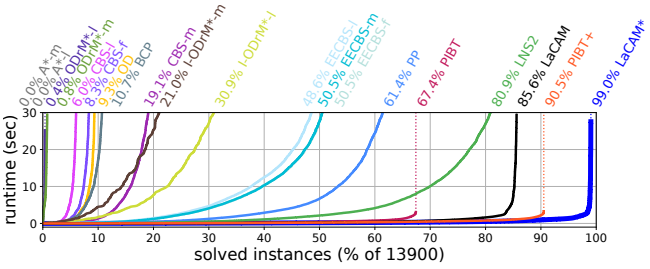
Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding

Keisuke Okumura[1,2]
[1]National Institute of Advanced Industrial Science and Technology (AIST)
[2]University of Cambridge
ko393@cl.cam.ac.uk

Abstract

This study extends the recently-developed LaCAM algorithm for multi-agent pathfinding (MAPF). LaCAM is a sub-optimal search-based algorithm that uses lazy successor generation to dramatically reduce the planning effort. We present two enhancements. First, we propose its anytime version, called LaCAM*, which eventually converges to optima, provided that solution costs are accumulated transition costs. Second, we improve the successor generation to quickly obtain initial solutions. Exhaustive experiments demonstrate their utility. For instance, LaCAM* sub-optimally solved 99% of the instances retrieved from the MAPF benchmark, where the number of agents varied up to a thousand, within ten seconds on a standard desktop PC, while ensuring eventual convergence to optima; developing a new horizon of MAPF algorithms.



	algorithm	reference	solvability	optimality	metrics
	A*	[Hart *et al.*, 1968]	complete	optimal	m, l
	OD	[Standley, 2010]	complete	sub-optimal	(greedy)
	ODrM*	[Wagner and Choset, 2015]	complete	optimal	m, l
	I-ODrM*	[Wagner and Choset, 2015]	complete	bnd. sub-opt.	m, l
	BCP	[Lam *et al.*, 2022]	solution cmp.	optimal	f
	CBS	[Sharon *et al.*, 2015]	solution cmp.	optimal	f, m, l
	EECBS	[Li *et al.*, 2021b]	solution cmp.	bnd. sub-opt.	f, m, l
	PP	[Silver, 2005]	incomplete	sub-optimal	
	LNS2	[Li *et al.*, 2022]	incomplete	sub-optimal	
	PIBT(+)	[Okumura *et al.*, 2022]	incomplete	sub-optimal	
	LaCAM	[Okumura, 2023]	complete	sub-optimal	
	LaCAM*	**this paper**	**complete**	**eventually opt.**	**m, l**

Figure 1: Performance on the MAPF benchmark. *upper:* The number of solved instances among 13,900 instances on 33 four-connected grid maps, retrieved from [Stern *et al.*, 2019]. The size of agents varies up to 1,000. '-f,' '-m,' and '-l' respectively mean that an algorithm tries to minimize flowtime, makespan, or sum-of-loss. The scores of LaCAM* are for initial solutions. *lower:* Representative or state-of-the-art MAPF algorithms. "solution cmp." means that an algorithm ensures to find solutions for solvable instances but it never identifies unsolvable ones. "bnd. sub-opt." means a bounded sub-optimal algorithm. Their sub-optimality was set to five.

1 Introduction

The *multi-agent pathfinding (MAPF)* problem aims to assign collision-free paths for multiple agents on a graph. To date, various MAPF algorithms have been developed, motivated by various applications such as warehouse automation [Wurman *et al.*, 2008]. Ideal MAPF algorithms will be complete, optimal, quick, and scalable. However, there is generally a trade-off between the former two and the latter two. Conversely, *the primary challenge of developments in MAPF algorithms is to guarantee solvability and solution quality, while suppressing planning efforts to secure speed and scalability*.

To break this tradeoff, we present two enhancements to the recently-developed algorithm called *LaCAM (lazy constraints addition search for MAPF)* [Okumura, 2023]. It is complete, sub-optimal, and search-based (akin to A* search) that uses lazy successor generation. The first enhancement is its anytime version called *LaCAM** that eventually converges to optima, provided that solution costs are accumulated transition costs. Since solving MAPF optimally is computationally intractable [Yu and LaValle, 2013], one practical approach to large instances is obtaining sub-optimal solutions and then refining their quality as time allows. LaCAM* meets such demands. The second enhancement is for the successor generation, i.e., tuning of the PIBT algorithm [Okumura *et al.*, 2022], so as to quickly obtain initial solutions.

With these enhancements, we empirically demonstrate that LaCAM* can break the trade-off. For instance, it sub-optimally solved 99% of the instances retrieved from the MAPF benchmark [Stern *et al.*, 2019] within ten seconds while guaranteeing the eventual optimality, on a standard desktop PC. As illustrated in Fig. 1, this result is beyond frontiers of existing MAPF algorithms. In what follows, we present preliminaries, LaCAM*, improved successor generation, empirical results, and discussion in order. The appendix and code are available at https://kei18.github.io/lacam2/.

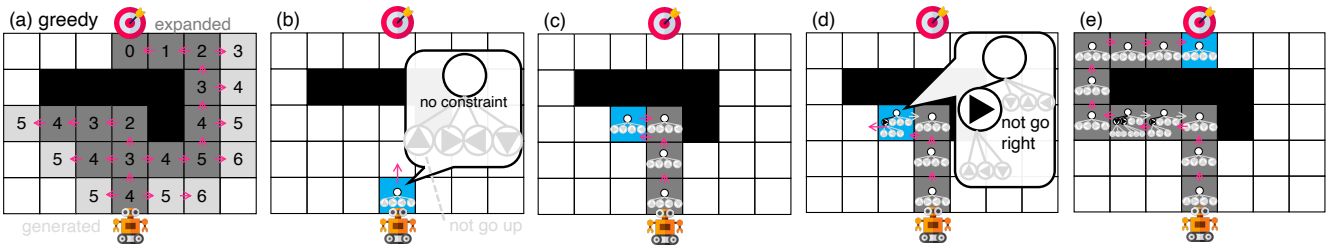


Figure 2: Illustration of LaCAM using single-agent grid pathfinding.

2 Preliminaries

2.1 Notation, Problem Definition, and Assumption

Notations. $S[k]$ denotes the k-th element of the sequence S, where the index starts at one. For convenience, we use $\bot$ as an "undefined" or "not found" sign.

Instance. An *MAPF instance* is defined by a graph $G = (V, E)$, a set of agents $A = \{1, \dots, n\}$, a tuple of distinct starts $\mathcal{S} = (s_i \in V)^{i \in A}$ and goals $\mathcal{G} = (g_i \in V)^{i \in A}$.

Configuration. A *configuration* is a tuple of locations for all agents. For instance, $\mathcal{Q} = (v_1, v_2, \dots, v_n) \in V^{|A|}$ is a configuration, where $\mathcal{Q}[i] = v_i$ is the location of agent $i \in A$. The start and goal configurations are $\mathcal{S}$ and $\mathcal{G}$, respectively.

