Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding

Keisuke Okumura[1] , **Manao Machida**[2] , **Xavier Défago**[1] and **Yasumasa Tamura**[1]
[1]Tokyo Institute of Technology
[2]NEC Corporation
okumura.k@coord.c.titech.ac.jp, manao-machida@ap.jp.nec.com, {defago, tamura}@c.titech.ac.jp

Abstract

The Multi-agent Path Finding (MAPF) problem consists in all agents having to move to their own destinations while avoiding collisions. In practical applications to the problem, such as for navigation in an automated warehouse, MAPF must be solved iteratively. We present here a novel approach to iterative MAPF, that we call Priority Inheritance with Backtracking (PIBT). PIBT gives a unique priority to each agent every timestep, so that all movements are prioritized. Priority inheritance, which aims at dealing effectively with priority inversion in path adjustment within a small time window, can be applied iteratively and a backtracking protocol prevents agents from being stuck. We prove that, regardless of their number, all agents are guaranteed to reach their destination within finite time, when the environment is a graph such that all pairs of adjacent nodes belong to a simple cycle of length 3 or more (e.g., biconnected). Our implementation of PIBT can be fully decentralized without global communication. Experimental results over various scenarios confirm that PIBT is adequate both for finding paths in large environments with many agents, as well as for conveying packages in an automated warehouse.

1 Introduction

In systems using physical moving agents, it is essential to allow agents to reach their own destinations smoothly without collisions, by providing valid paths while minimizing excess travel time. This problem, known as *Multi-agent Path Finding* (MAPF), is however computationally difficult due to the search space growing exponentially as the number of agents increases. MAPF is an important problem due to its applications in various domains, e.g., traffic control [Dresner and Stone, 2008], automated warehouse [Wurman *et al.*, 2008], or airport surface operation [Morris *et al.*, 2016], etc.

Previous research on MAPF focuses on solving a "one-shot" version of the problem, i.e., to make agents reach their goal from their initial position only once. In practical applications, such as conveying packages in a warehouse [Ma *et al.*, 2017], MAPF must however be solved *iteratively*. That is, whenever an agent reaches a goal, it receives a new one. This rules out any simple adaptation of offline and computationally-intensive optimal solutions due to prohibitive computations, and new goals typically appearing at runtime. Furthermore, centralized solutions being inherently problematic for systems with many agents due to scalability concerns, decoupled algorithms such as prioritized route planning are adequate, especially if fully decentralized. Decentralized solutions are highly attractive to multi-agent/robot systems for many reasons, including a higher potential for robustness and fault-tolerance, better scalability, and lower production cost [Yan *et al.*, 2013]. Finally, implementations without global communication are preferable since they provide better potential for scalability and concurrency.

In this paper, we present a novel algorithm for *iterative Multi-agent Path Finding* (iterative MAPF), called *Priority Inheritance with Backtracking* (PIBT), which focuses on the adjacent movements of multiple agents based on prioritized planning in a short time window. Priority inheritance is a well-known approach to deal effectively with priority inversion in real-time systems [Sha *et al.*, 1990], and is applied here to path adjustment. When a low-priority agent X impedes the movement of a higher-priority agent Y, agent X temporarily inherits the higher-priority of Y. To avoid a situation where agents are stuck waiting, priority inheritance is executed in combination with a backtracking protocol. Since PIBT assumes that agents can only communicate when located within two hops of each other (e.g., Manhattan distance 2 in a grid environment), it can be implemented in a fully decentralized way and inherits the above characteristics.

Our main contributions are two-folds: 1) we propose an algorithm ensuring that every agent always reaches its destination within finite time as long as the environment satisfies the condition that all pairs of adjacent nodes belong to a simple cycle of length 3 or more (includes undirected biconnected graphs); and 2) we evaluate that algorithm in various environments, showing its practicality. In particular, experimental results over various scenarios confirm its adequateness both for finding paths in large environments with many agents, as well as for conveying packages in an automated warehouse.

The paper is organized as follows. Section 2 reviews existing algorithms for MAPF and a variant called *Multi-agent Pickup and Delivery* (MAPD). Section 3 defines iterative MAPF, which provides an abstract framework for multiple moving agents. Section 4 presents the PIBT algorithm and its theoretical analysis. Section 5 presents empirical results on both path finding (MAPF) and pickup and delivery (MAPD). Section 6 concludes the paper and discusses future work.

2 Related Work

Many complete MAPF algorithms exist, such as A* with Operator Decomposition [Standley, 2010], Enhanced Partial Expansion A* [Goldenberg *et al.*, 2014], Increasing Cost Tree Search [Sharon *et al.*, 2013], Conflict-based Search [Sharon *et al.*, 2015], M* [Wagner and Choset, 2015], etc.

Finding an optimal solution is however NP-hard [Yu and LaValle, 2013], and optimal algorithms do not scale in the number of agents and are too costly for iterative use, hence the need for sub-optimal solvers, e.g., FAR [Wang *et al.*, 2008], MAPP [Wang and Botea, 2011], Tree-based Agent Swapping Strategy [Khorshid *et al.*, 2011], BIBOX [Surynek, 2009], CBS variants [Barer *et al.*, 2014; Cohen *et al.*, 2016].

Push and Swap/Rotate [Luna and Bekris, 2011; de Wilde *et al.*, 2013], which partly inspired our proposal, are sub-optimal centralized approaches that allow one agent to push another away from its path. However, they only allow a single agent or a pair of agents to move at each timestep. Enhanced Push and Swap include Parallel Push and Swap [Sajid *et al.*, 2012] where all agents can move simultaneously, or Push-Swap-Wait [Wiktor *et al.*, 2014] which takes a decentralized approach in narrow passages. DisCoF [Zhang *et al.*, 2016] combines decoupled prioritized planning with fallback to *coupled* push and swap in case of conflicts. PIBT, the proposed method, can be seen as a combination of safe "push" operations, thanks to backtracking and dynamic priorities.

