Engineering LaCAM*: Towards Real-Time, Large-Scale, and Near-Optimal Multi-Agent Pathfinding

Keisuke Okumura[1,2]

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

ABSTRACT

This paper addresses the challenges of real-time, large-scale, and near-optimal multi-agent pathfinding (MAPF) through enhancements to the recently proposed LaCAM* algorithm. LaCAM* is a scalable search-based algorithm that guarantees the eventual finding of optimal solutions for cumulative transition costs. While it has demonstrated remarkable planning success rates, surpassing various state-of-the-art MAPF methods, its initial solution quality is far from optimal, and its convergence speed to the optimum is slow. To overcome these limitations, this paper introduces several improvement techniques, partly drawing inspiration from other MAPF methods. We provide empirical evidence that the fusion of these techniques significantly improves the solution quality of LaCAM*, thus further pushing the boundaries of MAPF algorithms.

KEYWORDS

MAPF; Anytime algorithm; Real-time planning

1 INTRODUCTION

The *multi-agent pathfinding (MAPF)* problem [28] seeks to find a collection of collision-free paths for multiple agents on graphs, with appealing applications such as warehouse automation [32] and railway scheduling [12]. The core challenge for MAPF algorithms is to derive plausible solutions that minimize redundant agent motions, even with hundreds of agents or more, within a realistic computational timeframe (i.e., in real-time; e.g., 10 s). We address this ultimate objective by enhancing the recently introduced *LaCAM* (lazy constraint addition search for MAPF)* algorithm [18, 19].

LaCAM* is a search-based algorithm akin to the A* search [7]. It is an anytime algorithm that, after the initial solution discovery, gradually improves the solution quality and eventually converges to optimal ones, provided that the solution cost takes the form of cumulative transition costs. While LaCAM* has showcased remarkable

scalability for the number of agents, outperforming other MAPF methods, the initial solution quality is somewhat compromised [26]. Furthermore, the solution refinement of vanilla LaCAM* is notably slow and impractical [18]; achieving near-optimal solutions for large-scale MAPF in real-time remains extremely challenging.

To overcome these limitations, this paper explores various *engineering* techniques for enhancing LaCAM*, drawing inspiration from the literature on MAPF and search methodologies. An overview of each technique is as follows.

- Section 3 introduces a non-deterministic search node extraction to escape search-stuck situations. This accelerates the refinement process.
- Section 4 introduces an effective utilization of precomputed paths that are spatially dispersed. This aids in discovering better initial solutions.
- Section 5 introduces a Monte Carlo-style successor generation method. This aids in discovering better initial solutions.
- Section 6 introduces the dynamic incorporation of alternative solutions during the search. This boosts the refinement.
- Section 7 introduces the recursive use of LaCAM* to find alternative solutions. Combined with the technique in Sec. 6, this accelerates the refinement.

While each technique constitutes a modest enhancement, their combined employment significantly boosts the performance of LaCAM*, as evidenced in Fig. 1 and 2. In such challenging cases, not only do state-of-the-art optimal algorithms [9, 14] exhibit complete failure, but even bounded suboptimal approaches [15, 31] and affordable prioritized planning [20, 27] largely falter. Meanwhile, the original LaCAM* succeeded in solving *all* of them within the 30 s timeout. This paper enhances the appeal of LaCAM* by empowering it to generate near-optimal solutions. For instance, the improved version achieved an average cost reduction of approximately 30% in the *random-32-32-20* scenario with 409 agents. Based on these empirical achievements, we believe that this work opens a new frontier for real-time, large-scale, and near-optimal MAPF, an area that has been difficult to tackle.

Below, the paper provides preliminaries in Sec. 2, subsequently presenting each technique in order. Each technique is evaluated within its dedicated section and collectively evaluated in Sec. 8. The supplementary material appears at https://kei18.github.io/lacam3/.

2 PRELIMINARIES

For convenience, we use ⊥ as an "undefined" sign. The dist function returns the shortest path length between two vertices on a graph.
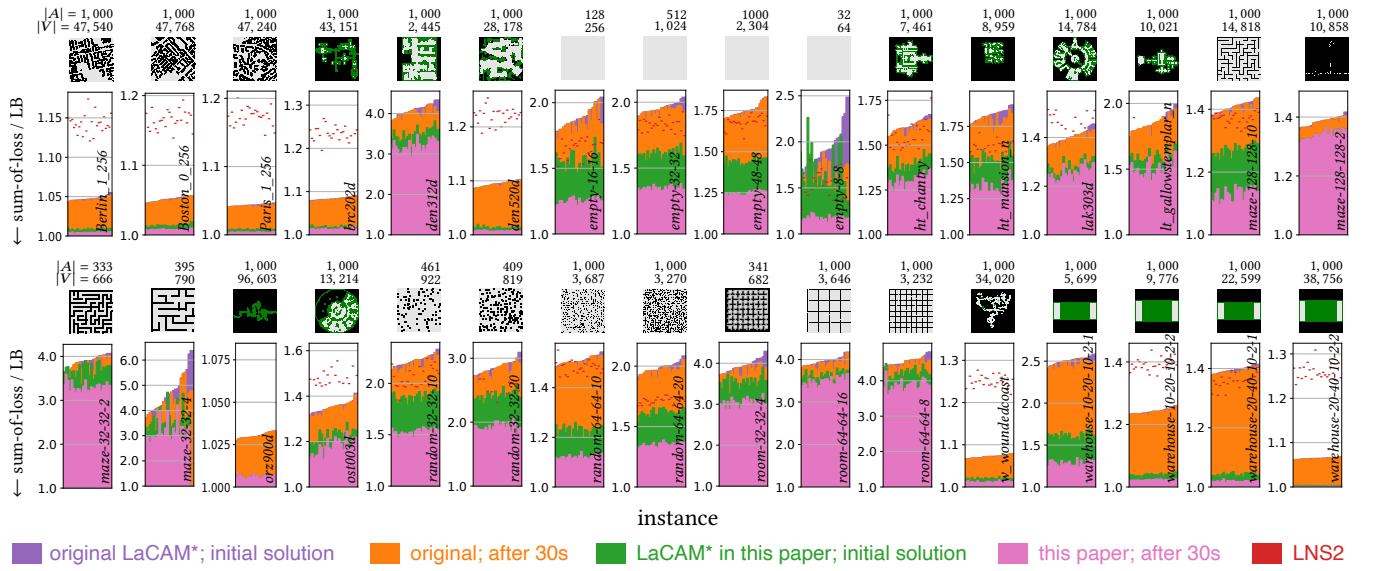
Figure 1: Solution quality improvements. Each chart illustrates the solution quality across 25 instances using a four-connected grid map, retrieved from the MAPF benchmark [28]. Solution quality is assessed through the *sum-of-loss*, quantifying the total number of actions that agents do not remain at their goals. The scores are normalized by division with their corresponding trivial lower bound, i.e., the sum of the shortest path lengths between the agents' start and goal vertices. Smaller scores are preferable (↓), with the minimum is one. For each grid, instances underwent evaluation with the maximum number of agents as specified in the benchmark, mostly a thousand agents. The instances are ordered based on the initial solution quality of the original LaCAM*. Since LaCAM* is an anytime algorithm, both the initial solution quality and the quality at the 30 s time limit are presented. Improved performance over the original LaCAM* is represented by the difference between orange and pink edges. For reference, the scores of LNS2 [11], an incomplete suboptimal approach, are included when instances were solved within the time limit (510/800 instances; 64%). The original LaCAM* solved all instances, while the improved version encountered failure in one instance of *maze-32-32-4*. The flowtime results corresponding to this figure are available in Fig. 2.

