Minimizing Makespan with LaCAM* for Optimal Multi-Agent Path Finding

Anonymous Author(s)
Submission Id: 636

ABSTRACT

Multi-Agent Path Finding (MAPF) requires collision-free paths for a set of agents. Optimal MAPF solutions often minimize one of two cost functions: (1) *sum-of-costs* (SOC), which is the sum of the costs of the paths, or (2) *makespan* (MKS), the maximum of these costs. *LaCAM** is a recent anytime MAPF solver, which eventually converges to the optimal solution. However, LaCAM* has a considerably slow convergence speed to the optimum and is considered a scalable near-optimal solver. While this is true for SOC, in this paper, we show that LaCAM* can quickly converge to the optimal MKS solution. We explore LaCAM* and an improved version LaCAM*2 with three search methods: DFBnB, A*, and IDA*. We present their superb performance for finding optimal MKS solutions, even with thousands of agents.

KEYWORDS

Multi-Agent Path Finding, Makespan, LaCAM*, A*, Combinatorial Search

ACM Reference Format:

Anonymous Author(s). 2026. Minimizing Makespan with LaCAM* for Optimal Multi-Agent Path Finding. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 8 pages.

1 INTRODUCTION

The task in the *Multi-Agent Path Finding* problem (MAPF) is to find conflict-free (collision-free) paths for multiple agents [44]. MAPF is derived from real-world applications of navigating multiple physical entities, such as drones, robots, vehicles, etc. There are two common objective functions that quantify the cost of MAPF solutions: *sum-of-costs* (SOC) and *makespan* (MKS). SOC is the sum of the costs of all paths, and MKS is the highest cost among all paths' costs. It is NP-hard to optimally solve MAPF for both SOC [51] and MKS [45]. However, advanced algorithms are capable of optimally solving MAPF for many agents [13, 20, 30, 39].

*LaCAM** [33] is a recent anytime solver, which extends the suboptimal algorithm *LaCAM* [34] and converges to the optimal MAPF solution (for SOC or MKS). Both LaCAM and LaCAM* search in a *vertex configuration space*, where each node represents the vertices of the entire set of agents. In contrast, *Conflict-Based Search* (CBS) [39], a common optimal MAPF algorithm, searches in a *path configuration space*, where each node represents a set of paths for all agents. Each of the two algorithms LaCAM* and CBS has its own different forte; while LaCAM* is capable of quickly finding (unbounded) near-optimal solutions (for SOC or MKS), CBS is better suited for finding optimal or bounded-suboptimal solutions (again, for SOC or MKS) [4, 7, 8, 11, 22, 23, 25, 30, 39].

A primary weakness of LaCAM* is that it explores the vertex configuration space, in which the number of successors of each node and, in particular, the size of the state space is exponential in the number of agents. Moreover, to prove optimality, as in standard heuristic search, every node that may lead to a better solution (according to a heuristic function) *must* be expanded [9]. In this paper, we challenge the weakness of LaCAM* and aim to use the algorithm to find optimal solutions. In particular, we show that LaCAM* can quickly find optimal MKS solutions when it is executed on large and sparse maps. As we explain, this is due to its accurate heuristic estimate for MKS. Moreover, we experimentally demonstrate that, in such maps, LaCAM* significantly outperforms CBSм, a CBS-based algorithm designed for MKS, making LaCAM* the state-of-the-art optimal MAPF solver for MKS in these maps. Therefore, we explore various search strategies for LaCAM*. Standard LaCAM* (denoted LaCAM*-DFBnB) employs a *depth-first branch-and-bound* search (DFBnB); nodes are explored in a *depth-first search* (DFS) manner until a solution is found and, then, additional nodes are explored until the optimal solution is determined. We then introduce two new versions of LaCAM*: LaCAM* with an *A** search (LaCAM*-A*) as well as LaCAM* with an *IDA** search (LaCAM*-IDA*). We also apply these two strategies to LaCAM*2 [35], an improved version of LaCAM*. We experimentally show that LaCAM*2 with the new search strategies (LaCAM*2-A* and LaCAM*2-IDA*) outperforms other methods in finding optimal MKS solutions, presenting state-of-the-art performance on various benchmark maps. For instance, with a time limit of 60 seconds, LaCAM*2-IDA* reaches 100% success rate for optimally solving MAPF for MKS on a city benchmark map (Berlin_1_256) containing 3,000 agents.

