LaCAM: Search-Based Algorithm for Quick Multi-Agent Pathfinding

Keisuke Okumura

Tokyo Institute of Technology
Tokyo, Japan
okumura.k@coord.c.titech.ac.jp

Abstract

We propose a novel complete algorithm for multi-agent pathfinding (MAPF) called *lazy constraints addition search for MAPF (LaCAM)*. MAPF is a problem of finding collision-free paths for multiple agents on graphs and is the foundation of multi-robot coordination. LaCAM uses a two-level search to find solutions quickly, even with hundreds of agents or more. At the low-level, it searches constraints about agents' locations. At the high-level, it searches a sequence of all agents' locations, following the constraints specified by the low-level. Our exhaustive experiments reveal that LaCAM is comparable to or outperforms state-of-the-art sub-optimal MAPF algorithms in a variety of scenarios, regarding success rate, planning time, and solution quality of sum-of-costs.

1 Introduction

The *multi-agent pathfinding (MAPF)* problem (Stern et al. 2019) aims to assign collision-free paths on graphs to each agent, which is the foundation of multi-robot coordination. In MAPF applications such as fleet operations in automated warehouses (Wurman, D'Andrea, and Mountz 2008), it is necessary to solve MAPF with thousands of agents in a real-time manner.

In general, the design of MAPF algorithms needs to consider a trade-off between quality and speed. From the quality side, solving MAPF optimally is NP-hard in various criteria (Yu and LaValle 2013). Although many effective optimal algorithms have been developed, it is still challenging to handle a few hundred of agents in real-time (i.e., short timeframes in this context) even with state-of-the-art methods (Li et al. 2021b; Lam et al. 2022). From the speed side, sub-optimal algorithms can quickly solve large MAPF instances while compromising solution quality. However, in real-time applications (i.e., scenarios with deadlines for planning time in this context) or in instances with massive agents such that optimal algorithms never handle, there is no choice but to use sub-optimal algorithms. In addition, recently proposed frameworks can effectively refine the quality of known MAPF solutions (Okumura, Tamura, and Défago 2021; Li et al. 2021a). Consequently, the development of quick sub-optimal algorithms has practical value.

To this end, this paper proposes a novel algorithm called *lazy constraints addition search for MAPF (LaCAM)*. From the theoretical side, LaCAM is complete. It returns a solution for solvable instances, otherwise reports the non-existence. From the empirical side, we demonstrate that LaCAM can solve a variety of MAPF instances in a very short time, including instances with large maps (e.g., 1491×656 four-connected grid), instances with massive agents (e.g., $10,000$), or dense situations. For instance, LaCAM solved *all* instances with 400 agents on a 32×32 grid with 20% obstacles from the MAPF benchmark (Stern et al. 2019), with a median runtime of $1\,\mathrm{s}$. In contrast, baseline sub-optimal MAPF algorithms (Silver 2005; Standley 2010; Okumura et al. 2022; Li, Ruml, and Koenig 2021; Li et al. 2022) mostly failed to solve the instances with the timeout of $30\,\mathrm{s}$.

LaCAM has an easily extensible structure, which comprises a two-level search. At the high-level, it searches a sequence of configurations, where a configuration is a tuple of locations for all agents. At the low-level, it searches constraints that specify which agents go where in the next configuration. Successors at the high-level (i.e., configurations) are generated in a lazy manner while following constraints from the low-level, leading to a dramatic reduction of the search effort. Similar to the popular CBS algorithm (Sharon et al. 2015), due to its simplicity, we consider that LaCAM can apply to many domains not limited to MAPF such as multi-robot motion planning (Solis et al. 2021).

Throughout the paper, we focus on sub-optimal LaCAM. The discussion of optimal LaCAM appears at the end, together with a qualitative discussion of why LaCAM is quick. In what follows, we present problem formulation, the algorithm, evaluation, and related work in order. The supplementary material is available on https://kei18.github.io/lacam.

2 Preliminaries

Problem Definition An *MAPF instance* is defined by a graph $G = (V, E)$, a set of agents $A = \{1, \ldots, n\}$, a tuple of distinct starts $\mathcal{S} = (s_1, \ldots, s_n)$ and goals $\mathcal{G} = (g_1, \ldots, g_n)$, where $s_i, g_i \in V$.

