Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

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 interchangeable agents must be connected at all times. This problem is fundamental to swarm robotics applications such as self-reconfiguration and marching, where standard MAPF is insufficient as it does not guarantee the connectivity constraint. Despite its simple structure, CUMAPF remains understudied and lacks practical algorithms. We first develop an INTEGER LINEAR PROGRAMMING (ILP) reduction to solve CUMAPF. Although this formulation provides a makespan-optimal plan, it is severely limited in terms of scalability and real-time responsiveness due to the large number of variables. We therefore propose a suboptimal but complete algorithm named *PULL*. 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. Empirically, PULL can quickly solve randomly generated instances containing hundreds of agents, which ILP cannot handle. Furthermore, PULL's solution substantially improves upon a naive approach to CUMAPF.

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 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 classified into *labeled*, *unlabeled*, and *colored* settings (Stern et al. 2019), and CUMAPF can be viewed in the context of the second setting. This setting can be applied when all agents can exchange roles or when they are functionally identical.

While unlabeled MAPF can be optimized in polynomial time for general cases (Yu and LaValle 2013a), adding connectivity constraints makes the problem significantly harder. Indeed, Fekete et al. (2023) showed that makespan optimization for CUMAPF is NP-hard even on simple 2D grids. Moreover, unlike the unlabeled case, various algorithms developed for labeled MAPF (see Table 1) do not work on CUMAPF, since they cannot explicitly handle connectivity constraints. This suggests that CUMAPF raises novel challenges unaddressed by existing frameworks, necessitating a new algorithmic framework.

In the context of MAPF, various algorithms have been proposed depending on the application and the characteristics of the solution needed: e.g., computationally intensive *optimal algorithms*, and lightweight *suboptimal algorithms*. Optimal algorithms, like CBS (Sharon et al. 2015), guarantee minimum 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 a suboptimal algorithm for practical applications. Table 1 shows several related works of MAPF and its variants. Although there are many algorithms for large-scale labeled and unlabeled MAPF, their counterparts for CUMAPF are missing; at present, we only know the NP-hardness, and an approximation algorithm that runs in restricted conditions.

Contribution. In this paper, we provide both optimal and suboptimal approaches for CUMAPF. We first present a reduction-based algorithm that utilizes an INTEGER LINEAR PROGRAMMING (ILP). Although this method finds a makespan-optimal solution, it is not scalable and is unsuitable for real-time applications due to the large number of variables and constraints. This motivates us to develop a suboptimal yet polynomial-time and complete algorithm. The

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

	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	**Reduction to ILP (this work)**, (PULL + LaCAM*; appendix)	**PULL (this work)**	NP-hard for optimization (Fekete et al. 2023)

Table 1: Categorization of MAPF variants and their solutions. While we focus on ILP and PULL for CUMAPF, we further explore an eventually optimal algorithm, which quickly finds initial suboptimal solutions and refines them towards the optimum. This is achieved by embedding PULL into a configuration generator-based MAPF algorithm called LaCAM* (Okumura 2023). We omitted it due to limited page space and the subtle practical effects of solution refinement; see the full version (Suzuki and Okumura 2025) for detailed discussion.

proposed *PULL* algorithm is a *configuration generator*, efficiently computing the next placement of the agents from the current ones and the target. PULL is computationally lightweight; it runs in $O(\Delta^2 n^2)$ time per one step, where Δ and n are the maximum degree of a graph and the number of agents, respectively. Furthermore, PULL outputs empirically efficient solutions compared to those obtained by a trivial approach (e.g., 350% improvement for 500 agents on a 32×32 grid). In what follows, we present problem formulation, related work, algorithms, 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, unweighted, 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. We use a notation for a set of continuous integers: let $[i, j]$ be a set $\{i, i+1, \dots, j-1, j\}$ for $i \leq j$.

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 *component* of G is a subgraph of the form $G[\mathsf{Reach}(G, v)]$ for some $v \in V$. 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 $A = [1, n]$ 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, while maintaining the connectivity of the entire configuration. Formally, a *valid* plan Π satisfies the following conditions:

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

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

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

MAPF with additional constraint. MAPF typically considers vertex or swap 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 physical and communication features are given separately.

CUMAPF. Fekete et al. (2023) show that finding optimal solutions to CUMAPF is NP-hard, even in 2D empty grid graphs. This hardness also applies to finding a good solution, specifically a 3/2-approximation of the makespan. The work also introduces a polynomial-time, constant-factor approximation algorithm for makespan in *empty* 2D grid environments. Despite this, its practical applicability is severely limited, as it requires a large number of theoretical steps, has quite a large approximation ratio, and works only under restricted conditions, motivating our study.

4 Optimal Approach: ILP

While the makespan-optimal unlabeled MAPF can be solved by reduction to the MAXIMUM FLOW problem (Yu and LaValle 2013a), the same idea does not apply to CUMAPF due to the connectivity constraint. We thus adopt the reduction to INTEGER LINEAR PROGRAMMING (ILP). This section first formulates ILP for the unlabeled case, and extends it to include the connectivity constraint.