Prioritized planning is computationally cheap and hence attractive for MAPF. Hierarchical Cooperative A* (HCA*) and Windowed HCA* (WHCA*) [Silver, 2005] are decoupled approaches in that they plan a path for each agent one after the other while avoiding collisions with previously computed paths. WHCA* uses a limited lookahead window. Our proposal, PIBT, is based on WHCA* with a window size of one. Decentralized solutions, i.e., where each agent computes its own path based on information from other agents, are inherently decoupled. Decentralized solutions for MAPF [Velagapudi *et al.*, 2010; Čáp *et al.*, 2015; Chouhan and Niyogi, 2015] are hence usually prioritized. The negotiation process for ordering priorities studied by Azarm et al. [Azarm and Schmidt, 1997] solves conflicts by having involved robots try all priority orderings, and deals with congestion by limiting negotiation to at most 3 robots and letting others wait. A recent theoretical analysis of prioritized planning [Ma *et al.*, 2018] identifies instances that fail for any order of static priorities. That study, which also presents a centralized approach called Priority-Based Search, actually provides a very strong case for planning based on *dynamic* priorities, such as the approach taken with PIBT.

The problem of Multi-agent Pickup and Delivery (MAPD) [Ma *et al.*, 2017] abstracts real scenarios such as an automated warehouse, and consists of both allocation and route planning. Agents are assigned to a task from a stream of delivery tasks and must consecutively visit both a pickup and a delivery location. The paper proposes two decoupled algorithms based on HCA* for MAPD, called respectively Token Passing (TP) and Token Passing with Task Swap. These algorithms can easily be adapted to be decentralized but require a certain amount of non-task endpoints where agents do not block other agents' motion.

3 Problem Definition

The problem of *iterative Multi-agent Path Finding* (iterative MAPF) is a generalization of problems addressing multiple moving agents, including both Multi-agent Path Finding (MAPF) and Multi-agent Pickup and Delivery (MAPD). Since iterative MAPF is an abstract model we do not intend to solve it directly, rather, it is necessary to embody task creation according to target domains.

The system consists of a set of agents, $A = \{a_1, \ldots, a_n\}$, and an environment given as a graph $G = (V, E)$, where agents occupy nodes in V and move along edges in E. Considering practical situations, G must be a *simple* (neither loops nor multi-edges) and *strongly-connected* (every node is reachable from every other node) directed graph. This includes all simple undirected graphs that are connected.

Let $v_i(t)$ denote the node occupied by agent a_i at discrete time t. The initial position of agent a_i is $v_i(0)$ and given as input. At each timestep, a_i selects one node $v_i(t+1) \in \{v | v \in V,\ (v_i(t), v) \in E\} \cup \{v_i(t)\}$ as its location for the next timestep. Agents must avoid 1) *collision*: $v_i(t) \neq v_j(t)$; and 2) *intersection* with others: $v_i(t) \neq v_j(t+1) \vee v_j(t+1) \neq v_j(t)$. We do not prohibit rotations, so $v_i(t+1) = v_j(t) \wedge v_j(t+1) = v_k(t) \wedge \cdots \wedge v_l(t+1) = v_i(t)$ is possible.

Let $\Gamma = \{\tau_1, \tau_2, \ldots\}$ be a dynamic set of tasks with new tasks being added over time, i.e., not all tasks are known initially. A task is defined as a finite set of goals $\tau_j = \{g_1, g_2, \ldots, g_m\}$ where $g_k \in V$, possibly with dependencies as a partial order on g_k. Let t_j be the timestep when τ_j is added to Γ. An agent is *free* when it has no assigned task. A task $\tau_j \in \Gamma$ can only be assigned to free agents. When τ_j is assigned to a_i, a_i starts visiting goals in τ_j. When all goals have been visited, τ_j is completed and a_i becomes free.

Let now $\pi(a_i, \tau_j) = (v_i(t), v_i(t+1), \ldots)$ be the path of a_i when τ_j is assigned at timestep t until a_i completes τ_j. When a_i is free, it is assigned a dummy task $\bar{\tau}_l$ with path $\pi(a_i, \bar{\tau}_l)$, where index l is for the uniqueness of π. The path must ensure no *collision* and no *intersection*, with the following additional condition; $g_k \in \pi(a_i, \tau_j),\ \forall g_k \in \tau_j$. The service time $\lambda(\pi(a_i, \tau_j), t_j)$ is defined as the time elapsed from start (t_j) to the completion of τ_j by a_i. Note that $\lambda(\pi(a_i, \tau_j), t_j)$ does not usually equal to $|\pi(a_i, \tau_j)|$ because τ_j is not always assigned immediately.

We say that iterative MAPF has a solution if and only if the service times of all tasks are finite. Iterative MAPF is a generic problem which, depending on its concrete instance, can rely on an objective function of the following general

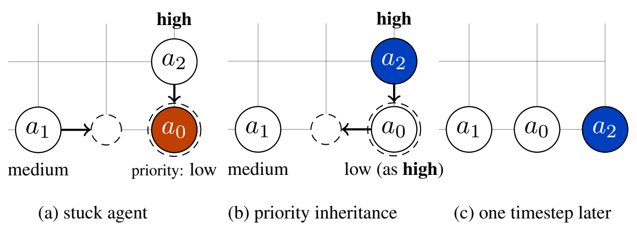


Figure 1: Examples of priority inheritance. Without inheritance (1a), a stuck agent (a_0) cannot give way to a high-priority agent (a_2) without risking collision into a third agent (a_1). With priority inheritance (1b), a_0 temporarily inherits the priority of a_2, forcing a_1 to give way, and solving the situation (1c).

form.

$$\begin{aligned} \underset{\Gamma_i,\pi}{\text{minimize}} \quad & \sum_{i=1}^{n} \sum_{\tau_j \in \Gamma_i \cup \{\bar{\tau}_l\}} \lambda(\pi(a_i, \tau_j), t_j) \\ \text{s.t.} \quad & \Gamma = \Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_n \\ & \Gamma_i \cap \Gamma_{i'} = \emptyset,\ \forall i \neq i' \in \{1, \ldots, n\} \end{aligned}$$

where Γ_i is the set of tasks allocated to a_i. Concrete instances of iterative MAPF include route planning and task allocation, for which objective functions are described in Section 5.