Collision and Connectivity. A configuration $\mathcal{Q}$ has a *vertex collision* when there is a pair of agents $i, j \in A, i \neq j$, such that $\mathcal{Q}[i] = \mathcal{Q}[j]$. Two configurations X and Y have an *edge collision* when there is a pair of agents $i, j \in A, i \neq j$, such that $X[i] = Y[j] \land Y[i] = X[j]$. Let $\mathsf{neigh}(v)$ be a set of vertices adjacent to $v \in V$. Two configurations X and Y are *connected* when $Y[i] \in \mathsf{neigh}(X[i]) \cup \{X[i]\}$ for all $i \in A$, and, there are neither vertex nor edge collisions in X and Y.

Decision Problem. Given an MAPF instance, an *MAPF problem* is a decision problem that asks existence of a sequence of configurations $\Pi = (\mathcal{Q}_0, \mathcal{Q}_1, \dots, \mathcal{Q}_k)$, such that $\mathcal{Q}_0 = \mathcal{S}$, $\mathcal{Q}_k = \mathcal{G}$, and any two consecutive configurations in Π are connected. A *solution* to MAPF is a Π that satisfies the conditions. To align with the literature, the index of Π exceptionally starts at zero (i.e., $\Pi[0] = \mathcal{Q}_0$). An algorithm is said to be *complete* when it returns solutions for solvable instances and reports non-existence for unsolvable instances. Otherwise, it is called *incomplete*.

Optimization Problems. Given a transition (or edge) cost between two configurations, $\mathsf{cost}_e : V^{|A|} \times V^{|A|} \mapsto \mathbb{R}_{\geq 0}$, we aim at minimizing accumulated transition costs of a solution Π, denoted as $\mathsf{cost}(\Pi) := \sum_{t=0}^{k-1} \mathsf{cost}_e\big(\Pi[t], \Pi[t+1]\big)$. This notation can represent various optimization metrics. For instance, cost is called *makespan* when $\mathsf{cost}_e(X, Y) := 1$. It is called *sum-of-fuels* (aka. *total travel distance*) when $\mathsf{cost}(X, Y) := |\{i \in A; X[i] \neq Y[i]\}|$. *Sum-of-loss* counts actions of non-staying at goals, defined by $\mathsf{cost}_e(X, Y) := |\{i \in A \mid \neg(X[i] = Y[i] = g_i)\}|$. A solution Π is *optimal* when there is no solution Π' such that $\mathsf{cost}(\Pi') < \mathsf{cost}(\Pi)$.

Remarks for Flowtime. Another common metric of MAPF is *flowtime* (aka. *sum-of-costs*): $\sum_{i \in A} t_i$, where $t_i \leq k$ is the earliest timestep such that $\Pi[t_i][i] = \Pi[t_i + 1][i] = \dots = \Pi[k][i] = g_i$. The difference from sum-of-loss is cost contribution of agents who once reach their goal and leave there temporarily. The flowtime is history-dependent on paths of each agent; hence, it is impossible to represent as it is with accumulative costs.[1] Instead, this paper considers sum-of-loss as seen in [Standley, 2010; Wagner and Choset, 2015].

Admissible Heuristic. We assume an *admissible heuristic* $\mathsf{h} : V^{|A|} \mapsto \mathbb{R}_{\geq 0}$, such that $\mathsf{h}(Q)$ is always the optimal cost from $\mathcal{Q}$ to $\mathcal{G}$ or less; e.g., $\mathsf{h}(\mathcal{Q}) := \sum_{i \in A} \mathsf{dist}(\mathcal{Q}[i], g_i)$ is available for the sum-of-{loss, fuels}, where $\mathsf{dist} : V \times V \mapsto \mathbb{N}_{\geq 0}$ is the shortest path length on G.

Understanding MAPF as Graph Pathfinding. Using configurations, consider a graph H comprising vertices that represent configurations, and edges that represent the connectivity of configurations. Then, by regarding $\mathcal{S}$ and $\mathcal{G}$ as start and goal vertices respectively, *optimal MAPF is equivalent to the shortest pathfinding problem on* H. This is a key perspective to understand LaCAM$^{(*)}$.

2.2 LaCAM

LaCAM [Okumura, 2023] was originally developed as a suboptimal complete MAPF algorithm. In a nutshell, *it is a graph pathfinding algorithm*, like A*, but some parts are specific to MAPF. Below, we provide the essence of the graph pathfinding part, using an example of single-agent grid pathfinding. The details of the MAPF-specific part are delivered in the appendix.

Classical Search. See Fig. 2a that illustrates how a typical search scheme solves grid pathfinding. Specifically, we show the greedy best-first search with a heuristic of the Manhattan distance. Here, a location of the agent corresponds to a *configuration* (aka. *state*) of the search. From the start configuration, the search generates three successor nodes: left, up, and right. Each node corresponds to one configuration. The search then takes one of the generated nodes according to the heuristic, and generates successors. This procedure continues until finding the goal configuration.

Branching Factor. Consider how many nodes are generated to estimate the search effort. Though the solution length is 8, 22 nodes are generated. This number is related to the number of connected configurations (i.e., *branching factor*). It is four in grid pathfinding, therefore, the number of node

[1] However, it is worth noting that flowtime can be defined by introducing virtual goals, where once an agent has arrived there, it cannot move anywhere in the future.

generations remains acceptable. However, the branching factor of MAPF is exponential for the number of agents. Consequently, the generation itself becomes intractable. This is why a vanilla A* is hopeless to solve large MAPF instances.

Configuration Generator. LaCAM tries to relieve this huge-branching-factor issue when a *configuration generator* is available. Given a configuration and *constraints*, it generates a connected configuration following constraints. Constraints should be embodied by each domain. In this example, consider a constraint as a prohibition of direction, such as *not* moving up, left, right, or down.

Constraint Tree. Each search node of LaCAM contains not only a configuration but also constraints, taking the form of a tree structure. For each node invoke, LaCAM gradually develops the tree by *low-level search*, implemented by, e.g., breadth-first search (BFS). A node on the tree has a constraint and represents several constraints by tracing a path to the root.

Search Flow. We now explain LaCAM using Fig. 2b–e, with a depth-first search (DFS) style. The attempt to find a sequence of configurations is called *high-level search*. At first, a search node of the start is examined (Fig. 2b). The node poses no constraints for the first invoke, meaning that the configuration generator can output any connected configuration. Suppose that the generator outputs an "up" configuration, following the Manhattan distance guide, illustrated by the pink arrow. Preparing for the second invoke, the node expands a constraint tree with new constraints (e.g., "not go up"). The high-level search does not discard the examined node immediately, rather, it discards when all connected configurations have been generated (i.e., when the low-level search completes). Next, Fig. 2c–d show an example of the second invoke of nodes. In Fig. 2c, the generator outputs an already-known configuration. Since this example assumes DFS, LaCAM examines the blue-colored node again in Fig. 2d. This time, the generator must follow a constraint "not go right" and its parent "no constraint." The example then generates a "left" configuration. The search continues until finding the goal (Fig. 2e) and obtains a solution path by backtracking.

Adaptation to MAPF. With appropriate designs of constraints and tree construction, LaCAM can be an exhaustive search and guarantees completeness for graph pathfinding problems. In [Okumura, 2023], such an example is shown for MAPF by letting a constraint specify which agent is where in the next configuration. Moreover, LaCAM can greatly decrease the number of node generations if the configuration generator is promising in outputting configurations that are close to the goal. This reduction could be a silver bullet to achieve quick planning, especially in planning problems where the branching factor is huge like MAPF. The remaining question is a realization of good configuration generators. In the original paper, *PIBT (priority inheritance with backtracking)* [Okumura *et al.*, 2022] served as it, explained in Sec. 2.3.

