Polynomial-time Configuration Generator for Connected Unlabeled Multi-Agent Pathfinding

Takahiro Suzuki[1,2], **Keisuke Okumura**[1,3]
[1]National Institute of Advanced Industrial Science and Technology (AIST), Japan
[2]Tohoku University, Japan
[3]University of Cambridge, UK
takahiro.suzuki.q4@dc.tohoku.ac.jp, okumura.k@aist.go.jp

Abstract

We consider Connected Unlabeled Multi-Agent Pathfinding (CUMAPF), a variant of MAPF where the agents must maintain connectivity at all times. This problem is fundamental to swarm robotics applications like self-reconfiguration and marching, where standard MAPF is insufficient as it does not guarantee the required connectivity between agents. While unlabeled MAPF is tractable in optimization, CUMAPF is NP-hard even on highly restricted graph classes. To tackle this challenge, we propose *PULL*, a complete and polynomial-time algorithm with a simple design. It is based on a rule-based one-step function that computes a subsequent configuration that preserves connectivity and advances towards the target configuration. *PULL* is lightweight, and runs in $O(n^2)$ time per step in 2D grid, where n is the number of agents. Our experiments further demonstrate its practical performance: *PULL* finds competitive solution qualities against trivial solutions for hundreds of agents, in randomly generated instances. Furthermore, we develop an eventually optimal solver that integrates *PULL* into an existing search-based MAPF algorithm, providing a valuable tool for small-scale instances.

1 Introduction

Modern robotics is increasingly tackling complex challenges that require tight coordination among multiple agents, where effective collaboration frequently proves essential for task success. This is especially apparent in demanding scenarios such as collaborative work in space or autonomous exploration at disaster sites, where maintaining the integrity of the group becomes a critical factor.

This motivates the study of Connected Unlabeled Multi-agent Pathfinding (CUMAPF): the problem of computing collision-free paths for a team of robots to reach target placements while always maintaining geometric connectivity. The importance of this problem is underscored by its applications in autonomous platooning (Martínez-Díaz et al. 2021), swarm navigation (Shankar, Okumura, and Prorok 2025), and programmable matter (Hinnenthal, Liedtke, and Scheideler 2024; Kostitsyna, Peters, and Speckmann 2023). CUMAPF, according to (Fekete et al. 2023), is NP-hard even on simple 2D grids, indicating its computational intractability.

CUMAPF can be viewed as a variant of the broader research field of *Multi-Agent Pathfinding* (MAPF), which is a general term for the problem of planning paths for multiple agents in a shared environment that do not conflict with each other. MAPF problems can be broadly classified into *labeled*, *unlabeled*, and *colored* settings (Stern et al. 2019), and CUMAPF belongs to the second context. This setting can be applied when all agents can exchange roles or when they are composed of the same functions.

While unlabeled MAPF is polynomial-time solvable in general cases (Yu and LaValle 2013a), adding connectivity constraints makes the problem significantly harder. Moreover, unlike the unlabeled case, various algorithms developed for labeled MAPF (see Table 1) do not work on CUMAPF due to the constraint, since these cannot handle it. This suggests that CUMAPF raises novel challenges unaddressed by existing frameworks, consequently necessitating a new one.

In the context of MAPF, various algorithms have been proposed depending on the application and the characteristics of the solution needed: e.g., optimal algorithms based on heuristic search, and scalable *suboptimal* algorithms. Optimal algorithms, like CBS (Sharon et al. 2015), guarantee minimal solution length; however, their computational cost is prohibitive for large-scale systems. In contrast, suboptimal algorithms do not guarantee optimality, but they provide fast output even for large-scale instances (Li et al. 2022; Okumura et al. 2022; Okumura 2023). Research on large-scale automation has become increasingly important, demanding suboptimal algorithms for practical applications. Table 1 shows several related works of MAPF and its variants. Although there are suboptimal algorithms for large-scale labeled and unlabeled MAPF, their counterpart for CUMAPF are missing.

In this study, we utilize this suboptimal algorithmic approach for CUMAPF and propose a rule-based algorithm *PULL*. *PULL* is a polynomial-time *configuration generator*; it efficiently computes the next placement of the agents from the current ones and the target. Moreover, it is computationally lightweight, and outputs empirically efficient solutions compared to those obtained by a trivial approach (e.g., 350% improvement for 500 agents on a 32×32 grid).

Since *PULL* employs a suboptimal rule-based approach, there are adversarial instances that generate solutions significantly deviating from the optimal. To address this challenge, we develop an algorithm that integrates *PULL* with LaCAM* (Okumura 2023), an existing search scheme which leads to large-scale, real-time MAPF studies. The algorithm iteratively refines the solution with heuristic search, eventually converging to an optimal one.

	optimal algorithm	suboptimal algorithm	theoretical work
Labeled MAPF	M* (Wagner and Choset 2015), CBS (Sharon et al. 2015) ...	PIBT (Okumura et al. 2022), LNS2 (Li et al. 2022) ...	Poly-time for existence (Röger and Helmert 2012), NP-hard for optimization (Yu and LaValle 2013b)
Unlabeled MAPF	Reduction to MAX-FLOW (Yu and LaValle 2013a)	TSWAP (Okumura and Défago 2023)	Poly-time for optimization (Yu and LaValle 2013a)
CUMAPF	This work + LaCAM* (eventually)	**This work**	NP-hard for optimization (Fekete et al. 2023)

Table 1: Categorization of MAPF variants and their solutions.

Contribution. We propose *PULL*, a suboptimal, complete algorithm for CUMAPF that runs in $O(\Delta^2 n^2)$ time, where Δ and n are the maximum degree of a graph and the number of agents, respectively. Our results indicate that CUMAPF always has a solution. We demonstrate that *PULL* produces effective solutions on instances generated using MAPF benchmarks commonly employed by the research community, especially when compared to a vanilla method. Additionally, we showcase its usefulness by integrating the search scheme LaCAM*. In what follows, we present problem formulation, related work, the algorithm, and evaluation in order. The code is available on https://github.com/su3taka/CUMAPF-PULL.

2 Preliminaries

Notation. Let $G = (V, E)$ be a simple, undirected, and finite connected graph. For a vertex $v \in V$, the *open neighborhood* is denoted by $N(v) = \{w \in V \mid vw \in E\}$, and its *degree* is $d(v) = |N(v)|$. The maximum degree of G is $\Delta = \max_{v \in V} d(v)$. For a vertex set $V' \subseteq V$, their neighborhoods are $N(V') = (\bigcup_{v \in V'} N(v)) \setminus V'$. For a vertex subset $S \subseteq V$, we denote the subgraph induced by S as $G[S]$. A graph is *connected* if there is a path between any two distinct vertices. A *component* of G is a connected subgraph that is not part of any larger connected subgraph.

Let $\mathsf{Reach}(G, v) \subseteq V$ be the set of all vertices reachable from v in G. This set can be computed in linear time by using breadth-first search (BFS), simultaneously computing their shortest path from $u \in \mathsf{Reach}(G, u)$ to v and its length $\mathsf{dist}(u, v)$. A vertex $v \in V$ is a *cut vertex* if $G[V \setminus \{v\}]$ has more components than G. The set of all cut vertices of G is denoted by $\mathsf{Cut}(G)$. This set can be computed in linear time by using depth-first search (DFS).

Problem Definition. Let $G = (V, E)$ be a graph and $A = \{1, 2, \dots n\}$ be a set of n agents. A *configuration* $\mathcal{Q} = (q_1, \dots, q_n) \in V^n$ is an assignment of each agent to a vertex. Although the configuration $\mathcal{Q}$ is defined as a list of vertices, we may also refer to it as a set of vertices for convenience. For the sake of simplicity, let $\mathcal{Q}(R)$ be $\{\mathcal{Q}[i] \mid i \in R\}$ for a set of agents $R \subseteq A$. A configuration $\mathcal{Q}$ is *connected* if the induced subgraph $G[\mathcal{Q}]$ is connected.

Given two sets of vertices $S, T \subset V$ with $|S| = |T| = n$, the goal of CUMAPF is to find a finite sequence of configurations (a *plan*) $\Pi = [\mathcal{Q}_0 = S, \mathcal{Q}_1, \dots, \mathcal{Q}_{t^*} = T]$. Throughout the plan, agents can only move to adjacent (or the same) vertices, and must avoid either vertex or swap conflicts,[1] while maintaining the connectivity of the entire configuration. Formally, a *valid* plan Π satisfies the following conditions:

1. $\mathcal{Q}_0 = S \wedge \mathcal{Q}_{t^*} = T$.
2. $\forall i \in A, \forall k \in \{1, \dots, t^*\} : \mathcal{Q}_k[i] \in N(\mathcal{Q}_{k-1}[i]) \cup \{\mathcal{Q}_{k-1}[i]\}$. (*reachability*)
3. $\forall k \in \{0, \dots, t^*\}, \forall i, j \in A, i \neq j : \mathcal{Q}_k[i] \neq \mathcal{Q}_k[j]$. (avoidance of *vertex conflict*)
4. $\forall k \in \{1, \dots, t^*\}, \forall i, j \in A, i \neq j : (\mathcal{Q}_k[i], \mathcal{Q}_k[j]) \neq (\mathcal{Q}_{k-1}[j], \mathcal{Q}_{k-1}[i])$. (avoidance of *swap conflict*)
5. $\forall k \in \{0, \dots, t^*\} : \mathcal{Q}_k$ is connected.