4.1 ILP Formulation of Unlabeled MAPF

In this section, we describe a reduction to the unlabeled MAPF to ILP. Note that unlabeled MAPF can be reduced to MAXIMUM FLOW, which in turn can be reduced to an ILP; however, for simplicity, we directly give an ILP formulation.

First, we define the *Bounded* variant of unlabeled MAPF: given a graph $G = (V, E)$, vertex subset $S, T \subseteq V$, and *time bound* $\tau \in \mathbb{N}$, it determines whether there exists a plan with conditions 1–4 in Section 2 and the length is at most τ.

Here, we assume that all variables are nonnegative. Let $x_{v,t}$ be a variable that indicates whether an agent is at vertex v at step t, that is, $x_{v,t} = 1$ when $v \in \mathcal{Q}_t$, and 0 otherwise. Also, $f_{u,v,t}$ represents whether an agent at vertex u at step t moves to $v \in N[u]$ at timestep $t+1$, using an edge $uv \in E$. Then, unlabeled MAPF can be formulated as follows.

1. For every $u, v \in V$, $x_{v,t} \leq 1$.
2. Initial configuration $\mathcal{Q}_0$ is S: for $v \in V$, $x_{v,0} = 1$ if $v \in S$, and 0 otherwise.
3. Final configuration $\mathcal{Q}_\tau$ is T: for $v \in V$, $x_{v,\tau} = 1$ if $v \in T$, and 0 otherwise.
4. For every timestep $t \in [0, \tau]$, at most x_u agents on u move to some vertex $v \in N[u]$: for every $u \in V$, $\sum_{v \in N[u]} f_{u,v,t} = x_{u,t}$
5. For every timestep $t \in [1, \tau+1]$, at most x_u agents on u come from some vertex $v \in N[u]$: for every $u \in V$, $\sum_{v \in N[u]} f_{v,u,t-1} = x_{u,t}$

Then we set an objective as a constant, to check whether these constraints are *feasible*, i.e., there is a satisfying assignment for these constraints. Note that swap conflicts can be resolved by a common postprocessing (Yu and LaValle 2013a).

By solving this bounded unlabeled MAPF incrementally from $\tau = 1$ at most $|V| + n - 2$ times, there exists at least one feasible instance (Yu and LaValle 2013a). Let τ' be the minimum time bound in the instance of unlabeled MAPF. Note that the assignment of $x_{v,t}$ corresponds to the position of each agent at each timestep. Based on the assignment for this instance, we can construct a valid plan Π with length at most τ' by a proper postprocessing.[1] Thus, this formulation leads to an optimal algorithm.

4.2 Connectivity Constraint

One can observe that all we have to do is to add the constraint to $V_t \coloneqq \{v \mid x_{v,t} = 1\}$ for every $t \in [0, \tau]$ such that $G[V_t]$ is connected, i.e., there is at least one (u, v)-path for every $u, v \in V_t$. The connectivity of the graph is encoded by a *Single-commodity Flow* model (Conrad et al. 2007): a single source at the root supplies one commodity to each selected vertex, with flow allowed only through the vertices in V_t.

First, we introduce variables $r_{v,t} \in [0, x_{v,t}]$ that represent whether v is chosen as a source, and $\ell_{uv,t}$ indicating the number of commodities that are carried through an edge from u to v such that $uv \in E$. Then, we can formulate the constraints as follows;

1. $\forall v \in V, r_{v,t} \leq x_{v,t}$,
2. $\sum_{v \in V} r_{v,t} = 1$,
3. $\forall u, v \in V. \ell_{uv,t} \leq (|A| - 1)x_{v,t}$, and
4. $\forall v \in V. \sum_{u \in N[v]} (\ell_{uv,t} - \ell_{vu,t}) = x_{v,t} - |A| r_{v,t}$.

We now justify these constraints, with reference to Fig. 1. First, in Constraints 1 and 2, we can restrict the source of flow to a single vertex per timestep. The last two constraints coincide with carrying commodities via a flow network that consists of vertices in V_t. To do so, Constraint 3 restricts usable edges to those that connect only vertices in V_t; if $x_{u,t} = x_{v,t} = 1$, then edge uv can carry $|A| - 1$ units of flow, that is, all the flow except the (u, u)-path. Regarding the last Constraint, consider adding unit of flow that goes along some (v_1, v_2)-path to the current flow, and see the value $\sum_{u \in N[v]} (\ell_{uv,t} - \ell_{vu,t})$ of the left-hand side in this case. At the source v_1, one unit of flow leaves, so the value decreases by 1; at v_2, one unit of flow enters, so the value increases by 1; and at intermediate vertices, one unit enters, and one unit leaves, thus the net change is 0. Here, in our network, we start from an empty flow, and add a (u, v)-path for the root u with $r_{u,t} = 1$ and every $v \in V_t$. Therefore, the final value is $1 - |A|$ at the root u, 1 for every $v \in V_t$, and 0 for every $v \notin V_t$. This is because u is selected $|A|$ times as an initial vertex and once as a target vertex, and $v \in V_t$ is selected once as a target vertex. The value $x_{v,t} - |A| r_{v,t}$ of the right-hand side coincides exactly with this value. The optimality is justified in the same way as for unlabeled MAPF.