Pseudocode. Algorithm 1 shows DFS-based LaCAM. Each search node $\mathcal{N}$ stores *(i)* a configuration, *(ii)* a constraint tree embodied by a queue (assuming BFS) and *(iii)* a pointer to a parent node (see Line 2). Nodes are stored in an *Open* list and *Explored* table, akin to general search schema, and are processed one by one. We abstract how to create constraint trees for MAPF by Line 9, which is elaborated in the appendix.

Algorithm 1 LaCAM. $\mathcal{C}_{\text{init}}$ means "no constraint."

input: MAPF instance
output: solution or `NO_SOLUTION`

1: initialize *Open*, *Explored* ▷ *stack, hash table*
2: $\mathcal{N}^{\text{init}} \leftarrow \langle\, config : \mathcal{S}, tree : [\![\, \mathcal{C}^{\text{init}} \,]\!], parent : \bot \,\rangle$
3: *Open*.push $(\mathcal{N}^{\text{init}})$; *Explored*$[\mathcal{S}] = \mathcal{N}^{\text{init}}$
4: **while** *Open* $\neq \emptyset$ **do**
5: $\quad \mathcal{N} \leftarrow$ *Open*.top()
6: $\quad$ **if** $\mathcal{N}.config = \mathcal{G}$ **then return** backtrack$(\mathcal{N})$
7: $\quad$ **if** $\mathcal{N}.tree = \emptyset$ **then** *Open*.pop(); **continue** ▷ *discard node*
8: $\quad \mathcal{C} \leftarrow \mathcal{N}.tree$.pop() ▷ *get constraint*
9: $\quad$ low_level_expansion$(\mathcal{N}, \mathcal{C})$ ▷ *proceed low-level search*
10: $\quad \mathcal{Q}^{\text{new}} \leftarrow$ configuration_generator$(\mathcal{N}, \mathcal{C})$
11: $\quad$ **if** $\mathcal{Q}^{\text{new}} = \bot$ **then continue** ▷ *generator may fail*
12: $\quad$ **if** *Explored* $[\mathcal{Q}^{\text{new}}] \neq \bot$ **then continue**
13: $\quad \mathcal{N}^{\text{new}} \leftarrow \langle\, config : \mathcal{Q}^{\text{new}}, tree : [\![\, \mathcal{C}^{\text{init}} \,]\!], parent : \mathcal{N} \,\rangle$
14: $\quad$ *Open*.push $(\mathcal{N}^{\text{new}})$; *Explored* $[\mathcal{Q}^{\text{new}}] = \mathcal{N}^{\text{new}}$
15: **return** `NO_SOLUTION`

Algorithm 2 PIBT

input: configuration $\mathcal{Q}^{\text{from}}$, agents A, goals $(g_1, \dots g_n)$
output: configuration $\mathcal{Q}^{\text{to}}$ (each element is initialized with $\bot$)

1: **for** $i \in A$ **do**; **if** $\mathcal{Q}^{\text{to}}[i] = \bot$ **then** PIBT(i)
2: **return** $\mathcal{Q}^{\text{to}}$
3: **procedure** PIBT(i) ▷ *return* `VALID` *or* `INVALID`
4: $\quad C \leftarrow$ neigh $\left(\mathcal{Q}^{\text{from}}[i]\right) \cup \left\{\mathcal{Q}^{\text{from}}[i]\right\}$ ▷ *candidate vertices*
5: $\quad$ sort C in ascending order of dist(u, g_i) where $u \in C$
6: $\quad$ **for** $v \in C$ **do**
7: $\quad\quad$ **if** collisions in $\mathcal{Q}^{\text{to}}$ supposing $\mathcal{Q}^{\text{to}}[i] = v$ **then continue**
8: $\quad\quad \mathcal{Q}^{\text{to}}[i] \leftarrow v$
9: $\quad\quad$ **if** $\exists j \in A$ s.t. $j \neq i \wedge \mathcal{Q}^{\text{from}}[j] = v \wedge \mathcal{Q}^{\text{to}}[j] = \bot$ **then**
10: $\quad\quad\quad$ **if** PIBT$(j) =$ `INVALID` **then continue**
11: $\quad\quad$ **return** `VALID` ▷ *assignment done*
12: $\quad \mathcal{Q}^{\text{to}}[i] \leftarrow \mathcal{Q}^{\text{from}}[i]$; **return** `INVALID`

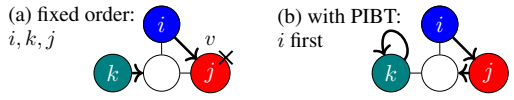


Figure 3: Concept of PIBT. $\mathcal{Q}^{\text{from}}$ is illustrated. Bold arrows represent assignments of $\mathcal{Q}^{\text{to}}$. *(a)* Consider a fixed assignment order of i, k, and j. If i and k are assigned following the illustrated arrows, j has no candidate vertex as $\mathcal{Q}^{\text{to}}[j]$ (annotated with $\times$). *(b)* This pitfall is overcome by doing the assignment for j prior to k, reacting to i's assignment request.

2.3 PIBT

PIBT [Okumura *et al.*, 2022] was originally developed to solve MAPF iteratively. In a nutshell, *it is a configuration generator, which generates a new connected configuration ($\mathcal{Q}^{to}$), given another ($\mathcal{Q}^{from}$) as input*. By continuously generating configurations, PIBT can generate a solution for MAPF.

Concept. To determine $\mathcal{Q}^{\text{to}}$, PIBT sequentially assigns a vertex to each agent while avoiding assignments that trigger collisions. This assignment order adaptively changes. Specif-

ically, before fixing $\mathcal{Q}^{\text{to}}[i]$ to $v \in \mathsf{neigh}(\mathcal{Q}^{\text{from}}[i]) \cup \{\mathcal{Q}^{\text{from}}[i]\}$, PIBT first checks the existence of $j \in A$ such that $\mathcal{Q}^{\text{from}}[j] = v$. If such j exists, j may have no candidate vertex for $\mathcal{Q}^{\text{to}}[j]$, due to collision avoidance (see Fig. 3a). Therefore, PIBT next determines $\mathcal{Q}^{\text{to}}[j]$ prior to locations of other agents (e.g., k in Fig. 3). This scheme is called *priority inheritance*. If $\mathcal{Q}^{\text{to}}[j]$ is successfully assigned, $\mathcal{Q}^{\text{to}}[i]$ is fixed to v (Fig. 3b); otherwise, $\mathcal{Q}^{\text{to}}[i]$ needs to take another vertex other than v.

Pseudocode. Algorithm 2 implements the concept above, by recursively calling a procedure PIBT (Line 3–), which takes an agent i and eventually assigns $\mathcal{Q}^{\text{to}}[i]$. The assignment attempts of Lines 6–11 are performed for candidate vertices regarding $\mathcal{Q}^{\text{from}}[i]$, in ascending order of the distance toward g_i. The attempts continue until $\mathcal{Q}^{\text{to}}[i]$ is determined, resulting in `VALID` outcome (Line 11). When all attempts failed, PIBT(i) assigns $\mathcal{Q}^{\text{from}}[i]$ to $\mathcal{Q}^{\text{to}}[i]$ and returns `INVALID` (Line 12). Priority inheritance is triggered as necessary, taking the form of calling PIBT for another agent j (Line 10). The success of priority inheritance triggers receipt of `VALID`; otherwise, `INVALID` is backed, and then i continues the assignment attempts.

