reverse operation of PUSH. When i applies a PULL operation on j, j moves to i's current vertex after i moves to another vertex.

OCCUPANT is a procedure that is used in Alg. 4 Line 3. It takes $(v, I_{curr}, s_{prev}, s_{next})$ as the input, and seeks to find an agent $j \in I_{curr}$ that currently occupies v and has not yet been planned, according to s_{prev} and s_{next}. If no such an agent j exists, **OCCUPANT** returns an empty set.

SWAP-REQUIRED-POSSIBLE

The procedure SWAP-REQUIRED-POSSIBLE is called in Alg. 2. Recall that v_g^i denotes the goal vertex of agent $i \in I$, and C denotes a set of vertices that agent i can reach from its current vertex, sorted according to the distance from v_g^i from the nearest to the furthest (Alg. 2).

Alg. 4 takes as input an agent $i \in I$ and a vertex v in C that is the closest to v_g^i. Here, $s_{prev}, s_{next}, I_{curr}$ are "global" variables that are created on in Alg. 1 at the beginning of each iteration of LSRP-SWAP. When $v = v^i$ (Line 2), it guarantees that $v = v^i = v_g^i$, which means agent i reaches the goal and i does not need to swap vertices with others to get closer to v_g^i, and Alg. 4 ends. Otherwise ($v \neq v^i$), it invokes OCCUPANT to find an agent j to swap with.

If OCCUPANT finds an agent j ($j \neq \emptyset$), it indicates that, agent i can swap with agent j to get closer to v_g^i. Then, it invokes SWAP-CHECK to check if it is enough to eventually swap agents i and j by just using a sequence of pull operations of agent j to agent i (Line 6). If SWAP-CHECK returns true, it means that i, j cannot be swapped by just using a sequence of pull operations. If SWAP-CHECK returns false, it means that i, j can be swapped by just using a sequence of pull operations, or agent i, j do not need to be swapped at all. On Line 6, when SWAP-CHECK returns true, another SWAP-CHECK is applied (Line 7) to check if agent i, j can be swapped by letting agent i pull j. After SWAP-CHECK at line 7 returns false, it means that using pull operations can swap the agents. Agent j is thus returned and the procedure ends (Line 8).

Subsequently, from Line 9 in Alg. 4, all adjacent vertices of v^i are iterated in arbitrary order. In each iteration, it tries to find an agent $k \in I_{curr}$ that currently occupies a neighbor vertex u of v^i and is not planned yet (Line 10). If u is unoccupied (i.e., $k = \emptyset$) or $u = v$ (i.e., $s_{prev}^k.v = v$), then these cases are already handled by Lines 3-8, and the current iteration of the for loop ends (Line 11). If k is found, on Lines 12-13 in Alg. 4, SWAP-REQUIRED-POSSIBLE first places agents k and i at vertices v^i and v respectively, and then invokes SWAP-CHECK twice (Line 12-13) to check whether agent k needs to swap with agent i. The first call of SWAP-CHECK checks whether agents k and i can swap their locations by repeatedly letting agent k pulls agent i (Line 12). If the SWAP-CHECK on Line 12 returns true, which indicates that swap operation cannot be done through agent k's pulling. Then, the second SWAP-CHECK is applied to check if these two agents can be swapped by letting agent i pulls agent k (Line 13). When the second SWAP-CHECK returns false, it indicates that the pull operations can swap agents k and i. Thus agent k is returned, indicating that agent i needs to swap with agent k, and the procedure ends (Line 14).

Algorithm 4: SWAP-REQUIRED-POSSIBLE

Input: i, v
Notation: $i, j, k \in I$
1: $v^i \leftarrow s_{prev}^i.v$
2: **if** $v = v^i$ **then return** $\emptyset$
3: $j \leftarrow$ OCCUPANT$(v, I_{curr}, s_{prev}, s_{next})$
4: **if** $j \neq \emptyset$ **then**
5: $\quad v^j \leftarrow s_{prev}^j.v$
6: $\quad$ **if** SWAP-CHECK(v^j, v^i) **then**
7: $\quad\quad$ **if** !SWAP-CHECK(v^i, v^j) **then**
8: $\quad\quad\quad$ **return** j
9: **for** $u \in$ Neigh(v^i) **do**
10: $\quad k \leftarrow$ OCCUPANT$(u, I_{curr}, s_{prev}, s_{next})$
11: $\quad$ **if** $k = \emptyset \vee s_{prev}^k.v = v$ **then continue**
12: $\quad$ **if** SWAP-CHECK(v^i, v) **then**
13: $\quad\quad$ **if** !SWAP-CHECK(v, v^i) **then**
14: $\quad\quad\quad$ **return** k
15: **return** $\emptyset$

