Time-Independent Planning for Multiple Moving Agents

Keisuke Okumura, Yasumasa Tamura, Xavier Défago
School of Computing, Tokyo Institute of Technology
Tokyo, Japan
{okumura.k, tamura, defago}@coord.c.titech.ac.jp

Abstract

Typical Multi-agent Path Finding (MAPF) solvers assume that agents move synchronously, thus neglecting the reality gap in timing assumptions, e.g., delays caused by an imperfect execution of asynchronous moves. So far, two policies enforce a robust execution of MAPF plans taken as input: either by forcing agents to synchronize or by executing plans while preserving temporal dependencies. This paper proposes an alternative approach, called time-independent planning, which is both online and distributed. We represent reality as a transition system that changes configurations according to atomic actions of agents, and use it to generate a time-independent schedule. Empirical results in a simulated environment with stochastic delays of agents' moves support the validity of our proposal.

Introduction

Multi-agent systems with physically moving agents are becoming gradually more common, e.g., automated warehouse (Wurman, D'Andrea, and Mountz 2008), traffic control (Dresner and Stone 2008), or self-driving cars. In such systems, agents must move smoothly without colliding. This is embodied by the problem of Multi-agent Path Finding (MAPF) (Stern 2019). Planning techniques for MAPF have been extensively studied in recent years.

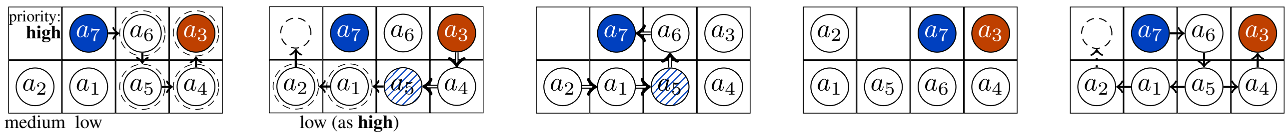
The output of such planning is bound to be executed in real-world situations with agents (robots). Typical MAPF is defined in discrete time. Agents are assumed to do two kinds of atomic actions synchronously: move to a neighboring location or stay at their current location. Perfect executions for the planning are however difficult to ensure since timing assumptions are inherently uncertain in reality, due to the difficulty of: 1) accurately predicting the temporal behavior of many aspects of the system, e.g., kinematics, 2) anticipating external events such as faults and interference, and 3) ensuring a globally consistent and accurate notion of time in the face of clock shift and clock drift. Even worse, the potential of unexpected interference increases with the number of agents, hence the need to prepare for imperfect executions regarding the timing assumptions.

There are two intuitive ways to tackle imperfect executions of MAPF plans taken as input. The first, and conservative idea is to forcibly synchronize agents' moves, globally or locally. Most decentralized approaches to MAPF take this approach implicitly (Wiktor et al. 2014; Kim et al. 2015; Okumura et al. 2019; Wang and Rubenstein 2020). This policy negatively affects the entire performance of the system with unexpected delays and lacks flexibility (Ma, Kumar, and Koenig 2017). The second policy makes agents preserve temporal dependencies of the planning (Hönig et al. 2016; Ma, Kumar, and Koenig 2017; Hönig et al. 2019; Atzmon et al. 2020). Two types of temporal dependencies exist: 1) internal events within one agent and, 2) order relation of visiting one node. This policy is sound but still vulnerable to delays. Consider an extreme example where one agent moves very slowly or crashes. Due to the second type of dependencies, the locations where the agent will be are constrained by the use of the other agents. Thus, the asynchrony of the movements is sensitive to the whole system.

We, therefore, propose an alternative approach, called time-independent planning, that aims at online and distributed execution focusing on agents' moves. We represent the whole system as a transition system that changes configurations according to atomic actions of agents, namely, 1) request the next locations ($requesting$), 2) move ($extended$), and, 3) release the past locations, or stay ($contracted$). In this time-independent model, any *a priori* knowledge for timings of atomic actions is unavailable, representing non-deterministic behaviors of the external environment. The challenge is to design algorithms tolerant of all possible sequences of actions.

The main contributions of this paper are: 1) the formalization of the time-independent model and Causal-PIBT, a proposed time-independent planning with guaranteed *reachability*, i.e., all agents are ensured to reach their destinations within finite time. Causal-PIBT, as a proof-of-concept, extends a recently-developed decoupled approach that solves MAPF iteratively, Priority Inheritance with Backtracking (PIBT) (Okumura et al. 2019). We also present how an offline MAPF plan enhances Causal-PIBT. 2) experimental results demonstrating the validity and robustness of the proposal through the simulation with stochastic delays of agents' moves, using MAPF-DP (with Delay Probabilities) (Ma, Kumar, and Koenig 2017).

The remainder of this paper consists of the following five sections: 1) preliminary including the formalization of



(a) priority inheritance (PI) (b) backtracking and PI again (c) backtracking (d) one timestep later (e) as DFST

Figure 1: Example of PIBT. Requests for the next timestep are depicted by dashed circles, determined greedily according to agents' destinations (omitted here). Flows of priority inheritance and backtracking are drawn as single-line and doubled-line arrows, respectively. First, a_7 (blue agent) determines the next desired node (current location of a_6). Then, priority inheritance happens from a_7 to a_6, making a_7 wait for backtracking and a_6 start planning; a_6, a_5 and a_4 do the same. a_3 (red), however, is stuck (1a). Thus, a_3 backtracks as invalid to a_4 (1b). a_4 tries to replan, however, a_4 is also stuck hence a_4 sends backtracking as invalid to a_5 (blue with diagonal lines). a_5, with success replanning, executes other priority inheritance to a_1 (1b). Finally, a_1, a_5, a_6 and a_7 receives backtracking as valid (1c) and then start moving (1d). Fig. 1e: virtual depth first search tree for this example, for explanation of Causal-PIBT. a_7 is a root. Solid arrows are drawn from a parent to a child. a_2 finds an empty node.