We aim to find a plan Π that minimizes the makespan t^*.

LaCAM* (Okumura 2023) is an eventually optimal MAPF solver that utilizes DFS over configuration space with some existing MAPF algorithms as a configuration generator. LaCAM* starts a DFS operation with a configuration $\mathcal{Q}_0$, i.e. generating a reachable successor $\mathcal{Q}_1$ from $\mathcal{Q}_0$, generating $\mathcal{Q}_2$ from $\mathcal{Q}_1$, to the target configuraiton. Unlike the normal DFS operation, LaCAM* allows to revisit a configuration: if $\mathcal{Q}_k$ is revisited from $\mathcal{Q}_{k'}$, LaCAM* introduces a constraint C for a subset of agents during successor generation, such that $\mathcal{Q}_k$ must produce a successor that is distinct from $\mathcal{Q}_{k+1}$. Constraints C are applied to configurations by specifying both which agents are constrained and their respective movements. Eventually, by enumerating all such constraints, the algorithm explores all possible neighboring configurations. Thus, with search tree rewiring, the traversal explores the entire configuration space, implying completeness and optimality.

3 Related Work

Unlabeled MAPF (a.k.a. *Anonymous MAPF*). Unlike the labeled setting, the unlabeled variant introduces the additional subproblem of assigning agents to target locations. Since the optimal goal assignment depends on the resulting path, these two subproblems –assignment and pathfinding– are tightly coupled and solved simultaneously. The seminal work (Yu and LaValle 2013a) showed that optimal solutions can be found in polynomial time via a reduction to the maximum-flow problem. However, this flow-based approach often suffers from high computational costs on large instances in practice, limiting its scalability. These scalability

[1] Swap conflict is omitted in common unlabeled MAPF since it can be resolved by postprocessing. However, we add this term since *PULL* does not require postprocessing.

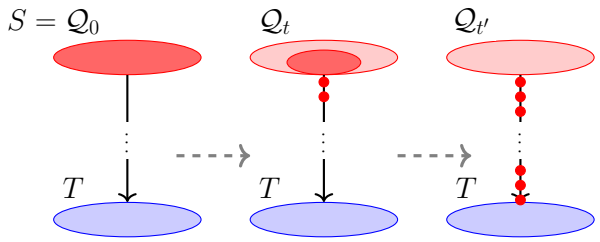

Figure 1: The naive approach for solving CUMAPF. Light red represents $S = \mathcal{Q}_0$, deep red represents the current configuration, and blue represents T. (*Left*) Initial configuration, (*Center*) intermediate configuration at step $t < t'$, (*Right*) configuration at step $t' = \mathsf{dist}(x, y)$.

challenges have motivated the development of fast suboptimal algorithms (Okumura and Défago 2023).

MAPF with additional constraint. MAPF typically considers vertex or edge conflicts; however, practical applications often require handling additional constraints due to communication or other limitations; for instance, (Li et al. 2020) aims for agents to move while maintaining a specific formation, and (Křišťan and Svoboda 2025) conducts theoretical studies on combinatorial constraints on graphs, e.g., dominating set and independent set. Tateo et al. (2018) show the PSPACE-hardness in the case where the edge sets representing the physical and communication features are given separately.

CUMAPF. Fekete et al. (2023) show CUMAPF is NP-hard, even determining whether the optimal makespan is two or more, in 2D empty grid graphs. This work also shows that $3/2$-approximation of makespan is also NP-hard. In contrast to the NP-hardness, there is a polynomial-time, constant-factor approximation algorithm for makespan in *empty* 2D grid environments; however, its practical applicability is severely limited. It requires a significant theoretical step and is exclusively designed for obstacle-free 2D grid environments. It points out the limits of versatility in complex scenarios, motivating this study for practical approaches.

4 Algorithm

This section proposes *PULL*, a suboptimal CUMAPF algorithm. The algorithm is designed as a configuration generator that takes the current agent configuration $\mathcal{Q}^{\text{from}}$ and the target vertex set T as input to compute the configuration for the next time step. By iteratively applying this function from S, the algorithm is guaranteed to converge to T in a finite number of steps. The pseudocode for the algorithm is provided in Algorithm 1. The following subsections supplement the pseudocode by explaining its main structure and core mechanisms. Herein, $\mathsf{next}(a, b)$ denotes the parent vertex of $a \in V$ on the BFS-tree rooted by $b \in V$. Cleary, we have $\mathsf{dist}(\mathsf{next}(a, b), b) = \mathsf{dist}(a, b) - 1$ when $a \neq b$.

4.1 Concept

We first introduce a simple way for configuration generation to CUMAPF, that is, a minimal method to obtain a better configuration $\mathcal{Q}^{\text{to}}$ than $\mathcal{Q}^{\text{from}}$. We extend this algorithm in Section 4.2. Consider the input where S and T satisfies that $S \cap T = \emptyset$ for convinience (Figure 1 left). Let $x \in S$ and $y \in T$ be a pair of vertices such that $(x, y) = \arg\min_{(s,t)\in S\times T} \mathsf{dist}(s, t)$. If the agent currently on x can move to $\mathsf{next}(x, y)$ without breaking connectivity, then we can guarantee that there is at least one agent that moves towards the goal.

If $S \setminus \{x\} \cup \{\mathsf{next}(x, y)\}$ is not connected, then we have to assign another agent on $u \in N(x)$ to x. This operation might generate another disconnected configuration; then we can repeat the same process until the configuration becomes connected. This recursive procedure allows the assigned agent to form a single path. It always terminates when it reaches a vertex that does not "cut" the configuration, since any leaf of a DFS-tree of a graph can not be a cut vertex. We can ensure that the agent closest to its goal in $\mathcal{Q}^{\text{from}}$ moves toward T, and we obtain a configuration that is "closer" to the goals than $\mathcal{Q}^{\text{from}}$.

By iteratively applying this process, the agent moves a distance of one per step on the shortest (x, y)-path towards y. We can see that there is a step t such that exactly one agent reaches y (Figure 1 right). By appropriately handling the case $S \cap T \neq \emptyset$, which will be described in Section 4.2, we can obtain a simple yet complete algorithm.

Despite the existence of this simple solution, it provides a highly suboptimal solution: a plan with a huge makespan. This is because the algorithm processes one movement of agents at each step. For example, it only executes the leftmost column in Figure 2, resulting in the configuration shown in the upper part of the second column. One can observe that we can yield the configuration that is closer to the goal by moving other unconstrained agents; *PULL* accomplishes this.

4.2 Description

The previous algorithm found a single path in the induced subgraph $G[\mathcal{Q}^{\text{from}}]$ when generating the next configuration. However, there may be any number of paths, as long as they are pairwise disjoint and the resulting configuration $\mathcal{Q}^{\text{to}}$ is connected. Here, we describe the way to compute such multiple vertex-disjoint paths iteratively.

The core of *PULL* lies in the procedure `PULL` (t, R, V') for $t \in N(\mathcal{Q}^{\text{from}})$, $R \subseteq A$, $V' \subseteq \mathcal{Q}^{\text{from}}$ (see Algorithm 1). This function computes a path on $G[\mathcal{Q}^{\text{from}}]$ to "pull" an agent toward a specified vertex t, avoiding the agents in the set R. Furthermore, the path never starts from any vertex in V'. The most critical aspect of this process is performing the assignment while preserving connectivity. Figure 2 provides an example of the whole process to output $\mathcal{Q}^{\text{to}}$ from $\mathcal{Q}^{\text{from}}$.

Enumerating available vertices. To achieve this, `PULL` performs two operations. First, it identifies the set of vertices of the graph $G[\mathcal{Q}^{\text{to}} \cup \{t\}]$ which can be reached at t without crossing the agent $i \in R$ (line 15). Thus, the BFS operation must not look at any vertex v in the vertex set $\mathcal{Q}^{\text{to}}(R)$. Since the assignment of agents forms a single path, this BFS operation characterizes the set of vertices that satisfies a necessary condition to be an initial vertex of the path (denoted by F).