Given an MAPF instance, let $\pi_i[t] \in V$ denote the location of an agent i at discrete time $t \in \mathbb{N}_{\geq 0}$. At each timestep t, i can move to an adjacent vertex, or can stay at its current vertex, i.e., $\pi_i[t+1] \in \mathsf{neigh}(\pi_i[t]) \cup \{\pi_i[t]\}$, where $\mathsf{neigh}(v)$ is the set of vertices adjacent to $v \in V$. Agents
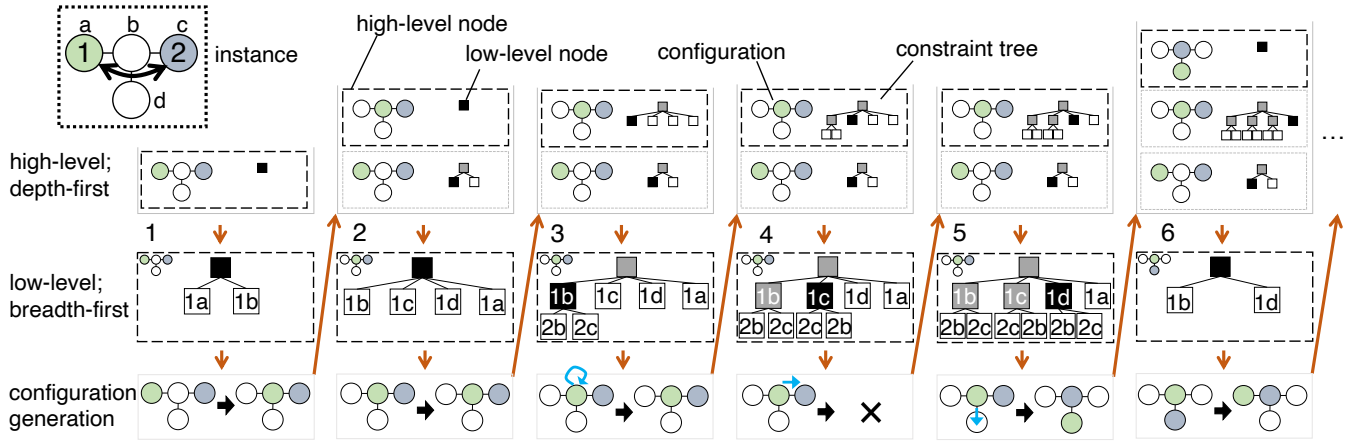
注意 这里constrain是必须往哪走

Figure 1: Running example of LaCAM. Orange arrows represent the search progress order. Selected and searched low-level nodes are filled with black and gray, respectively. Constraints are shown by blue-colored arrows.

must avoid two types of conflicts: 1) *vertex conflict*: $\pi_i[t] = \pi_j[t]$, and, 2) *swap conflict*: $\pi_i[t] = \pi_j[t+1] \wedge \pi_i[t+1] = \pi_j[t]$. A set of paths is *conflict-free* when no conflicts exist.

A *solution* to MAPF is a set of conflict-free paths $\{\pi_1, \ldots, \pi_n\}$ such that all agents reach their goals at a certain timestep $T \in \mathbb{N}_{\geq 0}$. More precisely, the problem assigns a path $\pi_i = (\pi_i[0], \pi_i[1], \ldots, \pi_i[T])$ to each agent such that $\pi_i[0] = s_i$, $\pi_i[T] = g_i$, and is conflict-free.

This paper considers a *sum-of-costs (SOC)* metric to rate solution quality: $\sum_{i \in A} T_i$, where $T_i \leq T$ is the earliest timestep such that $\pi_i[T_i] = \pi_i[T_i + 1] = \ldots = \pi_i[T] = g_i$.

Notations   We use $S[k]$ to denote the k-th element of the sequence S, where the index starts at one. Δ is the maximum degree of the graph. A *configuration* refers to a tuple of locations for all agents, e.g., the start and goal configurations are $\mathcal{S}$ and $\mathcal{G}$, respectively. Two configurations X and Y are *connected* when $\{(X[1], Y[1]), \ldots, (X[n], Y[n])\}$ constitutes a set of conflict-free paths. For convenience, we use $\perp$ as an "undefined" or "not found" sign.

3   Algorithm

This section first presents the concept of LaCAM, followed by the pseudocode and implementation details.

3.1   Concept

LaCAM is a two-level search. At the high-level, it explores a sequence of configurations; each search node corresponds to one configuration. For each high-level node, it also performs a low-level search that creates *constraints*. A constraint specifies which agent is where in the next configuration. The low-level search proceeds lazily, creating a minimal successor each time the corresponding high-level node is invoked. Figure 1 presents an illustration of LaCAM. The example MAPF instance is depicted on the left upper side. The following part explains the figure step by step.

High-Level Search   As in general search schemes, LaCAM progresses by updating an *Open* list that stores the high-level nodes. *Open* is implemented by data structures of stack, queue, or priority queue. We use the stack throughout the paper. Thus, LaCAM is explained as a depth-first search style. The first row of Fig. 1 shows *Open*. For each search iteration, LaCAM selects one node from *Open*. Different from general search schemes, LaCAM does not immediately discard the selected node, as explained after three paragraphs.

Low-Level Search   Each high-level node comprises a configuration and a *constraint tree*. The constraint tree gradually grows each time invoking the high-level node; this is the low-level search of LaCAM. Throughout the paper, we use a breadth-first search for the low-level. The middle row of Fig. 1 visualizes this step. Each node of the constraint tree has a constraint, except for the root node. For instance, in the first column, the root node has two successors: '1a' and '1b.' This means that agent-1 must go to vertex-a or vertex-b in the next configuration. Successors of the low-level search are created by two steps: 1) Select an agent i. Let v be the vertex of i in the configuration. 2) Create successors that specifies i is on $u \in \text{neigh}(v)$ or v. The agent is selected so that each path from each low-level node to the root does not contain duplicated agents. Therefore, those paths specify constraints for several agents. In addition, no successors are created when the depth of the node is beyond $|A|$ because constraints have been assigned for all agents.