MAPF and related work, 2) the time-independent model, 3) examples of time-independent planning, 4) empirical results of the proposals using MAPF-DP, and 5) conclusion and future directions.

Preliminaries

This section first defines MAPF. Then, we explain the MAPF variant emulating asynchrony of movements, called MAPF-DP, which we later use in experiments. We also explain two policies that execute MAPF plans and PIBT, the original form of Causal-PIBT.

MAPF

In an environment represented as a graph $G = (V, E)$, the MAPF problem is defined as follows. Let $\pi_i[t] \in V$ denote the location of an agent a_i at discrete time $t \in \mathbb{N}$. Given distinct initial locations $\pi_i[0] \in V$ and destinations $g_i \in V$ for each agent, assign a path $\pi_i = (\pi_i[0], \pi_i[1], \ldots, \pi_i[T])$ such that $\pi_i[T] = g_i$ to each agent minimizing some objective function (see below).

At each timestep t, a_i can move to an adjacent node, or, can stay at its current location, i.e., $\pi_i[t{+}1] \in \mathit{Neigh}(\pi_i[t]) \cup \{\pi_i[t]\}$, where $\mathit{Neigh}(v)$ is the set of nodes neighbor to $v \in V$. Agents must avoid two types of conflicts (Stern et al. 2019): 1) *vertex conflict*: $\pi_i[t] \neq \pi_j[t]$, and, 2) *swap conflict*: $\pi_i[t] \neq \pi_j[t+1] \lor \pi_i[t+1] \neq \pi_j[t]$.

Two kinds of objective functions are commonly used to evaluate MAPF solutions: 1) *sum of cost* (SOC), where the cost is the earliest timestep T_i such that $\pi_i[T_i] = g_i, \ldots, \pi_i[T] = g_i, T_i \leq T$, 2) *makespan*, i.e., T. This paper focuses on the sum of cost (SOC).

MAPF-DP (with Delay Probabilities)

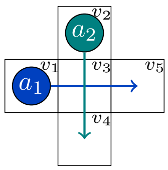
MAPF-DP (Ma, Kumar, and Koenig 2017) emulates imperfect executions of MAPF plans by introducing the possibility of unsuccessful moves. Time is still discrete. At each timestep, an agent a_i can either stay in place or move to an adjacent node with a probability p_i of being unsuccessful. The definition of conflicts is more restrictive than with normal MAPF: 1) vertex conflict is as defined in MAPF, and 2) *following conflict*: $\pi_i[t+1] \neq \pi_j[t]$. The rationale is that, without the later restriction, two agents might be in the same node due to one failing to move. Note that following conflict contains swap conflict.

Execution Policies Ma *et al.* (Ma, Kumar, and Koenig 2017) studied two robust execution policies using MAPF plans for the MAPF-DP setting. The first one, called *Fully Synchronized Policies (FSPs)*, synchronizes the movements of agents globally, i.e., a_i waits to move to $\pi_i[t{+}1]$ ($\neq \pi_i[t]$) until all move actions of $\pi_j[t']$, $t' \leq t$ are completed. The second approach, called *Minimal Communication Policies (MCPs)*, executes a plan while maintaining its temporal dependencies. There are two kinds of dependencies: 1) *internal events*, i.e., the corresponding action of $\pi_i[t]$ is executed prior to that of $\pi_i[t+1]$, and 2) *node-related events*, i.e., if $\pi_i[t] = \pi_j[t']$ and $t < t'$, the event of $\pi_i[t]$ is executed prior to that of $\pi_j[t']$. As long as an MAPF plan is valid, both policies make agents reach their destinations without conflicts, despite of delay probabilities.

Priority Inheritance with Backtracking (PIBT)

PIBT (Okumura et al. 2019) repeats one-timestep planning until it terminates, aiming at solving MAPF iteratively (without delays). Unlike complete solvers for MAPF, PIBT does not ensure that all agents are on their goals simultaneously, rather it ensures *reachability*; all agents are ensured to reach their destinations eventually. Thus, an agent that reaches its goal potentially moves from there while other agents are moving. Reachability plays a crucial role in situations where destinations are given continuously such as lifelong MAPF (Ma et al. 2017) because all tasks (i.e., destinations) assigned to agents are ensured to be completed.

In PIBT, for each timestep, agents are provided with unique priorities and they sequentially determine their next locations in decreasing order of priorities while avoiding to use nodes that have requested by higher-priority agents. *Priority inheritance*, originally considered in resource scheduling problems (Sha, Rajkumar, and Lehoczky 1990), is introduced; when a low-priority agent X obstructs the movement of a higher-priority agent Y, X temporarily inherits the higher-priority of Y. Priority inheritance can be applied iteratively. Agents giving their priority (Y) must wait for



activated agent → ← interacted agent | c: *contracted*, r: *requesting*, e: *extended*

state	σ_1	σ_2	σ_1	σ_2	σ_1	σ_2	σ_2	σ_1	σ_1	σ_1	σ_1	σ_2	σ_2	σ_2	σ_2	σ_2
mode	c	c	r	r	e	c	r	c	r	e	c	e	c	r	e	c
head	$\bot$	$\bot$	v_3	v_3	v_3	$\bot$	v_3	$\bot$	v_5	v_5	$\bot$	v_3	$\bot$	v_4	v_4	$\bot$
tail	v_1	v_2	v_1	v_2	v_1	v_2	v_2	v_3	v_3	v_3	v_5	v_2	v_3	v_3	v_3	v_4

initial states | interaction

Figure 2: Example of an execution of the model. a_1 goes to v_5 and a_2 goes to v_4. In the table, time progresses from left to right. There is an interaction, which makes a_2 be back to *contracted*.

backtracking from agents inheriting the priority (X). Backtracking has two outcomes: valid or invalid. A *valid* situation occurs when Y can successfully move to the location of X in the next timestep. An *invalid* situation occurs when X is stuck, i.e., neither going anywhere nor staying at its current location without colliding, forcing Y to replan its path. Fig. 1 shows an example of PIBT in one timestep.