2.1 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 start vertices $\mathcal{S} = (s_i \in V)^{i \in A}$ and goal vertices $\mathcal{T} = (g_i \in V)^{i \in A}$. A *configuration* $Q = (v_1, v_2, \ldots, v_n) \in V^n$ is a tuple of locations for all agents, where $Q[i] = v_i$ is the location of agent $i \in A$. For instance, $\mathcal{S}$ and $\mathcal{T}$ represent the start and goal configurations, respectively. A configuration Q has a *vertex collision* when there is a pair of agents $i \neq j$ such that $Q[i] = Q[j]$. Two configurations X and Y have an *edge collision* when there is a pair of agents $i \neq j$ such that $X[i] = Y[j] \wedge Y[i] = X[j]$. Let neigh$(v)$ denote a set of vertices adjacent to $v \in V$. Two configurations X and Y are *connected* when $Y[i] \in \text{neigh}(X[i]) \cup \{X[i]\}$ for all $i \in A$, and there are neither vertex nor edge collisions in X and Y. Given an MAPF instance, a *solution* is a sequence of configurations $\Pi = (Q_0, Q_1, \ldots, Q_k)$, such that $Q_0 = \mathcal{S}$, $Q_k = \mathcal{T}$, and any two consecutive configurations in Π are connected. As a solution quality metric, this paper considers minimizing *sum-of-loss*, the number of agent actions that do not stay at goals. Formally, it is defined by $\sum_{t=0}^{k-1} \text{cost}_e(Q_t, Q_{t+1})$, where $\text{cost}_e(X, Y) := |\{i \in A \mid \neg(X[i] = Y[i] = g_i)\}|$.

Remarks. Finding sum-of-loss optimal solutions is NP-hard; the proof in [33] is applicable. Another popular metric to evaluate solution quality is the *flowtime* (aka. *sum-of-costs*), which accounts for the sum of the timesteps in which each agent stops at its goal and

remains there permanently. This metric is history-dependent for a sequence of configurations, making its representation in terms of cumulative transition costs intricate. However, due to the similarity in their representation, we empirically observe that the pursuit of near-optimal solutions in terms of sum-of-loss also yield near-optimal outcomes for flowtime, as evidenced in Fig. 1 and 2.

2.2 LaCAM$^{(*)}$

Given start and goal configurations, *LaCAM* [19] is a graph pathfinding algorithm with the search space defined by the configurations and their interconnections. LaCAM is complete; it returns a solution for solvable instances within a finite timeframe, otherwise reports the non-existence. *LaCAM** [18] is its anytime variant, designed to eventually find the shortest path for cumulative transition costs, even when the transition cost (i.e., cost$_e$) is not in the sum-of-loss form. Algorithm 1 provides an overview of LaCAM*, with the anytime components shaded. For clarity, two subprocedures have been omitted from the pseudocode and are detailed in the appendix. This subsection first describes the concept of LaCAM, followed by the extensions used in LaCAM* and the pseudocode.

2.2.1 Overview. Similar to the general search scheme such as A*, LaCAM performs the search by sequentially processing *search nodes* one by one, which are stored within an *Open* list. Each node corresponds to a configuration Q and maintains a pointer to a parent
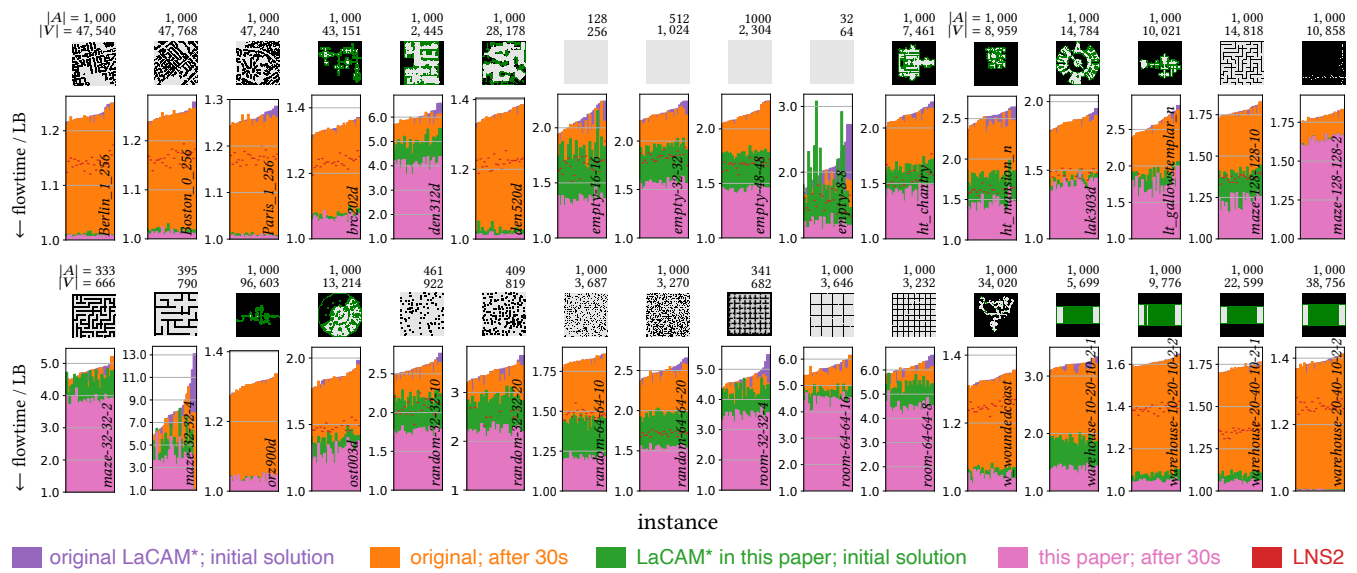
Figure 2: Solution quality improvements of flowtime (aka. sum-of-costs). See also the caption in Fig. 1. Flowtime is a widely-used metric to evaluate MAPF solutions, computed as the sum of the earliest timesteps that each agent permanently stays at its destination. Although LaCAM* primarily focuses on minimizing the sum-of-loss metric, the resultant solutions also excel in terms of flowtime.