Depending on context, Γ_i is determined *a priori*. For instance, in the literature on MAPF, Γ_i is provided as $\{\tau_i = \{g_i\}\}$ where g_i is a goal node of a_i. A start node is defined by $v_i(0)$. In the basic definition of MAPF, termination is achieved once all agents are at their goal in the same timestep. To satisfy this requirement, when a_i that once reached g_i leaves it, a new task $\tau = \{g_i\}$ is added to Γ_i.

MAPD is an instance of iterative MAPF, where every task is a tuple of two nodes (pickup and delivery) instead of a set.

4 Priority Inheritance with Backtracking

This section introduces the concept of Priority Inheritance with Backtracking (PIBT), describes the algorithm and its properties, as well as how to apply PIBT to specific problems such as MAPF and MAPD. To avoid unnecessarily complex explanations, we introduce PIBT as a centralized algorithm and focus on the analysis of the prioritization scheme itself. PIBT relies on a decoupled approach making it easily amenable to decentralization. We briefly discuss later how to adapt it to a decentralized case.

4.1 Basic Concept

Prioritized approaches are effective for iterative use, so PIBT essentially implements WHCA* [Silver, 2005] with a time window of one. At every timestep, each agent updates its own unique priority. Paths are computed one by one while avoiding collisions with previously computed paths.

Priority Inheritance

Priorities alone can still result in a deadlock (see Fig. 1a) resulting from a case of *priority inversion*. Priority inversion occurs when a low-priority agent (a_0) fails to obtain a resource held by a medium-priority agent (a_1), even though it holds a second resource claimed by a higher-priority agent (a_2). This is typically addressed by *priority inheritance* [Sha *et al.*, 1990] (Fig. 1b). The rationale is that, a low-priority agent (a_0) temporarily inherits the higher priority of agents (a_2) claiming resources it holds, thus forcing medium-priority agents (a_1) out of the way (Fig. 1c).

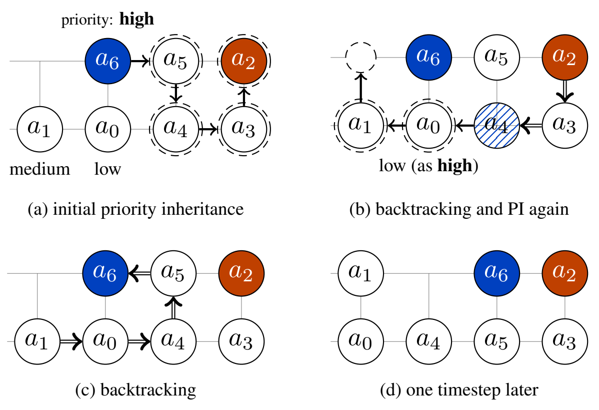


Figure 2: Example of PIBT. Flows of back tracking are drawn as doubled-line arrows. Since a_2 is stuck (2a), back tracking returns invalid to a_3, subsequently a_4. a_4 executes other priority inheritance to a_0 (2b). a_4, a_0, a_5 and a_6 wait for the results of back tracking (2c) and then start moving (2d).

Backtracking

Priority inheritance deals effectively with priority inversion, but it does not completely ensure deadlock-freedom. For instance, as shown in Fig. 2a, agent a_2 finds no escape as a result of consecutive priority inheritance ($a_6 \to a_5 \to a_4 \to a_3$).

The solution relies on *backtracking*: i.e., any agent a that gives its priority must wait for an outcome (valid or invalid). If valid, a successfully moves to the desired node. Otherwise, it must change its target, excluding 1) nodes requested from higher priority agents, and 2) nodes denied from inherited agents. Upon finding no valid or unoccupied nodes, a sends back an invalid outcome to the agent from which it inherited its priority. In Fig. 2b, a_2 first sends invalid back to a_3, which in turn sends invalid to a_4. Since a_0 has lower priority, a_4 can let a_0 inherit its priority as an alternative, which leads to a valid outcome (Fig. 2c), and agents can move (Fig. 2d).

4.2 Algorithm

We begin by defining the notion of groups of interacting agents, since the algorithm and its proofs rely on it. When two agents located within two hops of each other move, they might collide in the next step, and they are said to be directly interacting in that step. For an agent a_i, a group of interacting agents $\mathcal{A}_i(t) \subseteq A$ is then defined by transitivity over direct interactions in a timestep t. Note that, given a_i and $\mathcal{A}_i(t)$, then for any other agent $a_j \in \mathcal{A}_i(t)$ we have $\mathcal{A}_i(t) = \mathcal{A}_j(t)$. Whenever obvious from context, we use $\mathcal{A}(t)$ or $\mathcal{A}$.

Since agents belonging to different groups cannot affect each other, path planning can effectively occur in parallel. As a result, PIBT is decentralized and only relies on local communication, i.e., it is sufficient that two agents in close proximity can talk directly and utilize multi-hop communication.

Algorithm 1 PIBT (code at timestep t)

```
1: UNDECIDED ← $\mathcal{A}(t)$                                ▷ agents list
2: OCCUPIED ← $\emptyset$                                       ▷ nodes list
3: update all priorities $p_i(t)$
4: **while** UNDECIDED $\neq \emptyset$ **do**
5:     let $a$ be the agent with highest priority in UNDECIDED
6:     PIBT$(a, \bot)$
7: **end while**
8:
9: **function** PIBT$(a_i, a_j)$
10:    UNDECIDED ← UNDECIDED $\setminus \{a_i\}$
11:    $C_i \leftarrow (\{v \mid (v, v_i(t)) \in E\} \cup \{v_i(t)\})$
               $\setminus(\{v_j(t)\} \cup$ OCCUPIED$)$
12:    **while** $C_i \neq \emptyset$ **do**
13:        $v_i^* \leftarrow \arg\max_{v \in C_i} f_i(v)$
14:        OCCUPIED ← OCCUPIED $\cup \{v_i^*\}$
15:        **if** $\exists a_k \in$ UNDECIDED such that $v_i^* = v_k(t)$ **then**
16:            **if** PIBT$(a_k, a_i)$ is valid **then**
17:                $v_i(t+1) \leftarrow v_i^*$
18:                **return** valid
19:            **else**
20:                $C_i \leftarrow C_i \setminus$ OCCUPIED
21:            **end if**
22:        **else**
23:            $v_i(t+1) \leftarrow v_i^*$
24:            **return** valid
25:        **end if**
26:    **end while**
27:    $v_i(t+1) \leftarrow v_i(t)$
28:    **return** invalid
29: **end function**
```