With these protocols, the agent with highest priority is ensured to move to an arbitrary neighbor node if the graph satisfies the adequate property, e.g., biconnected. Subsequently, such an agent moves to its goal along the shortest path to its goal. To ensure reachability, PIBT uses dynamic priorities, where the priority of an agent increments gradually at each timestep until it drops upon reaching its goal, meaning that, agents not reaching their goals yet eventually get the highest.

Time-Independent Model

The time-independent model and the terminology used in this paper are partly inspired by the model for distributed algorithms with synchronous message passing (Tel 2000) and the Amoebot model (Derakhshandeh et al. 2014), an abstract computational model for programmable matter. Our model is however different in that agents are physically embodied and move on a graph, have destinations, and know their locations.

Components The system consists of a set of agents $A = \{a_1, \dots, a_n\}$ and a graph $G = (V, E)$. We assume that each agent knows G to plan their respective paths.

Configuration and State The whole system is represented as a transition system according to atomic actions of agents. Each agent a_i is itself a transition system with its *state*, denoted as σ_i, consisting of its internal variables, its current location, and destination. A *configuration* $\gamma = (\sigma_1, \dots, \sigma_n)$ of the whole system at a given time consists of the states of all agents at that time. A change in the state of some agents causes a change of configuration of the system, e.g., $\gamma = (\dots, \sigma_{i-1}, \sigma_i, \sigma_{i+1}, \dots) \rightarrow \gamma' = (\dots, \sigma_{i-1}, \sigma'_i, \sigma_{i+1}, \dots)$.

Mode Each agent at any time occupies at least one node and occupies two nodes during moves between nodes. We use two terms to represent agents' locations: $tail_i \in V$ and $head_i \in V \cup \{\bot\}$, where $\bot$ is void. They are associated with a mode $mode_i$ which can be: $contracted$, $requesting$, and $extended$.

- $contracted$: a_i stays at one node $tail_i$, and $head_i = \bot$; location of a_i is $tail_i$
- $requesting$: a_i attempts to move to $head_i \neq \bot$, being at $tail_i$; location of a_i is $tail_i$
- $extended$: a_i is moving from $tail_i$ to $head_i$; locations of a_i are both $tail_i$ and $head_i$.

Initially, all agents are $contracted$ and have distinct initial locations. $head_i$ ($\neq \bot$) is always adjacent to $tail_i$.

Transition Agents move by changing modes. These transitions are accompanied by changing $tail$ and $head$ as described below.

- from $contracted$, a_i can become $requesting$ by setting $head_i$ to $u \in Neigh(tail_i)$, to move to a neighbor node.
- from $requesting$, a_i can revert to $contracted$ by changing its $head_i$ to $\bot$.
- from $requesting$, agent a_i can become $extended$ when $\neg occupied(head_i)$, where $occupied(v)$ holds when there is no agent a_j such that $tail_j = v$ and no agent a_j in $extended$ such that $head_j = v$.
- from $extended$, a_i can become $contracted$, implying that the movement is finished. This comes with $tail_i \leftarrow head_i$, then $head_i \leftarrow \bot$.

Transitions are atomic so, e.g., if agents in $contracted$ become $extended$ through $requesting$, at least two actions are required; see an example later. Other modes transitions are disallowed, e.g., an agent in $contracted$ cannot become $extended$ directly.

Conflict-freedom and Deadlock In the above transition rules, the model implicitly prohibits vertex and following conflicts of MAPF. Rather, the model is prone to *deadlocks*; A set of agents $\{a_k, \dots, a_l\}$ are in a deadlock when all of them are $requesting$ and are in a cycle $head_k = tail_{k+1}$, $\cdots$, $head_l = tail_k$.

Activation Agents perform a single atomic action as a result of being *activated*. The nature and outcome of the action depend on the state of the agent (e.g., mode transition, local variable update). Activations occur non-deterministically, and there is no synchronization between agents. For simplicity, we assume that at most one agent is activated at any time.

In other words, the simultaneous activation of two agents a_i and a_j results in a sequence $(\dots, \sigma_i, \dots, \sigma_j, \dots) \rightarrow (\dots, \sigma'_i, \dots, \sigma_j, \dots) \rightarrow (\dots, \sigma'_i, \dots, \sigma'_j, \dots)$. Activations are supposed to be *fair* in the sense that, in any sufficiently long period, all agents must be activated at least once.

Interaction An activation may affect not only variables of the activated agent, but also affect nearby agents indirectly. For instance, if two *requesting* agents have the same *head*, one wins and becomes *extended* whereas the other loses and becomes *contracted*, atomically. This type of activation is called an *interaction*. Interactions include activations such that the activated agents change their variables referring to the variables of other agents. We say that the agents involved in the interaction, except a_i itself, are *interacted agents*. Given an activated agent a_i and an interacted agent a_j, the system transitions as follows: $(\dots, \sigma_i, \dots, \sigma_j, \dots) \rightarrow (\dots, \sigma'_i, \dots, \sigma'_j, \dots)$ with the state of all other agents unchanged. Except for interactions, the configuration is changed by the state change of a single agent. We assume that interactions are performed by communication between agents, but the detailed implementation is not relevant to this paper.