Configuration Generation   Once both the high- and low-level nodes are specified, a new configuration is generated. The new one must satisfy the constraints of the low-level node, which are specified by a path to the root node. Excluding that, any connected configuration from the original configuration can be generated. *We complement how to generate new configurations following constraints later in Sec. 3.3, but for now, regard this as a black-box function.* The generation step is visualized in the third row of Fig. 1. According to the new configuration, a new high-level node is created. For instance, at the end of the first column of Fig. 1, a new configuration (b, c) is generated and inserted to *Open*.

Discarding High-Level Nodes When finished searching all low-level nodes, the corresponding high-level node has been generating all configurations connected to its configuration. Therefore, this high-level node is discarded.

Example LaCAM continues the above search operations until finding the goal configuration $\mathcal{G}$. Once $\mathcal{G}$ is found, it is trivial to obtain a solution by backtracking high-level nodes. We next describe Fig. 1 in detail step by step of columns.

1. Initially, $Open$ contains only a high-level node for the start configuration $\mathcal{S}$. At the low-level, two successors are created. In this step, any agent can be selected; we choose agent-1. Next, LaCAM generates a configuration connected to the original one (a, c). Since the target low-level node is the root, there is no constraint. Assume that a new configuration (b, c) is generated. Then, the corresponding new high-level node is inserted into $Open$.

2. The high-level node generated in the previous iteration is selected. At the low-level, we again choose agent-1. This time, LaCAM generates four successors: '1a', '1b', '1c,' and '1d.' We add them to the low-level tree in order of '1b', '1c', '1d,' and '1a' to make the example interesting. Assume that the same configuration (b, c) is generated. Then, a high-level node is not created since the generated configuration has already appeared in the search.

3. The same high-level node is selected as the previous iteration. The low-level search generates two nodes for agent-2: '2b' and '2c.' This time, the configuration generation must follow the constraint of '1b.' Consequently, the same configuration (b, c) is generated and no new high-level node is created.

4. According to the selected low-level node, it is impossible to generate a connected configuration due to conflicts; this iteration skips the creation of a high-level node.

5. The constraint makes agent-1 move to vertex-d. Then, a new configuration (d, b) is generated. The corresponding high-level node is created and inserted into $Open$.

6. The high-level node for (d, b) is selected, and then, a new configuration (b, a) is generated. The search can find the goal configuration in the next iteration.

3.2 Pseudocode

Algorithm 1 shows an example implementation of LaCAM. In the pseudocode, $\mathcal{N}$ and $\mathcal{C}$ correspond to high- and low-level nodes, respectively. The low-level search uses queue ($tree$) because it is breadth-first. Several details are below.

Configuration Generation This is performed by a black-box function get_new_config [Line 14]. The function returns a configuration connected to a configuration of a high-level node, following constraints specified by a low-level node. It returns $\bot$ when failing to generate such configurations (e.g., the fourth column of Fig. 1). Note that, at the bottom of the low-level tree, all agents have constraints. Therefore, exactly one configuration is specified without freedom.

High-Level Node Management To manage already known configurations, Alg. 1 uses an $Explored$ table that takes a configuration as a key and stores a high-level node.

Algorithm 1: LaCAM

input: MAPF instance ($\mathcal{S}$: starts, $\mathcal{G}$: goals)
output: solution or NO_SOLUTION
preface: $\mathcal{C}_{\text{init}} := \langle\, parent : \bot, who : \bot, where : \bot \,\rangle$

1: initialize $Open$, $Explored$ ▷ $Open$: stack
2: $\mathcal{N}_{\text{init}} \leftarrow \langle config : \mathcal{S}, tree : [\![\, \mathcal{C}_{\text{init}} \,]\!],$ ▷ $tree$: queue
        $order : \mathsf{get\_init\_order}(), parent : \bot \rangle$
3: $Open.\mathsf{push}(\mathcal{N}_{\text{init}})$; $Explored[\mathcal{S}] = \mathcal{N}_{\text{init}}$
4: **while** $Open \neq \emptyset$ **do**
5:    $\mathcal{N} \leftarrow Open.\mathsf{top}()$
6:    **if** $\mathcal{N}.config = \mathcal{G}$ **then return** $\mathsf{backtrack}(\mathcal{N})$ 一个搜索树
7:    **if** $\mathcal{N}.tree = \emptyset$ **then** $Open.\mathsf{pop}()$; **continue**
8:    $\mathcal{C} \leftarrow \mathcal{N}.tree.\mathsf{pop}()$
9:    **if** $\mathsf{depth}(\mathcal{C}) \leq |A|$ **then**
10:      $i \leftarrow \mathcal{N}.order[\mathsf{depth}(\mathcal{C})]$; $v \leftarrow \mathcal{N}.config[i]$ 直接找到最低层
11:      **for** $u \in \mathsf{neigh}(v) \cup \{v\}$ **do**
12:        $\mathcal{C}_{\text{new}} \leftarrow \langle\, parent : \mathcal{C}, who : i, where : u \,\rangle$
13:        $\mathcal{N}.tree.\mathsf{push}(\mathcal{C}_{\text{new}})$
14:    $\mathcal{Q}_{\text{new}} \leftarrow \mathsf{get\_new\_config}(\mathcal{N}, \mathcal{C})$
15:    **if** $\mathcal{Q}_{\text{new}} = \bot$ **then continue** 若没有新的configuration 跳过
16:    **if** $Explored[\mathcal{Q}_{\text{new}}] \neq \bot$ **then continue** 若已经加入过 跳过
17:    $\mathcal{N}_{\text{new}} \leftarrow \langle config : \mathcal{Q}_{\text{new}}, tree : [\![\, \mathcal{C}_{\text{init}} \,]\!],$
        $order : \mathsf{get\_order}(\mathcal{Q}_{\text{new}}, \mathcal{N}), parent : \mathcal{N} \rangle$ order确保生成的每条路径不会有重复
的agent 路径指的是从当
前到root结点
18:    $Open.\mathsf{push}(\mathcal{N}_{\text{new}})$; $Explored[\mathcal{Q}_{\text{new}}] = \mathcal{N}_{\text{new}}$
19: **return** NO_SOLUTION