2 BACKGROUND

2.1 Path Finding

A *path* π in graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ from start vertex $s \in \mathcal{V}$ to goal vertex $g \in \mathcal{V}$ is a list of vertices such that the list starts with vertex s and ends with vertex g, and any two consecutive vertices in the list must be traversable. Let $\pi(t)$ denote the t-th vertex in path π. Formally, $\pi(0) = s$, $\pi(|\pi| - 1) = g$, and $\forall t : (\pi(t), \pi(t+1)) \in \mathcal{E}$. A *Path Finding* problem (PF) is defined by a tuple $\langle \mathcal{G}, s, g \rangle$ and requires such a path. The cost $C(\pi) = |\pi| - 1$ of path π is equal to the number of transitions performed in it.[1] The *optimal* solution (optimal path) to PF is the lowest-cost path among all PF solutions (paths).

Finding the optimal path to PF can be efficiently accomplished using various search algorithms, which track their search by nodes (where each node represents a vertex in the given graph), maintained in a data structure called Open.

[1] This work focuses on a unit-cost graph, which is common in Multi-Agent Path Finding.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus.

A.* A* [16], a well-known *Best-First Search* (BFS) heuristic search algorithm, uses a heuristic function $h(v)$ that estimates the cost to reach the goal vertex g from any given vertex v. Let $C^*(v)$ denote the cost of the optimal path from a given vertex v to vertex g. A heuristic function is called *admissible* if it never overestimates, i.e., $\forall v : h(v) \leq C^*(v)$. Until the goal is expanded, A* iteratively expands the node with the lowest $f(v) = g(v) + h(v)$, where $g(v)$ is the cost of reaching v from the start vertex s, and generates its successors. Given an admissible heuristic function, A* is guaranteed to return the optimal path.

Depth-First Branch-and-Bound (DFBnB). DFBnB [18] is an anytime PF algorithm, which converges to the optimal solution. It performs a *Depth-First Search* (DFS), which explores as far as possible along each branch before backtracking. After a solution is found, DFBnB continues the search and improves the quality of the solution while discarding nodes whose f-value exceeds the cost of the lowest-cost solution found. DFBnB halts when Open is empty.

Iterative Deepening A (IDA*). IDA** [17] is a heuristic search algorithm, which performs a DFS in iterations, according to the lowest f-value observed, ensuring that the first solution found is optimal. All three algorithms A*, DFBnB, and IDA* often use another data structure, called Closed, to avoid re-expanding duplicate nodes.

2.2 Multi-Agent Path Finding

The *Multi-Agent Path Finding* problem (MAPF) [44], which generalizes PF for multiple agents, is defined by the tuple $\langle \mathcal{G}, A, S, G \rangle$, where $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph, $A = (a_1, \ldots, a_k)$ is a set of k agents and $S = (s_1, \ldots, s_k)$ and $G = (g_1, \ldots, g_k)$ are lists of start and goal vertices for the k agents, respectively. A path π_i for agent a_i is a path from start vertex s_i to goal vertex g_i, representing the movement actions of the agent between each consecutive timesteps t and $t+1$, and it is composed of *move* actions (where $\pi_i(t) \neq \pi_i(t+1)$) and *wait* actions (where $\pi_i(t) = \pi_i(t+1)$).

A *plan* $\Pi = (\pi_1, \ldots, \pi_k)$ is a list of paths for the agents. A *solution* to MAPF is a *conflict-free* plan Π; that is, any two paths in plan Π do not *conflict*. For any two paths π_i and π_j of agents a_i and a_j, we consider the following two common types of conflicts.