4.3 Limitation – Scalability

One can observe that, given an instance of bounded CUMAPF, we solve an ILP with $O((|V| + |E|)\tau)$ variables and $O(|V|\tau)$ constraints.

Although this method can efficiently find optimal solutions on small instances, the computational costs on large-scale problems become prohibitively high. Table 2 shows the empirical results for two maps retrieved from the well-known MAPF benchmarks (Stern et al. 2019), using Gurobi as the

[1] Since the information about agent indices has been lost, we use a perfect matching to recover it. Detailed implementations are in https://github.com/su3taka/CUMAPF-PULL.

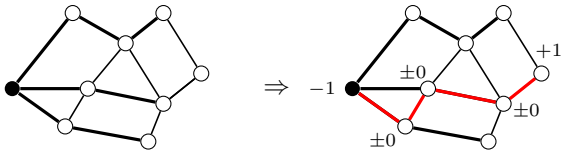


Figure 1: Constraint 4 for ILP reduction. Circles represent vertices of the graph, and a black circle represents the sources determined by Constraint 1. Lines represent edges, and bold lines indicate edges used by the flow. (*Left*) An intermediate network and several flows. (*Right*) The network with a new flow (the red path). We indicate the net change of $\sum_{u \in N[v]}(\ell_{uv,t} - \ell_{vu,t})$ for each vertex on the red path.

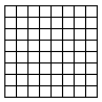
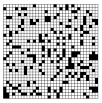
Map	n	ILP solved (%)	time(s) ILP	time(s) PULL	ratio speedup	ratio makespan
empty-8-8 	10	100	0.137	0.005	27.36	1.347
	15	100	0.226	0.010	18.73	1.484
	20	100	0.233	0.013	15.15	1.540
	25	100	0.202	0.015	11.95	1.678
	30	100	0.190	0.018	9.934	1.713
random-32-32-20 	10	82	56.34	0.020	1660	1.131
	20	48	75.36	0.050	1499	1.460
	30	34	57.43	0.094	862.1	1.650
	40	24	77.75	0.157	804.1	1.676
	50	4	93.66	0.231	N/A	N/A

Table 2: Evaluation on the optimal algorithm, using 300 s timeout. For each map, we prepared five types of n, generated 50 random instances for each. The time column reports the average time to find a solution over the solved instances. We also report the average running time of PULL (s), the ratio of time improvement (ILP/PULL), and the suboptimality of makespan (PULL/ILP) on the instances successfully solved by ILP. The ratio is omitted in the last row, due to the small sample size.

ILP solver. On a small map (*empty-8-8*), the solver can find an optimal solution within a second, while on a larger map (*random-32-32-20*), the solver often fails to find a solution, and the number of failed instances is increasing as n increases. Moreover, the average time to find a solution also increases as n increases. This empirical result motivates us to explore a fast and scalable suboptimal approach, which we will introduce next.

5 Suboptimal Approach: PULL

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 timestep. 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

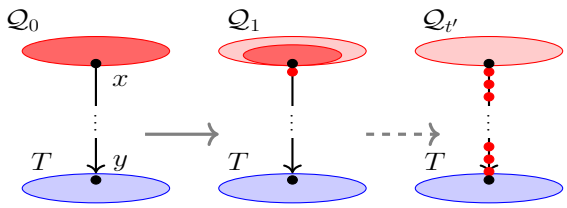


Figure 2: The naive approach for solving CUMAPF. Light red represents $\mathcal{Q}^{\text{from}} = \mathcal{Q}_0$, deep red represents the current configuration, and blue represents T. (*Left*) Initial configuration, (*Center*) generated configuration $\mathcal{Q}_1$ from $\mathcal{Q}_0$, (*Right*) configuration at step $t' = \mathsf{dist}(x, y)$.

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 $a \neq b$.

5.1 Concept

We begin the description of PULL with a bare minimum idea to solve CUMAPF. Here, given an input configuration $\mathcal{Q}^{\text{from}}$ that is not duplicated with a target T (i.e., $\mathcal{Q}^{\text{from}} \cap T = \emptyset$), we are interested in obtaining a transitionable configuration $\mathcal{Q}^{\text{to}}$ from $\mathcal{Q}^{\text{from}}$ that is closer to T.