node with another configuration Q' connected to Q. Upon encountering the goal configuration during the search, a solution is derived by backtracking of parent pointers. The structure of the *Open* list dictates the search progression. This study uses the *stack* structure, thereby LaCAM is described similarly to depth-first search (DFS).

Primary deviations from the general search scheme arise from the method of generating successors. *LaCAM generates a maximum of one successor node upon invoking a node.* In essence, it produces successors in a *lazy* fashion, and a node may be invoked multiple times during the search. The details are explained below.

2.2.2 Lazy Constraints Addition. In LaCAM, a search node for a configuration Q also encompasses the management of *constraints*, which delineate the procedure for generating a successor configuration. A constraint is a pair $i \in A$ and $v \in \text{neigh}(Q[i]) \cup \{Q[i]\}$. When this constraint is specified, a successor configuration must satisfy the condition that agent-i occupies vertex v. By carefully revising the specified constraints with each invocation of a node, it becomes possible to systematically generate all interconnected configurations stemming from that particular node.

More specifically, a search node of LaCAM preserves a tree structure of constraints, where its root has no constraints. Other tree nodes correspond to a constraint and encode multiple constraints by tracing a path to the root. During each invocation of the node, a leaf node C is chosen from this tree in a breadth-first manner. The tree will then grow incrementally by creating child tree nodes for C. These new tree nodes must specify an agent that is different from any ancestor as their constraint, otherwise, the children will not be added. After updating the tree, C is used in a successor generation to dictate the relevant constraints.

2.2.3 Configuration Generation. The final element of LaCAM is how to generate a successor configuration. *This is achieved through*

the implementation of adaptive versions of other MAPF algorithms, designed to produce a fixed-length sequence of configurations, as exemplified in [16, 20, 27]. If the configuration generator is promising in producing configurations that are close to the goal configuration, LaCAM can greatly reduce the number of node generations, which is a bottleneck in planning domains with large branching factors such as MAPF. From an empirical standpoint, the *priority inheritance with backtracking (PIBT)* algorithm [20], explained in Sec. 2.3, demonstrated outstanding performance for this sake.

2.2.4 Attaching Eventual Optimality. LaCAM* builds upon LaCAM with two extensions: *(i)* it continues the search even after encountering the goal configuration, and *(ii)* it rewrites parent pointers as required. A solution is always obtainable by backtracking after encountering the goal configuration. Furthermore, it is ensured that an optimal solution is constructed through backtracking when the *Open* list becomes empty. The objective of the rewriting process is to maintain the optimal path from the start configuration to each configuration in the already explored search space. This task is efficiently executed through the utilization of an adaptive version of Dijkstra's algorithm with additional search node components, namely, a set of nodes with a connected configuration and cost-to-come (i.e., g-value).

2.2.5 Pseudocode. Algorithm 1 embodies the concept discussed thus far. The creation of constraints is encapsulated within LowLevelSearch (Line 9), and the rewriting procedure is represented by DijkstraUpdate (Line 19). Refer to the appendix for further details of these subprocedures. The configuration generation is denoted as configuration_generator (Line 10).

The algorithm functions as follows. Following the initialization (Lines 1–3), LaCAM* proceeds by sequentially processing a search node (Lines 4–20). Invoked nodes are not immediately discarded.

Algorithm 1 LaCAM*

---

input: MAPF instance, edge cost cost_e, admissible heuristic h
output: solution, NO_SOLUTION, or FAILURE
notations: $f(\mathcal{N}) := \mathcal{N}.g + h(\mathcal{N}.Q)$; $\spadesuit := (\mathcal{N}^{\text{goal}} \neq \perp)$

1: initialize *Open* (stack), *Explored* (hash table); $\mathcal{N}^{\text{goal}} \leftarrow \perp$
2: $\mathcal{N}^{\text{init}} \leftarrow \langle Q : \mathcal{S}, parent : \perp, tree : [\![\perp]\!] \text{ (queue)}, neigh : \emptyset, g : 0\rangle$
3: *Open*.push($\mathcal{N}^{\text{init}}$); *Explored*[$\mathcal{S}$] = $\mathcal{N}^{\text{init}}$
4: **while** *Open* $\neq \emptyset$ $\wedge$ $\neg$interrupt() **do**
5:    $\mathcal{N} \leftarrow$ *Open*.top()
6:    **if** $\mathcal{N}.Q = \mathcal{T}$ **then** $\mathcal{N}^{\text{goal}} \leftarrow \mathcal{N}$
7:    **if** $\spadesuit \wedge f(\mathcal{N}^{\text{goal}}) \leq f(\mathcal{N})$ **then** *Open*.pop(); **continue**
8:    **if** $\mathcal{N}.tree = \emptyset$ **then** *Open*.pop(); **continue**
9:    $C \leftarrow$ LowLevelSearch($\mathcal{N}$)   ▷ *constraints generation*
10:    $Q^{\text{new}} \leftarrow$ configuration_generator($\mathcal{N}, C$)
11:    **if** $Q^{\text{new}} = \perp$ **then continue**
12:    **if** *Explored*[Q^{new}] = $\perp$ **then**   ▷ *new configuration*
13:      $g \leftarrow \mathcal{N}.g + \text{cost}_e(\mathcal{N}.Q, Q^{\text{new}})$
14:      $\mathcal{N}^{\text{new}} \leftarrow \langle Q : Q^{\text{new}}, parent : \mathcal{N}, tree : [\![\perp]\!], neigh : \emptyset, g : g\rangle$
15:      *Open*.push($\mathcal{N}^{\text{new}}$); *Explored*[Q^{new}] = $\mathcal{N}^{\text{new}}$
16:      $\mathcal{N}.neigh$.append($\mathcal{N}^{\text{new}}$)
17:    **else**   ▷ *known configuration*
18:      $\mathcal{N}.neigh$.append(*Explored*[Q^{new}])
19:      DijkstraUpdate($\mathcal{N}$)
20:      *Open*.push(*Explored*[Q^{new}] or *Explored*[$\mathcal{S}$])
21: **if** $\spadesuit$ **then return** backtrack($\mathcal{N}^{\text{goal}}$)
22: **else if** *Open* = $\emptyset$ **then return** NO_SOLUTION
23: **else return** FAILURE