Algorithm 1: *PULL*

Input: configuration $\mathcal{Q}^{\text{from}}$, agents A, goals T
Output: configuration $\mathcal{Q}^{\text{to}}$ ▷ *initialized with* $\mathcal{Q}^{from}$

1: $R \leftarrow \emptyset$
2: **if** $\mathcal{Q}^{\text{from}} \cap T \neq \emptyset$ **then**
3: $\quad \mathcal{P} \leftarrow$ Components of $(G[\mathcal{Q}^{\text{from}} \cap T])$
4: $\quad$ Sort $P_i \in \mathcal{P}$ in descending order of $|P_i|$
5: $\quad$ **for** $P_i \in \mathcal{P}$ **do**
6: $\quad\quad$ **for** $v \in (N(P_i) \cap T)$ **do**
7: $\quad\quad\quad$ **if** v is unoccupied **then** $R \leftarrow$ PULL (v, R, P_i)
8: $\quad\quad$ **for** $v \in P_i$ **do**
9: $\quad\quad\quad$ **if** $\exists i$ s.t. $\mathcal{Q}^{\text{to}}[i] = \mathcal{Q}^{\text{from}}[i]$ **then** $R \leftarrow R \cup \{i\}$
10: Sort $N(\mathcal{Q}^{\text{from}})$ by $\min_{t \in T}(\mathsf{dist}(u, t))$ in ascending order
11: **for** $u \in N(\mathcal{Q}^{\text{from}})$ **do**
12: $\quad$ **if** u is unoccupied **then** $R \leftarrow$ PULL $(u, R, \emptyset)$
13: **return** $\mathcal{Q}^{\text{to}}$
14: **function** PULL(t, R, V')
15: $\quad F \leftarrow \mathsf{Reach}(G[\mathcal{Q}^{\text{to}} \cup \{t\} \setminus \mathcal{Q}^{\text{to}}(R)], t)$
16: $\quad B \leftarrow \mathsf{Cut}(G[\mathcal{Q}^{\text{to}} \cup \{t\}])$
17: $\quad$ **if** $V_S \leftarrow (F \setminus B) \setminus (V' \cup \{t\}) = \emptyset$ **then return** R
18: $\quad \mathsf{cur} \leftarrow \arg\max_{v \in V_S}(\min_{y \in T}(\mathsf{dist}(v, y)))$
19: $\quad$ **while** $\mathsf{cur} \neq t$ **do**
20: $\quad\quad i \leftarrow$ agents s.t. $\mathcal{Q}^{\text{from}}[i] = \mathsf{cur}$
21: $\quad\quad Q^{to}[i] \leftarrow \mathsf{next}(\mathsf{cur}, t)$ ▷ *assignment*
22: $\quad\quad R \leftarrow R \cup \{i\}$, $\mathsf{cur} \leftarrow Q^{to}[i]$
23: $\quad$ **return** R

Second, we compute the set of cut vertices of $G[\mathcal{Q}^{\text{to}} \cup \{t\}]$ (line 16). If a vertex $v \in \mathcal{Q}^{\text{to}}$ is a cut vertex, then the agent on v is forced to stay on v to maintain the connectivity. Thus, this operation characterizes the set of vertices that must not be chosen as the initial vertex of the path (denoted by B).

By the above observations, $F \setminus B$ represents the set of vertices that can be chosen as an initial vertex. Line 17 determines whether the procedure PULL succeeds or not. If the condition holds, the procedure PULL(t, R, V') fails to assign the agent, since there is no appropriate vertex. Then procedure PULL returns the same configuration and set of agents. If not, it is possible to form a new path that does not overlap with the existing path, while preserving connectivity.

Selection. Next, the procedure selects the initial vertex of the path. From the set of vertices V_S, the algorithm selects the one that is currently farthest from any target vertex, that is, cur holds $\min_{y \in T}(\mathsf{dist}(v, y)) \leq \min_{y \in T}(\mathsf{dist}(\mathsf{cur}, y))$ for every $v \in V_S$ (line 18). A chain of assignments is then executed. For an agent i such that $\mathcal{Q}^{\text{from}}[i] = \mathsf{cur}$, PULL assigns $\mathcal{Q}^{\text{to}}[i]$ to $\mathsf{next}(\mathsf{cur}, t)$, which is already computed in line 15. Then agent i is added to set R, and the procedure moves to the next vertex $\mathsf{next}(\mathsf{cur}, t) = \mathcal{Q}^{\text{to}}[i]$. After performing assignments iteratively, PULL generates the next configuration and returns a set R of agents already assigned the next position.

Top level procedure. The algorithm *PULL* repeatedly applies PULL and updates $\mathcal{Q}^{\text{to}}$ sequentially. We can observe that *PULL* can generate the configuration that approaches T, if $\mathcal{Q}^{\text{from}} \cap T = \emptyset$. However, it is important to consider whether applying *PULL* results in convergence to T or not.

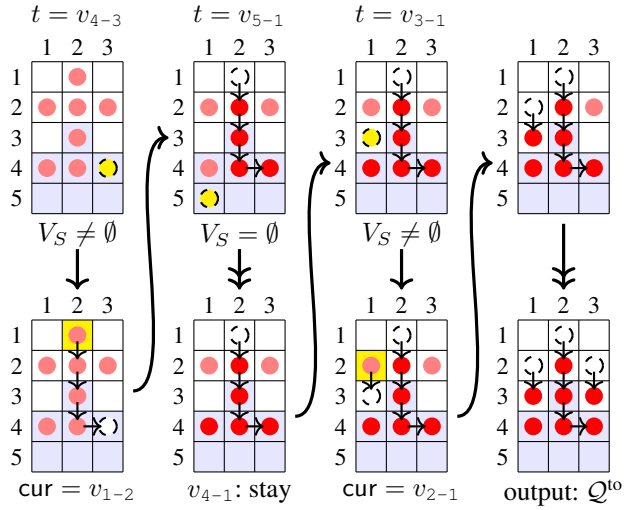

Figure 2: An example of the *PULL*'s operation. The graph G is 5×3 grid, and $v_{\text{i-j}}$ denotes the vertex with row i, column j. Light red indicates $\mathcal{Q}^{\text{from}}$, and blue represents T. (*Column 1*) first *PULL* call PULL$(v_{4-3}, \emptyset, \{v_{3-2}, v_{4-1}, v_{4-2}\})$ (v_{4-3} is marked by yellow). Then $V_S = \{v_{1-2}, v_{2-1}, v_{2-2}, v_{2-3}\}$, and cur $= v_{1-2}$. PULL returns $R = \{0, 2, 4, 6\}$. (*Column 2*) *PULL* call PULL$(v_{5-1}, R, \{v_{3-2}, v_{4-1}, v_{4-2}\})$. Then $V_S = \emptyset$, and PULL return R. The call PULL with $t = v_{5-2}$ also fails. We set $R = \{0, 2, 4, 5, 6\}$. (*Column 3*) *PULL* calls PULL$(v_{3-1}, R, \emptyset)$. $V_S = \{v_{2-1}\}$, and PULL returns $R = \{0, 1, 2, 4, 5, 6\}$. (*Column 4*) Repeatedly calling PULL, finally *PULL* returns $\mathcal{Q}^{\text{to}} = [v_{2-2}, v_{3-1}, v_{3-2}, v_{3-3}, v_{4-2}, v_{4-1}, v_{4-3}]$.

If $\mathcal{Q}^{\text{from}} \cap T \neq \emptyset$, we first computes the list of components of $G[\mathcal{Q}^{\text{from}} \cap T]$ (denoted by $\mathcal{P}$). Note that even if $\mathcal{Q}^{\text{from}}$ is connected, $G[\mathcal{Q}^{\text{from}} \cap T]$ can consist of multiple components, as illustrated in Figure 3. We next sort the elements of $\mathcal{P}$ by their size of vertex set, and call PULL in decreasing order. We now observe that, if $\mathcal{Q}^{\text{from}} \cap T \neq \emptyset$, *PULL* first calls procedure PULL$(t, *, *)$ with $t \in N(P_0)$. Thus, $\mathcal{Q}^{\text{to}}$ holds that $G[\mathcal{Q}^{\text{to}} \cap T]$ has the larger component than $G[\mathcal{Q}^{\text{from}} \cap T]$ (see Figure 3). Thus, the size of the largest component becomes larger step by step. Combining this method and assignments at line 9, we can prove the convergence.

4.3 Theoretical analysis

Let $R \subseteq A$ be the set of agents already been assigned their next position. We abuse cur_k, t_k, F_k, $\mathcal{Q}^{\text{to}}_k$, B_k and R_k, where k is the number of calls to the PULL within a single execution of algorithm 1 for convinience. Due to the space limitation, some proofs (denoted by $*$) are omitted in the manuscript.