Let $x \in \mathcal{Q}^{\text{from}}$ and $y \in T$ be a pair of vertices such that $(x, y) = \arg\min_{(s,t) \in \mathcal{Q}^{\text{from}} \times T} \mathsf{dist}(s, t)$ (Figure 2 left). If the agent currently on x can move to $z := \mathsf{next}(x, y)$ without breaking connectivity, then we can guarantee that there is at least one agent that moves towards the goal.

If $(\mathcal{Q}^{\text{from}} \setminus \{x\}) \cup \{z\}$ is not connected, then we have to move 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 is connected. This recursive procedure allows the moved agent to form a single path that follows the DFS-tree over connected agents. It always terminates when it reaches a vertex that does not "cut" the configuration, since any leaf of a DFS-tree cannot 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 $\mathcal{Q}^{\text{from}}$ (Figure 2 center).

By iteratively applying this process, the agent moves a distance of one per step on the shortest (x, y)-path towards y. Then, there is a step t such that exactly one agent reaches y (Figure 2 right). The remaining procedure is to "pour" the agents into the goal, which will be elaborated in Section 5.2.

While this naive approach can solve CUMAPF, it provides a highly makespan-suboptimal solution in practice. This is because the algorithm only moves the agents along a simple path, starting from $\mathsf{next}(x, y)$, at each step, and other agents remain unchanged. For example, it only executes the leftmost column in Figure 3, resulting in the configuration shown in the upper part of the second column.

The aforementioned idea uses a single path in the induced subgraph $G[\mathcal{Q}^{\text{from}}]$ when generating the next configuration. However, there may exist multiple paths towards the goals as long as they are pairwise disjoint and the resulting configuration $\mathcal{Q}^{\text{to}}$ is connected. That is, we can synthesise a configuration much closer to the goal by moving other unconstrained

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: $R \leftarrow \emptyset$

(when the configuration and goals overlapped)
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** $\nexists j$ s.t. $\mathcal{Q}^{\text{to}}[j] = v$ **then** $R \leftarrow \texttt{PULL}(v, R, P_i)$
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** $R \leftarrow R \cup \{i\}$

(main operation)
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** $\nexists j$ s.t. $\mathcal{Q}^{\text{to}}[j] = u$ **then** $R \leftarrow \texttt{PULL}(u, R, \emptyset)$
13: **return** $\mathcal{Q}^{\text{to}}$

(pull agents from $t \in V$)
14: **procedure** $\texttt{PULL}(t, R, V')$
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: $\quad\quad Q^{to}[i] \leftarrow \mathsf{next}(\mathsf{cur}, t)$ ▷ *determining* $\mathcal{Q}^{to}$
22: $\quad\quad R \leftarrow R \cup \{i\}, \mathsf{cur} \leftarrow Q^{to}[i]$
23: $\quad$ **return** R

agents. This idea constitutes the approach of PULL.

5.2 Algorithm

Algorithm 1 illustrates PULL, which consists of three blocks: *(i)* Lines 14–23 denote a subprocedure PULL that "pulls" the agents, *(ii)* Lines 10–13 call PULL to repeatedly identify pullable paths, and; *(iii)* Lines 2–9 handle a configuration that overlaps with the goal configuration by calling PULL with the non-movable, constrained agents.

Our core is the procedure $\texttt{PULL}(t, R, V')$ for $t \in N(\mathcal{Q}^{\text{from}})$, $R \subseteq A$, $V' \subseteq \mathcal{Q}^{\text{from}}$. This computes a path on $G[\mathcal{Q}^{\text{from}}]$ to "pull" agents toward a specified vertex t, while avoiding the subset of agents R and preserving connectivity. Furthermore, the path never starts from any vertex in V'. Figure 3 provides an example of the whole process to output $\mathcal{Q}^{\text{to}}$ from $\mathcal{Q}^{\text{from}}$. First, we provide detailed descriptions of PULL.

Pull process – Enumerating available vertices. To identify a pullable path, PULL firstly 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). BFS conducts this starting from t over $G[\mathcal{Q}^{\text{to}} \cup \{t\} \setminus \mathcal{Q}^{\text{to}}(R)]$. Since the chain of moves of agents forms a single path (similar to the discussion in Section 5.1), this BFS operation characterizes the set of vertices that satisfy a necessary condition to be an initial vertex of the path (denoted by F).

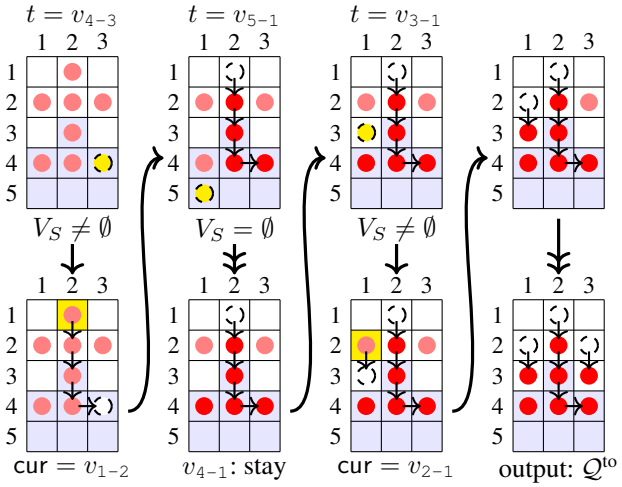


Figure 3: An example of the PULL's operation. The graph G is 5×3 grid, and $v_{\texttt{i-j}}$ denotes the vertex with row i, column j. Light red indicates $\mathcal{Q}^{\text{from}}$, and blue represents T. (*Column 1*) first PULL call $\texttt{PULL}(v_{4-3}, \emptyset, \{v_{3-2}, v_{4-1}, v_{4-2}\})$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

Next, PULL computes 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).