Rather, they are set aside after the assessment of potential constraint combinations is complete (Line 8). Upon the constraint determination (Line 9), the configuration generation ensues (Line 10). If the configuration is not known for the search, a new node is created (Lines 12–16); otherwise, the rewriting process occurs (Lines 17–20). Finally, LaCAM* concludes by reporting a result (Lines 21–23).

2.2.6 Implementation Tips. The pseudocode also includes several enhancements beyond the minimal implementation.

When a known configuration is encountered, Line 20 reinstates the corresponding node to the top of the *Open* list, a strategy that has been empirically demonstrated to improve solution quality [19]. Furthermore, instead of solely reinserting the rediscovered node, inserting the initial node with a very low probability (e.g., 0.001) proves advantageous in circumventing search-stuck situations [18].

Line 7 engages in pruning by discarding a node if the sum of its cost-to-come and the estimated cost-to-go surpasses the present best solution cost. Such a node does not contribute to quality improvement. The estimation is facilitated by an *admissible* heuristic h : $V^n \mapsto \mathbb{R}_{\geq 0}$, which is admissible if the estimation consistently remains equal to or lower than the true value. For instance, $\sum_{i \in A} \text{dist}(Q[i], g_i)$ serves as an effective choice.

2.3 PIBT

As an effective implementation, the original LaCAM$^{(*)}$ papers [18, 19] employed PIBT [20] as their configuration generator. Following their success, this study also uses PIBT. The following offers a basic understanding of PIBT, sufficient for subsequent discussions.

Given a configuration Q, let ϕ_i denote an enumeration of the possible subsequent vertices for agent-i, i.e., $C_i := \text{neigh}(Q[i]) \cup \{Q[i]\}$. PIBT is conceptualized as a function that takes $\phi_1, \dots, \phi_n$ as inputs and yields a connected configuration with Q. The assignment for agent-i is executed while adhering to the order established by ϕ_i. In other words, PIBT allocates the first vertex from ϕ_i in the absence of collisions. The original PIBT arranges vertices in ascending order based on $\text{dist}(u, g_i)$, where $u \in C_i$, using randomization when resolving tie-breaking situations. Hence, it intends to prioritize the allocation of a vertex from C_i that is closest to the goal g_i.

While the original LaCAM [19] employs a vanilla PIBT, the subsequent study [18] remarkably improves the planning success rate by devising the construction of ϕ_i. Specifically, drawing inspiration from the "swap" operation in rule-based MAPF methods [4, 17], this adjustment involves reversing the order of ϕ_i when two agents require a location exchange within narrow passages. This can mostly prevent LaCAM from getting stuck in bottleneck situations where the search process repeatedly visits some specific configurations. The implementation of LaCAM* in this study also uses this trick.

2.4 Evaluation Environment

All experiments in this paper were performed on a 28-core desktop PC with Intel Core i9-10940X 3.3 GHz CPU, and 64 GB RAM. Each MAPF instance was solved sequentially; we never did a concurrent run. Meanwhile, some techniques in this study assume parallel computation, and they were implemented using multi-threading.

Hereinafter, we assess each technique under three scenarios, all of which employ a four-connected grid map obtained from the MAPF benchmark [28]: *(i)* random: *random-32-32-20* with 409 agents, *(ii)* empty: *empty-48-48* with 1,000 agents, and *(iii)* chantry: *ht_chantry* with 1,000 agents. Images of these grids are shown in Fig. 1. For each scenario, a total of 25 instances were considered. Since the implementation of LaCAM* incorporates non-deterministic elements, such as tie-breaking in PIBT, each method was executed for every instance using four distinct random seeds. Hence, the results below are for 100 trial runs. Each trial had a runtime limit of 10 s. The objective is to find the best possible solutions for the sum-of-loss within this time limit.

3 NON-DETERMINISTIC NODE EXTRACTION

We start engineering by a lightweight technique, namely, rethinking the order of node extraction from *Open* at Line 5. LaCAM* can quickly find an initial solution, facilitated by a DFS-like mechanism utilizing a standard stack as *Open*. However, sustaining the same mechanism after the initial solution discovery is not mandatory. In fact, with the vanilla stack, if the search encounters problematic configurations such that most of their neighbors have worse costs than the present solution cost, the search will stay on the nodes for a very long time due to the pruning at Line 7. Consequently, the solution refinement of LaCAM* stops.

3.1 Method

A straightforward approach to tackle this issue is to select a node from *Open* that differs from the original extraction (i.e., the top entry) with a small probability (e.g., 0.01), provided the search has

already found an initial solution. Such techniques that rely on non-determinism have been seen in other MAPF methods [1, 3], as well as LaCAM* itself (Line 20). There are design choices about which node to extract instead. We empirically tested two candidates: *(i)* a start node, i.e., $Explored[\mathcal{S}]$, denoted as "restart," partially inspired by [24], or *(ii)* a randomly selected node from $Open$, denoted as "random." Note that these modifications do not break the optimal search structure of LaCAM*.

3.2 Evaluation

Figure 3 shows that both the "restart" and "random" strategies contribute to steady improvements in the solution quality compared to the original LaCAM*, even though they started from the same initial solutions. This is because, as anticipated, the non-deterministic node extraction can escape from search-stuck situations.