Lemma 1 ($*$)**.** *If $\mathcal{Q}^{from}$ is connected, then $\mathcal{Q}^{to}$ is connected, collision-free, and reachable.*

Proof. Proofs that $\mathcal{Q}^{\text{to}}$ is conflict-free and reachable are provided in the Appendix. Here, we present the proof for connectivity. We prove the lemma by induction on k, the number

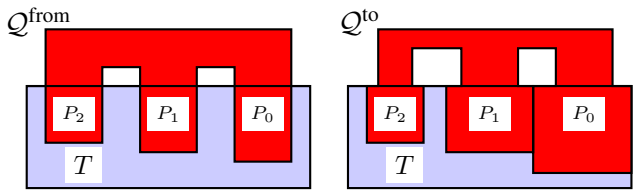

Figure 3: A conceptual illustration of line 4. In the initial iteration, *PULL* calls PULL on $N(P_0)$, which increases the size of the largest component. (**Left**) initial configuration, (**Right**) itermediate configuration before executing line 10. Two connected components might merge (e.g., P_0 and P_1 in the figure), and these are then computed as a larger component in the next step.

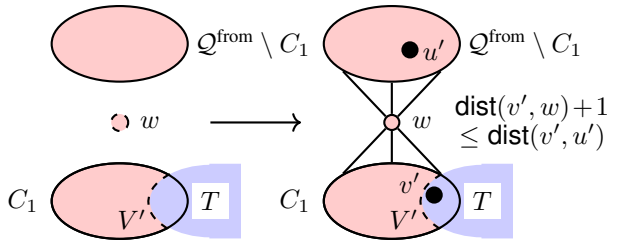

Figure 4: A brief explanation of the proof in Lemma 2. (*Left*) Graph $G[\mathcal{Q}^{\text{from}} \cup \{t_1\} \setminus \{w\}]$ is divided into multiple components, shown as two for convenience. (*Right*) A graph $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$ has a single component, and every (v', u')-paths must pass through w since w is a cut vertex.

of calls to the PULL. For the base case, we have $\mathcal{Q}^{\text{to}}_0 = \mathcal{Q}^{\text{from}}$, which is connected and collision-free by the assumption.

Assume that $\mathcal{Q}^{\text{to}}_k$ is connected and collision-free. When $V_S = \emptyset$ at line 17, then $\mathcal{Q}^{\text{to}}_k = \mathcal{Q}^{\text{to}}_{k+1}$ since $R_k = R_{k+1}$. Then $\mathcal{Q}^{\text{to}}_{k+1}$ is also connected. Consider the case when $(F_k \setminus B_k) \setminus (V' \cup \{t\}) \neq \emptyset$. We can see that the set of occupied vertices after the move is $\mathcal{Q}^{\text{to}}_{k+1} = (\mathcal{Q}^{\text{to}}_k \setminus \{\mathsf{cur}\}) \cup \{t\}$. The procedure PULL selects cur from a set of vertices V_S that excludes any cut vertices of the graph $G[\mathcal{Q}^{\text{to}}_k \cup \{t\}]$. Since removing a non-cut vertex from a connected graph preserves its connectivity, the graph $G[\mathcal{Q}^{\text{to}}_{k+1}]$ is also connected. Thus, we can prove that $G[\mathcal{Q}^{\text{to}}]$ is connected by induction. □

Next, we show that $\mathcal{Q}^{\text{to}} \neq \mathcal{Q}^{\text{from}}$, that is, at least one call succeeds when $V' = \emptyset$ or $G[V']$ is connected.

Lemma 2. *Let t_1 be the vertex that* PULL $(t_1, \emptyset, V')$ *is called in step τ. There is an agent i such that $\mathcal{Q}^{to}[i] = t_1$ in line 21.*

Proof. To prove the lemma, we must show that this set V_S is always non-empty when $R = \emptyset$. This is equivalent to showing that there is at least one vertex $v \notin B_1$ in the set of $(\mathcal{Q}^{\text{from}} \cup \{t_1\}) \setminus (V' \cup \{t_1\}) = \mathcal{Q}^{\text{from}} \setminus V'$.

First, we show the case when $V' = \emptyset$. Any connected graph with two or more vertices has at least one vertex that is not a cut vertex. Thus $|F \setminus B| \geq 2$ holds for the graph $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$. Since $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$ is connected, t_1 cannot be the cut vertex of $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$. Thus $|V_S| \geq 2 - 1 = 1$ holds when $V' = \emptyset$.

We move to the case where $V' \neq \emptyset$ and $G[V']$ is connected. Note that $t_1 \in N(V')$ when *PULL* calls the procedure, thus $G[V' \cup \{t_1\}]$ is connected. We assume that every vertex in $\mathcal{Q}^{\text{from}} \setminus V'$ is a cut vertex of $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$ for a contradiction. Let w be a vertex that holds $w = \arg\max(\min_{y \in T \cup \{t_1\}} \mathsf{dist}(u, y))$. From the assumption, w is a cut vertex of $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$. Thus, $G' = G[\mathcal{Q}^{\text{from}} \cup \{t_1\} \setminus \{w\}]$ consists of at least two components, and all vertices in $V' \cup \{t_1\}$ are contained in the same component.

Let C_1 be a vertex set that $V' \cup \{t_1\} \subseteq C_1$ and $G[C_1]$ is a component of G', and u be a vertex $u' \in V \setminus C_1$ (Figure 4 left). Since w is a cut vertex and $v' \in V'$ and u' is in different component in G' for every vertex $v' \in V'$, every (v', u')-path must pass through w. The same applies to shortest paths. Thus, u' holds that $\min_{y \in T \cup \{t_1\}}(\mathsf{dist}(u', y)) \geq \min_{y \in T \cup \{t_1\}}(\mathsf{dist}(w, y)) + 1$ (Figure 4 right). This leads a contradiction to the assumption that $w = \arg\max(\min_{y \in T \cup \{t_1\}} \mathsf{dist}(u, y))$. Therefore there is at least one vertex in $(F_1 \setminus B_1) \setminus (V' \cup \{t_1\})$.

Then the call PULL $(t_1, \emptyset, V')$ invokes line 21. Thus, there is an assignment to t_1, completing the proof. □

From Lemma 2, we can show the following lemmas, which are necessary to guarantee the completeness of our algorithm.

Lemma 3 (∗). *If $\mathcal{Q}^{from} \cap T = \emptyset$, then*

$$\min_{(s,t) \in \mathcal{Q}^{to} \times T} \mathit{dist}(s, t) = \min_{(s,t) \in \mathcal{Q}^{from} \times T} \mathit{dist}(s, t) - 1$$

Then there exists a step τ' such that $\mathcal{Q}_\tau \cap T \neq \emptyset$. Thus, the condition at line 3 is satisfied at some step. From here, let $p^{\max}_\tau$ be the size of the largest component of $G[\mathcal{Q}_\tau \cap T]$.

Lemma 4 (∗). *If $\mathcal{Q}_\tau \cap T \neq \emptyset$ at step τ, $p^{max}_{\tau+1} \geq p^{max}_\tau + 1$.*

Then there is a step $\tau^\star$, where $p^{max}_{\tau^\star} = n$. At step τ^*, it holds that $\mathcal{Q}^{\text{from}}_{\tau^\star} = T$, which implies that a plan $[\mathcal{Q}_0, \mathcal{Q}_1, ..., \mathcal{Q}_{\tau^\star}]$ is valid. Furthermore, $\tau^\star$ has an upper bound: $\tau' + (n-1)$. This immediately leads to the following. Here, $\mathsf{diam}(G)$ is defined by $\max_{(s,t) \in V \times V} \mathsf{dist}(s, t)$.

Theorem 5. *PULL is complete for* CUMAPF.

Proposition 6. CUMAPF *has the upper bound of makespan:* $\mathit{diam}(G) + n - 1$.

For the running time of *PULL*, we have the following:

Proposition 7 (∗). *PULL runs in $O(\Delta^2 n^2)$ time.*

It is worth noting that evaluation of $\min_{t \in T} \mathsf{dist}(u, t)$ for all vertices $u \in V$ can be done in $O(|V| + |E|) = O(\Delta|V|)$ time, by proper preprocessing. Thus, combining Proposition 6 and 7, we have the following corollary.

Corollary 8. *PULL solves* CUMAPF *in $O(|V| \cdot \Delta^2 n^2)$ time.*

Tightness, Adversarial instances. The makespan upper bound established is tight; an example is shown in Figure 5(a). Furthermore, we show that for some instances, *PULL* outputs a plan with a makespan of $O(|A|)$, even when the optimal makespan is two (see Figure 5(b)). A detailed description is provided in the Appendix.

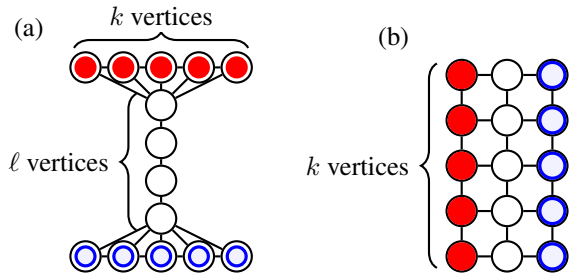

Figure 5: (a) An example of a tight instance: the optimal makespan is $\mathsf{diam}(G) + |A| - 1$. (b) An example of a adversarial instance: *PULL* outputs plan with $O(|A|)$ makespan, while the optimal makespan is two.