Consequently, $F \setminus B$ 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, $\texttt{PULL}(t, R, V')$ fails to find the agents to move, 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.

Pull process – Selection. Next, PULL selects the initial vertex of the pulling path. From the set of vertices V_S, PULL 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). Then PULL starts to determine next positions sequentially; for an agent i such that $\mathcal{Q}^{\text{from}}[i] = \mathsf{cur}$, PULL determines $\mathcal{Q}^{\text{to}}[i]$ as $\mathsf{next}(\mathsf{cur}, t)$, which is already computed in line 15. Then agent i is added to set R of agents already determined next position, and the procedure moves to the next vertex $\mathsf{next}(\mathsf{cur}, t) = \mathcal{Q}^{\text{to}}[i]$. After performing the decisions iteratively, PULL generates the next configuration and returns a set R of agents already determined their next position.

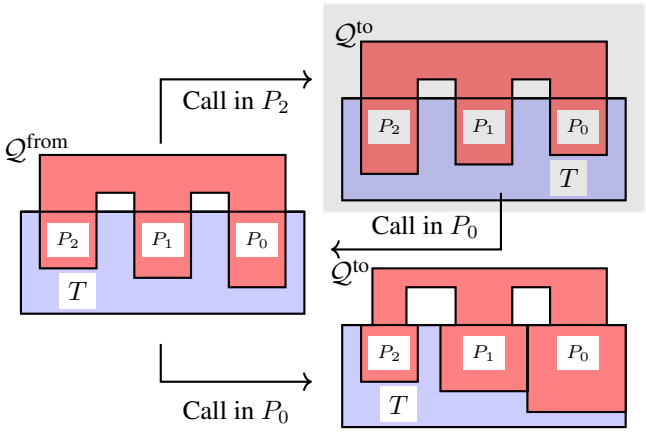


Figure 4: A conceptual illustration of line 4. (**Left**) initial configuration. (**Upper right**) intermediate configuration without prioritization. If there is no proper prioritization, for example, at step τ, P_2 pulls the agents of P_0, and at the step $\tau+1$, P_0 pulls the agents of P_2, causing a livelock. (**Lower right**) intermediate configuration before executing line 10. In the initial iteration, PULL calls PULL on $N(P_0)$, which increases the size of the largest component. Two connected components might merge (e.g., P_0 and P_1 in the figure), and these are computed as a larger component in the next step.

Top-level procedure (lines 2–12). PULL repeatedly applies PULL and updates $\mathcal{Q}^{\text{to}}$. Doing so, PULL can generate the configuration that approaches T, if $\mathcal{Q}^{\text{from}} \cap T = \emptyset$.

When $\mathcal{Q}^{\text{from}}$ overlaps with T. In this case, we need to care about which vertex to pull first. Figure 4 (left) shows such an example. When PULL first pulls an agent in P_2 at step t and P_0 at step $t+1$, the iteration causes a livelock. To resolve this issue, we prioritize "blocks of agents that have reached T," i.e., the components of $G[\mathcal{Q}^{\text{from}} \cap T]$, and then PULL perform PULL to enlarge the largest one further (Figure 4 lower right).

To be concrete, 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 4. 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 4). Thus, the size of the largest component becomes larger step by step. Combining this method and process at line 9, we can prove the convergence.

5.3 Theoretical Analysis

We write 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 space limitations, we omit the proofs in the paper, and we give some proof sketches for key lemmas. See (Suzuki and Okumura 2025) for full proofs.

We first show that our proposed algorithm correctly generates the configuration $\mathcal{Q}^{\text{to}}$ from the input $\mathcal{Q}^{\text{from}}$.