Algorithm 1 describes PIBT, where $p_i(t) \in \mathbb{R}$ is the priority of agent a_i at timestep t. $f_i(v), v \in V$ is the valuation function of nodes for a_i at the current timestep, where a larger value of $f_i(v)$ means that a_i has a higher preference to move to node v at the next timestep. PIBT can be described recursively, especially with respect to priority inheritance and backtracking. We assume that the groups of interacting agents $\mathcal{A}(t)$ are fully identified prior to starting the algorithm. UNDECIDED is the set of agents that have not manifested an intention (initially $\mathcal{A}(t)$). At each timestep, agents update their priorities in the way mentioned later [Line 3]. Subsequently, select the agent with highest priority in UNDECIDED and call function PIBT [Lines 5,6]. This loops until all agents in $\mathcal{A}$ determines their nodes for next timestep.

The function PIBT takes two arguments: the agent a_i making a decision and an agent a_j from which a_i inherits its priority, or $\bot$ if there is none. a_i must select a node for $v_i(t+1)$ from the set of candidates nodes C_i. Here, C_i consists of $v_i(t)$ and its neighbors while excluding 1) nodes requested from higher priority agents, 2) nodes denied from inherited agents, and 3) $v_j(t)$ for avoiding *intersection* [Line 11].

If C_i is empty [Line 12], this means that a_i is stuck like a_2 in Fig. 2b. In that case, a_i selects $v_i(t)$ as $v_i(t+1)$ and returns invalid as outcome [Lines 27,28]. Otherwise (C_i non-empty), a_i selects the most valuable node v_i^* [Line 13]. If v_i^* is occupied by an agent a_k in UNDECIDED [Line 15], a_i applies priority inheritance and waits for the result of backtracking [Line 16], otherwise, v_i^* is set to $v_i(t+1)$ and returns valid [Lines 23,24]. Upon a valid outcome, a_i can move to v_i^* [Line 17]. Otherwise, a_i updates C_i to exclude prohibited nodes [Line 20] and repeats the process [Lines 12–26].

We now prove that the agent with highest priority can always move. The intuition is that it can always move along a simple cycle.

Lemma 1. *Let a_1 denote the agent with highest priority at timestep t and v_1^* an arbitrary neighbor node of $v_1(t)$. If there exists a simple cycle $\mathbf{C} = (v_1(t), v_1^*, \dots)$ and $|\mathbf{C}| \geq 3$, Algorithm 1 makes a_1 move to v_1^* in the next timestep.*

Proof. After deciding v_1^*, it is added to OCCUPIED [Line 13,14]. From the definition of C_i, $\forall i \neq 1$ [Line 11], no other agent can select v_1^*.

Consider first the case when v_1^* is unoccupied. No other agent can enter v_1^* and a_1 is guaranteed to move to v_1^*.

Consider now that v_1^* is occupied by some agent a_2. a_2 inherits a_1's priority which is highest. The existence of cycle $\mathbf{C}$ ensures that C_2 is not initially empty ($|\mathbf{C}| \geq 3$, $\mathbf{C}$ contains $v_1(t)$ and $v_2(t)$). From the definition of C_2, *collision* and *intersection* with a_1 are implicitly prevented. If a_2 selects $v_2^* \in C_2$ such that v_2^* is unoccupied by another agent, then a_2 is guaranteed to move to v_2^* according to the same logic as for a_1. Consequently, a_1 successfully moves to v_1^*. This forms the basis and the remaining now proves the induction step.

Following this, suppose that a_i grants its priority to a_j ($a_i, a_j \in \mathcal{A}_1(t), 2 \leq i < j$). From the definition of C_j, *collision* and *intersection* with other agents are implicitly prevented. If a_j selects an unoccupied node $v_j^* \in C_j$, a_j can move to v_j^* and a series of agents that will receive the result of backtracking as valid, including a_i and a_2, can move to its current target node based on the same argument.

Now, by contradiction, suppose that a_2 fails to move to any node, i.e. a_2 receives invalid as the result of backtracking and C_2 becomes empty. Let DECIDED denote $\mathcal{A}_1(t) \setminus \{$UNDECIDED $\cup \{a_1\}\}$. The assumption says all agents in DECIDED failed to move. Since $a_i \in$ DECIDED ($\forall i \geq 2$) tried to move other nodes until C_i became empty, neighbor nodes of $v_2(t)$ exclusive of $v_1(t)$ are in OCCUPIED, and all nodes adjacent to $v_j(t)$ ($\forall j \geq 3$) are in OCCUPIED. Incidentally, the existence of $\mathbf{C}$ indicates at least one agent $a^* \neq a_2$ on $\mathbf{C}$ had initially at least one free neighbor node including $v_1(t)$. At the beginning of the original priority inheritance from a_1, even though all nodes in $\mathbf{C}$ is occupied of someone, the agent on the last node of $\mathbf{C}$ has a free neighbor node, i.e., $v_1(t)$, otherwise, it is obviously there exists a^*. Considering the mechanism of priority inheritance, DECIDED must contain all agents on $\mathbf{C}$ exclusive of a_1 because $\mathbf{C}$ contains $v_2(t)$. This is contradiction; a^* should be in DECIDED but a^* initially had an free neighbor node. Therefore, a_2 can finally move to a node not $v_1(t)$ or $v_2(t)$ (equals to v_1^*). Thus, a_1 can move to v_1^*. □

Prioritization

Let τ_i denote the task currently assigned to a_i, and g_i the current goal in τ_i. Let $\eta_i(t) \in \mathbb{N}$ be the number of timesteps elapsed since a_i last updated g_i and until time t. Note that $\eta_i(0) = 0$ or when a_i is free. Let $\epsilon_i \in [0, 1)$ be a value unique to each agent a_i. At every timestep, $p_i(t) \in \mathbb{R}$ is computed as follows; $p_i(t) \leftarrow \eta_i(t) + \epsilon_i$. Obviously, $p_i(t)$ is unique among agents at every timestep.