Low-Level Agent Selection To generate low-level search trees, a high-level node includes $order$, an enumeration of all agents sorted by specific criteria, specified by two functions get_init_order [Line 2] and get_order [Line 17]. The agent is selected following $order$ and depth of the low-level search tree (starting at one; obtained by a function depth) [Line 10]. Doing so ensures that each path of the constraint tree has no duplicate agents.

Theorem 1 (completeness). *Algorithm 1 returns a solution for solvable MAPF instances; otherwise, it reports* NO_SOLUTION.

Proof. A search space is finite. For the high-level, the number of configurations is $O\left(|V|^{|A|}\right)$. For the low-level, the number of search iterations is upper bounded by $1 + \Delta + \Delta^2 + \ldots + \Delta^{|A|} = O\left(\Delta^{|A|+1}\right)$. When the low-level search is finished, the corresponding high-level node has been generating all configurations connected to its configuration. Consequently, all *reachable* configurations from the start configuration, defined by transitivity over connections of two configurations, are examined, deriving the theorem. $\square$

3.3 Implementation Details

Configuration Generation The heart of LaCAM is how to generate configurations following constraints [Line 14]. Ideally, this sub-procedure should be sufficiently quick and generate a promising configuration to reach the goal configuration. This can be realized by adapting existing MAPF algorithms that can compute a partial solution, i.e., a set
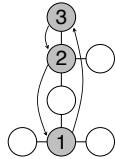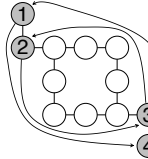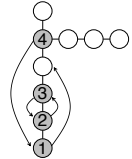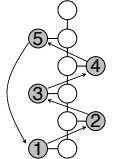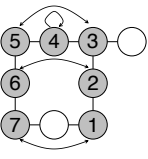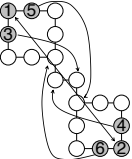
	tree		*corners*		*tunnel*		*string*		*loop-chain*		*connector*		
	Time(ms)	SOC	Time(ms)	SOC	Time(ms)	SOC	Time(ms)	SOC	Time(ms)	SOC	Time(ms)	SOC	Solved
LaCAM(med)	0	19	17	47	173	190	0	66	34	1752	0	168	**6/6**
LaCAM(worst)	0	41	31	70	208	254	0	68	124	2593	3	281	
PP	N/A	N/A	0	32	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1/6
OD	0	31	0	47	30	191	0	22	5882	2269	27	129	**6/6**
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	55	0	54	0	227	0	52	N/A	N/A	0	130	5/6
EECBS	1	16	1	32	N/A	N/A	0	20	N/A	N/A	N/A	N/A	3/6
LNS2	N/A	N/A	0	32	N/A	N/A	0	20	N/A	N/A	160	80	3/6
A*(SOC-optimal)	0	16	16	32	24	53	1	20	17752	121	391138	80	**6/6**

Table 1: Results of the small complicated instances.

of paths until a certain timestep. Regarding the target configuration as a start configuration and producing a partial solution by these algorithms, we can extract a configuration at timestep one in the partial solution. For instance, a naive approach is to adapt prioritized planning with a limited planning horizon, such as windowed HCA* (Silver 2005) with single-step window size. A rolling horizon approach for lifelong MAPF (Li et al. 2021c) is also available to generate configurations. An aggressive approach is to adapt PIBT (Okumura et al. 2022), a scalable MAPF algorithm that repeats planning for one-timestep; we used PIBT in the experiments. *Those algorithms, originally developed to solve MAPF, are available to create promising successors in the high-level search of LaCAM.*

Order of Agents　In our implementation, the agent order of the initial high-level node was in descending order of the distance between the start and goal [Line 2], influenced by commonly used heuristics of prioritized planning (Van Den Berg and Overmars 2005). In other high-level nodes [Line 17], we prioritized agents who are not on their goal, aiming to create constraints for those agents earlier in the low-level search. A tiebreak used the last arrival timestep of agents in the high-level search so that agents who have been apart from goals for a long time were prioritized, akin to PIBT (Okumura et al. 2022).