5 Combining *PULL* and LaCAM*

As Figure 5(b) illustrates, this algorithm performs poorly when *vertex connectivity* (the minimum number of vertices whose removal disconnects the graph) of $G[S]$ is small. It is challenging to address this by improving the method used in *PULL*. This is because *PULL* employs a method that gradually modifies the configuration within a single step while preserving connectivity; however, as seen in the instance in Figure 5(b), proceeding in that manner allows at most two agents to move forward for any k. Therefore, we propose a search-based method that utilizes LaCAM* (Okumura 2023) towards an eventually optimal algorithm.

LaCAM* was originally developed for labeled MAPF, using PIBT (Okumura et al. 2022) as a configuration generator, but it is easily attainable CUMAPF counterpart with *PULL*. Configuration generator for LaCAM* needs to output the next configuration $\mathcal{Q}^{\text{to}}$, taking the current configuration $\mathcal{Q}^{\text{from}}$, target T, and constraints C as input. Here, the constraints C include agents with predetermined destinations for the next step, along with their respective destinations. Since *PULL* cannot handle the constraints, we modified it slightly to accommodate them (see Algorithm 2). Note that these constraints may invalidate the lemmas in Section 4.3, potentially preventing the attainment of a conflict-free configuration. The adaptation for completeness is described as follows.

A key change is in line 5, which uses the list pri instead of the distance to T. If there is a pair of agents i, j that $i \notin R \land j \in R \land \mathcal{Q}^{\text{from}}[i] = \mathcal{Q}^{\text{to}}[j]$, we should move agent i to avoid the vertex conflict. Then the Algorithm 2 sets the priority of vertex $\mathcal{Q}^{\text{from}}[i]$ high, so that vertex $\mathcal{Q}^{\text{from}}[i]$ tends to be chosen as the initial vertex of the path. If $\mathcal{Q}^{\text{from}}[i]$ is chosen, agent i moves to another vertex, which avoids the vertex-conflict between i and j. If there is are any conflicts in the resulting configuration, Algorithm 2 returns $\bot$.

If $\mathcal{Q}^{\text{to}}$ is not a valid configuration: i.e. $\mathcal{Q}^{\text{to}}$ is not connected, or there is any conflict, Algorithm 2 returns $\bot$, meaning it cannot generate the configuration in lines 7 and 8. Note that the search process ensures all generated configurations are valid. Specifically: (1) if $\mathcal{Q}^{\text{to}}$ is not connected, it is rejected in line 7; (2) any configuration with a vertex conflict is pruned by the search in line 8; and (3) any swap conflicts are resolved in line 10. Consequently, every resulting configuration is guaranteed to be connected, collision-free, and reachable. This leads to the following, which is a direct consequence of (Okumura 2023).

Algorithm 2: *PULL* for LaCAM*

Input: $\mathcal{Q}^{\text{from}}$, A, T, and constraints C ▷ *C is given by set of tuples $(i \in A, v \in N[\mathcal{Q}^{from}[i]])$*
Output: configuration $\mathcal{Q}^{to}$, or $\bot$ ▷ *initialized by $\bot^n$*

```
1: R ← ∅, pri[v] ← min_{t∈T} dist(v, t) for v ∈ Q^from
2: for (i, v) ∈ C do
3:     Q^to[i] = v, R ← R ∪ {i}
4: if ∃i, j ∈ A s.t. i ∉ R ∧ j ∈ R ∧ Q^from[i] = Q^to[j] then
5:     pri[Q^from[i]] ← |V|
6: perform PULL, with modified PULL
7: if Q^to is not connected then return ⊥
8: if there is a vertex conflict then return ⊥
9: if there is a swap conflict between i, j ∈ A then
10:    Q^to[i], Q^to[j] ← Q^to[j], Q^to[i]
11: return Q^to
12: function PULL(t, R, V')
13:    perform lines 15 and 16 of Algorithm 1
14:    if V_S ← (F \ B) \ (V' ∪ {t}) = ∅ then return R
15:    cur ← arg max_{v∈V_S} (pri[v])
16:    Assign agents on v ∈ V in (cur, t)-path to next(v, t)
17:    return R
```

Proposition 9. *The algorithm, which uses Algorithm 2 as a subroutine of LaCAM*, is complete and eventually optimal; i.e., the output eventually converges to optima.*

6 Empirical analysis

This section presents an empirical evaluation of *PULL*. Specifically, we conduct the following two: *(i)* An evaluation of the running time and solution quality (i.e. makespan) of *PULL* against a simple solution in Section 4.1 on a variety of benchmark maps. *(ii)* A validation of the makespan-optimality of our search-based solver on adversarial instances. Note that *PULL* is the first suboptimal algorithm that addresses CUMAPF – no prior methods available. Thus, we used an intuitive method (Section 4.1) as a baseline to provide a reference for solution quality. The algorithm is coded in Python, and the experiments were run on a Mini PC with Intel Core i9-13900H 2.6GHz CPU, and 32GB RAM.

6.1 Performance of *PULL*

This section empirically evaluates the *PULL* algorithm on two key metrics: computation time and makespan. For our evaluation, we use the *random-32-32-20*, *random-64-64-20*, and *warehouse-10-20-10-2-2* in MAPF benchmarks (Stern et al. 2019), and randomly generated map *random-48-48-20*. We prepare instances with a varying number of agents from 100 to 1,000, in increments of 100, by generating two random connected subgraphs (means S and T).

As finding the optimal plan is computationally hard for these instances, we benchmark our solution quality against a trivial lower bound from the value of bottleneck matching (denoted by LB). This represents the length of the plan when

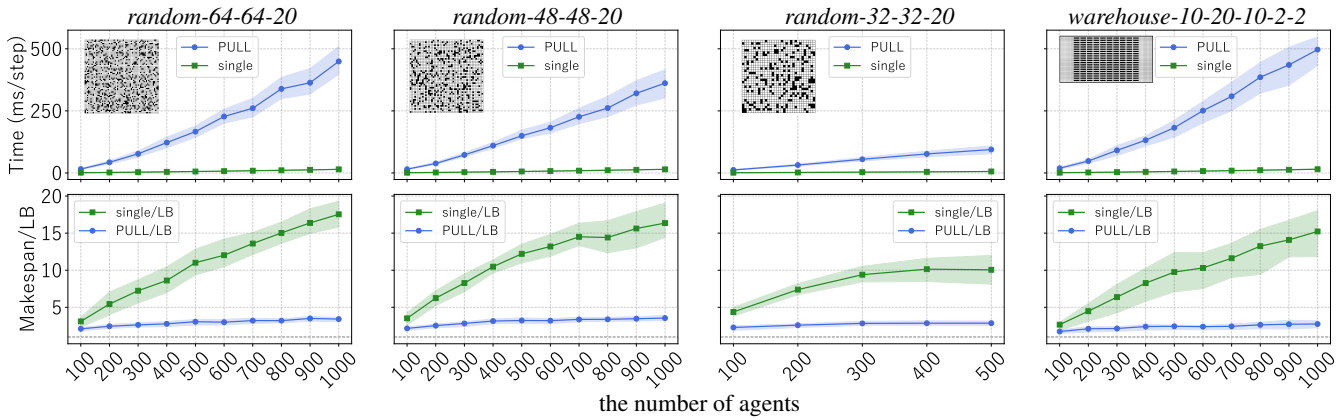

Figure 6: Experimental results for time and makespan/LB across four maps. Blue represents Algorithm 1, and green represents the algorithm in Section 4.1. Lines plot the average values, and the filled areas indicate the interquartile range.

each agent moves toward its goal, disregarding conflicts or connectivity. We also used the simple solution described in the Section 4.1 (denoted as single) as a comparison target for *PULL*.

Figure 6 displays the experimental results for the time and makespan/LB for the two algorithms *PULL* and single, based on 100 instances per setting. The main observation is as follows; *(i)* For each map, the average computational time of *PULL* increases with the number of agents; however, its growth is more gradual than $O(\Delta^2 n^2) = O(n^2)$. *(ii)* The average makespan/LB increases for both algorithms as the density (i.e., $n/|V|$) increases; however, *PULL* exhibits a considerably more suppressed increase. Specifically, across all maps, when $n = 500$, the average makespan/LB of *PULL* is 0.3 times or less than that of the simple solution. Moreover, in sparse conditions, *PULL* consistently outputs solutions closer to the lower bound than single.