This prioritization leads to the following theorem.

Theorem 2. *If G has a simple cycle $\mathbf{C}$ for all pairs of adjacent nodes and $|\mathbf{C}| \geq 3$ then, with PIBT, all agents reach their own destination within $\text{diam}(G) \cdot |A|$ timesteps after being given.*

Proof. From Lemma 1, the agent with highest priority reaches its own goal within $\text{diam}(G)$ timesteps. Based on the definition of $\eta_i(t)$, once some agent a_i has reached its goal, $p_i(t)$ is reset and is lower than that of all other agents that have not reached their goal yet. Those agents see their priority increase by one. As long as such agents remain, exactly one of them must have highest priority. In turn, that agent reaches its own goal after at most $\text{diam}(G)$ timesteps. This repeats until all agents have reached their goal at least once, which takes at most $\text{diam}(G) \cdot |A|$ timesteps in total. $\square$

A typical example that satisfies the above condition is any biconnected undirected graph. The opposite is not true however, and Theorem 2 is expressed more generally to account for directed graphs. For instance, a directed ring satisfies the condition even though it is not biconnected. Note that we neither ensure nor require that all agents be on their goal simultaneously.

Complexity Analysis

Now we consider the computational complexity of PIBT. To simplify, assume that PIBT performs in a centralized way. Let $\Delta(G)$ denote the maximum degree of G, and F be the maximum time required to compute the most valuable node for the next timestep [Line 13]. F depends on both G and the node evaluation function f_i.

Proposition 3. *The computational complexity of PIBT in one timestep is $O(|A| \cdot \Delta(G) \cdot F)$.*

Proof. For an agent a_i, the function $\mathsf{PIBT}(a_i, \cdot)$ is called once in one timestep because $\mathsf{PIBT}(a_i, \cdot)$ is called if and only if $a_i \in$ UNDECIDED [Line 5 and Line 15,16], and a_i is removed from UNDECIDED after calling [Line 10]. The main loop in Line 12 is repeated at most $\Delta(G) + 1$ times. If priority inheritance occurs and succeeds [Line 16], the loop is broken [Line 18], otherwise $|C_i|$ decreases [Lines 14 and 20]. If the absence of priority inheritance, the loop is also broken [Line 24]. At each iteration of the loop, the node selection [Line 13] occurs and is itself of complexity F. Thus, we derive the complexity of PIBT as $O(|A| \cdot \Delta(G) \cdot F)$. $\square$

Communication

Since PIBT is based on priorities and assumes only local interactions, it is easily adapted to a decentralized context. The part of priority inheritance and backtracking is done by propagation of information. In a decentralized context, PIBT requires agents to know others' priorities before they decide their next nodes. Usually, this requires $|\mathcal{A}|^2$ communication between agents, however, updating rule of p_i relaxes this effort, e.g., stores other agents priories and communicates only when η_i becomes zero. Therefore, the communication cost of PIBT mainly depends on the information propagation phase.

Let us now consider this phase. In Algorithm 1, communication between agents corresponds to calling function PIBT [Line 6,16] and backtracking [Line 18,24,28]. $\mathsf{PIBT}(a_i, \cdot)$ is never called twice in each timestep, as discussed in the previous section. Moreover, each agent sends a backtracking message at most once in each timestep. Overall, the communication cost for PIBT at each timestep is linear w.r.t. the number of agents, that is, $O(|A|)$. In reality, this can be even less because the figures depend on the number of interacting agents $|\mathcal{A}|$ which can be much smaller than $|A|$.

4.3 Applying to Specific Problems

The valuation function of nodes, $f_i(v)$, must be defined based on the concrete problem. In the two scenarios introduced later, we use $-\text{cost}(v, g_i)$ as f_i, where $\text{cost}(u, u')$ is the length of the shortest path from u to u', and g_i is the current destination if a_i has a task. [1] To avoid unnecessary priority inheritance, the presence (or not) of an agent is used as a tie-breaking rule. If a_i is free, make $v_i(t)$ to be the highest and use the same tie-breaking rule.

Multi-agent Path Finding (MAPF)

In any graph G, PIBT with above f_i does not ensure that all agents are located on their goals simultaneously, which MAPF requires. We confirmed a certain kind of livelock situations in our experiment. It might be possible to ensure completeness for some classes of graphs through a more complex f_i, e.g., akin to "swap" operation [Luna and Bekris, 2011], but we do not address this issue here. Note that PIBT is aimed at iterative use rather than one-shot use.

Multi-agent Pickup and Delivery (MAPD)

The MAPD problem is a typical example of iterative MAPF. Since PIBT only provides coordinated movements of agents, we need to complement the method of task allocation.

Different from the proposed algorithms by Ma *et al.* [Ma *et al.*, 2017], deadlocks never occur with PIBT even without non-task endpoints, as long as G follows the condition stated in Theorem 2. Moreover, PIBT does not require well-formed instances which guarantees that for any two endpoints including pickup and delivery locations, a path exists between them that traverses no other endpoints. Practically, MAPD is performed in orderly environments, i.e., the assumption we stated of a graph is adequate. Therefore, there is no need to care about the behavior of free agents.

From the above reasons, we propose a simple allocation process; at every timestep, each free agent moves to the near-

[1] Experiments on MAPF rely on A* to obtain the shortest paths. Those on MAPD used the Floyd-Warshal algorithm, to build the all-pairs distance matrix, during a preprocessing phase. The comparison (TP) also utilized the matrix.