Order of Low-Level Nodes　As seen in Fig. 1, the order of inserting low-level nodes affects search progress. We provisionally made the order random. This part requires further investigation in the future.

Reinsert High-Level Node　Algorithm 1 takes a naive depth-first search style. Instead, when finding an already known configuration, we observed that reinserting the corresponding high-level node to *Open* [Line 16] can improve solution quality. The reason is that repeatedly appearing configurations in the search can be seen as a bottleneck; it makes sense to advance the low-level search of the high-level node,

which is performed by the reinsert operation. Note that LaCAM does not lose completeness with this modification.

4 Evaluation

This section evaluates LaCAM. Specifically, we show five empirical results: 1) evaluation with small complicated MAPF instances, 2) evaluation with the MAPF benchmark, 3) showing the current limitation of LaCAM using an adversarial instance, 4) scalability test with up to 10,000 agents, and 5) investigating other implementation designs.

4.1 Experimental Setup

Baselines　We carefully selected the following six suboptimal MAPF algorithms as baselines.

- **Prioritized Planning (PP)** (Erdmann and Lozano-Perez 1987; Silver 2005) as a basic approach for MAPF. PP used distance heuristics (Van Den Berg and Overmars 2005) for the planning order and A* (Hart, Nilsson, and Raphael 1968) for single-agent pathfinding.

- **A* with operator decomposition (OD)** (Standley 2010) as an adaptation of the general search scheme to MAPF. We used a greedy search to obtain solutions as much as possible (i.e., neglecting g-value of A*). The heuristic (i.e., h-value) was the sum of distance towards goals.

- **PIBT** (Okumura et al. 2022), which repeats one-timestep planning to solve MAPF. We tested a vanilla PIBT because the LaCAM implementation used PIBT as a subprocedure (Sec. 3.3). To detect planning failure, we regarded that PIBT failed to solve an instance when it reached pre-defined sufficiently large timesteps.

- **PIBT^{+}** (Okumura et al. 2022) as a state-of-the-art scalable MAPF solver, which uses PIBT until a certain timestep. The rest of planning is performed by another MAPF algorithm. The complement phase used a rule-based solver, Push and Swap (Luna and Bekris 2011).

- **EECBS** (Li, Ruml, and Koenig 2021) as a state-of-the-art search-based solver that bases on a celebrated MAPF
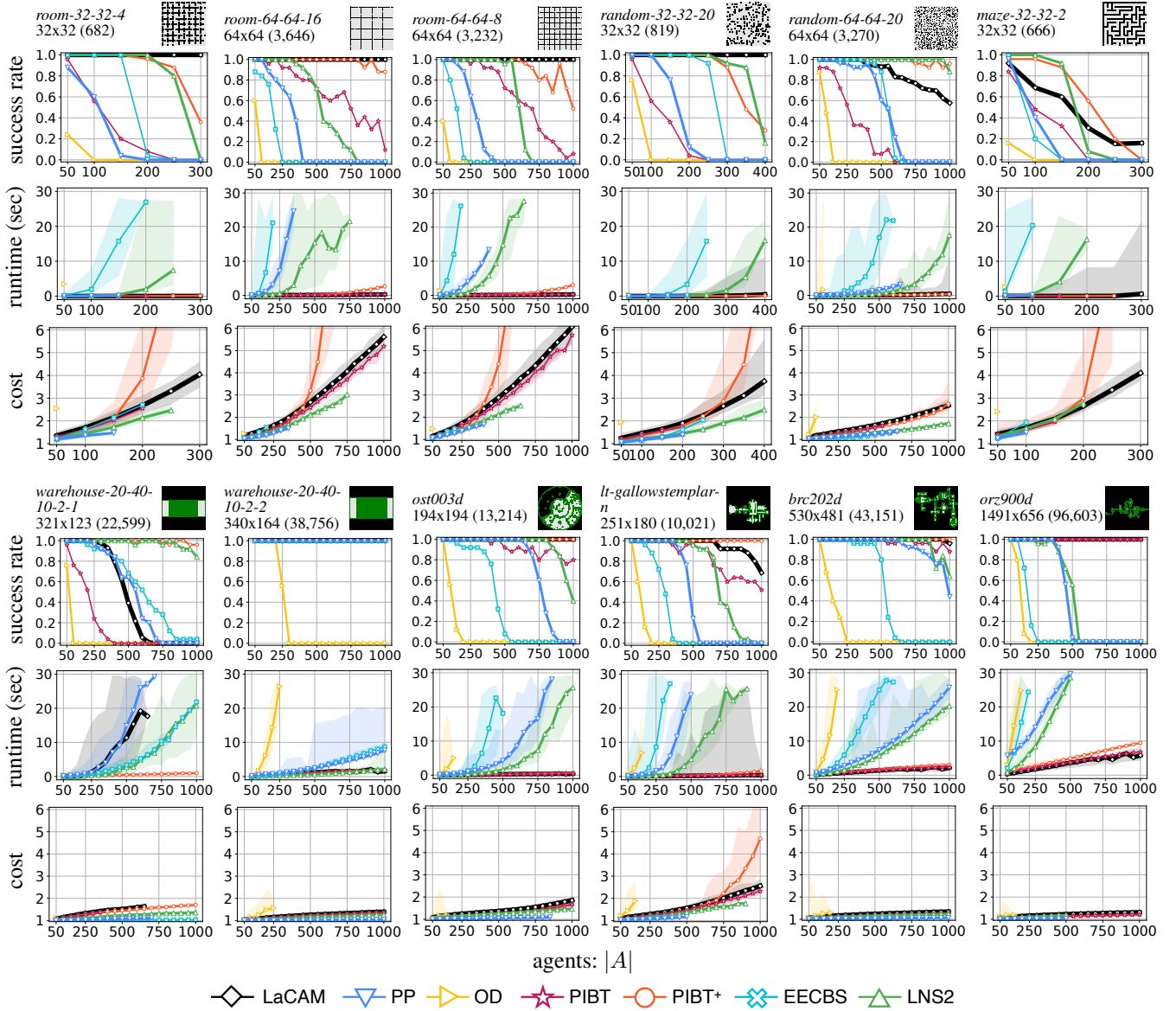
11658