Dynamic Priorities. In addition to priority inheritance, PIBT prioritizes assignments for agents that are not on their goal, which is done by sorting A of Line 1 in that way. This scheme is convenient for lifelong scenarios wherein all agents are not necessarily being goals simultaneously.

3 LaCAM*: Eventually Optimal Algorithm

Algorithm 3 presents LaCAM*. The same lines as LaCAM (Alg. 1) are grayed out. The blue-colored lines are not necessary from the theoretical side but are effective in speeding up the search. The main differences from LaCAM are two: *(i)* it continues the search when finding the goal configuration $\mathcal{G}$, and, *(ii)* it rewrites parent relations between search nodes as necessary. For convenience, the transition cost cost_e and admissible heuristic h can take nodes as arguments, instead of configurations. Below, the updated parts are explained.

Keeping Goal Node. LaCAM* retains the goal node $\mathcal{N}^{\text{goal}}$, rather than immediately returning solutions when first finding the goal $\mathcal{G}$ (Line 6). The search terminates when there is no remaining node in *Open*; otherwise, there is an interruption from users such as timeout (Line 4). A solution is then constructed by backtracking from $\mathcal{N}^{\text{goal}}$ (Lines 27–28). Doing so makes LaCAM* an anytime algorithm, that is, *after finding the goal node, it is interruptible whenever a solution is required, while gradually refining solution quality as time allows*.

Search Node Ingredients. Each high-level node contains a set *neigh* that stores connected configurations (i.e., nodes) and g-value that represents cost-to-come from the start $\mathcal{S}$ (Line 2). They are initialized and updated appropriately when finding a new configuration (Lines 24–26).

Updating Parents and Costs. To maintain the optimality, when finding an already known configuration, LaCAM* updates *neigh* (Line 14). This is followed by updates of g-value and *parent*, performed by Dijkstra's algorithm (Lines 15–21). Figure 4 illustrates the update.

Algorithm 3 LaCAM*. $\mathcal{C}_{\text{init}}$ means "no constraint."

input: MAPF instance, edge cost cost_e, admissible heuristic h
output: solution, `NO_SOLUTION`, or `FAILURE`
notation: $\mathsf{f}(\mathcal{N}) := \mathcal{N}.g + \mathsf{h}(\mathcal{N})$; $\spadesuit := \left(\mathcal{N}^{\text{goal}} \neq \bot\right)$

1: initialize *Open*, *Explored*; $\mathcal{N}^{\text{goal}} \leftarrow \bot$
2: $\mathcal{N}^{\text{init}} \leftarrow \langle config : \mathcal{S}, tree : [\![\mathcal{C}^{\text{init}}]\!], parent : \bot, neigh : \emptyset, g : 0 \rangle$
3: *Open*.push $\left(\mathcal{N}^{\text{init}}\right)$; *Explored*$[\mathcal{S}] = \mathcal{N}^{\text{init}}$
4: **while** *Open* $\neq \emptyset \ \wedge \ \neg$interrupt() **do**
5: $\quad \mathcal{N} \leftarrow$ *Open*.top()
6: $\quad$ **if** $\mathcal{N}.config = \mathcal{G}$ **then** $\mathcal{N}^{\text{goal}} \leftarrow \mathcal{N}$
7: $\quad$ **if** $\spadesuit \wedge \mathsf{f}\left(\mathcal{N}^{\text{goal}}\right) \leq \mathsf{f}\left(\mathcal{N}\right)$ **then** *Open*.pop(); **continue**
8: $\quad$ **if** $\mathcal{N}.tree = \emptyset$ **then** *Open*.pop(); **continue**
9: $\quad \mathcal{C} \leftarrow \mathcal{N}.tree$.pop()
10: $\quad$ low_level_expansion$(\mathcal{N}, \mathcal{C})$
11: $\quad \mathcal{Q}^{\text{new}} \leftarrow$ configuration_generator$(\mathcal{N}, \mathcal{C})$
12: $\quad$ **if** $\mathcal{Q}^{\text{new}} = \bot$ **then continue**
13: $\quad$ **if** *Explored* $[\mathcal{Q}^{\text{new}}] \neq \bot$ **then**
14: $\quad\quad \mathcal{N}.neigh$.append $($*Explored* $[\mathcal{Q}^{\text{new}}])$
15: $\quad\quad \mathcal{D} \leftarrow [\![\mathcal{N}]\!]$ ▷ *Dijkstra, priority queue of g-value*
16: $\quad\quad$ **while** $\mathcal{D} \neq \emptyset$ **do**
17: $\quad\quad\quad \mathcal{N}^{\text{from}} \leftarrow \mathcal{D}$.pop()
18: $\quad\quad\quad$ **for** $\mathcal{N}^{\text{to}} \in \mathcal{N}^{\text{from}}.neigh$ **do**
19: $\quad\quad\quad\quad g \leftarrow \mathcal{N}^{\text{from}}.g + \mathsf{cost}_e\left(\mathcal{N}^{\text{from}}, \mathcal{N}^{\text{to}}\right)$
20: $\quad\quad\quad\quad$ **if** $g < \mathcal{N}^{\text{to}}.g$ **then**
21: $\quad\quad\quad\quad\quad \mathcal{N}^{\text{to}}.g \leftarrow g; \mathcal{N}^{\text{to}}.parent \leftarrow \mathcal{N}^{\text{from}}; \mathcal{D}.\mathsf{push}(\mathcal{N}^{\text{to}})$
22: $\quad\quad\quad\quad\quad$ **if** $\spadesuit \wedge \mathsf{f}\left(\mathcal{N}^{\text{to}}\right) < \mathsf{f}\left(\mathcal{N}^{\text{goal}}\right)$ **then** *Open*.push $\left(\mathcal{N}^{\text{to}}\right)$
23: $\quad$ **else**
24: $\quad\quad \mathcal{N}^{\text{new}} \leftarrow \langle config : \mathcal{Q}^{\text{new}}, tree : [\![\mathcal{C}^{\text{init}}]\!], parent : \mathcal{N}, neigh : \emptyset, g : \mathcal{N}.g + \mathsf{cost}_e\left(\mathcal{N}, \mathcal{Q}^{\text{new}}\right) \rangle$
25: $\quad\quad$ *Open*.push $\left(\mathcal{N}^{\text{new}}\right)$; *Explored* $[\mathcal{Q}^{\text{new}}] = \mathcal{N}^{\text{new}}$
26: $\quad\quad \mathcal{N}.neigh$.append $\left(\mathcal{N}^{\text{new}}\right)$
27: **if** $\spadesuit \wedge$ *Open* $= \emptyset$ **then return** backtrack $\left(\mathcal{N}^{\text{goal}}\right)$ ▷ *optimal*
28: **else if** $\spadesuit$ **then return** backtrack $\left(\mathcal{N}^{\text{goal}}\right)$ ▷ *sub-optimal*
29: **else if** *Open* $= \emptyset$ **then return** `NO_SOLUTION`
30: **else return** `FAILURE`

Discarding Redundant Nodes. Once the goal node is found, LaCAM* discards nodes that do not contribute to improving solution quality (Line 7). It also revives nodes as necessary when their g-values are updated (Line 22).