4 SPACE UTILIZATION OPTIMIZATION

The solution quality of LaCAM* depends heavily on the underlying configuration generator, i.e., PIBT. Recalling the process, PIBT endeavors to allocate the first vertex in the enumeration ϕ_i to agent-i as much as possible, given a configuration Q and ϕ_i, which lists the potential subsequent positions of agent-i. In terms of implementation, ϕ_i is constructed in an ascending order based on the distance to the goal vertex g_i. This design ensures that PIBT aims to steer agents along the shortest paths to their respective goals. However, the shortest path connecting two vertices is typically not unique; numerous paths may yield identical costs. This section delves into the methodology for selecting between these equivalent paths, a decision that can significantly influence solution quality.

Figure 4 provides a concrete motivating example. Consider agent-1, which has at least two shortest paths originating from its start vertex, designated as path-a and path-b. Employing PIBT to build ϕ_1 around path-a would impose a higher cost on agent-2 compared to the alternative of selecting path-b. For larger teams of agents, this phenomenon exerts a noteworthy influence on solution quality.

A better construction strategy of ϕ_i is imperative to circumvent such issues. In pursuit of this objective, this section introduces *space utilization optimization (SUO)* [6] to PIBT, a concept that involves the use of a precomputed collection of less congested paths as an effective aid to MAPF algorithms.[1]

4.1 Method

4.1.1 Finding Scattered Paths. Such scattered paths encourage agents to evenly traverse spatiotemporal locations. Specifically, we are interested in finding a collection of paths $\Pi_{\text{SUO}} = [\pi_1, \dots, \pi_n]$, one for each agent, where *(i)* π_i is the shortest path from s_i to g_i, while *(ii)* minimizing the number of collisions in Π_{SUO}. For example, in Fig. 4, path-b should be selected for agent-1, because otherwise a collision with the unique shortest path for agent-2 will be included.

Identifying Π_{SUO} is computationally demanding, given its nature as a combinatorial optimization task, wherein the objective is to find the optimal collection of paths, with each agent potentially possessing numerous shortest paths. Rather than seeking an exact solution, [6] introduces an approximation that can be computed

---

[1]A recent preprint paper [2] also employs a similar strategy to optimize ϕ_i in PIBT for lifelong MAPF, while this paper provides a simpler yet effective method.
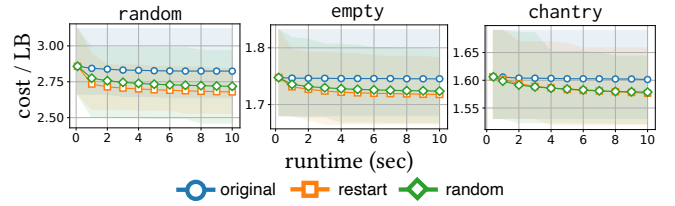
Figure 3: Effect of non-deterministic node extraction after finding initial solutions. "original" refers to the original LaCAM* [18]. All methods solved every trial within 10 s. For each second, the average sum-of-loss within the solved instances up to that point is depicted. The scores are normalized by dividing by their trivial lower bound, $\sum_i \text{dist}(s_i, g_i)$. The charts include the average scores of the initial solutions. Furthermore, they encompass the minimum and maximum scores achieved within the solved instances up to that specific time, visualized with transparent regions. For each method, we tested several probabilities for non-deterministic node extraction, and show those with consistently provided outcomes with smaller costs ("restart": 0.01, "random": 0.01).
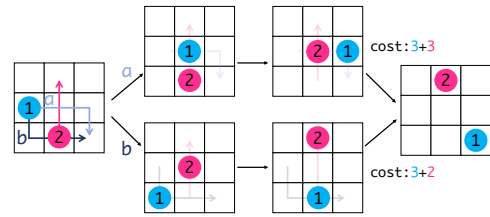
Figure 4: Motivation to compute spatially dispersed paths.

Algorithm 2 Finding scattered paths

input: MAPF instance;   **output:** Π_{SUO} (path for each agent)
hyperparameters: $m \in \mathbb{N}_{\geq 0}$
1: $\Pi \leftarrow [\bot, \dots, \bot]$
2: **while** Π is updated in the last iteration **do**
3:   **for** $i \in A$ **do**
4:     $\Pi[i] \leftarrow$ a path from s_i to g_i, such that:
        - has a length equal to or less than $\text{dist}(s_i, g_i) + m$
        - minimizes #collisions with other paths in Π
5: **return** Π

---

within a reasonable timeframe. The pseudocode for this approach is presented in Alg. 2. For the present moment, the hyperparameter m can be regarded as zero. Then, Alg. 2 iteratively computes individual shortest paths while diligently evading collisions with other paths, until no further updates are noted. The single-agent replanting process at Line 4 can be executed through A*. Note that analogous algorithmic methodologies can be seen in other MAPF methods, such as finding individual paths for the initial solution in conflict-based search [25], or deconfliction of infeasible solutions through large neighborhood search [11]. Let Π_{SUO} represent the acquired scattered paths stemming from Alg. 2, with $\Pi_{\text{SUO}}[i]$ signifying the corresponding path attributed to agent-i.

4.1.2 Exploitation. Upon the creation of Π_{SUO}, it serves as an effective guide for the PIBT configuration generator, elaborated as follows. Given a configuration Q, the construction of ϕ_i involves sorting vertices in $\text{neigh}(Q[i]) \cup \{Q[i]\}$ in an ascending order based on f_i, which is defined as:

$$f_i(u) = \begin{cases} 0 & \text{if an ordered edge } (Q[i], u) \text{ exists in } \Pi_{SUO}[i] \\ \text{dist}(u, g_i) & \text{otherwise} \end{cases}$$

This allows PIBT to prioritize the use of scattered paths in Π_{SUO} and enables it to solve an example of Fig. 4 optimally.

4.1.3 Allowing Non-Shortest Paths. The original SUO paper [6] presupposes the shortest paths at Line 4. However, our empirical observations have indicated that relaxing this condition can yield further improvements to solution quality. To elaborate, with the introduction of a hyperparameter $m \in \mathbb{N}_{\geq 0}$, we grant Line 4 the margin to seek out a path whose length extends up to $\text{dist}(s_i, g_i) + m$ for agent-i. The computation of such a path can still be executed by A^*. Larger m should be avoided due to the increase in search space, but a reasonable m allows the single-agent pathfinding to search for more scattered paths.