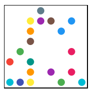
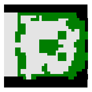
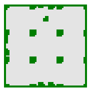
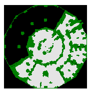
		PIBT				WHCA*				PPS			
field	agents	path	makespan	success	runtime	path	makespan	success	runtime	path	makespan	success	runtime
simple map 8×8 	10	6.7 (6.7)	11.2 (11.2)	99	0 (0)	5.8 **(5.8)**	10.5 **(10.5)**	98	1 (1)	8.3 (8.3)	13.7 (13.7)	100	0 **(0)**
	15	7.5 (7.5)	12.7 **(12.5)**	99	1 (1)	6.2 **(6.2)**	12.6 (12.5)	85	2 (2)	11.0 (11.1)	18.3 (18.5)	100	1 **(1)**
	20	8.4 (7.9)	15.0 (13.9)	89	2 **(2)**	6.5 **(6.4)**	13.8 **(13.5)**	44	4 (3)	13.8 (12.9)	23.0 (22.2)	100	2 (2)
	25	9.5 (10.0)	18.1 (20.3)	85	2 **(2)**	6.8 **(6.8)**	15.0 **(15.0)**	4	5 (5)	16.9 (16.7)	26.9 (27.8)	100	4 (4)
	30	10.7	19.7	75	3	-	-	0	-	21.3	32.7	99	7
	40	14.3	25.9	41	5	-	-	0	-	33.3	47.4	99	21
	50	33.3	67.0	37	12	-	-	0	-	58.0	72.7	95	73
lak105d 25×31 	10	20.3 (20.2)	34.2 **(34.0)**	100	2 (2)	19.0 **(19.0)**	34.0 **(34.0)**	99	4 (4)	21.1 (21.1)	35.7 (35.6)	100	2 **(2)**
	25	23.0 (23.0)	41.1 (40.8)	99	8 (8)	19.9 **(19.9)**	40.4 **(40.5)**	87	22 (22)	26.0 (26.2)	47.0 (47.1)	100	9 **(6)**
	50	26.9 (26.4)	49.0 **(47.6)**	95	19 **(19)**	21.1 **(21.1)**	47.9 (47.9)	36	92 (92)	36.5 (35.4)	67.1 (64.5)	100	40 (26)
	75	31.7 (32.1)	59.4 (64.5)	77	36 **(37)**	22.8 **(23.7)**	54.0 **(57.0)**	3	210 (230)	52.1 (47.7)	97.4 (91.0)	98	118 (62)
	100	37.1	67.6	62	55	-	-	0	-	69.9	130.3	98	284
arena 49×49 	10	32.4 (32.4)	56.5 **(56.5)**	100	5 (5)	31.6 **(31.6)**	56.5 **(56.5)**	100	5 (5)	32.6 (32.6)	56.9 (56.9)	100	3 **(3)**
	25	33.1 (33.1)	63.7 **(63.7)**	100	12 (12)	31.5 **(31.5)**	64.0 (64.0)	100	17 (17)	33.8 (33.8)	65.1 (65.1)	100	9 **(9)**
	50	35.0 (35.0)	68.7 **(68.7)**	100	25 (25)	32.2 **(32.2)**	69.3 (69.3)	100	59 (59)	36.6 (36.6)	72.7 (72.7)	100	20 **(20)**
	100	37.6 (37.6)	74.1 **(74.0)**	98	52 **(52)**	32.4 **(32.4)**	74.8 (74.8)	98	253 (253)	41.1 (41.1)	82.1 (82.0)	100	53 (53)
	200	42.1 (42.1)	77.5 **(77.7)**	96	121 **(121)**	33.3 **(33.3)**	79.8 (79.9)	82	1292 (1294)	51.0 (50.9)	96.6 (96.4)	100	185 (183)
	300	46.0 (46.2)	80.6 **(81.6)**	90	210 **(212)**	34.3 **(34.3)**	83.8 (83.8)	26	3482 (3492)	62.0 (62.1)	111.9 (112.2)	100	477 (463)
	400	49.4 (47.5)	84.1 (84.0)	74	318 **(305)**	35.1 **(35.1)**	83.0 **(83.0)**	1	6703 (6703)	74.3 (72.6)	129.4 (139.0)	100	1048 (847)
	500	52.7	88.4	61	458	-	-	0	-	87.6	147.0	99	2122
ost003d 194×194 	10	163.3 (163.3)	316.4 (316.4)	100	208 (208)	159.6 **(159.6)**	316.3 **(316.3)**	100	290 (290)	163.8 (163.8)	317.3 (317.3)	100	192 **(192)**
	25	170.6 (170.6)	356.3 **(356.3)**	100	586 (586)	162.1 **(162.1)**	356.4 (356.4)	100	1179 (1179)	172.4 (172.4)	360.3 (360.3)	100	485 **(485)**
	50	173.4 (173.0)	370.9 (370.8)	100	1413 (1407)	160.3 **(160.3)**	370.2 **(370.2)**	99	3317 (3317)	177.2 (176.8)	379.0 (378.8)	100	1053 **(1050)**
	100	179.6 (178.7)	381.6 (381.2)	100	3298 (3224)	158.8 **(158.8)**	380.2 **(380.2)**	91	8436 (8436)	189.0 (188.0)	403.7 (403.3)	100	2796 **(2748)**
	200	190.5 (189.8)	393.1 (392.2)	100	8550 **(8392)**	160.1 **(160.1)**	391.1 **(391.1)**	57	22853 (22853)	215.6 (214.7)	453.1 (454.1)	100	9506 (9276)
	300	198.9 (194.1)	402.3 **(396.2)**	98	14614 **(13069)**	155.2 **(155.2)**	397.8 (397.8)	6	38453 (38453)	243.3 (233.1)	498.4 (488.8)	100	21630 (18219)
	400	207.4	410.7	99	21813	-	-	0	-	274.8	551.3	98	42966
	500	215.7	424.9	94	29834	-	-	0	-	308.3	606.4	100	76312

Table 1: The results of MAPF experiments. "path" corresponds to path cost. "success" means the percentage that solver successfully solved within 100 instances. The unit of "runtime" is ms. In "path", "makespan" and "runtime", average only in the case of success within a solver (no decoration) and average over instances that were successfully solved by all solvers (in parentheses) are both shown.

est pickup location of the non-assigned task, ignoring the behavior of other agents, and is assigned to a task if and only if it reaches the pickup location and the task remains. The following theorem directly follows from Theorem 2.