The first observation implies that the actual number of calls of `PULL` is $o(\Delta n)$ in some practical situations. This suggests the algorithm is scalable to even larger instances, enabling it to find solutions within a practical time. Regarding the second point, we can conclude that *PULL* produces solutions that are comparatively close to the lower bound when compared to the simple solution, leading to more efficient operations in real-world scenarios.

We conducted an additional experiment to demonstrate that the computational time is largely independent of the number of vertices $|V|$. Theoretically, the execution time per one step depends solely on n in MAPF benchmark, while the map size is only involved in distance calculations during preprocessing. Therefore, the average time is expected to be mostly the same when n is fixed. Figure 7 presents the running time and makespan/LB for a fixed $n = 100$, on a set of random maps of increasing size (e.g., from 16x16 to 64x64, incrementing width by 8). Note that all of the maps (except *random-32-32-20* and *random-64-64-20*) are randomly generated with 20% obstacles (see Appendix). We observe that $|V|$ has a small impact on *PULL*'s running time. We can also see that makespan/LB decreases as $n/|V|$ gets smaller, consistent with the previous observation.

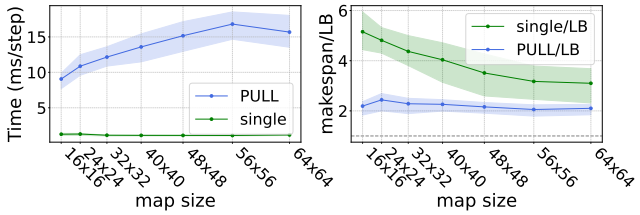

Figure 7: The result of Algorithm 1 in several maps with $n = 100$.

6.2 LaCAM* for adversarial instances

Finally, we validate our eventually optimal solver on three adversarial instances, with their results shown in Figure 8. We can observe that LaCAM* obtains an optimal solution for sufficiently small cases. Instances such as *10-3-0* reveal LaCAM*'s struggle to produce an optimal solution within two hours, indicating a need for more time to explore the entire configuration space. We also tested larger maps, such as *random-32-32-20*, but observed the same trend without notable improvements in makespan. We regard this as an initial reference point for future research. Indeed, experimental results show that it is not efficient, which indicates room for further improvement.

7 Conclusion

This paper tackled CUMAPF, a variant of MAPF that introduces a connectivity constraint on the entire agent configuration. This makes conventional MAPF algorithms inapplicable. Moreover, there are no algorithms readily applicable to CUMAPF in practical settings. We introduced a rule-based algorithm *PULL*, that empirically finds solutions superior to vanilla approaches. Additionally, we develop a search-based solver that employs *PULL* and is ensured to find an optimal solution eventually.

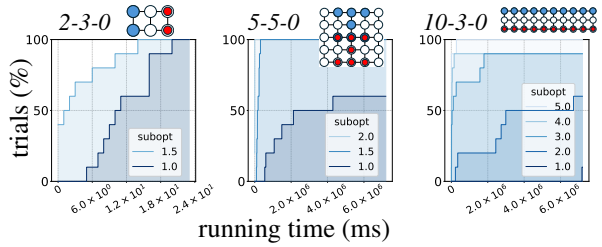

Figure 8: Experimental results of an eventually optimal algorithm using LaCAM*. The results are obtained from 10 runs with different random seeds. Each plot represents the percentage of trials that reached a specific makespan/LB (subopt).

This work contributes to the literature on MAPF with additional constraints, a promising area for future research. Moreover, our successful application of the LaCAM* framework to this non-trivial variant highlights its power and broad applicability to a wider class of constrained MAPF problems.

Several directions exist to improve *PULL*. For example, one might consider implementations using recursive functions, similar to PIBT (Okumura et al. 2022), or approaches that reduce the problem to flow-like problems. We determined that neither of these two methods can efficiently handle connectivity, making a significant algorithmic speedup challenging. The latter approach, however, specifically the escape problem (Cormen et al. 2022), might be applicable to refinement algorithms.

References

Cormen, T.; Leiserson, C.; Rivest, R.; and Stein, C. 2022. *Introduction to Algorithms, fourth edition*. MIT Press. ISBN 9780262046305.

Fekete, S. P.; Keldenich, P.; Kosfeld, R.; Rieck, C.; and Scheffer, C. 2023. Connected coordinated motion planning with bounded stretch. *Autonomous Agents and Multi-Agent Systems*, 37(2).

Hinnenthal, K.; Liedtke, D.; and Scheideler, C. 2024. Efficient Shape Formation by 3D Hybrid Programmable Matter: An Algorithm for Low Diameter Intermediate Structures. In *Proceeding of Symposium on Algorithmic Foundations of Dynamic Networks (SAND)*, 15:1–15:20.

Kostitsyna, I.; Peters, T.; and Speckmann, B. 2023. Fast Reconfiguration for Programmable Matter. In Oshman, R., ed., *37th International Symposium on Distributed Computing (DISC)*, volume 281, 27:1–27:21.

Křišťan, J. M.; and Svoboda, J. 2025. Reconfiguration Using Generalized Token Jumping. In Nakano, S.-i.; and Xiao, M., eds., *WALCOM: Algorithms and Computation*, 244–265.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9): 10256–10265.

Li, J.; Sun, K.; Ma, H.; Felner, A.; Kumar, T. K. S.; and Koenig, S. 2020. Moving Agents in Formation in Congested Environments. In *Proceedings of International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 726–734.

Martínez-Díaz, M.; Al-Haddad, C.; Soriguera, F.; and Antoniou, C. 2021. Platooning of connected automated vehicles on freeways: a bird's eye view. *Transportation Research Procedia*, 58: 479–486.

Okumura, K. 2023. Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding. In Elkind, E., ed., *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, 243–251.

Okumura, K.; and Défago, X. 2023. Solving simultaneous target assignment and path planning efficiently with time-independent execution. *Artificial Intelligence*, 321: 103946.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2022. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence*, 310: 103752.

Röger, G.; and Helmert, M. 2012. Non-Optimal Multi-Agent Pathfinding Is Solved (Since 1984). In Felner, A.; Sturtevant, N. R.; Bekris, K. E.; and Stern, R., eds., *Multiagent Pathfinding, Papers from the 2012 AAAI Workshop*, volume WS-12-10 of *AAAI Technical Report*.

Shankar, A.; Okumura, K.; and Prorok, A. 2025. LF: Online Multi-Robot Path Planning Meets Optimal Trajectory Control. arXiv:2507.11464.

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

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search (SoCS)*, 151–158.

Tateo, D.; Banfi, J.; Riva, A.; Amigoni, F.; and Bonarini, A. 2018. Multiagent Connected Path Planning: PSPACE-Completeness and How to Deal With It. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

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

Yu, J.; and LaValle, S. M. 2013a. Multi-agent Path Planning and Network Flow. In *Algorithmic Foundations of Robotics X*, 157–173.

Yu, J.; and LaValle, S. M. 2013b. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, 1443–1449.

A Omitted proofs of Section 4.3

A.1 Remaining proof of Lemma 1

Lemma 1 ($*$). *If $\mathcal{Q}^{from}$ is connected, then $\mathcal{Q}^{to}$ is connected, collision-free, and reachable.*

Proof. We show that there is no vertex conflict and swap conflict in the $(k+1)$-th call. First, assume that there is a vertex conflict, i.e., there is a vertex v in (cur, t)-path such that $v \in \mathcal{Q}^{\text{to}}(R_k)$. If $v = t$, t was already occupied by another agent, since $t \notin \mathcal{Q}^{\text{from}}$. However, this contradicts to the statement in line 7, which excludes the occupied vertices. If $v \neq t$, it holds that $v \in \mathcal{Q}^{\text{to}}(R_k)$ and $v \in F_{k+1} \subseteq \mathcal{Q}^{\text{to}} \setminus \mathcal{Q}^{\text{to}}(R)$, by the construction of F_k. This fact immidiately contradicts that $\mathcal{Q}^{\text{to}}(R_k) \cap (\mathcal{Q}^{\text{to}} \setminus \mathcal{Q}^{\text{to}}(R)) = \emptyset$. Therefore, no two agents can occupy the same vertex in $\mathcal{Q}^{\text{to}}_{k+1}$. Second, assume that there is a swap conflict, that is, there is a pair of agents $i, j \in R$ such that $\mathcal{Q}^{\text{to}}_{k+1}[i] = \mathcal{Q}^{\text{from}}[j] \wedge \mathcal{Q}^{\text{to}}_{k+1}[j] = \mathcal{Q}^{\text{from}}[i]$. Note that there is no collision occurs in $\mathcal{Q}^{\text{to}}_k$. Since both agents are moved by separate `PULL` calls within the same one-step function, one must be moved before the other. Assume without loss of generality that agent i is moved before agent j. Note that there is no collision in $\mathcal{Q}^{\text{to}}_k$, we can assume that the move of agent j occurs in $(k+1)$-th call, by induction hypothesis. Now consider the configuration $\mathcal{Q}^{\text{to}}_k$, which is the configuration that right before agent j is moved. In $\mathcal{Q}^{\text{to}}_k$ agent j has not yet moved, thus $\mathcal{Q}^{\text{to}}_k[j] = \mathcal{Q}^{\text{from}}_k[j]$ holds. Furthermore, agent i has already been assigned. By our initial assumption of a swap conflict, we know that $\mathcal{Q}^{\text{to}}_{k+1}[i] = \mathcal{Q}^{\text{from}}[j]$ and $\mathcal{Q}^{\text{to}}_{k+1}[i] = \mathcal{Q}^{\text{to}}_k[i]$, we have $\mathcal{Q}^{\text{to}}_k[i] = \mathcal{Q}^{\text{from}}[j]$. Then we have $\mathcal{Q}^{\text{to}}_k[j] = \mathcal{Q}^{\text{to}}_k[i]$ at the k-th call, which immediately contradicts to the fact that there is no vertex conflict in $\mathcal{Q}^{\text{to}}_k$.