Termination Assuming that each agent a_i has its own destination $g_i \in V$, termination can be defined in two different ways.

- *Strong termination* occurs when reaching a configuration such that $mode_i = contracted \wedge tail_i = g_i$ for any agent a_i.
- *Weak termination* is when all agents have been at least once in a state where $mode_i = contracted \wedge tail_i = g_i$.

Strong termination corresponds to usual MAPF termination, whereas weak termination corresponds to the reachability property of PIBT. Strong termination implies weak termination.

In this paper, we refer to weak termination as *reachability*. Note that the properties of either deadlock-freedom or deadlock-recovery are required to ensure reachability.

Remarks Figure 2 illustrates an execution in the time-independent model. Although the example focuses on (classical) MAPF, many other problem variants, e.g., iterative MAPF (Okumura et al. 2019), can be addressed simply by adapting termination and goal assignments.

Algorithm

This section presents two examples of time-independent planning: GREEDY and Causal-PIBT. We also present how to enhance Causal-PIBT with offline MAPF plans.

Greedy Approach

GREEDY performs only basic actions; it can be a template for another time-independent planning. We simply describe its implementation for a_i as follows.

- when *contracted*: Choose the nearest node to g_i from $Neigh(tail_i)$ as new $head_i$, then become *requesting*.
- when *requesting*: Become *extended* when the $head_i$ is unoccupied, otherwise, do nothing.
- when *extended*: Become *contracted*.

Obviously, GREEDY is prone to deadlocks, e.g, when two adjacent agents try to swap their locations, they block eternally. The time-independent planning without deadlock-freedom or deadlock-recovery properties is impractical, motivating the next algorithm.

Causal-PIBT

The Causal-PIBT algorithm extends both algorithms PIBT and GREEDY. Although PIBT is timing-based relying on synchronous moves, Causal-PIBT is event-based relying on causal dependencies of agents' actions.

Concept Two intuitions are obtained from PIBT to design a time-independent algorithm that ensures reachability; 1) Build a depth-first search tree rooted at the agent with highest priority, using the mechanism of priority inheritance and backtracking. 2) Drop priorities of agents that arrive at their goals to give higher priorities to all agents not having reached their goals yet, thus resulting in that all agents eventually reach their goals. We complement the first part as follows.

The path adjustment of PIBT in one timestep can be seen as the construction of a (virtual) depth-first search tree consisting of agents and their dependencies. The tree is rooted at the first agent starting priority inheritance, i.e., locally highest priority agent, e.g., a_7 in Fig. 1. When an agent a_j inherits a priority from another agent a_i, a_j becomes a *child* of a_i and a_i becomes its *parent*. We show this virtual tree in Fig. 1e. Once an empty node adjacent to the tree is found, all agents on the path from the root to that empty node can move toward one step forward (a_2, a_1, a_5, a_6 and a_7). This enables the agent with highest priority, being always a root, to move from the current node to any arbitrary neighbor nodes. The invalid outcome in backtracking works as backtracking in a depth-first search, while the valid outcome notifies the search termination.

Description Causal-PIBT basically performs GREEDY, i.e., at each activation an agent a_i tries to move towards its goal; in addition, Causal-PIBT uses priority inheritance. When a_i is *contracted* or *requesting*, a_i potentially inherits priority from another agent a_j in *requesting* with $head_j = tail_i$. We now explain the details. The pseudocode is split into Algorithm 1 and 2. Interactions and the corresponding interacted agents are explicitly marked in comments. In Algo. 1, procedures with activation are denoted for each mode.

Variants: We first introduce local variants of a_i.

- $parent_i$ and $children_i$: for maintaining tree structures. a_i is a root when $parent_i = a_i$. The algorithm updates these variants so that $a_i = parent_j \Leftrightarrow a_j \in children_i$.
- C_i and S_i: for searching unoccupied neighbor nodes. C_i is candidate nodes of next locations. a_i in *contracted* selects $head_i$ from C_i. S_i represents already searched nodes by a tree to which a_i belongs. S_i is propagated in the tree. C_i is updated to be disjoint from S_i.

Algorithm 1 Causal-PIBT

$parent_i \in A$: initially a_i; $children_i \subset A$: initially $\emptyset$
$pori_i$: original priority; $ptmp_i$: temporal priority, initially $pori_i$
$C_i \subseteq V$: candidate nodes, initially $Neigh(tail_i) \cup \{tail_i\}$
$S_i \subseteq V$: searched nodes, initially $\emptyset$