4.2 Evaluation

Figure 5 vividly illustrates the impact of employing spatially scattered paths in improving the quality of the initial solutions. Employing Π_{SUO} with the shortest paths already leads to a reduction in solution cost, yet greater improvements are achieved by permitting the utilization of non-shortest paths. It is worth noting that the computational overhead for computing the scattered paths does exist but remains modest. In fact, the derivation of initial solutions for scenarios involving 1,000 agents was attained within seconds. Note that the non-deterministic node extraction detailed in Sec. 3 was *not* applied to isolate the influence of SUO.
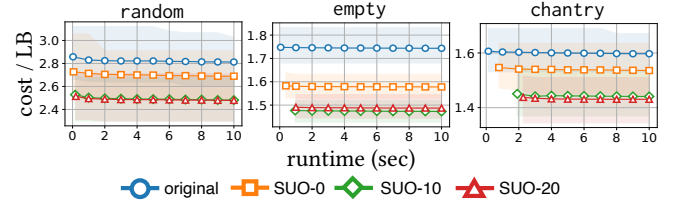
5 MONTE-CARLO CONFIGURATION GENERATION

To assign a vertex to each agent, PIBT employs an enumeration based on distances to the goal and resolves ties *non-deterministically*. Even with an identical input configuration, PIBT has the potential to produce distinct configurations. Among these configurations, certain ones might yield better initial solutions than others.

Motivated by this insight, this section presents the *Monte-Carlo configuration generator*. This concept draws partial inspiration from the Monte-Carlo tree search [8, 30], and its applicability extends to non-PIBT configuration generators.

5.1 Method

Transitioning from the original LaCAM* is straightforward; substitute the configuration generator at Line 10 in Alg. 1 with Alg. 3. The procedure entails gathering k configurations through k iterations of the generator. Subsequently, the best configuration is chosen based on the sum of the cost-to-come and the estimated cost-to-go.

Parallel Computation. Line 1 can take advantage of *parallel computation*, allowing for the simultaneous generation of numerous configurations. This can improve the quality of initial solutions without incurring a significant slowdown compared to the original

Figure 5: Effect of space utilization optimization. See also the caption of Fig. 3. The runtime includes computing Π_{SUO}. "SUO-x" means to set m in Alg. 2 to x. All methods solved all trials.

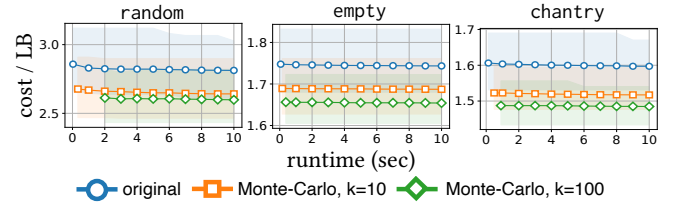**Algorithm 3** monte_carlo_configuration_generator

input: search node $\mathcal{N}$, constraints C; **output:** configuration
hyperparameters: $k \in \mathbb{N}_{>0}$ (number of samples)
1: $\mathcal{P} \leftarrow (\text{configuration\_generator}(\mathcal{N}, C))^k$ ▷ *possibly parallel*
2: **return** $argmin_{Q \in \mathcal{P}} \text{cost}_e(\mathcal{N}.Q, Q) + \text{h}(Q)$

Figure 6: Effect of Monte-Carlo configuration generation. 5/100 attempts in random failed with $k = 100$; all others succeeded.

LaCAM*. Given the prevalence of multi-core machines, leveraging the capabilities of parallel computing presents a logical strategy for the advancement of powerful MAPF methodologies.

5.2 Evaluation

Figure 6 shows the effect of the Monte-Carlo configuration generator with multi-threading. As before, the techniques introduced in previous sections were *not* applied here. Even with just ten Monte-Carlo runs, LaCAM* exhibits an enhanced capability to refine the quality of initial solutions. As the number of samples k is increased, a continued enhancement in solution quality is observed. However, there have been instances of failure when $k = 100$. This occurrence can be attributed to the behavior of the configuration generator, which becomes more deterministic with an increased number of Monte-Carlo samples. Then LaCAM* may find it difficult to escape from the wrong search direction to reach the goal. This phenomenon also accounts for the delay in acquiring initial solutions in the random scenario under the $k = 100$ condition. The empirical results of speedup by multi-threading is available in the appendix.

6 DYNAMIC INCORPORATION OF ALTERNATIVE SOLUTIONS

In optimization problems, once a feasible solution is found, improving its quality using known solutions is generally achievable. This procedure is commonly referred to as *local search*. For instance, a

traveling salesman problem (TSP) has many algorithms available to improve solution quality [23]. Analogous to TSP, multiple methods exist for refining solution quality in MAPF [4, 10, 21, 29]. While these approaches may not ensure convergence to optimal solutions, when combined with LaCAM*, they can offer a theoretical assurance that optimal solutions will eventually be obtained.

6.1 Method

During the search of LaCAM*, suppose an alternate solution Π is discovered by some means. Such scenarios can occur when a different MAPF algorithm, operating concurrently with LaCAM*, finds a solution. Another practical illustration, particularly assumed in this study, involves initiating a local search from the present LaCAM* solution using parallel computation. To integrate a new solution Π into the search process, LaCAM* can employ Alg. 4 and then proceed with its search. In essence, this entails sequentially incorporating configurations from Π to create a new search node or rewrite the search tree structure.

Each time a new solution is fed in, Alg. 4 should be called. This operation preserves the integrity of LaCAM*'s complete and optimal search structure. Furthermore, it efficiently propels the search forward when the provided solution surpass the present one.

This technique is more powerful than it seems because it offers theoretical support for various suboptimal MAPF algorithms; i.e., when coupled with LaCAM*, *it is theoretically possible to eventually find optimal solutions from arbitrary suboptimal solutions*, provided that the solution cost accounts for cumulative transition costs.

6.2 Evaluation

6.2.1 Implementation. The experiments used an iterative refinement framework for arbitrary MAPF solutions [10, 21]. This method initially selects a subset of agents and exclusively applies MAPF algorithms to them, yielding refined solutions, while treating unselected agents as dynamic obstacles. If the refined solution proves superior, it supplants the original one. At the implementation level, a random subset of $1-30$ agents was chosen for each refinement iteration. The refinement process employed prioritized planning [5, 27] combined with SIPP [22] for single-agent pathfinding, as seen in [12]. Once an initial solution is found by LaCAM*, the refinement procedures, called *refiners*, operated concurrently with LaCAM* through multi-threading, using the initial solution as input. Upon the completion of a refiner, Alg. 4 was invoked to incorporate alternative solutions, followed by the execution of another refiner utilizing the best solution at that time.