Theorem 1. *LaCAM* (Alg. 3) is complete and optimal.*

Proof. Consider Alg. 3 without blue lines (Lines 7 and 22); they just speed up the search without breaking the optimal search structure. In this proof, the term "path" refers to a sequence of connected configurations.

First, we introduce a directed graph H, where its vertex corresponds to a configuration. Initially, H has only a start vertex $\mathcal{S}$. Then, the search iterations gradually develop H. When a new node is created (Line 24), its configuration is added to H. An arc (X, Y) of H occurs when the search finds a connection from X to Y, i.e., $\mathcal{N}^Y \in \mathcal{N}^X.neigh$.

We now prove that: *(♣) for any configuration $\mathcal{Q}$ in H, a path from $\mathcal{S}$ to $\mathcal{Q}$ constructed by backtracking (i.e., by following $\mathcal{N}.parent$) is the shortest path in H, regarding accumulative transition costs, at the beginning of each search iteration*. This is proven by induction. Initially, H comprises only $\mathcal{S}$, satisfying ♣. Assume now that ♣ is satisfied in the previous iteration of Lines 4–26. In the next iteration, H is

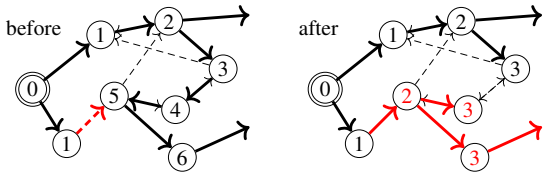


Figure 4: Updating parents and costs. Each circle is a search node (i.e., configuration), including its g-value of makespan. Arrows represent known neighboring relations. Among them, solid lines represent *parent*. The updated parts are red-colored. *left*: A new neighbor relationship, a red dashed arrow, has been found. *right*: Rewrite the search tree. Observe that the rewriting occurs in a limited part of the tree due to g-value pruning (Line 20).

updated by: *(i)* generating a new configuration (Lines 24–26), or *(ii)* finding a known configuration (Lines 14–21). Case *(i)* holds ♣ because only a new vertex and an arc toward the vertex are added to H. Case *(ii)* holds ♣ because only an arc is added for the node $\mathcal{N}$ (Line 14), and then Lines 15–21 perform Dijkstra's algorithm starting from $\mathcal{N}$ that maintains the tree structure of the shortest paths.

The search space of LaCAM* is finite (see [Okumura, 2023]); the search terminates in finite time. Each node eventually examines all connected configurations. Consequently, when terminated, H includes all possible paths from $\mathcal{S}$ to $\mathcal{G}$ for solvable instances. Together with ♣, LaCAM* returns an optimal solution, otherwise, reports the non-existence. □

Implementation Tips. Following [Okumura, 2023], when finding an already known configuration at Line 13, our implementation reinserts the corresponding node to *Open*. Moreover, with a small probability (e.g., 0.1%), the implementation reinserts a node of $\mathcal{S}$ instead of the found one. Doing so enables the search to "escape" from configurations being bottlenecks. Such techniques relying on non-determinism have been seen in other search problems [Kautz *et al.*, 2002] and MAPF studies [Cohen *et al.*, 2018b; Andreychuk and Yakovlev, 2018]. Indeed, we informally observed that this random replacement slightly improved the success rate. Note that the optimality still holds with these modifications.

4 Improving Configuration Generator

The performance of LaCAM heavily relies on a configuration generator, therefore, the development of good generators is critical. The implementation in [Okumura, 2023] uses a vanilla PIBT of Alg. 2, resulting in poor performances in several scenarios, especially in instances with narrow corridors. This is because PIBT itself often fails such scenarios, hence being an ineffective guide for LaCAM. This section elaborates on this phenomenon and presents an improved version.

4.1 Failure Analysis of PIBT

As seen in Sec. 2.3, PIBT sequentially assigns the next locations for agents. Since this order prioritizes agents being not on their goals, livelock situations might be triggered. See Fig. 5; two agents reach their goal vertex periodically but PIBT never reaches the goal configuration.

Algorithm 4 procedure PIBT with swap

1: $C \leftarrow \mathsf{neigh}\left(\mathcal{Q}^{\text{from}}[i]\right) \cup \left\{\mathcal{Q}^{\text{from}}[i]\right\}$
2: sort C in increasing order of $\mathsf{dist}(u, g_i)$ where $u \in C$
3: $j \leftarrow \mathsf{swap\_required\_and\_possible}\left(i, C[1], \mathcal{Q}^{\text{from}}\right)$
4: **if** $j \neq \bot$ **then** reverse C
5: **for** $v \in C$ **do**
6: Lines 7–10 of Alg. 2
7: **if** $v = C[1] \wedge j \neq \bot \wedge \mathcal{Q}^{\text{to}}[j] = \bot$ **then** $\mathcal{Q}^{\text{to}}[j] \leftarrow \mathcal{Q}^{\text{from}}[i]$
8: **return** VALID
9: $\mathcal{Q}^{\text{to}}[i] \leftarrow \mathcal{Q}^{\text{from}}[i]$; **return** INVALID

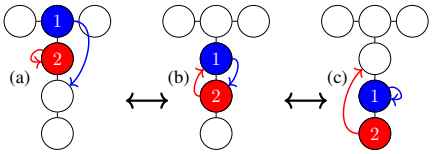


Figure 5: Failure example of PIBT. Goals are represented by arrows.

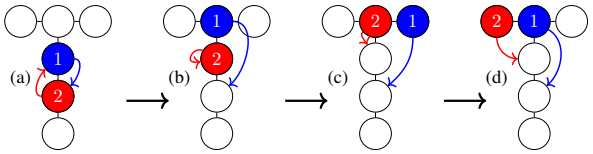
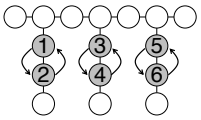


Figure 6: Swap operation. The last two steps are omitted because of just moving two agents toward their goal.



	$\lvert A\rvert$=2	$\lvert A\rvert$=4	$\lvert A\rvert$=6
w/Alg. 2	128	23,907	287,440
w/Alg. 4	6	8	8

Table 1: The number of search iterations of LaCAM to solve the instances. When $|A| = 2$, only agents-$\{1, 2\}$ appear, and so forth.

LaCAM can break such livelocks by posing constraints. However, it may require significant effort because appropriate combinations of constraints should be explored. Even worse, with more agents, the search effort dramatically increases, as demonstrated in Table 1.

4.2 Enhancing PIBT by Swap

Livelocks in PIBT can be resolved by *swap* operation, originally developed in rule-based MAPF algorithms [Luna and Bekris, 2011; De Wilde *et al.*, 2014]. In short, this operation swaps locations of two agents using a vertex with a degree of three or more. Figure 6 shows an example. Here, we extract its essence and incorporate it into PIBT. Specifically, this is done by adjusting vertex scoring at Line 5 in Alg. 2.