1: **when** $mode_i = contracted$
2:   **if** $C_i = \emptyset \wedge parent_i = a_i$ **then**
3:     RELEASECHILDREN, RESET
4:   **end if**
5:   PRIORITYINHERITANCE
6:   **if** $C_i = \emptyset$ **then**
7:     let a_j be $parent_i$
8:     **if** $head_j = tail_i$ **then**
9:       $S_j \leftarrow S_j \cup S_i, C_j \leftarrow C_j \setminus S_j$ ▷ with a_j
10:       $head_j \leftarrow \bot, mode_j \leftarrow contracted$
11:     **end if**
12:     **return**
13:   **end if**
14:   $u \leftarrow$ the nearest node to g_i in C_i
15:   **if** $u = tail_i$ **then**
16:     RELEASECHILDREN, RESET
17:     **return**
18:   **end if**
19:   $C_i \leftarrow C_i \setminus \{u\}, S_i \leftarrow S_i \cup \{u, tail_i\}$
20:   $head_i \leftarrow u, mode_i \leftarrow requesting$
21: **end when**
22: **when** $mode_i = requesting$
23:   PRIORITYINHERITANCE
24:   **if** $parent_i \neq a_i \wedge head_i \in S_{parent_i}$ **then**   ▷ $parent_i$
25:     $head_i \leftarrow \bot, mode_i \leftarrow contracted$
26:     **return**
27:   **end if**
28:   **if** $occupied(head_i)$ **then return**
29:   $A' \leftarrow \{a_j \mid a_j \in A, mode_j = requesting, head_j = head_i\}$
30:   $a^\star \leftarrow \arg\max_{a_j \in A'} ptmp_j$
31:   **for** $a_j \in A' \setminus \{a^\star\}$ **do**   ▷ agents in A'
32:     $head_j \leftarrow \bot, mode_j \leftarrow contracted$
33:   **end for**
34:   **if** $a^\star \neq a_i$ **then return**
35:   $children_{parent_i} \leftarrow children_{parent_i} \setminus \{a_i\}$ ▷ $parent_i$
36:   $parent_i \leftarrow a_i$
37:   RELEASECHILDREN
38:   $mode_i \leftarrow extended$
39: **end when**
40: **when** $mode_i = extended$
41:   $tail_i \leftarrow head_i, head_i \leftarrow \bot, mode_i \leftarrow contracted$
42:   update $pori_i$, RESET
43: **end when**

- $pori_i$ and $ptmp_i$: priorities. They are components of a total order set. $ptmp_i$ is basically equal to $pori_i$, however, it is changed by priority inheritance. $ptmp_i \geq pori_i$ in any time, and only $ptmp_i$ is used for interaction.

Structure: The procedures in *contracted* consist of:

- Relaying release [Line 2–4]. An agent in stuck like a_4 in Fig. 1e is "released" when its parent (a_5) moves. If it has children (a_3), it needs to relay the release to its children, then initialize.
- Priority inheritance [Line 5]. When a_i is activated and there exists an agent a_j with higher priority such that $head_j = tail_i$, then priority inheritance happens from a_j to a_i, e.g., from a_7 and a_6 in Fig. 1a.
- Backtracking [Line 6–13]. When a_i detects stuck ($C_i = \emptyset$), making its parent a_j *contracted* while propagating already searched nodes S_i, then stops. This procedures correspond to invalid case of backtracking in PIBT, e.g., from a_3 to a_4 in Fig. 1b.
- Prioritized planning [Line 14–20]. Pickup one node u for the next location from C_i, update searched nodes, and become *requesting*. Initialize when u is $tail_i$.

The procedures in *requesting* consist of:

- Priority inheritance [Line 23]. The condition is the same as priority inheritance in *contracted*. This implicitly contributes to detecting deadlocks.
- Deadlock detection and resolving [Line 24–27]. This part detects the circle of requests, then go back to *contracted*.
- Winner determination [Line 28–33]. When $head_i$ is unoccupied, there is a chance to move there. First, identify agents requesting the same node then select one with highest priority as a winner. Let losers back to be contracted. See "interaction" in Fig. 2.
- Preparation for moves [Line 34–38]. If a_i itself is a winner, after isolating itself from a tree belonged to, a_i becomes *extended*.

The procedures in *extended* are to update priority and back to *contracted*.

Subprocedures: We use three subprocedures, as shown in Algo. 2. PRIORITYINHERITANCE first determines whether priority inheritance should occur [Line 2–5], then updates the tree structure and inherits both the priority and the searched nodes from the new parent [Line 6–10]. RELEASECHILDREN just cuts off the relationship with the children of a_i. RESET initializes the agent's status.

Overview: Causal-PIBT constructs depth-first search trees, each rooted at the agents with locally highest priorities, using *parent* and *children*. They are updated through priority inheritance and backtracking mechanisms. All agents in the same tree have the same priority *ptmp*, equal to the priority *pori* of the rooted agent. When a tree with higher priority comes in contact with a lower-priority tree (by some agent becoming *requesting*), the latter tree is decomposed and is partly merged into the former tree (by PRIORITYINHERITANCE). In the reverse case, the tree with lower priority just waits for the release of the area. Backtracking occurs when a child has no candidate node to move due to requests from agents with higher priorities [Line 8–11 in Algo. 1].

Dynamic Priorities: We assume that $pori_i$ is unique between agents in any configuration and updated upon reaching a goal to be lower than priorities of all agents who have not yet reached their goals [Line 42 in Algo. 1]. This is realized by a prioritization scheme similar to PIBT.

Properties Next, two useful properties of Causal-PIBT are shown: deadlock-recovery and the reachability. We also state that Causal-PIBT is quick enough as atomic actions in terms of time complexity.