Figure 2: Results of the MAPF benchmark. The number of vertices for each grid is shown with parentheses. 'cost' represents SOC divided by the total distance of start-goal pairs, $\sum_{i \in A} \text{dist}(s_i, g_i)$, where $\text{dist}(u, v)$ is the shortest path length from $u \in V$ to $v \in V$ on the graph. This score works as the upper bound of sub-optimality, where the minimum is one. For 'runtime' and 'cost,' we show median scores of solved instances within each solver. We further show minimum and maximum scores using semi-transparent regions. The success rate of LaCAM was based on the number of successful trials over total trials.

algorithm, CBS (Sharon et al. 2015). The sub-optimality was set to five to find solutions as much as possible. In Sec. 4.5, it was set to the default value (1.2) of the authors' implementation due to the better performance.

• **MAPF-LNS2 (LNS2)** (Li et al. 2022) as another excellent MAPF solver based on large neighborhood search.

It is worth mentioning that PP, PIBT$^{(+)}$, EECBS, and LNS2 are incomplete; they cannot detect unsolvable instances, unlike LaCAM. For PIBT$^{(+)}$, EECBS, and LNS2, we used the implementations coded by their respective authors.[1] For PP, we used an implementation included in (Okumura et al. 2022). OD was own-coded in C++.

Evaluation Environment LaCAM was coded in C++, available in the online supplementary material. The experiments were run on a desktop PC with Intel Core i9-7960X 2.8 GHz CPU and 64 GB RAM. We performed a maximum of 32 different instances run in parallel using multithreading.

4.2 Small Complicated Instances

First, we tested LaCAM with instances used in (Luna and Bekris 2011), shown in Table 1. The runtime limit was $10\,\mathrm{s}$. Since our LaCAM implementation used non-determinism (see Sec. 3.3), it was run five times for the same instance while changing random seeds.

Table 1 summarizes the results. As reference records, we also show SOC-optimal solutions obtained by a vanilla A^*, ignoring the runtime limit. Although most baseline methods failed several instances, LaCAM solved all the instances regardless of random seeds, within reasonable timeframes. Regarding solution quality (i.e., SOC), LaCAM compromises the quality compared to PP, EECBS, and LNS2. This is due to the nature of LaCAM, which progresses the search with a short planning horizon.

4.3 MAPF Benchmark

Next, we tested LaCAM using the MAPF benchmark (Stern et al. 2019), which includes a set of four-connected grids and start–goal pairs for agents. We selected twelve grids with different portfolios (e.g., size, sparseness, and complexity). For each grid, 25 "random scenarios" were used while increasing the number of agents by 50 up to the maximum. Therefore, identical instances were tried for the solvers in all settings. The runtime limit was set to $30\,\mathrm{s}$ following (Stern et al. 2019). LaCAM was run five times for each setting. For reference, we report that A^* used in Table 1 failed to solve an instance with ten agents in *random-32-32-20*.

Figure 2 summarizes the results. In most scenarios, LaCAM outperforms PP, OD, EECBS, and LNS2 in both success rate and runtime, while compromising the solution quality. The runtime of LaCAM is comparable with $\mathrm{PIBT}^{(+)}$, furthermore, LaCAM outperforms a vanilla PIBT in the success rate. The most competitive results with LaCAM were scored by PIBT^+. However, overall, the SOC scores of LaCAM are better than those of PIBT^+, especially in dense situations. Furthermore, LaCAM solved challenging scenarios, such as *random-32-32-20* with 400 agents, where the baseline methods almost failed to solve. In summary, LaCAM can solve many instances within short timeframes, with acceptable solution quality. Meanwhile, we observed that LaCAM scored poor performance in several grids, such as *random-64-64-20* and *warehouse-20-40-10-2-1*. We investigate this reason in the next.