This claim holds for all $k \in \mathbb{N}$, then $\mathcal{Q}^{\text{to}}_\tau$ is connected and collision-free.

To the last, it is clear that $\mathcal{Q}^{\text{to}}[i] \in N(\mathcal{Q}^{\text{from}}) \cup \mathcal{Q}^{\text{from}}$ for all the agents, thus $\mathcal{Q}^{\text{to}}_\tau$ is a reachable configuration from $\mathcal{Q}^{\text{from}}$. This completes the proof. □

A.2 Proof of Lemma 3

Lemma 3 ($*$). *If $\mathcal{Q}^{from} \cap T = \emptyset$, then*

$$\min_{(s,t) \in \mathcal{Q}^{to} \times T} \mathsf{dist}(s,t) = \min_{(s,t) \in \mathcal{Q}^{from} \times T} \mathsf{dist}(s,t) - 1$$

Proof. From the assumption, note that lines 3–9 are not executed. Consider the vertex u_0, the first vertex in the vertex set sorted on line 10. We can see that $\min_{t \in T} \mathsf{dist}(u_0, t) = \min_{(s,t) \in \mathcal{Q}^{\text{from}} \times T} \mathsf{dist}(s,t) - 1$. Note that R is initialized with $\emptyset$ in line 1. We can apply Lemma 2, and we can see that there is an agent a_i such that $\mathcal{Q}^{\text{to}}[i] = u_0$. □

A.3 Proof of Lemma 4

Lemma 4 ($*$). *If $\mathcal{Q}_\tau \cap T \neq \emptyset$ at step τ, $p^{max}_{\tau+1} \geq p^{max}_\tau + 1$.*

Proof. Let $\mathcal{P}$ be a list of components in $G[\mathcal{Q}_\tau \cap T]$, sorted according to their size. Note that $\mathcal{Q}^{\text{from}} = \mathcal{Q}_\tau$ and $\mathcal{Q}^{\text{to}} = \mathcal{Q}_{\tau+1}$. First, we show $\mathcal{P}[0] \subseteq \mathcal{Q}^{\text{to}}_k$ after the k-th call of procedure `PULL` for every k call by induction. Note that $\mathcal{P}[0] \subseteq \mathcal{Q}^{\text{to}}_0$ since $\mathcal{Q}^{\text{to}}_0 = \mathcal{Q}^{\text{from}}$. Assume that $\mathcal{P}[0] \subseteq \mathcal{Q}^{\text{to}}_{k'}$ holds for some k'. First we consider the case where $t_{k'+1} \in N(\mathcal{P}[0])$. Then it clearly holds that $\mathcal{Q}^{\text{to}}_{k+1} = \mathcal{Q}^{\text{to}}_k \cup \{t_{k'+1}\} \setminus \{\mathsf{cur}\}$ for some vertex $\mathsf{cur} \notin \mathcal{P}[0]$, since we restrict that the initial vertex cur is not in $\mathcal{P}[0]$ in line 17. Thus $\mathcal{P}[0]$ suffices that $\mathcal{P}[0] \subseteq \mathcal{Q}^{\text{to}}_{k'+1}$. Second, consider the case where $t_{k'+1} \notin N(\mathcal{P}[0])$. All agents $i \in A$ which holds $Q^{from}[i] \in \mathcal{P}[0]$ suffices $i \in R_{k'+1}$ in line 11, then every vertex in $\mathcal{P}[0]$ never be chosen as $\mathsf{cur}_{k'+1}$. Thus, it is clear that $\mathcal{P}[0] \subseteq \mathcal{Q}^{\text{to}}_{k'+1}$ by the same discussion as Lemma 1.

There is a call of function `PULL` $(u_0, \emptyset, P[0])$ for a vertex $u \in N(\mathcal{P}[0])$. This assignment always succeeds, by lemma 2. Then it is clear that $p^{max}_{\tau+1} \geq |\mathcal{P}[0]| + 1 = p^{max}_\tau + 1$, which completes the proof. □

A.4 Proof of Proposition 7

Proposition 7 ($*$). *PULL runs in $O(\Delta^2 n^2)$ time.*

Proof. It takes $O(E(G[\mathcal{Q}^{\text{from}}])) = O(\Delta n)$ time for BFS, DFS in line 15, 16, and line 21 takes $O(n)$ time. Thus, it takes $O(\Delta n)$ time for each call of procedure `PULL`.

Line 4 takes $O(\Delta n)$ time, and the number of calls is upper-bounded by Δn. Also, sorting the agent by distance takes $O(n)$ time by bucket sort by $O(n)$ bukcets, since

$$\begin{aligned}\min_{(s',t) \in (\mathcal{Q}^{\text{from}} \times T)} \mathsf{dist}(s',t) &\leq \min_{t \in T} \mathsf{dist}(s,t) \\ &\leq \min_{(s',t) \in (\mathcal{Q}^{\text{from}} \times T)} \mathsf{dist}(s',t) + n - 1\end{aligned}$$

for all $s \in \mathcal{Q}^{\text{from}}$. Thus, it holds that

$$\begin{aligned}\min_{(s',t) \in (\mathcal{Q}^{\text{from}} \times T)} \mathsf{dist}(s',t) - 1 &\leq \min_{t \in T} \mathsf{dist}(v,t) \\ &\leq \min_{(s',t) \in (\mathcal{Q}^{\text{from}} \times T)} \mathsf{dist}(s',t) + n\end{aligned}$$

for every $v \in N(\mathcal{Q}^{\text{from}})$. Then the total running time is $O(\Delta^2 n^2)$. □

A.5 Proofs for Tightness, Adversarial instances.

Proposition 10. *There is an instance where the optimal makespan is $\mathsf{diam}(G) + |A| - 1$, even when the given graph is planar.*

Proof. consider the graph, like the one illustrated in Figure 5(a). The optimal makespan is $k + \ell$, where $\mathsf{diam}(G) + |A| - 1 = (\ell + 1) + k - 1 = \ell + k$. We can observe that *PULL* returns a plan with makespan $\ell + k$, since the procedure `PULL` succeeds to assign exactly once at each time step. □

Proposition 11. *There is an instance where optimal makespan is 2, and PULL returns the plan with makespan $O(|A|)$.*

Proof. consider the graph, like the one illustrated in Figure 5(b). Even if the optimal makespan is 2, *PULL* produces a solution with a makespan of at best $|A|/2 + O(1)$. □

B Experiments

B.1 Used maps for experiments in Figure 7

Figure 9 shows the additional maps used in the experiments where the map size was varied. All of these, including agent placements, were randomly generated.

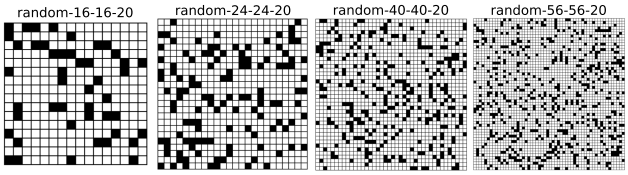

Figure 9: Used maps. All maps are randomly-generated, with 20% obstacles.

B.2 Table of results

Tables 2 to 6 present the detailed average values and interquartile ranges for each experiment. The interquartile range is presented in parentheses.

B.3 Supplementary Experiments for Algorithm 2

Figure 10 illustrates the trajectory of makespan for small instances, with the number of iterations on the x-axis. For instance like 10-3-0, we can observe that the number of iterations is small for seeds that converge to the optimal solution.

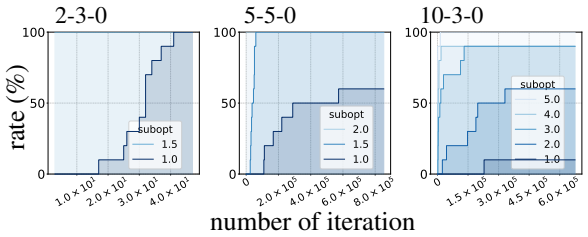

Figure 10: Experimental results of an eventually optimal algorithm using LaCAM*. The results are obtained from 10 runs with different random seeds. Each plot represents the percentage of trials that reached a specific makespan/LB.