Algorithm 2 Procedures of Causal-PIBT

```
1:  procedure PRIORITYINHERITANCE
2:      A'' ← {a_j | a_j ∈ A, mode_j = requesting, tail_i = head_j}
3:      if A'' = ∅ then return
4:      a_k ← arg max_{a_j ∈ A''} ptmp_j
5:      if ptmp_k ≤ ptmp_i then return
6:      RELEASECHILDREN
7:      children_{parent_i} ← children_{parent_i} \ {a_i}        ▷ parent_i
8:      parent_i ← a_k, children_k ← children_k ∪ {a_i}          ▷ a_k
9:      ptmp_i ← ptmp_k
10:     S_i ← S_k ∪ {head_i}, C_i ← Neigh(tail_i) ∪ {tail_i} \ S_i
11: end procedure
12: procedure RELEASECHILDREN
13:     for a_j ∈ children_i do parent_j ← a_j                    ▷ a_j
14:     children_i ← ∅
15: end procedure
16: procedure RESET
17:     S_i ← ∅, C_i ← Neigh(tail_i) ∪ {tail_i}
18:     ptmp_i ← pori_i
19: end procedure
```

Theorem 1 (deadlock-recovery)**.** *Causal-PIBT ensures no deadlock situation can last forever.*

Proof. Assume that there is a deadlock. When an agent a_i changes from *contracted* to *requesting*, it updates S_i such that S_i includes $head_i$ and $tail_i$ [Line 19 in Algo. 1]. After finite activations, all agents involved in the deadlock must have the same priority $ptmp$ due to priority inheritance. Every priority inheritance is accompanied by the propagation of S_i [Line 10 in Algo. 2]. When an agent in *requesting* detects its head in S_{parent_i}, it changes back to *contracted* [Line 24–27 in Algo. 1]. These imply the statement. □

Once an agent a_i in a deadlock comebacks to *contracted*, a_i cannot request the same place as before due to the update of C_i [Line 19 in Algo. 1], preventing a reoccurrence of the deadlock situation; i.e., a livelock situation if repeated indefinitely.

Theorem 2 (reachability)**.** *Causal-PIBT has the reachability in biconnected graphs if $|A| < |V|$.*

Proof sketch. Let a_i be the agent with highest priority $pori$. If a_i is *contracted*, this agent does not inherit any priority. Thus, a_i can be *requesting* so that $head_i$ is any neighbor node of $tail_i$. If a_i is *requesting*, a_i eventually moves to $head_i$ due to the construction of the depth-first search tree in a biconnected graph. These imply that a_i can move to an arbitrary neighbor node in finite activations. By this, a_i moves along the shortest path to its goal. Due to the prioritization scheme, an agent that has not reached its goal eventually inherits the highest priority $pori$, and then starts moving along its shortest path to its goal. This ensures reachability. The detailed proof is found in the long version (Okumura, Tamura, and Défago 2020). □

Proposition 1 (time complexity)**.** *Assume that the maximum time required for one operation to update S_i or C_i (i.e., union, set minus) is α and for an operation of Line 14 in Algo. 1 is β. Let $\Delta(G)$ denote the maximum degree of G. For each activation, time complexity of Causal-PIBT in contracted is $O(\Delta(G)+\alpha+\beta)$, in requesting is $O(\Delta(G)+\alpha)$, and in extended is $O(1)$.*

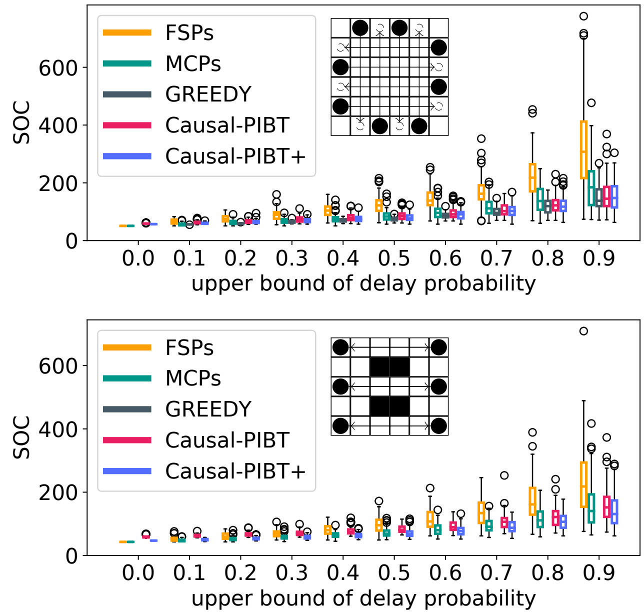


Figure 3: The results in small benchmarks, shown as boxplots. The number of successful instances for GREEDY are 0, 5, 8, 11, 17, 26, 30, 36, 40, 51 for each $\bar{p}$ (from 0 to 0.9) in the upper plot. GREEDY failed all cases in the lower plot.

Proof sketch. $|children_i|$, $|A'|$ used in *requesting*, and $|A''|$ used in PRIORITYINHERITANCE are smaller than equal to $\Delta(G)$. Then, the complexity of PRIORITYINHERITANCE is $O(\Delta(G)+\alpha)$. RELEASECHILDREN is $O(\Delta(G))$. RESET is $O(1)$. We can derive the statements using the above. □

MAPF Plans as Hints

Although GREEDY and Causal-PIBT are for online situations, they can optionally use offline MAPF plans as hints while still being online and distributed schemes during execution. This can contribute to relaxing congestion because time-independent planning is shortsighted, i.e., planning paths anticipating only a single step ahead. Finding optimal MAPF plans is NP-hard (Yu and LaValle 2013; Ma et al. 2016), however, powerful solvers have been developed so far (Stern 2019); time-independent planning can use them. We describe how Causal-PIBT is enhanced by MAPF plans. Note that GREEDY can be extended as well. The intuition is to make agents follow original plans whenever possible.

Given a MAPF plan $\boldsymbol{\pi}$, assume that a_i knows π_i *a priori*. Before execution, a_i makes a new path $\tilde{\pi}_i$ by removing a node $\pi_i[t]$ such that $\pi_i[t] = \pi_i[t-1]$ from π_i while keeping the order of nodes, since the action "stay" is meaningless in the time-independent model. During execution, a_i manages its own discrete time t_i internally (initially $t_i = 0$).