Algorithm 5: SWAP-CHECK

Input: v^i, v^j
Notation: $i, j \in I$
1: $v^{pl} \leftarrow v^i; v^{pd} \leftarrow v^j$
2: **while** $v^{pl} \neq v^j$ **do**
3: $\quad n \leftarrow (\text{Neigh}(v^{pl})).size()$
4: $\quad$ **for** $u \in$ Neigh(v^{pl}) **do**
5: $\quad\quad$ **if** $u = v^{pd}$ **then**
6: $\quad\quad\quad n - 1$
7: $\quad\quad\quad$ **continue**
8: $\quad\quad v^{pl} \leftarrow u$
9: $\quad$ **if** $n >= 2$ **then return false**
10: $\quad$ **if** $n <= 0$ **then return true**
11: $\quad$ **if** $v^{pd} = v_g^j$ **then**
12: $\quad\quad$ **if** $argmin(h(i,u))_{u \in Neigh(v^{pl})} = v^{pd}$ **then**
13: $\quad\quad\quad$ **return true**
14: $\quad v^{pd} \leftarrow v^{pl}$
15: **return true**

If no such an agent j or k exists, it indicates that either there is no need for agent i to swap with another agent, or there is no way for agent i to swap with another agent. Thus no agent is returned and Alg. 4 ends (Line 15).

SWAP-CHECK

SWAP-CHECK is a procedure to predict if a sequence of pull operations can swap agent i and j. In SWAP-CHECK, let $v^{pl} \leftarrow v^i$, where "pl" stands for pull, indicating that the agent attempts to pull another agent, and let $v^{pd} \leftarrow v^j$, where "pd" stands for pulled, indicating that the agent is pulled by another agent. To do this, we continuously let i pull j (Line 3-14). Specifically, in each while iteration, all neighbor vertices u of the current vertex v^{pl} of the pulling agent i are considered, and u is skipped if u is same as the vertex v^{pd} of the agent j that is pulled. If u is different from v^{pd}, then u is assigned to be the next vertex that the pulling agent i will move to (Line 8). Then, the pulled agent moves to the current vertex of the pulling agent (Line 14). This while loop ends in four cases.

- (1) Agent i's current vertex has at least 2 neighbor vertices, and i can move to a vertex different from agent j's current vertex. In this case, swap is possible through pull operations from agent i to agent j, and no other operations are required. SWAP-CHECK thus returns false (Line 9).
- (2) Agent i has no neighbor vertices besides agent j's current vertex. In this case, solely using push cannot achieve swap, and pull operation is thus required. SWAP-CHECK thus returns true (Line 10).
- (3) Let $h(i, u)$ denote the shortest path distance from vertex u to agent i's goal v_g^i. When the pulled agent j is at its goal vertex v_g^i (Line 11), if for the pulling agent i, the nearest vertex to its goal v_g^i among the neighbor vertices Neigh(v^{pl}) of i's current vertex v^{pl} is v^{pd} (Line 12), SWAP-CHECK returns true, and swap operation is required.
- (4) When $v^{pl} = v^j$, it indicates that agent i has pulled agent j through a cycle (as defined in Sec. 5). It means that the two agents have not swapped their vertices. Therefore, additional pull operation is required, and true is thus returned.

Toy Example

This section presents an example illustrating how LSRP-SWAP solves the instance in Fig. 8 that requires swapping the location of two agents in a tree-like graph. Let b denote the blue agent and y denote the yellow agent. Initially, agent y is assigned with the highest priority, while agent b is assigned with the lowest priority.

LSRP-SWAP for (a) As shown in Fig. 8, starting from the initial state (Start), after a few push operations, LSRP-SWAP reaches (a). In (a), agent y is at its goal vertex, so its priority is reset, while agent b's priority increases by 1. Now agent b has the highest priority. LSRP-SWAP invokes

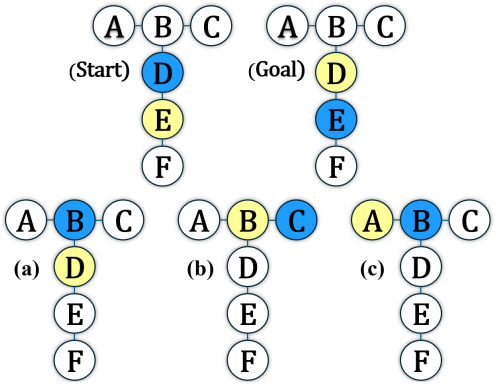

Figure 8: The two agents with color blue and yellow start like (Start), going through a series of collision-free operations, and finally reach (Goal). The only feasible plan is to swap the location of two agents via operations (a), (b) and (c), where both agents need to move away from their goal locations. ASY-PUSH cannot find a feasible solution in this case as it always lead to one of the agent moves towards to its goal location.

ASY-PUSH-SWAP to plan for agent b and y in decreasing order of priority, and agent b is planned at first.