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

Proof. (sketch) Regarding reachability, it is clear since the decision of $\mathcal{Q}^{\text{to}}[i]$ is made according to $\mathcal{Q}^{\text{to}}[i] \in N(\mathcal{Q}^{\text{from}}[i]) \cup \{\mathcal{Q}^{\text{from}}[i]\}$ for every $i \in A$, from line 21. Furthermore, for each agent $i \in A$, once its destination $\mathcal{Q}^{\text{to}}[i]$ is fixed, it is added to the set R, so its destination will not be changed. From here, we show that k-th call of PULL generates a connected, collision-free configuration $\mathcal{Q}^{\text{to}}_k$ from $\mathcal{Q}^{\text{from}}_k = \mathcal{Q}^{\text{to}}_{k-1}$. Upon a successful pull by PULL, each agent on the (cur, t)-path moves one step forward. Ignoring labels, this is equivalent to moving an agent at cur to t, hence $\mathcal{Q}^{\text{to}}_{k'+1} = (\mathcal{Q}^{\text{to}}_{k'} \cup \{t\}) \setminus \{\mathsf{cur}\}$. Since cur is not a cut vertex of $G[\mathcal{Q}^{\text{to}}_{k'} \cup \{t\}]$, we can conclude $\mathcal{Q}^{\text{to}}_{k'+1}$ is connected. Next, there is no conflict within a procedure in line 21. Moreover, in another process, we search paths while avoiding agents in R and destinations $\mathcal{Q}^{\text{to}}(R)$ that have already been determined next positions, thus there is no path crossing with an existing one. Therefore, no conflict occurs throughout, and we conclude that $\mathcal{Q}^{\text{to}}$ is conflict-free. □

Next, we show that $\mathcal{Q}^{\text{to}} \neq \mathcal{Q}^{\text{from}}$, that is, at least one PULL call succeeds. Note that, for every call of PULL, we can assume that the third argument V' is either empty or forms a connected induced subgraph.

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. (sketch) To prove the lemma, we must show that the set V_S (line 17) is always non-empty for the first call of PULL, i.e., when $R = \emptyset$, since $F = \mathcal{Q}^{\text{from}}$. The case when $V' = \emptyset$ is clear since every graph has at least two vertices that are not the cut vertex. Even after removing t, at least one vertex remains; hence V_s is nonempty.

Next, we consider when $V' \neq \emptyset$, that is, PULL is called in line 7. The proof is conducted by contradiction. Consider the vertex $w \in \mathcal{Q}^{\text{from}} \setminus V'$ such that w holds that the largest minimum distance $\min_{t \in V'}(\mathsf{dist}(v, t))$ among every vertex $v \in \mathcal{Q}^{\text{from}} \setminus V'$ (♠). Suppose that w is a cut vertex of $G[\mathcal{Q}^{\text{from}} \cup \{t\}]$ for a contradiction. Then, the graph $G[(\mathcal{Q}^{\text{from}} \cup \{t\}) \setminus \{w\}]$ consists of more than one component (Figure 5 left). Let C_1 be a component that contains V' (note that, since $G[V']$ is connected, it is contained in a single component), and consider the vertex u' in another component. Since w is a cut vertex, every path connecting u' and $v' \in V'$ contains w as an internal vertex, and thus the shortest (u', v')-path also does. Therefore, $\mathsf{dist}(u', v') \geq \mathsf{dist}(w', v') + 1$ holds (Figure 5 right); a contradiction to the maximality of w (♠), thus w does not cut $\mathcal{Q}^{\text{from}}$. □

From Lemma 2, we can show the following lemmas, which are necessary to guarantee the completeness of our algorithm. The first case is when $\mathcal{Q}^{\text{from}}$ does not overlap T. As the approach introduced in Section 5.1, we show that at least one of the agents that are currently closest to T moves in a direction that reduces its distance to T by one.

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

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

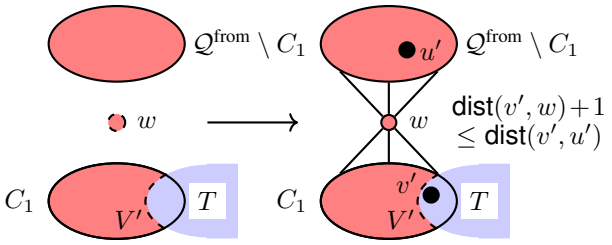


Figure 5: A brief explanation of the proof in Lemma 2. (*Left*) Graph $G[\mathcal{Q}^{\text{from}} \cup \{t_1\} \setminus \{w\}]$ is divided into multiple components, shown as two for convenience. (*Right*) A graph $G[\mathcal{Q}^{\text{from}} \cup \{t_1\}]$ has a single component, and every (v', u')-paths must pass through w since w is a cut vertex.

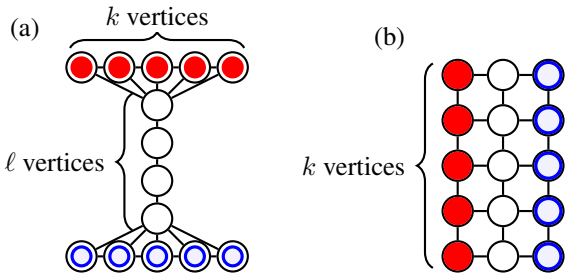


Figure 6: (a) An example of a tight instance: the optimal makespan is $\mathsf{diam}(G)+n-1$. (b) An example of a adversarial instance: PULL outputs plan with $O(n)$ makespan, while the optimal makespan is two.