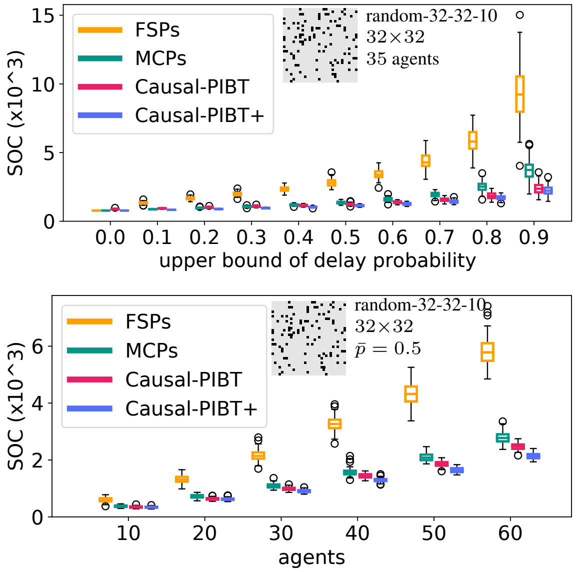


Figure 4: Executions in random grids.

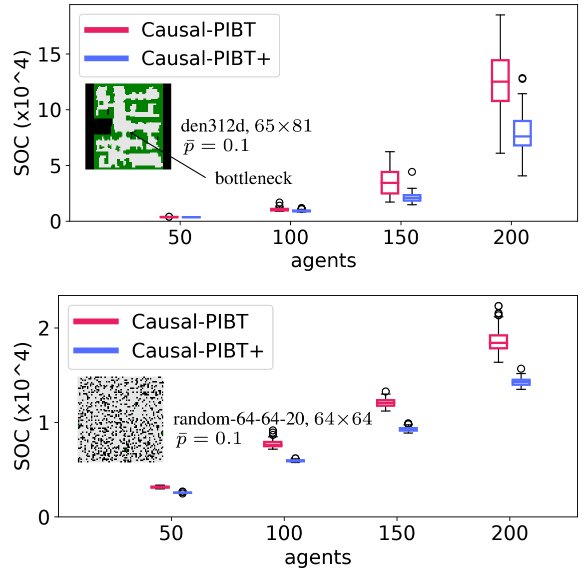


Figure 5: Executions in large fields.

The node selection phase [Line 14 in Algo. 1] is modified as follows. If $t_i \geq |\tilde{\pi}_i| - 1$, i.e., a_i finished $\tilde{\pi}_i$, a_i follows the original policy; chooses a next node greedily from C_i. Otherwise, if $tail_i = \pi_i[t_i]$ and $\pi_i[t_i + 1] \in C_i$, i.e., when a_i can follow $\tilde{\pi}_i$, a_i selects $\pi_i[t_i + 1]$. Or else, a_i selects the nearest node on the rest of $\tilde{\pi}_i$, formally; $\arg\min_{v \in C_i} \left\{ \min_{u \in \tilde{\pi}[t_i+1:\,]} \mathrm{cost}(v, u) \right\}$ where $\tilde{\pi}[t\colon] = (\tilde{\pi}[t], \tilde{\pi}[t+1], \ldots)$.

t_i is updated when a_i in $extended$ is activated. If $head_i \in \tilde{\pi}_i[t_i + 1\colon]$, t_i becomes the corresponding timestep t' such that $\tilde{\pi}_i[t'] = head_i$; otherwise, do nothing.

Evaluation

The experiments aim at: 1) showing the robustness of time-independent planning, i.e., even though delays of agents are unpredictable, their trajectories are kept efficient, and, 2) verifying the usefulness to use MAPF plans as hints during executions.

We used the MAPF-DP problem because it can emulate imperfect executions. To adapt the time-independent model to MAPF-DP, rules of activation are required. We repeated the following two phases: 1) Each agent a_i in $extended$ is activated with probability $1 - p_i$. As a result, a_i successfully moves to $head_i$ with probability $1 - p_i$ and becomes $contracted$. 2) Pickup one agent in $contracted$ or in $requesting$ randomly then makes it activated, and repeat this until the configuration becomes stable, i.e, all agents in $contracted$ and $requesting$ do not change their states unless any agent in $extended$ is activated. The rationale is that time required by executing atomic actions except for agents' moves is much smaller than that of the moves. We regard a pair of these two phases as one timestep. The strong termination was used to compare with existing two approaches for MAPF-DP: FSPs (Fully Synchronized Policies) and MCPs (Minimum Communication Policies). The evaluation is based on the sum of cost (SOC) metric. The simulator was developed in C++ [1], and all experiments were run on a laptop with Intel Core i9 2.3GHz CPU and 16GB RAM. In all settings, we tried 100 repetitions.

Small Benchmarks First, we tested the time-independent planning in two small benchmarks shown in Fig. 3. The delay probabilities p_i were chosen uniformly at random from $[0, \bar{p}]$, where $\bar{p}$ is the upper bound of p_i. The higher $\bar{p}$ means that agents delay often. $\bar{p}$=0 corresponds to perfect executions without delays. We manipulated $\bar{p}$. GREEDY, Causal-PIBT, and the enhanced one using MAPF plans as hints (Causal-PIBT+) were performed. Those three were regarded to fail after 10000 activations in total, implying occurring deadlocks or livelocks. FSPs and MCPs were also tested as comparisons. To execute FSPs, MCPs, and Causal-PIBT+, valid MAPF plans such that minimize SOC, which assumes following conflicts, were computed by an adapted version of Conflict-based Search (CBS) (Sharon et al. 2015), prior to performing MAPF-DP.