Additionally we performed searches with map *random-32-32-20*, with randomly-generated instance ($|A| = 100$) with LB $= 22$. Figure 11 shows the results of the refinement. We can observe that there is a slight change in the solution quality. We obtain similar results for other instances on *random-32-32-20*.

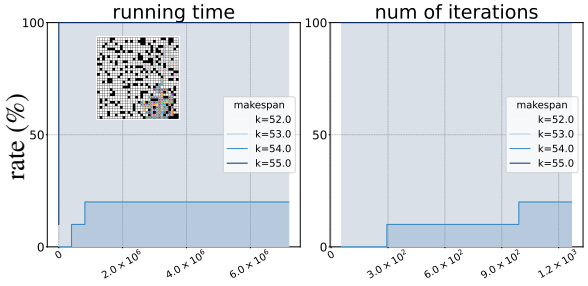

Figure 11: Experimental results of an eventualy optimal algorithm using LaCAM*, on *random-32-32-20*. The results are obtained from 10 runs with different random seeds.

Table 2: random-64-64-20

metrics	method	$\|A\|$ 100	200	300	400	500	600	700	800	900	1000
time	*PULL*	15.7 (13.5 – 18.0)	42.9 (36.3 – 49.2)	77.0 (64.5 – 88.7)	122.3 (101.6 – 144.3)	166.5 (143.6 – 189.2)	227.4 (200.4 – 257.1)	260.7 (226.2 – 300.5)	338.6 (299.4 – 385.1)	363.3 (317.9 – 420.5)	449.2 (397.3 – 509.3)
	single	1.2 (1.1 – 1.2)	2.1 (2.0 – 2.1)	3.2 (3.1 – 3.3)	4.5 (4.4 – 4.6)	5.9 (5.7 – 6.0)	7.3 (7.2 – 7.4)	9.0 (8.9 – 9.2)	10.7 (10.5 – 10.9)	12.6 (12.4 – 12.8)	14.5 (14.3 – 14.7)
makespan/LB	*PULL*	2.102 (1.844 – 2.261)	2.445 (2.155 – 2.684)	2.629 (2.340 – 2.908)	2.774 (2.398 – 3.082)	3.056 (2.617 – 3.362)	2.997 (2.626 – 3.370)	3.208 (2.809 – 3.542)	3.202 (2.919 – 3.433)	3.500 (3.121 – 3.773)	3.414 (3.084 – 3.639)
	single	3.100 (2.317 – 3.681)	5.439 (3.992 – 7.057)	7.238 (5.574 – 8.723)	8.615 (6.979 – 10.453)	11.013 (9.369 – 12.832)	12.022 (10.478 – 14.199)	13.598 (12.104 – 15.079)	15.028 (13.617 – 16.496)	16.380 (14.956 – 18.277)	17.537 (15.812 – 19.305)

Table 3: random-48-48-20

metrics	method	$\|A\|$ 100	200	300	400	500	600	700	800	900	1000
time	*PULL*	15.2 (13.0 – 17.2)	38.3 (34.5 – 43.3)	72.8 (65.3 – 81.7)	110.1 (99.2 – 123.0)	149.6 (138.2 – 172.5)	182.2 (160.5 – 204.6)	226.1 (198.6 – 260.5)	261.4 (226.8 – 308.5)	320.8 (273.1 – 372.0)	361.6 (302.5 – 415.2)
	single	1.1 (1.0 – 1.1)	2.1 (2.0 – 2.1)	3.2 (3.1 – 3.2)	4.5 (4.3 – 4.5)	5.9 (5.8 – 6.1)	7.5 (7.3 – 7.6)	9.2 (9.0 – 9.3)	11.0 (10.8 – 11.1)	12.7 (12.6 – 12.9)	14.6 (14.4 – 14.8)
makespan/LB	*PULL*	2.159 (1.907 – 2.333)	2.530 (2.242 – 2.745)	2.826 (2.409 – 3.056)	3.148 (2.789 – 3.431)	3.222 (2.794 – 3.587)	3.203 (2.795 – 3.480)	3.370 (3.069 – 3.648)	3.382 (3.111 – 3.654)	3.464 (3.151 – 3.730)	3.547 (3.180 – 3.896)
	single	3.511 (2.595 – 4.271)	6.243 (5.233 – 7.241)	8.268 (7.102 – 9.505)	10.478 (9.621 – 11.403)	12.209 (10.985 – 13.510)	13.203 (11.884 – 14.840)	14.505 (13.330 – 16.336)	14.410 (12.360 – 16.683)	15.625 (13.000 – 17.916)	16.355 (14.414 – 19.040)

Table 4: random-32-32-20

metrics	method	$\|A\|$ 100	200	300	400	500
time	*PULL*	12.2 (10.9 – 13.6)	31.9 (27.9 – 35.4)	55.1 (48.3 – 62.5)	76.6 (65.0 – 87.7)	94.3 (77.1 – 108.6)
	single	1.1 (1.0 – 1.1)	2.1 (2.0 – 2.1)	3.2 (3.1 – 3.3)	4.6 (4.4 – 4.7)	6.0 (5.8 – 6.1)
makespan/LB	*PULL*	2.284 (1.892 – 2.500)	2.592 (2.282 – 2.775)	2.832 (2.574 – 3.108)	2.856 (2.579 – 3.167)	2.862 (2.563 – 3.143)
	single	4.371 (3.805 – 5.000)	7.384 (6.709 – 8.195)	9.400 (8.424 – 10.535)	10.140 (8.475 – 11.601)	10.059 (8.122 – 12.000)

Table 5: warehouse-10-20-10-2-2

metrics	method	$\|A\|$ 100	200	300	400	500	600	700	800	900	1000
time	*PULL*	19.0 (15.3 – 23.3)	48.0 (39.1 – 54.4)	90.9 (69.8 – 110.1)	131.9 (108.8 – 149.2)	182.1 (144.0 – 212.0)	250.9 (202.0 – 288.2)	308.7 (250.4 – 365.8)	385.6 (321.8 – 444.8)	435.1 (353.3 – 506.2)	496.9 (432.8 – 547.7)
	single	1.2 (1.1 – 1.2)	2.2 (2.1 – 2.2)	3.3 (3.2 – 3.4)	4.6 (4.4 – 4.7)	6.1 (5.9 – 6.2)	7.5 (7.3 – 7.6)	9.3 (9.0 – 9.5)	11.1 (10.7 – 11.3)	13.0 (12.6 – 13.4)	15.1 (14.5 – 15.5)
makespan/LB	*PULL*	1.739 (1.488 – 1.953)	2.102 (1.671 – 2.388)	2.145 (1.769 – 2.365)	2.404 (1.921 – 2.692)	2.441 (1.989 – 2.595)	2.394 (1.967 – 2.566)	2.436 (2.013 – 2.735)	2.635 (2.103 – 2.985)	2.718 (2.098 – 3.252)	2.748 (2.230 – 3.175)
	single	2.660 (1.911 – 2.967)	4.478 (3.130 – 5.520)	6.378 (4.251 – 8.100)	8.263 (5.809 – 10.265)	9.750 (7.083 – 12.395)	10.310 (7.527 – 12.352)	11.616 (9.031 – 13.637)	13.244 (9.439 – 15.504)	14.091 (11.802 – 16.729)	15.227 (11.837 – 18.048)

Table 6: $|A| = 100$, 20% obstacles

metrics	method	map size 16×16	24×24	32×32	40×40	48×48	56×56	64×64
time	*PULL*	9.1 (7.7 – 9.9)	10.9 (9.6 – 12.5)	12.2 (10.9 – 13.6)	13.6 (11.5 – 15.4)	15.2 (13.0 – 17.2)	16.8 (14.7 – 18.5)	15.7 (13.5 – 18.0)
	single	1.3 (1.1 – 1.4)	1.3 (1.1 – 1.4)	1.1 (1.0 – 1.1)	1.1 (1.0 – 1.2)	1.1 (1.0 – 1.1)	1.1 (1.0 – 1.1)	1.2 (1.1 – 1.2)
makespan/LB	*PULL*	2.19 (1.83 – 2.37)	2.44 (2.00 – 2.69)	2.28 (1.89 – 2.50)	2.26 (1.99 – 2.45)	2.16 (1.91 – 2.33)	2.05 (1.79 – 2.24)	2.10 (1.84 – 2.26)
	single	5.15 (4.44 – 5.95)	4.81 (4.30 – 5.32)	4.37 (3.81 – 5.00)	4.04 (3.14 – 4.71)	3.51 (2.60 – 4.27)	3.17 (2.46 – 3.79)	3.10 (2.32 – 3.68)