6.2.2 Results. The effect of this technique is depicted in Fig. 7 with a single refiner or four refiners. With the inclusion of a refiner, LaCAM* effectively improves solutions in the short term. Introducing additional refiners can further improve solution quality under specific circumstances. The refinement in the chantry scenario was not as fast as the other two scenarios because the subproblems for the refiners were challenging due to the complicated grid structure.

7 RECURSIVE CALL OF LaCAM*

Through the techniques introduced thus far, LaCAM* can rapidly find plausible solutions. As a result, LaCAM* itself can be harnessed to discover alternate solutions for the technique in Sec. 6. Such a

---

Algorithm 4 Incorporating new solution

1: **procedure** INCOPORATENEWSOLUTION($\Pi := (Q_0, Q_1, \ldots, Q_k)$)
2:     $\mathcal{N}^{\text{from}} \leftarrow Explored[Q_0]$; $\mathcal{N}^{\text{to}} \leftarrow \bot$
3:     **for** Q in $Q_1, Q_2, \ldots, Q_k$ **do**
4:         **if** $Explored[Q] = \bot$ **then**
5:             create $\mathcal{N}^{\text{new}}$ using Lines 13–16 in Alg. 1
6:             $\mathcal{N}^{\text{from}} \leftarrow \mathcal{N}^{\text{new}}$
7:         **else**
8:             $\mathcal{N}^{\text{to}} \leftarrow Explored[Q]$; $\mathcal{N}^{\text{from}}.neigh.\text{append}(\mathcal{N}^{\text{to}})$
9:             DIJKSTRAUPDATE($\mathcal{N}^{\text{from}}$); $\mathcal{N}^{\text{from}} \leftarrow \mathcal{N}^{\text{to}}$
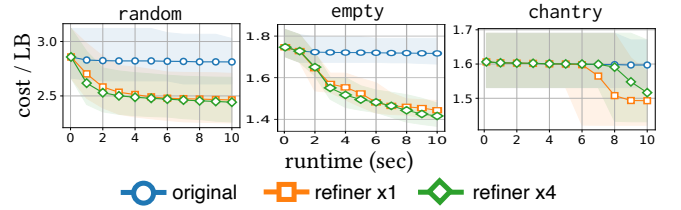

Figure 7: Effect of incorporating alternative solutions.

recursive utilization of algorithms has already surfaced in several MAPF studies [13, 31].

This concept is embodied as follows. We can generate another MAPF instance based on the present solution by selecting a configuration Q randomly from it and treating Q as a new start configuration. This new instance, where the goal configuration is unchanged from the original, is expected to discover a better-quality solution more readily than the original, owing to that Q is closer to the goal configuration compared to the start configuration of the original. Subsequently, another LaCAM* is invoked concurrently to address this new instance. Upon the completion of this recursive call, the invocation of Alg. 4 facilitates the incorporation of its outcomes into the ongoing search process.

Implementation. Each recursive call was assigned a timeout of 1 s. The recursive call did not call another LaCAM*. Our informal observations indicated that a single level of recursion was generally adequate. The recursive call was implemented as one of the refiners. More specifically, the recursive call was conducted instead of the standard refiner with a certain probability (0.2). The empirical results of this technique are detailed in Sec. 8.

8 PUTTING EVERYTHING TOGETHER

8.1 Effect of Combined Use

This section jointly evaluates the introduced techniques. Figure 8 shows the results corresponding to the previous evaluations. The combined use substantially improves solution quality, with the recursive call further amplifying the improvement. Notably, a nearly 30% reduction in solution cost was realized in the random scenario.

Figure 9 shows the ablation results for the combined use of the techniques in this paper, indicating that all techniques contribute to improving solution quality. Specifically, SUO and the dynamic incorporation of alternative solutions have a significant impact.

8.2 Evaluation on Large-Scale MAPF Instances

We applied the combined use, including the recursive call, to various challenging MAPF instances retrieved from the MAPF benchmark [28]. The benchmark encompasses 33 grid maps, each housing 25 "random" scenario files, each of which outlines a list of start-goal pairs. We procured 800 instances from the benchmark by using 32 grid maps,[2] each with 25 instances, while adhering to the specified maximum number of agents outlined in the scenario files. The allotted time limit stood at 30 s, in alignment with [28]. A single random seed was employed during testing. We also tested LNS2 [11], an incomplete suboptimal MAPF algorithm, as reference records. LNS2 was chosen because it was the only algorithm we were aware of, other than PIBT and LaCAM, that could handle such difficult cases. Note that LNS2 still failed to solve 290 instances (36%).

The results are summarized in Fig. 1 and illustrate the significant improvement in solution quality achieved by the proposed techniques. Furthermore, we present flowtime, a commonly used metric for assessing MAPF solutions, in Fig. 2. Although LaCAM* is designed to minimize the sum-of-loss, these techniques also result in a dramatic reduction in the flowtime metric.

Additional observations include the following. In certain instances where agents are not densely placed, e.g., *Berline_1_256*, the enhanced version generates solutions close to optimal, despite the significant disparities present in the original LaCAM*. Moreover, the quality of the initial solution is already comparable to or better than that of LNS2 in most cases. Meanwhile, the enhancements do not invariably guarantee cost reduction, especially about the initial solution quality, e.g., *empty-8-8*. Some cases are still difficult to refine; see *maze-128-128-2*. Note that the enhanced LaCAM* encountered *one* scenario it could not solve in the *maze-32-32-4* instance for the same reason stated in Sec. 5.

8.3 Evaluation with Ten Thousand Agents

We finally ventured into an extreme scenario – MAPF instances involving 10,000 agents. For this test, we prepared 25 instances using the *warehouse-20-40-10-2-2* map. This time, the time limit was set to 5 min. Such instances were unsolvable in a realistic timeframe for search-based MAPF algorithms prior to the advent of LaCAM [19]. Given the enormity of the problem instances, we implemented a timeout mechanism for the SUO technique, restricting its execution to half of the time limit; i.e., Alg. 2 compelled the premature generation of scattered paths, even before the iterative process stabilized. Furthermore, a timeout of 10 s was instituted for the recursive call.