The results are shown in Fig. 3. Although GREEDY failed in most cases due to deadlocks, Causal-PIBT (+) succeeded in all trials due to the deadlock-recovery and the reachability. FSPs resulted in larger SOC compared to MCPs. The results of Causal-PIBT (+) were competitive to those of MCPs.

Random Grid Next, we tested Causal-PIBT (+) using one scenario from MAPF benchmarks (Stern et al. 2019), shown in Fig. 4. We manipulated two factors: 1) changing $\bar{p}$ while fixing the number of agents (=35), and, 2) changing the number of agents while fixing $\bar{p}$ (=0.5). When the number of

[1] https://github.com/Kei18/time-independent-planning

agents increases, the probability that someone delays also increases. We set sufficient upper bounds of activations. FSPs and MCPs were also tested. In this time, an adapted version of Enhanced CBS (ECBS) (Barer et al. 2014), bounded suboptimal MAPF solver, was used to obtain valid MAPF-DP plans, where the suboptimality was 1.1.

Fig. 4 shows the results. The proposals succeeded in all cases even though the fields were not biconnected. The results demonstrated the time-independent planning outputs robust executions while maintaining the small SOC in the severe environment as for timing assumptions. In precisely, when $\bar{p}$ or the number of agents is small, MCPs was efficient, on the other hand, when these number increases, the time-independent plannings had an advantage.

Large Fields Finally, we tested proposals using large fields from the MAPF benchmarks, as shown in Fig. 5. We respectively picked one scenario for each field, then tested while changing the number of agents. $\bar{p}$ is fixed to 0.1. Regular ECBS (suboptimality: 1.1) was used for Causal-PIBT+. Note that Causal-PIBT+ does not require MAPF plans that assume following conflicts.

The results (in Fig. 5) demonstrate the advantage of using MAPF plans in Causal-PIBT. The proposed methods succeeded in all cases. We observed a very large SOC in map den312d due to the existence of a bottleneck. Such a structure critically affects execution delays.

Conclusion

This paper studied the online and distributed planning for multiple moving agents without timing assumptions. We abstracted the reality of the execution as a transition system, then proposed time-independent planning, and Causal-PIBT as an implementation that ensures reachability. Simulations in MAPF-DP demonstrate the robustness of time-independent planning and the usefulness of using MAPF plans as hints.

Future directions include the following: 1) Develop time-independent planning with strong termination, e.g., by adapting Push and Swap (Luna and Bekris 2011) to our model. 2) Address communication between agents explicitly. In particular, this paper neglects delays caused by communication by treating interactions as a black box, and the next big challenge is there.

Acknowledgments

We thank the anonymous reviewers for their many insightful comments and suggestions. This work was partly supported by JSPS KAKENHI Grant Numbers 20J23011 and 20K11685, and Yoshida Scholarship Foundation.

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2020. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research* 67: 549–579.

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 *Proc. Annual Symp. on Combinatorial Search (SoCS)*, 19–27.

Derakhshandeh, Z.; Dolev, S.; Gmyr, R.; Richa, A. W.; Scheideler, C.; and Strothmann, T. 2014. Brief announcement: amoebot–a new model for programmable matter. In *Proc. Symp. on Parallelism in Algorithms and Architectures*, 220–222. ACM.

Dresner, K.; and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research* 31: 591–656.

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.

Hönig, W.; Kumar, T. S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-agent path finding with kinematic constraints. In *Proc. Intl. Conf. on Automated Planning and Scheduling (ICAPS)*, 477–485.

Kim, K.; Campbell, J.; Duong, W.; Zhang, Y.; and Fainekos, G. 2015. DisCoF+: Asynchronous DisCoF with flexible decoupling for cooperative pathfinding in distributed systems. In *Proc. IEEE Intl. Conf. on Automation Science and Engineering (CASE)*, 369–376.

Luna, R.; and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proc. Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 294–300.

Ma, H.; Kumar, T. S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *Proc. AAAI Conf. on Artificial Intelligence*, 3605–3612.

Ma, H.; Li, J.; Kumar, T.; and Koenig, S. 2017. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proc. Intl. Joint Conf. on Autonomous Agents & Multiagent Systems (AAMAS)*, 837–845.

Ma, H.; Tovey, C.; Sharon, G.; Kumar, T. S.; and Koenig, S. 2016. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *Proc. AAAI Conf. on Artificial Intelligence*, 3166–3173.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2019. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. In *Proc. Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 535–542.

Okumura, K.; Tamura, Y.; and Défago, X. 2020. Time-Independent Planning for Multiple Moving Agents. *CoRR* abs/2005.13187. URL https://arxiv.org/abs/2005.13187.

Sha, L.; Rajkumar, R.; and Lehoczky, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers* 39(9): 1175–1185.

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. 2019. Multi-Agent Path Finding–An Overview. In *Artificial Intelligence - 5th RAAI Summer School*, volume 11866 of *LNCS*, 96–115. Springer.

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 *Proc. Intl. Symp. on Combinatorial Search (SoCS)*, 151–159.

Tel, G. 2000. *Introduction to distributed algorithms*. Cambridge university press.

Wang, H.; and Rubenstein, M. 2020. Walk, Stop, Count, and Swap: Decentralized Multi-Agent Path Finding With Theoretical Guarantees. *IEEE Robotics and Automation Letters* 5(2): 1119–1126.

Wiktor, A.; Scobee, D.; Messenger, S.; and Clark, C. 2014. Decentralized and complete multi-robot motion planning in confined spaces. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 1168–1175.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine* 29(1): 9–20.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proc. AAAI Conf. on Artificial Intelligence*.