Theorem 4. *If G has a simple cycle $\mathbf{C}$ for all pairs of adjacent nodes and $|\mathbf{C}| \geq 3$, then PIBT with the above-mentioned allocation process solves MAPD.*

5 Evaluation

This section assesses the effectiveness of PIBT quantitatively, through computer simulation. The simulator was developed in C++, [2] and all experiments were run on a laptop with Intel Core i7 2.2GHz CPU and 8GB RAM. The experiments address both the MAPF and MAPD problems.

5.1 Multi-agent Path Finding (MAPF)

We carefully chose four undirected graphs, shown in Table 1, as testbeds for MAPF. Three of them (lak105d, arena, ost003d) come from the game *Dragon Age: Origins* [Sturtevant, 2012] and are commonly used to evaluate multi-agent algorithms. For comparison with PIBT, we also experimented using two sub-optimal MAPF solvers, namely WHCA* [Silver, 2005] and Parallel Push and Swap (PPS) [Sajid *et al.*, 2012]. The window size of WHCA* was set to 5 on the simple map, and to 10 otherwise. All solvers were executed on the same set of 100 settings with random initial positions and goal locations for each configuration while changing the number of agents. An execution was considered failed if the solver yielded a failure or failed to provide a result after a timeout of 5 minutes. The evaluation metrics are the following four, referring to other MAPF studies: 1) path cost, i.e., the timestep when *each* agent reaches its given goals and never move from then; 2) makespan, i.e., the timestep when *all* agents reach their given goals; 3) runtime; and 4) success rate over 100 instances. Note that the sum of service time for one agent defined in Section 3 is different from path cost, e.g., assume that an agent reaches its goal and apart there. The service time does not augment when the agent is on the goal since the task is not issued, whereas the path cost does count that time. Stated in section 3, the typical objective function of iterative MAPF is service time and path cost is the specific metric of the one-shot version. However, following the literature on one-shot MAPF, we address here path cost and makespan, which is also indirect criteria of service time.

The results, shown in Table 1, indicate, for each solver, both an average over the instances successful with that solver (no decoration), and an average over only the instances that were successfully solved by all three solvers (in parentheses). In summary, PIBT compares well with other methods; Path efficiency is reasonable with a high success rate in most cases, moreover, PIBT is comparable with PPS in terms of runtime cost. In general, PIBT appears to have an advantage in huge fields with many agents since it relies only on local communication, and here, the experimental results support the strength of our proposal (see ost003d in Table 1). Conversely, PIBT suffers from a somewhat low success rate in dense situations (see the simple map). PIBT fails on MAPF either because the graph property mentioned before is not met or due to livelock situations.

[2] The code is available at https://kei18.github.io/pibt/

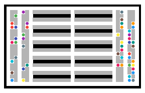
		PIBT			TP		
freq.	agents	make-span	service time	run-time	make-span	service time	run-time
0.2	10	**2531**	**29**	**255**	2543	38	3007
	20	**2527**	**26**	**405**	2541	38	3864
	30	**2525**	**25**	**534**	2540	38	4790
	40	**2524**	**25**	**648**	2542	38	5971
	50	**2524**	**24**	**776**	2541	38	6990
0.5	10	**1224**	**116**	**142**	1240	124	882
	20	**1038**	**28**	**218**	1058	40	5709
	30	**1033**	**25**	**326**	1059	40	7211
	40	**1031**	**24**	**410**	1058	40	8019
	50	**1031**	**24**	**494**	1058	40	9629
1	10	**1135**	**296**	**144**	1154	309	907
	20	**652**	**77**	**152**	672	85	1778
	30	**552**	**33**	**178**	576	41	4258
	40	**540**	**27**	**229**	580	43	11309
	50	**537**	**25**	**286**	581	44	13953
2	10	**1115**	**403**	**145**	1134	415	933
	20	**609**	**167**	**154**	631	177	1852
	30	**448**	**92**	**164**	478	101	2917
	40	**370**	**58**	**175**	404	67	4187
	50	**328**	**41**	**184**	369	51	5810
5	10	**1105**	**470**	**143**	1125	479	976
	20	**597**	**231**	**155**	616	238	1889
	30	**429**	**152**	**164**	454	159	2895
	40	**346**	**114**	**172**	381	121	4022
	50	**299**	**92**	**183**	341	99	5619
10	10	**1103**	**492**	**144**	1122	501	884
	20	**596**	**252**	**153**	612	259	1779
	30	**425**	**173**	**163**	452	179	2717
	40	**344**	**135**	**173**	371	140	4058
	50	**294**	**112**	**182**	336	118	5461

Table 2: The results of MAPD experiments. Each value is an average of 100 instances. The unit of runtime is ms. The graph is 21×35 4-connected grid. Gray cells in the MAPD map are task endpoints, i.e., pickup and delivery locations.

Experimental results in arena show drastic improvements in both runtime and makespan for PPS and WHCA* when compared to the original paper of PPS [Sajid *et al.*, 2012].

The improvement in runtime is easily explained by heavy optimization of our implementation and, in particular, of the underlying A* algorithm and its use (aggressive caching, elimination of redundant computations across agents, ...).

The improvement in makespan is difficult to isolate but is likely to come as a side-effect of the aggressive caching mentioned above. The rationale is that, when many agents need to move in the same direction, the caching helps create one-way flows of agents and hence reduce interactions and contention between groups of agents moving in opposite directions. In contrast, since A* is a stochastic algorithm, this could result in increased contention. This is interesting but largely beyond the scope of this paper.

All three solvers determine paths one at a time, but PPS and PIBT differ from WHCA* in that the former two have a look ahead of a single step, whereas the WHCA* computes paths multi-steps ahead. Due to this, WHCA* is naturally expected to compute better paths than PIBT and PPS. We however observe that PIBT sometimes results in a better makespan than WHCA*, which can be explained by the fairness provided by dynamic prioritization (i.e., more uniformity in path lengths among agents). In PIBT, once an agent a reaches its goal, its priority drops below other agents that have yet to reach their goal, letting them push a away rather than taking a detour. This likely contributes to reducing makespan over WHCA*, in which agents hold their priority. Further, the "Swap" operation in Push and Swap [Luna and Bekris, 2011] solvers, while effective in narrow passages, produces unnecessary steps in open spaces such as arena.