Algorithm 4 extends the PIBT procedure of Alg. 2. The modification is simple; if an agent i and neighboring agent j are judged to require swapping locations (Line 3), i reverses the order of candidate vertices C (Line 4); that is, i tries to be apart from g_i. Then, if i successfully moves to the first vertex in the candidates C, i *pulls* j to the current occupying vertex (Line 7). With an appropriate implementation of the function swap_required_and_possible, PIBT does not fall into the livelock of Fig. 5, rather, it can generate a sequence

of configurations shown in Fig. 6.

The swap_required_and_possible function is a pattern detector and implementation-depending. *We do not aim at designing complete detectors because pitfalls of the detectors can be complemented by LaCAM*. However, a well-tuned implementation can relax the search effort of LaCAM. Below, we illustrate our example implementation, while omitting tiny fine-tunings.

Pattern Detector Implementation

Assume that the detector is called for i. Assume further that another agent j is on $C[1]$ for i at Line 2 of Alg. 4, such that the degree is 2 or less. Our detector uses two emulations.

The first emulation asks about the necessity of the swap. This is done by continuously moving i to j's location while moving j to another vertex not equal to i's location, ignoring the other agents. The emulation stops in two cases: *(i)* The swap is *not required* when j's location has a degree of more than two. *(ii)* The swap is *required* when j's location has a degree of one, or, when i reaches g_i while j's nearest neighboring vertex toward its goal is g_i.

If the swap is required, the second emulation asks about the possibility of the swap. This is done by reversing the emulation direction; that is, continuously moving j to i's location while moving i to another vertex. It stops in two cases: *(i)* The swap is *possible* when i's location has a degree of more than two. *(ii)* The swap is *impossible* when i is on a vertex with degree of one.

The function returns j when the swap is required and possible. For instance, in configurations of Fig. 6(a,b), the swap is required for both agents. However, the swap is possible only for agent-1, then, the order of candidate vertices is reversed for agent-1 (Line 4). Consequently, Alg. 4 generates configurations of Fig. 6(b,c). An exception is the case of Fig. 6c, where agent-2 needs to reverse its candidates to generate a configuration of Fig. 6d. We note that this case is also possible to be detected by applying the two emulations.

5 Evaluation

This section empirically assesses the two improvements, comprising: *(i)* how the improved configuration generator reduces planning effort, *(ii)* how LaCAM* refines solution, *(iii)* how discarding redundant nodes speeds up the convergence, *(iv)* evaluation with small complicated instances, *(v)* evaluation with the MAPF benchmark, *(vi)* comparison with another anytime MAPF algorithm, and *(vii)* evaluation with extremely dense scenarios.

Setup. The experiments were run on a desktop PC with Intel Core i7-7820X 3.6 GHz CPU and 32 GB RAM. A maximum of 16 different instances were run in parallel using multi-threading. LaCAM* was coded in C++. All experiments used four-connected grid maps retrieved from the MAPF benchmark [Stern *et al.*, 2019]. Unless mentioned, this section uses a timeout of 30 s for solving MAPF. Baseline MAPF algorithms are summarized in Fig. 1. Their implementation details are available in the appendix.

	search iterations		runtime (ms)	
$\|A\|$	w/Alg. 2	w/Alg. 4	w/Alg. 2	w/Alg. 4
100	374 (344,54468)	366 (338,401)	65 (31,1218)	112 (34,216)
300	54802 (388,369131)	392 (357,482)	3049 (291,18858)	301 (187,409)
500	181459 (44534,268724)	410 (391,432)	18063 (4598,29820)	500 (347,574)

Table 2: Effect of configuration generators. For each $|A|$, median, min, and max scores are presented for instances solved by both algorithms among 25 instances retrieved from [Stern *et al.*, 2019], on *warehouse-20-40-10-2-1*, illustrated in Fig. 8.

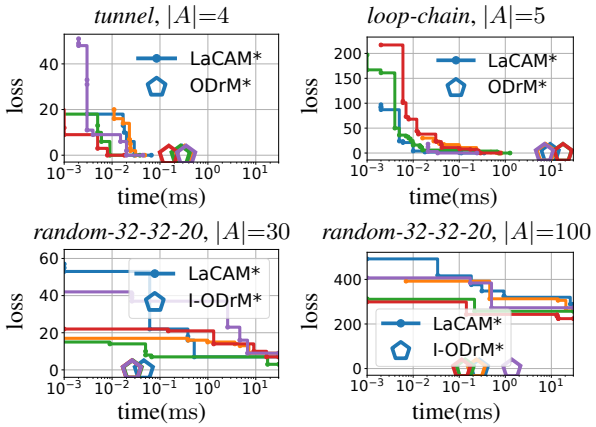


Figure 7: Refinement of LaCAM*. Three maps were used, shown in Table 4 and Fig. 8. For each chart, five identical instances were used where starts and goals were set randomly. The optimization was for sum-of-loss. "loss" shows the gaps from scores of (I-)ODrM*. In *random-32-32-20*, the bounded sub-optimal version with suboptimality of 1.5 was used because ODrM* failed to solve the instances. LaCAM* used Alg. 2 as a configuration generator.

	tree	*corners*	*tunnel*	*string*	*loop-chain*	*connector*
no discard	10K	2M	410M	19M	N/A	N/A
w/Alg. 2	1K	28K	287K	103K	N/A	N/A
w/Alg. 4	**1K**	**1K**	**199K**	**103K**	N/A	N/A

Table 3: The number of search iterations for termination. "no discard" is blue-lines omitted version of LaCAM*. The metric was for makespan. The instances are displayed in Table 4. In the last two instances, LaCAM* did not terminate before the timeout.

Effect of Improved Configuration Generator. Table 1 presents the number of search iterations of LaCAM on an instance that requires "swap," using a vanilla PIBT (Alg. 2) and the improved one (Alg. 4) as a configuration generator. Table 2 further compares the generators with larger instances. The results show that Alg. 4 dramatically reduced the search iterations of LaCAM, contributing to smaller computation time in large instances. Note however that the pattern detector has runtime overhead, as seen in $|A| = 100$ of Table 2.

Refinement of LaCAM*. Figure 7 shows how LaCAM* refines solutions. As baselines, we used scores of a complete and optimal algorithm called (I-)ODrM* [Wagner and Choset, 2015]. In the small instances, LaCAM* quickly