4.4 Adversarial Instance

In the previous experiment, LaCAM quickly solved various scenarios but scored poor performance in several grids. Specifically, LaCAM solved all instances of *warehouse-20-40-10-2-2* while it failed frequently in *warehouse-20-40-10-2-1*. The two maps differ in the width of corridors: the former is two while the latter is one. Therefore, we considered the existence of narrow corridors such that two agents cannot pass through could be a bottleneck of LaCAM.

For investigation, we prepared an adversarial instance for LaCAM (see Table 2), where two pairwise agents need to swap their locations in narrow corridors. We counted the number of search iterations of the high-level search while
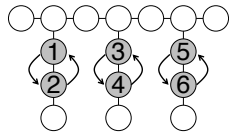

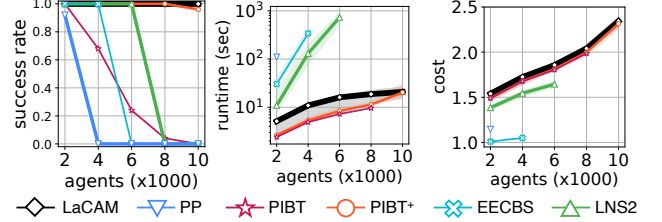| $|A|$	search iteration
2	128
4	23,907
6	287,440

Table 2: LaCAM performance in an adversarial instance.

Figure 3: Results with massive agents. The used map was *warehouse-20-40-10-2-2*.

changing the number of agents (two: only agents-$\{1, 2\}$ appear, four, and six). LaCAM solved all instances, however, the search effort dramatically increases with more agents. This is because, with more agents, the high-level search can contain a huge number of *slightly different configurations*. Those configurations differ only one or two agents differ in their locations and disturb the progression of the search.

From this observation, we consider the poor performance in several grids of Fig. 2 as follows. The LaCAM implementation used a depth-first search style, therefore, once a configuration similar to Table 2 appears during the search, resolving this configuration towards the goal configuration requires significant search effort. Consequently, LaCAM reaches a timeout. Overcoming this limitation is one promising future direction. One resolution might be developing a better configuration generator other than PIBT.

4.5 Scalability Test

We evaluated the scalability of the number of agents, using instances with up to $10,000$ agents in *warehouse-20-40-10-2-2*. The runtime limit was set to $1000\,\mathrm{s}$. OD was excluded since it run out of memory. Figure 3 summarizes the result. Only LaCAM solved all instances. Furthermore, the runtime was in at most $30\,\mathrm{s}$ even with 10,000 agents, demonstrating the excellent scalability of LaCAM.[2]

4.6 Design Choice of LaCAM

Finally, we investigated the design choice of LaCAM implementation. Specifically, we assessed two variants:

- **DFS** does not use the reinsert operation at the high-level.
- **GREEDY** uses another configuration generator instead of PIBT, such that agents greedily determines the loca-

---

[2]We briefly report that LaCAM can be faster depending on computational environments. As a pilot study, we tested the same setting with a single-thread run in a laptop with Intel Core i9 $2.3\,\mathrm{GHz}$ CPU and $16\,\mathrm{GB}$ RAM. Even with 10,000 agents, LaCAM solved all instances at most in $10\,\mathrm{s}$ in the worst case.
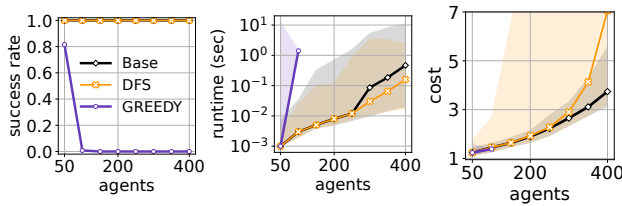
Figure 4: Results with different LaCAM designs. The used map was *random-32-32-20*.

tion according to the priority order. This is equivalent to prioritized planning with a single-step planning horizon.

Figure 4 shows the result in *random-32-32-20*. There are two observations: 1) The reinsert operation improves the SOC metric. 2) The choice of configuration generator significantly affects the search as seen in that GREEDY failed in most settings. Consequently, the excellent performance of LaCAM in Fig. 2 relies on promising successor generation at the high-level, which is done by PIBT.

5   Related Work

The multi-agent pathfinding (MAPF) problem (Stern et al. 2019) has been extensively studied since the 2010s. LaCAM borrows several ideas from prior art. We here review the relationship between LaCAM and existing MAPF algorithms.

In general, MAPF algorithms are categorized into four: 1) *Search-based* approaches search solutions in a coupled manner among agents (Standley 2010; Sharon et al. 2015). 2) *Compiling-based* approaches reduce MAPF to well-known problems such as SAT (Surynek 2019; Lam et al. 2022). 3) *Prioritized planning* approaches sequentially plan individual paths for agents in a decoupled manner (Erdmann and Lozano-Perez 1987; Silver 2005). 4) *Rule-based* approaches make agents move step-by-step following ad-hoc rules (Luna and Bekris 2011). LaCAM is search-based, while the configuration generation is not limited to. Indeed, PIBT (Okumura et al. 2022) used in the experiments is categorized into prioritized planning and rule-based approaches.