5.2 Multi-agent Pickup and Delivery (MAPD)

Following the experimental setup of the original study [Ma *et al.*, 2017], we used the same undirected graph as testbed, as shown in Table 2. We generated a sequence of 500 tasks by randomly choosing their pickup and delivery locations from all task endpoints. We used 6 different task frequencies, which numbers of tasks are added to the task set Γ: 0.2 (one task every 5 timestep), 0.5, 1, 2, 5 and 10 where the number of agents increases from 10 to 50. TP algorithm [Ma *et al.*, 2017] was also tested for comparison. All experimental settings were performed in 100 instances which initial positions of agents were set randomly. We evaluated following three: 1) makespan which is the timestep when all tasks are completed; 2) service times, i.e., terms from issue to completion of tasks; and 3) runtime.

The results are shown in Table 2. Clearly, PIBT significantly outperforms TP in the view from all three metrics. PIBT is efficient because 1) it ignores the positions of free agents; 2) the computational cost of planning a path is cheap.

6 Conclusion

This paper introduces PIBT, an algorithm for iterative MAPF. PIBT focuses on the adjacent movements of multiple agents, and hence can be applied to many domains. Intuitively, PIBT is well-suited to large environments because such environments result in intensive communication and path efficiency improves with a lower density of agents. Empirical results support this aspect, moreover, PIBT outperforms current solutions to pickup and delivery (MAPD).

Future research include the following: 1) relaxing conditions on the graph environment; 2) finding the effective valuation function of nodes for each domain; 3) adapting the model to asynchronous communication and movements; 4) expanding the time window; and 5) applying PIBT to applications other than MAPF and MAPD.

Acknowledgments

Research partly supported by JSPS KAKENHI Grants No. 17K00019 and 17K12734, and by Japan Science and Technology Agency (JST) SICORP.

References

[Azarm and Schmidt, 1997] Kianoush Azarm and Günther Schmidt. Conflict-free motion of multiple mobile robots based on decentralized motion planning and negotiation. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, volume 4, pages 3526–3533, 1997.

[Barer *et al.*, 2014] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proc. 7th Annual Symp. on Combinatorial Search*, 2014.

[Čáp *et al.*, 2015] Michal Čáp, Jiří Vokřínek, and Alexander Kleiner. Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. In *25th Intl. Conf. on Automated Planning and Scheduling*, 2015.

[Chouhan and Niyogi, 2015] Satyendra Singh Chouhan and Rajdeep Niyogi. Dmapp: A distributed multi-agent path planning algorithm. In *Australasian Joint Conference on Artificial Intelligence*, pages 123–135, 2015.

[Cohen *et al.*, 2016] Liron Cohen, Tansel Uras, TK Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. Improved solvers for bounded-suboptimal multi-agent path finding. In *IJCAI*, pages 3067–3074, 2016.

[de Wilde *et al.*, 2013] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. Push and rotate: cooperative multi-agent path planning. In *AAMAS*, pages 87–94, 2013.

[Dresner and Stone, 2008] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res.*, 31:591–656, 2008.

[Goldenberg *et al.*, 2014] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert Holte, and Jonathan Schaeffer. Enhanced partial expansion A*. *J. Artif. Intell. Res.*, 50:141–187, 2014.

[Khorshid *et al.*, 2011] Mokhtar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Fourth Annual Symp. on Combinatorial Search*, 2011.

[Luna and Bekris, 2011] Ryan Luna and Kostas E Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, pages 294–300, 2011.

[Ma *et al.*, 2017] Hang Ma, Jiaoyang Li, TK Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *AAMAS*, pages 837–845, 2017.

[Ma *et al.*, 2018] Hang Ma, Daniel Harabor, Peter J Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. *arXiv preprint arXiv:1812.06356*, 2018.

[Morris *et al.*, 2016] Robert Morris, Corina S Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, TK Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *AAAI Workshop: Planning for Hybrid Systems*, 2016.

[Sajid *et al.*, 2012] Qandeel Sajid, Ryan Luna, and Kostas E Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *SoCS*, 2012.

[Sha *et al.*, 1990] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[Sharon *et al.*, 2013] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artif.Intell.*, 195:470–495, 2013.

[Sharon *et al.*, 2015] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artif.Intell.*, 219:40–66, 2015.

[Silver, 2005] David Silver. Cooperative pathfinding. *AIIDE*, 1:117–122, 2005.

[Standley, 2010] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, volume 1, pages 28–29, 2010.

[Sturtevant, 2012] Nathan R Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Trans. on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[Surynek, 2009] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 3613–3619, 2009.

[Velagapudi *et al.*, 2010] Prasanna Velagapudi, Katia Sycara, and Paul Scerri. Decentralized prioritized planning in large multirobot teams. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pages 4603–4609, 2010.

[Wagner and Choset, 2015] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artif.Intell.*, 219:1–24, 2015.

[Wang and Botea, 2011] Ko-Hsin Cindy Wang and Adi Botea. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *J. Artif. Intell. Res.*, 42:55–90, 2011.

[Wang *et al.*, 2008] Ko-Hsin Cindy Wang, Adi Botea, et al. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.

[Wiktor *et al.*, 2014] Adam Wiktor, Dexter Scobee, Sean Messenger, and Christopher Clark. Decentralized and complete multi-robot motion planning in confined spaces. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1168–1175, 2014.

[Wurman *et al.*, 2008] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9, 2008.

[Yan *et al.*, 2013] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. A survey and analysis of multi-robot coordination. *Intl. Journal of Advanced Robotic Systems*, 10:399, 2013.

[Yu and LaValle, 2013] Jingjin Yu and Steven M LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*, 2013.

[Zhang *et al.*, 2016] Yu Zhang, Kangjin Kim, and Georgios Fainekos. Discof: Cooperative pathfinding in distributed systems with limited sensing and communication range. In *Distributed Autonomous Robotic Systems*, pages 325–340, 2016.