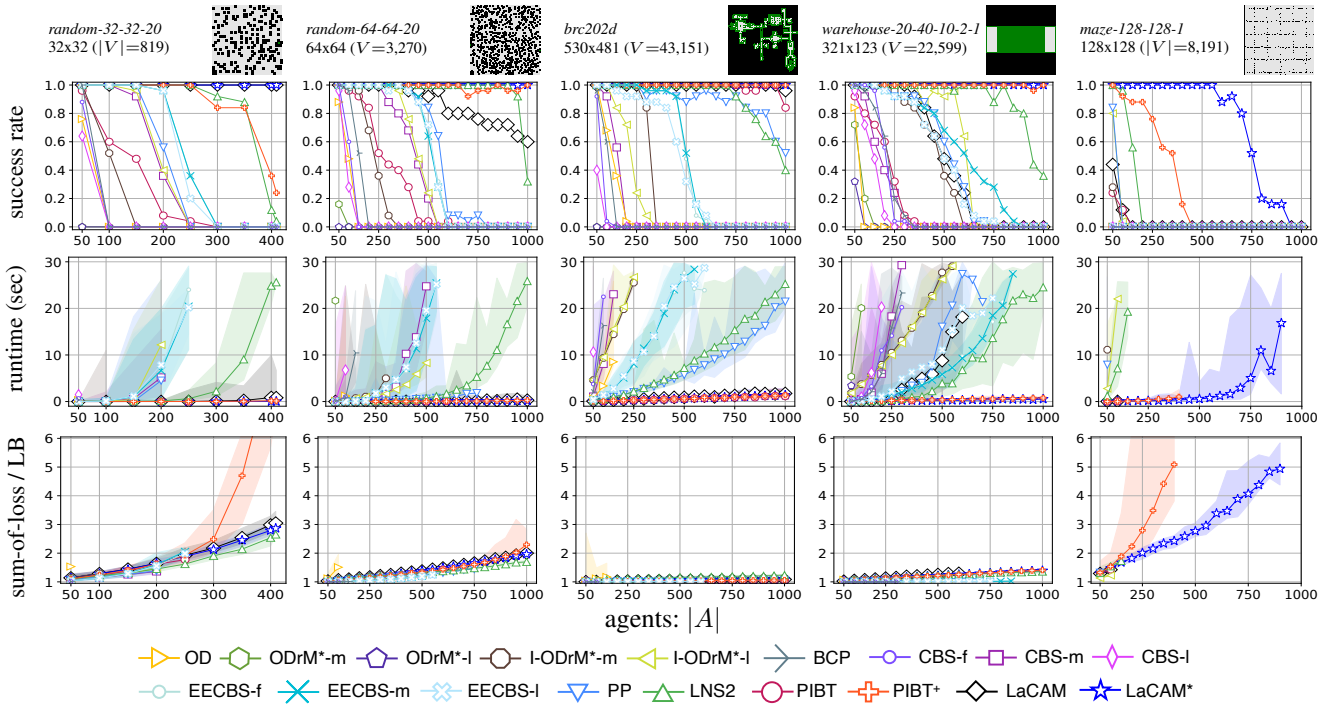


Figure 8: Results of the MAPF benchmark. Scores of sum-of-loss are normalized by $\sum_{i \in A} \mathsf{dist}(s_i, g_i)$. For runtime and sum-of-loss, median, min, and max scores of solved instances within each solver are displayed. Scores of LaCAM* are from initial solutions.

unit of time: ms	*tree*		*corners*		*tunnel*		*string*		*loop-chain*		*connector*		
	time	s-opt	time	s-opt	time	s-opt	time	s-opt	time	s-opt	time	s-opt	solved
LaCAM*	0	1.20	0	1.23	0	1.41	0	1.81	2	6.58	0	1.62	**6/6**
after 1 s	0	1.00	2	1.00	6	1.00	7	1.00	578	1.35	226	1.00	
A*	0	1.00	0	1.00	30	1.00	27	1.00	11125	1.00	N/A	N/A	5/6
ODrM*	5	1.00	2	1.00	396	1.00	402	1.00	N/A	N/A	N/A	N/A	4/6
I-ODrM*	1	1.00	0	1.50	70	1.07	2	1.25	N/A	N/A	N/A	N/A	4/6
CBS	71	1.00	0	1.00	N/A	N/A	149	1.00	N/A	N/A	N/A	N/A	3/6
EECBS	2	1.00	1	1.00	N/A	N/A	0	1.00	N/A	N/A	N/A	N/A	3/6
OD	0	1.00	0	1.88	14	2.73	0	1.25	2133	31.22	5	1.50	**6/6**
LaCAM	0	1.17	1	2.12	92	2.00	0	2.25	55	17.83	0	1.56	**6/6**
PP	N/A	N/A	0	1.00	N/A	N/A	0	1.00	N/A	N/A	N/A	N/A	2/6
LNS2	N/A	N/A	0	1.00	N/A	N/A	0	1.00	N/A	N/A	29	1.00	3/6
PIBT	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0/6
PIBT+	0	3.50	0	1.25	0	4.07	0	2.12	N/A	N/A	0	1.81	5/6
BCP	194	-	150	-	N/A	-	117	-	N/A	-	N/A	-	3/6

Table 4: Results of the small complicated instances. "s-opt" is makespan normalized by optimal ones. The minimum is one. The sum-of-loss version appears in the appendix. Two rows show results of LaCAM*: *(i)* scores for initial solutions and *(ii)* solution quality at 1 s and the runtime when that solution was obtained; they are an average of 10 trials with different random seeds. Algorithms are categorized into LaCAM*, those optimizing makespan, sub-optimal ones, and BCP optimizing another metric (i.e., flowtime).

found initial solutions and converged to optimal ones. Meanwhile, the convergence speed was slow in large instances with many agents. This is due to finding new connections between known configurations becoming rare, hence reducing the chance of rewriting the search tree.

Effect of Discarding Redundant Nodes. Table 3 shows how discarding redundant search nodes (blue lines of Alg. 3) affects the search to identify optimal solutions. Regardless of the generators, the discarding dramatically reduced the search effort. The reduction was larger with Alg. 4 because initial solutions can be found with smaller search iterations than Alg. 2. Note that without the discarding, the numbers of search iterations are equivalent between Alg. 2 and Alg. 4 because the search spaces are identical. In the remaining, LaCAM* uses Alg. 4 while LaCAM denotes the original implementation that uses Alg. 2.

Small Complicated Instances. Table 4 shows the results of LaCAM* with instances retrieved from [Luna and Bekris, 2011]. Overall, it immediately found not only initial solutions but also (near-)optimal ones. In contrast, the baselines failed some instances or returned low-quality solutions.

MAPF Benchmark. We tested LaCAM* on the MAPF benchmark that includes 33 maps, each having 25 "random scenarios" which specify start-goal pairs. From each scenario, we extracted instances by increasing the number of agents by 50 up to the maximum (1,000 in most cases) and obtained 13,900 instances in total. The percentage of solved instances is summarized in Fig. 1. Figure 8 presents partial results for each map. LaCAM* only failed in the instances of *maze-128-128-1* and sub-optimally solved all the other instances within 10 s, outperforming the other algorithms. The failures might be reduced by improving the pattern detector; however, we consider such implementations are too opti-

mized for the benchmark. As shown in Fig. 9, the refinement was steady but not dramatic due to the same reason of Fig. 7. Further discussions are available in the appendix.

Comparison with Anytime MAPF Solver. We compared LaCAM* with AFS [Cohen *et al.*, 2018a], a CBS-based anytime MAPF solver that guarantees to converge optima. Table 5 summarizes the results. Contrary to LaCAM*, AFS can obtain plausible solutions from the beginning, however, it compromises scalability. We consider this quality gap can be overcome by developing better generators other than PIBT.

Extremely Dense Scenarios. Table 6 reports LaCAM* in very congested scenarios that existing solvers mostly fail. Even with such challenging cases, LaCAM* solved many instances, demonstrating its excellent scalability.

6 Conclusion and Discussion

The primary challenge of MAPF is to maintain solvability and solution quality while suppressing planning efforts. To break this tradeoff, this paper presented two enhancements to LaCAM, namely, LaCAM* which eventually converges to optima and an effective configuration generator. The enhancements were thoroughly assessed, achieving remarkable results. From the empirical evidence, we believe that LaCAM* has developed a new frontier in MAPF.