In the ASY-PUSH-SWAP procedure of b, it invokes SWAP-REQUIRED-POSSIBLE with input $i = b, v = D$, since vertex D is the closest to b's goal vertex (Line 3, Alg. 2). The procedure finds that agent y occupies D through OCCUPANT (Line 3, Alg. 4) and invokes SWAP-CHECK (Line 6, Alg. 4). Here, SWAP-CHECK predicts whether the two agents can swap their locations by iteratively moving b to y's current vertex and moving y to another vertex. The input to this SWAP-CHECK is that $v^i = B$ and $v^j = D$, meaning agent y is at vertex D and agent b is at vertex B. After a few iterations, y arrives at vertex F and b arrives at vertex E. y finds no vertex to move to other than vertex E that is occupied by agent b (Line 10, Alg. 5). Therefore, additional operation is required, and SWAP-CHECK returns true.

Then, SWAP-REQUIRED-POSSIBLE invokes another SWAP-CHECK (Line 7, Alg.4) to predict whether two agents can swap their locations by iteratively moving y to b's current vertex and moving b to another vertex. The input to this SWAP-CHECK is that agent $v^i = D$ and $v^j = B$, meaning agent y is at vertex D and agent b is at vertex B. SWAP-CHECK finds that agent b's current vertex B has two occupied vertices (A and C) that are different from y's current vertex D, and agent b can move to either of these two occupied vertices. Swapping the location of both agents is thus possible since B has two unoccupied neighbor vertices, and SWAP-CHECK thus returns true (Line 9, Alg. 5).

After SWAP-CHECK returns true, SWAP-REQUIRED-POSSIBLE ends and returns agent y (Line 8, Alg. 4), indicating that agent y is able to swap vertices with agent b. After SWAP-REQUIRED-POSSIBLE returns agent y, the ASY-PUSH-SWAP procedure (Line 4, Alg. 2) reverses the order of vertices in C to start the swapping process, and b moves to $C[0]$, which is the vertex C in this toy exam-

ple. Then, agent b pulls y to b's current vertex (Line 24-25, Alg. 2) as shown in (b). So far, agents b and y are planned, and the LSRP-SWAP procedure for (a) in Fig. 8 ends.

LSRP-SWAP for (b) In the next iteration of LSRP-SWAP, when planning for (b) in Fig. 8, agent b, y are both in I_{curr} and are planned in decreasing order of priority. The procedure invokes ASY-PUSH-SWAP to plan for agent b at first. Here, ASY-PUSH-SWAP seeks to move agent b to vertex B and pushes agent y away, thus a recursive call to ASY-PUSH-SWAP of agent y is conducted.

In this recursive call of ASY-PUSH-SWAP on agent y, it invokes the SWAP-REQUIRED-POSSIBLE procedure with input $i = y, v = D$, since D is the closest vertex to agent y's goal. No agent occupies D, so the procedure arrives at Line 9 of Alg. 4. SWAP-REQUIRED-POSSIBLE (Line 9 in Alg. 4) iterates all neighbor vertices of y's current vertex B (in arbitrary order) . For example, it checks vertex D and A and finds they are unoccupied, which means the OCCUPANT procedure (Line 11 in Alg. 4) returns an empty set for vertices D and A. Then, when iterating vertex C, OCCUPANT finds that agent b occupies vertex C. SWAP-REQUIRED-POSSIBLE then predicts the case that agent b moves to vertex B after agent y moves to vertex D by invoking SWAP-CHECK with input $v^i = B, v^j = D$ (Line 12, Alg. 4). In SWAP-CHECK, after a few while-loop iterations, agent y reaches vertex F and has no neighbor vertex to move to, swap is required and true is thus returned (Line 10, Alg. 5). Then, SWAP-POSSIBLE is invoked with input $v^i = D, v^j = B$. In this SWAP-POSSIBLE, it finds that when agent b is at vertex B, b has 2 vertices to move to, so a swap is possible , true is thus returned (Line 9, Alg. 5). After SWAP-POSSIBLE, agent b is returned as the return argument k, the agent to swap vertices with agent y, and SWAP-REQUIRED-POSSIBLE ends (Line 14, Alg. 4). In y's ASY-PUSH-SWAP, its C is reversed, and C[0] = vertex A. A is identified first; ASY-PUSH-SWAP finds that A is unoccupied, thus moving y to A. Agent b moves to B after y arrives at A, as shown in (c). Agents b and y are planned, and the LSRP-SWAP procedure for (b) ends.

LSRP-SWAP for (c) Starting from (c), after a few while-loop iterations of LSRP-SWAP, where push operation is applied in each iteration, agents b and y eventually reach their goal vertices D and E, respectively. LSRP-SWAP successfully swaps the locations of the two agents.