Then there exists a step τ' such that $\mathcal{Q}_{\tau'}$ overlaps T. Thus, the condition at line 3 is satisfied at some step. From here, let $p_\tau^{\max}$ be the size of the largest component of $G[\mathcal{Q}_\tau \cap T]$. We finalise the proof by showing that $p_\tau^{\max}$ is strictly increasing.

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

Then there is a step $\tau^\star$, where $p_{\tau^\star}^{max} = n$. At step τ^*, it holds that $\mathcal{Q}_{\tau^\star}^{\text{from}} = T$, 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.

Theorem 5. *PULL is complete for* CUMAPF.

Proposition 6. *The makespan of* CUMAPF *is bounded by $\mathsf{diam}(G)+n-1$, where $\mathsf{diam}(G) := \max_{(s,t)\in V^2} \mathsf{dist}(s,t)$.*

For the time complexity of PULL, we have the following:

Proposition 7. *PULL runs in $O(\Delta^2 n^2)$ time, where Δ is the maximum degree of G.*

Proof. (sketch) The worst-case running time can be computed from an upper bound on the number of times `PULL` is invoked and the time required for a single execution of `PULL`. The number of times `PULL` is invoked is at most $|N[\mathcal{Q}^{\text{from}}]| \leq \Delta n$. The running time of a single execution of `PULL` is dominated by the BFS, DFS, and is $O(|\mathcal{Q}^{\text{from}}| + |E(G[\mathcal{Q}^{\text{from}}])|) = O(\Delta n)$. Therefore, the running time is upper bounded by $O(\Delta n) \times O(\Delta n) = O(\Delta^2 n^2)$. □

Combining Proposition 6 and 7, we have the following.

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

Note that, the distance $\min_{t\in T} \mathsf{dist}(u,t)$ for every $u \in V(G)$, used in lines 10 and 18, can be precomputed in $O(|V| + |E|) = O(\Delta|V|)$ time.

Tightness, Adversarial instances. The makespan upper bound established is tight; an example is shown in Figure 6(a). Furthermore, we show that for some instances, PULL outputs a plan with a makespan of $O(n)$, even when the optimal makespan is two (see Figure 6 (b)).

6 Empirical Analysis

This section presents an empirical evaluation of PULL. Specifically, we conduct the following two: a runtime evaluation against the optimal ILP algorithm, and an evaluation of the running time and solution quality (i.e., makespan) of PULL against a simple solution on a variety of benchmark maps. 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 Comparison between ILP and PULL

As shown in Table 2, the ILP-based algorithm is effective for small instances, but its runtime becomes prohibitively large for larger ones. In contrast, PULL finds a solution within the time limit for all instances in "empty-8-8," and on the instances solvable by ILP, its average running time was almost 10 times faster in the worst. In ILP, we add $O(|V|)$ constraints when a step is incremented, thus the runtime increases by a factor of $O(2^{|V|})$. In contrast, PULL computes a succeeding configuration with an additional time overhead of $O(\Delta^2 n^2)$, resulting in rapid planning. Moreover, the makespan *suboptimality*, defined as makespan/optimum, is at worst on average 1.7, indicating that it outputs practically useful solutions.

6.2 PULL on Large-scale Problem

We now demonstrate the effectiveness of PULL on large-scale instances. 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 *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 (S, T). The evaluation of ILP is omitted here due to the lack of scalability.

As finding the optimum is hard for these instances, we benchmark our solution quality against a trivial lower bound from the value of bottleneck matching (see (Gabow and Tarjan 1988) for details), which is denoted by LB. This represents the plan length required to transform S into T, disregarding any conflicts and connectivity. Note that PULL is the first suboptimal algorithm that addresses CUMAPF– no prior methods available. Thus, we used a simple solution in the Section 5.1 (denoted as single), the one that calls `PULL` only once at each step, as a baseline for solution quality.

Figure 7 displays the empirical 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:

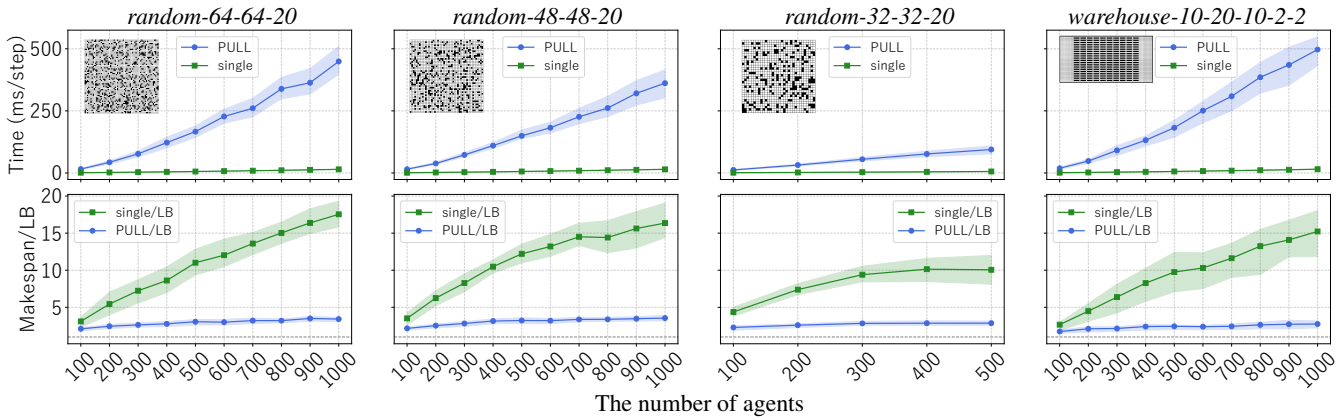


Figure 7: Empirical results for time and makespan/LB across four maps. Blue represents Algorithm 1, and green represents the algorithm in Section 5.1. Lines plot the average values, and the filled areas indicate the interquartile range.

- For each map, the average computational time of PULL increases with the number of agents; however, its growth is more gradual than $O(\Delta^2 n^2) = O(n^2)$. It 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.
- The average suboptimality 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 makespan of PULL is 0.3 times or less than that of the simple solution. Moreover, in sparse conditions, PULL outputs solutions closer to the lower bound than single. We can conclude that PULL produces solutions that are comparatively close to the LB when compared to the simple solution, leading to more efficient operations.

Sensitivity to the map size. We conducted an additional experiment to demonstrate that the computational time is largely independent of the number of vertices $|V|$. Theoretically, the execution time depends solely on n in the 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 8 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. We observe that $|V|$ has a small impact on PULL's running time. We can also see that makespan/LB decreases as density gets smaller, consistent with the previous observation.

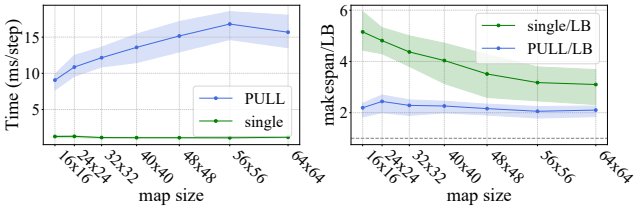


Figure 8: Algorithm 1 in several maps with $n = 100$.

7 Conclusion and Discussion

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 first formulate CUMAPF (also Unlabeled MAPF) for ILP, and obtain an optimal algorithm. Next, we introduced a rule-based algorithm PULL, that empirically finds solutions superior to vanilla approaches. This work contributes to the literature on MAPF with additional constraints, a promising area for future research. The discussion is as follows.

Algorithm improvement? Several directions exist to improve PULL. For example, one might consider implementations using recursive functions, similar to PIBT (Okumura et al. 2022), or reduction to flow-like problems. Our early attempts observed that neither of these two methods can efficiently handle connectivity, making a significant algorithmic speedup challenging. Nevertheless, the latter approach—specifically the escape problem (Cormen et al. 2022)—might be applicable to refinement algorithms.

Anytime algorithm. As noted in Table 1, we also developed an eventually optimal algorithm for CUMAPF that integrates PULL with a search-based MAPF algorithm LaCAM* (Okumura 2023). We use PULL as a configuration generator and perform an anytime search over configurations, which is guaranteed to converge to an optimal solution eventually. While this approach finds an optimal solution for small instances, the refinement effect for larger instances is almost nothing in practice. Developing a sophisticated anytime scheme for CUMAPF remains an open question.

References

Conrad, J.; Gomes, C. P.; van Hoeve, W.-J.; Sabharwal, A.; and Suter, J. 2007. Connections in Networks: Hardness of Feasibility Versus Optimality. In *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, 16–28. Berlin, Heidelberg: Springer Berlin Heidelberg.

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).

Gabow, H. N.; and Tarjan, R. E. 1988. Algorithms for Two Bottleneck Optimization Problems. *J. Algorithms*, 9(3): 411–417.

Hinnenthal, K.; Liedtke, D.; and Scheideler, C. 2024. Efficient Shape Formation by 3D Hybrid Programmable Matter: An Algorithm for Low Diameter Intermediate Structures. In *Proceedings 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 *Proceedings of the 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 *Proceedings of International Conference and Workshops on Algorithms and Computation (WALCOM)*, 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 AAAI Conference on Artificial Intelligence (AAAI)*, 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 *Proceedings of 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 *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. In *Proceedings of Annual Symposium on Combinatorial Search (SoCS)*, 151–158.

Suzuki, T.; and Okumura, K. 2025. Polynomial-time Configuration Generator for Connected Unlabeled Multi-Agent Pathfinding. *CoRR*, abs/2510.19567.

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

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 AAAI Conference on Artificial Intelligence (AAAI)*, 1443–1449.