Related Work. LaCAM$^{(*)}$ relates to partial successor expansion during the search, as seen in [Goldenberg *et al.*, 2014; Wagner and Choset, 2015], because it also generates a subset of successors, but differs in the use of constraints and a configuration generator. Anytime MAPF algorithms that converge to optima have been studied [Standley and Korf, 2011; Cohen *et al.*, 2018a; Vedder and Biswas, 2021]. However, their scalability is limited; they often fail to derive initial solutions, as we empirically saw. Techniques to refine arbitrary MAPF solutions have also been studied [Surynek, 2013; De Wilde *et al.*, 2014; Okumura *et al.*, 2021; Li *et al.*, 2021a] but they do not ensure optimality. Rewriting the search tree structure is popular in optimal motion planning [Karaman and Frazzoli, 2011; Shome *et al.*, 2020], by which LaCAM* is partially inspired. Incorporating swap into PIBT is inspired by sub-optimal rule-based MAPF algorithms [Luna and Bekris, 2011; De Wilde *et al.*, 2014]. Meanwhile, the solution quality of rule-based approaches themselves is often severely compromised.

Future Directions. We are interested in more effective configuration generators than PIBT variants which can output near-optimal initial solutions. Improving the convergence speed of LaCAM* is also important. Moreover, LaCAM* in MAPF variants, e.g., multi-robot motion planning [Okumura and Défago, 2023], is worth to be studied. Other than MAPF, since LaCAM* is just a graph pathfinding algorithm, applying its concept to other planning domains might be exciting.

Acknowledgments

I am grateful to Yasumasa Tarmura for his comments on the initial manuscript. This work was partly supported by JSPS KAKENHI Grant Numbers 20J23011 and JST ACT-X Grant Number JPMJAX22A1. I thank the support of the Yoshida Scholarship Foundation when I was a Ph.D. student.

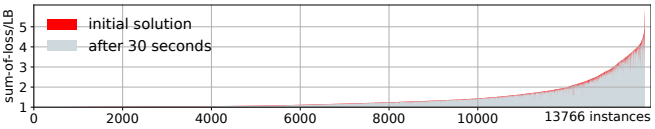


Figure 9: Refinement by LaCAM* for the MAPF benchmark. On the x-axis, the figure sorts 13,766 solved instances out of 13,900 by initial solution quality and displays the scores in red bars. For each instance, we also plot the solution quality at 30 seconds using gray bars. Hence, the effect of refinement is visualized by tiny red areas.

	solved(%)		time-init(ms)		loss-init		loss-30 s	
$\|A\|$	AFS	LaCAM*	AFS	LaCAM*	AFS	LaCAM*	AFS	LaCAM*
50	100	100	88	**1**	**25**	159	**24**	118
100	56	**100**	7223	**2**	**140**	609	**139**	545
150	0	**100**	N/A	**4**	N/A	1463	N/A	1368

Table 5: Comparison of anytime MAPF algorithms. We used sum-of-loss and 25 "random" scenarios of *random-32-32-20*. "init" shows scores related to initial solutions. "loss" is the gap scores from $\sum_{i \in A} \mathsf{dist}(s_i, g_i)$. The scores are averaged for instances solved by both solvers, except for $|A| = 150$ because AFS failed all.

map	$\|A\|$	%	time(s)	other algorithms
empty-8-8	58	100	0.00	PIBT&LaCAM (100%; 0.00 s)
random-32-32-20	737	100	0.63	LaCAM (4%; 14.81 s)
random-64-64-20	2943	68	11.64	N/A
maze-128-128-10	9772	100	55.54	N/A

Table 6: Results on extremely dense scenarios. $|A|$ was adjusted so that $|A|/|V| = 0.9$. For each scenario, 25 instances were prepared while randomly placing starts and goals. "%" is the success percentage by LaCAM* with timeout of 60 s. "time" is the median runtime to obtain initial solutions. We also tested the other solvers in Fig. 1 and report solvers that solved in at least one instance.

References

[Andreychuk and Yakovlev, 2018] Anton Andreychuk and Konstantin Yakovlev. Two techniques that enhance the performance of multi-robot prioritized path planning. In *Proc. Int. Joint Conf. on Autonomous Agents & Multiagent Systems (AAMAS)*, 2018.

[Cohen *et al.*, 2018a] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernández, Ariel Felner, TK Satish Kumar, and Sven Koenig. Anytime focal search with applications. In *Proc Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2018.

[Cohen *et al.*, 2018b] Liron Cohen, Glenn Wagner, David Chan, Howie Choset, Nathan Sturtevant, Sven Koenig, and TK Satish Kumar. Rapid randomized restarts for multi-agent path finding solvers. In *Proc. Annu. Symp. on Combinatorial Search (SOCS)*, 2018.

[De Wilde *et al.*, 2014] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. Push and rotate: a

complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research (JAIR)*, 2014.

[Goldenberg *et al.*, 2014] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert Holte, and Jonathan Schaeffer. Enhanced partial expansion A∗. *Journal of Artificial Intelligence Research (JAIR)*, 2014.

[Hart *et al.*, 1968] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968.

[Karaman and Frazzoli, 2011] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research (IJRR)*, 2011.

[Kautz *et al.*, 2002] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*, 2002.

[Lam *et al.*, 2022] Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter J Stuckey. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research (COR)*, 2022.

[Li *et al.*, 2021a] Jiaoyang Li, Zhe Chen, Daniel Harabor, P Stuckey, and Sven Koenig. Anytime multi-agent path finding via large neighborhood search. In *Proc Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2021.

[Li *et al.*, 2021b] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*, 2021.

[Li *et al.*, 2022] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Mapf-lns2: Fast repairing for multi-agent path finding via large neighborhood search. In *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*, 2022.

[Luna and Bekris, 2011] Ryan Luna and Kostas E Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proc Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2011.

[Okumura and Défago, 2023] Keisuke Okumura and Xavier Défago. Quick multi-robot motion planning by combining sampling and search. In *Proc Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2023.

[Okumura *et al.*, 2021] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Iterative refinement for real-time multi-robot path planning. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2021.

[Okumura *et al.*, 2022] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence (AIJ)*, 2022.

[Okumura, 2023] Keisuke Okumura. Lacam: Search-based algorithm for quick multi-agent pathfinding. In *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*, 2023.

[Sharon *et al.*, 2015] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*, 2015.

[Shome *et al.*, 2020] Rahul Shome, Kiril Solovey, Andrew Dobson, Dan Halperin, and Kostas E Bekris. drrt∗: Scalable and informed asymptotically-optimal multi-robot motion planning. *Autonomous Robots (AURO)*, 2020.

[Silver, 2005] David Silver. Cooperative pathfinding. In *Proc. AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2005.

[Standley and Korf, 2011] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *Proc Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2011.

[Standley, 2010] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*, 2010.

[Stern *et al.*, 2019] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proc. Annu. Symp. on Combinatorial Search (SOCS)*, 2019.

[Surynek, 2013] Pavel Surynek. Redundancy elimination in highly parallel solutions of motion coordination problems. *International Journal on Artificial Intelligence Tools (IJAIT)*, 2013.

[Vedder and Biswas, 2021] Kyle Vedder and Joydeep Biswas. X∗: Anytime multi-agent path finding for sparse domains using window-based iterative repairs. *Artificial Intelligence (AIJ)*, 2021.

[Wagner and Choset, 2015] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence (AIJ)*, 2015.

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

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