Popular MAPF algorithms use a two-level search (Sharon et al. 2013, 2015; Surynek 2019; Lam et al. 2022). At the high-level, these algorithms search *negative* constraints that specify which agents cannot use where and when, while at the low-level, they perform single-agent pathfinding following constraints. LaCAM also relies on a two-level search, however, constraints are addressed at the low-level. Furthermore, our implementation uses *positive* constraints that specify who to be where. Positive constraints are not new in the literature; we can see an example in CBS (Li et al. 2019). Note that LaCAM with negative constraints is possible to implement, however, the search space for the low-level can be dramatically larger than using positive ones.

Apart from two-level search, A* variants for MAPF have been also developed (Standley 2010; Goldenberg et al. 2014; Wagner and Choset 2015). In these studies, successors of a search node are generated without *aggressive* coordination between agents beyond collision checking. In contrast, LaCAM considers aggressive coordination when generating

successors at the high-level, which is incorporated by the use of other MAPF algorithms as a configuration generator.

The manner of adding constraints is partially influenced by A* with operator decomposition (Standley 2010). This algorithm creates successor nodes that correspond to one action of one agent, instead of actions for the entire agents. LaCAM adds constraints in a similar scheme.

6   Conclusion and Discussion

This paper introduced LaCAM, a novel, complete, and quick search-based MAPF algorithm. Our exhaustive experiments reveal that LaCAM can solve various instances in a very short time, even with complicated, dense, and challenging scenarios that other state-of-the-art sub-optimal solvers cannot handle. Furthermore, LaCAM is scalable; even with 10,000 agents, it solved instances in tens of seconds. In the following, we list discussions and future directions.

Why is LaCAM quick?   In general, the average branching factor largely determines the search effort. A vanilla A* for MAPF generates $O(\Delta^{|A|})$ configurations from one search node, which is intractable especially when $|A|$ is large. In contrast, each high-level node of LaCAM initially generates only one successor (i.e., configuration), and if necessary, gradually generates other successors in a lazy manner. This scheme virtually suppresses the branching factor of LaCAM. If the generated successor is promising (i.e., closer toward the goal configuration from the original), it can dramatically reduce the number of node generations. Promising successors can be quickly obtained by techniques of recent MAPF studies such as PIBT. This is a trick of quickness in LaCAM. Although virtually reducing the branching factor in A* for MAPF has been proposed (Standley 2010; Goldenberg et al. 2014; Wagner and Choset 2015), these studies never achieve the dramatic reductions as LaCAM.

Optimal LaCAM   This paper aimed at developing a quick MAPF solver; we retain the discussion of optimality. Since LaCAM is search-based, we consider that it is possible to develop optimal LaCAM. Indeed, using a breadth-first search style instead of a depth-first style can solve makespan-optimal MAPF, where makespan is the maximum traveling time of agents. We are also interested in developing an *asymptotically optimal* version of LaCAM, that is, eventually converging to optima. This is a common approach in motion planning algorithms (Karaman and Frazzoli 2011). Since solving MAPF optimally is NP-hard (Yu and LaValle 2013), such asymptotically-optimal approaches are practical for massive instances. In the MAPF literature, such approaches have already appeared for other search-based algorithms (Standley and Korf 2011; Cohen et al. 2018).

Improvements of Technical Components   We investigated the design choices of LaCAM in Sec. 4.6, while other components should be further explored, such as how to add effective constraints in the low-level search and how to design effective configuration generators. Those improvements will overcome the current limitation as seen in Sec. 4.4.

Acknowledgments

References

Cohen, L.; Greco, M.; Ma, H.; Hernández, C.; Felner, A.; Kumar, T. S.; and Koenig, S. 2018. Anytime Focal Search with Applications. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Erdmann, M.; and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica*.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N.; Holte, R.; and Schaeffer, J. 2014. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research (JAIR)*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*.

Karaman, S.; and Frazzoli, E. 2011. Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research (IJRR)*.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2022. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research (COR)*.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P.; and Koenig, S. 2021a. Anytime multi-agent path finding via large neighborhood search. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

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. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

Li, J.; Harabor, D.; Stuckey, P. J.; Felner, A.; Ma, H.; and Koenig, S. 2019. Disjoint splitting for multi-agent path finding with conflict-based search. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*.

Li, J.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021b. Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search. *Artificial Intelligence (AIJ)*.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021c. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

Luna, R.; and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2022. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. *Artificial Intelligence (AIJ)*.

Okumura, K.; Tamura, Y.; and Défago, X. 2021. Iterative Refinement for Real-Time Multi-Robot Path Planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*.

Silver, D. 2005. Cooperative Pathfinding. In *Proceedings of AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Solis, I.; Motes, J.; Sandström, R.; and Amato, N. M. 2021. Representation-Optimal Multi-Robot Motion Planning using Conflict-based Search. *IEEE Robotics and Automation Letters (RA-L)*.

Standley, T.; and Korf, R. 2011. Complete Algorithms for Cooperative Pathfinding Problems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Standley, T. S. 2010. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.; et al. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*.

Surynek, P. 2019. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Van Den Berg, J. P.; and Overmars, M. H. 2005. Prioritized motion planning for multiple robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Wagner, G.; and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artificial Intelligence (AIJ)*.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*.

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