Displayed in Fig. 10, the outcomes highlight the superiority of the enhanced version of this paper over the original. We further assessed multiple versions with specific features omitted to examine the individual contributions to the enhancements achieved. Among them, the SUO technique demonstrated a substantial impact, despite the initial computational overhead involved. While the other techniques exhibited discernible performance enhancements for LaCAM* in moderately large instances as presented so far, their effects were less pronounced in the context of these immense instances (hence, omitted from the figure). For such huge instances, it remains challenging to refine solution quality effectively.

---

[2]The *maze-128-128-1* map was excluded due to the original LaCAM*'s inability to solve instances featuring the maximum number of agents from this map [18].
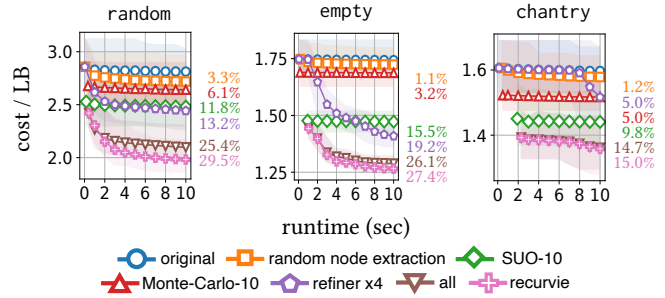
Figure 8: Effect of combining techniques. "all" denotes a combined use of the techniques in Sec. 3–6. "recursive" uses the recursive call of LaCAM* described in Sec. 7, in addition to the techniques in "all." The graphs also show the results of each technique for comparison. Their hyperparameters are shown on the label. "all" and "recursive" are also based on these parameters. The improvement rates are shown on the right side of each chart, where the baseline is the original LaCAM* (blue circles).
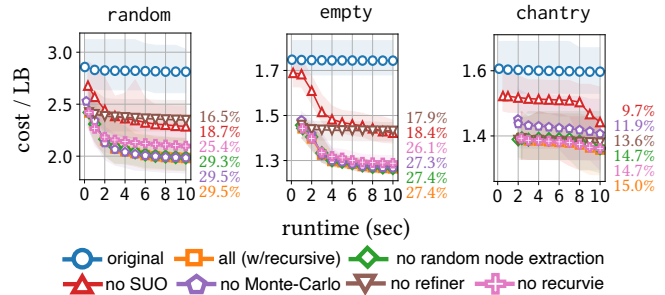
Figure 9: Ablation study. "no SUO" drops the SUO feature from "all"; same for others. The same parameters were used with Fig. 8.
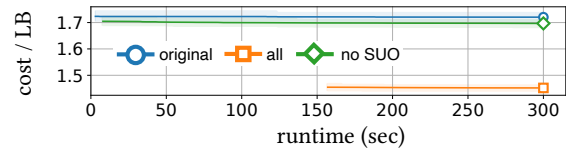
Figure 10: Results for 10,000 agents in *warehouse-20-40-10-2-2*. "all" includes all techniques including the recursive call.

9 CONCLUSION

This study introduced various techniques to improve the solution quality of LaCAM*. While each technique itself is an adjustment influenced by established methods from search or MAPF literature, their amalgamated utilization yielded substantial improvements over the original LaCAM*. With these results, we believe that this study continues to push the boundaries of MAPF, signifying a stride towards real-time, large-scale, and near-optimal MAPF.

REFERENCES

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

[2] Zhe Chen, Daniel Harabor, Jioyang Li, and Peter J Stuckey. 2023. Traffic Flow Optimisation for Lifelong Multi-Agent Path Finding. *arXiv preprint arXiv:2308.11234* (2023).

[3] Liron Cohen, Glenn Wagner, David Chan, Howie Choset, Nathan Sturtevant, Sven Koenig, and TK Satish Kumar. 2018. Rapid randomized restarts for multi-agent path finding solvers. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*.

[4] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. 2014. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research (JAIR)* (2014).

[5] Michael Erdmann and Tomas Lozano-Perez. 1987. On multiple moving objects. *Algorithmica* (1987).

[6] Shuai D Han and Jingjin Yu. 2022. Optimizing space utilization for more effective multi-robot path planning. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*.

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

[8] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European Conference on Machine Learning (ECML)*.

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

[10] Jiaoyang Li, Zhe Chen, Daniel Harabor, P Stuckey, and Sven Koenig. 2021. Anytime multi-agent path finding via large neighborhood search. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

[11] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

[12] Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig. 2021. Scalable rail planning and replanning: Winning the 2020 flatland challenge. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*.

[13] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search.. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

[14] Jiaoyang Li, Daniel Harabor, Peter J Stuckey, and Sven Koenig. 2021. Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search. *Artificial Intelligence (AIJ)* (2021).

[15] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. 2021. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

[16] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. 2021. Lifelong multi-agent path finding in large-scale warehouses.

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

[17] Ryan Luna and Kostas E Bekris. 2011. Push and swap: Fast cooperative pathfinding with completeness guarantees. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

[18] Keisuke Okumura. 2023. Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

[19] Keisuke Okumura. 2023. LaCAM: Search-Based Algorithm for Quick Multi-Agent Pathfinding. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

[20] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. 2022. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. *Artificial Intelligence (AIJ)* (2022).

[21] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. 2021. Iterative Refinement for Real-Time Multi-Robot Path Planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[22] Mike Phillips and Maxim Likhachev. 2011. Sipp: Safe interval path planning for dynamic environments. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*.

[23] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. 2011. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research (EJOR)* (2011).

[24] Silvia Richter, Jordan Thayer, and Wheeler Ruml. 2010. The joy of forgetting: Faster anytime search via restarting. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*.

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

[26] Bojie Shen, Zhe Chen, Muhammad Aamir Cheema, Daniel D Harabor, and Peter J Stuckey. 2023. Tracking Progress in Multi-Agent Path Finding. *arXiv preprint* (2023).

[27] David Silver. 2005. Cooperative Pathfinding. In *Proceedings of AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

[28] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*.

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

[30] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. 2023. Monte Carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review* (2023).

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

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

[33] Jingjin Yu and Steven M LaValle. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs.. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.