1. A *vertex conflict* $\langle a_i, a_j, v, t \rangle$ exists when the agents are simultaneously at vertex v at timestep t ($\exists t : \pi_i(t) = \pi_j(t) = v$).

2. A *swapping conflict* $\langle a_i, a_j, e, t \rangle$ exists when the agents traverse edge e in opposite directions between timesteps t and $t+1$ ($\exists t : \pi_i(t) = \pi_j(t+1) \wedge \pi_j(t) = \pi_i(t+1) \wedge (\pi_i(t), \pi_i(t+1)) = e$).

Objective functions. The objective functions for MAPF are defined as follows. The *sum-of-costs* (SOC) of plan Π is

$$C_{SOC}(\Pi) = \sum_{\pi_i \in \Pi} C(\pi_i).$$

The *makespan* (MKS) of plan Π is

$$C_{MKS}(\Pi) = \max_{\pi_i \in \Pi} C(\pi_i).$$

An *optimal* MAPF solution minimizes either SOC or MKS.

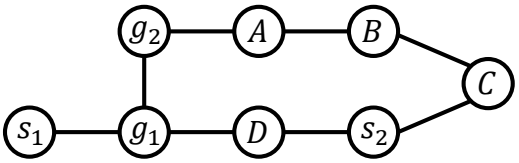

Figure 1: MAPF problem instance with different SOC and MKS optimal solutions.

Example. Figure 1 presents a MAPF problem instance containing two agents a_1 and a_2, with start vertices s_1 and s_2, and goal vertices g_1 and g_2. Let $\Pi^1 = (\pi_1^1, \pi_2^1)$ such that $\pi_1^1 = (s_1, g_1)$ and $\pi_2^1 = (s_2, C, B, A, g_2)$, and let $\Pi^2 = (\pi_1^2, \pi_2^2)$ such that $\pi_1^2 = (s_1, s_1, s_1, g_1)$ and $\pi_2^2 = (s_2, D, g_1, g_2)$. For Π^1, $C_{SOC}(\Pi^1) = 5$ and $C_{MKS}(\Pi^1) = 4$. For Π^2, $C_{SOC}(\Pi^2) = 6$ and $C_{MKS}(\Pi^2) = 3$. Therefore, Π^1 is better (and optimal) for minimizing SOC while Π^2 is better (and optimal) for minimizing MKS.

2.3 LaCAM

LaCAM [34] is a MAPF algorithm, capable of quickly finding suboptimal solutions. LaCAM has two search levels. On the high level, the algorithm is executed in a composite graph $\mathcal{G}^c = (\mathcal{V}^c, \mathcal{E}^c)$, where each composite vertex $v^c \in \mathcal{V}^c$ contains a *configuration* (a list of the vertices of the entire set of agents). The start and goal composite vertices are S and G, respectively (the start and goal vertices of the agents). A composite edge $(v_1^c, v_2^c) \in \mathcal{E}^c$ from $v_1^c = (v_{11}^c, \ldots, v_{1k}^c)$ to $v_2^c = (v_{21}^c, \ldots, v_{2k}^c)$ is any possible movement of all agents: $\forall v_{1i}^c : (v_{1i}^c, v_{2i}^c) \in \mathcal{E}^c \vee v_{1i}^c = v_{2i}^c$, excluding that where all agents wait $v_1^c \neq v_2^c$ or where two agents conflict.

Each high-level node contains a specific configuration (composite vertex). The high level iteratively chooses a node from Open and only partially expands it. That is, given a node n it generates a single successor of n, and n remains in Open until all successors are generated. This method of partially expanding a node while only generating a single successor is also known as *Enhanced partial expansion A** (EPEA*) [15]. Choosing the configuration for the next successor to be generated is performed by (1) a low-level search and (2) a configuration generator. To ensure that all possible configurations are considered for the successors, the low level uses a positive constraint, enforcing an agent to perform a specific move. Starting from no constraints, the low level gradually constrains the agents to perform any possible movement. This is done using a tree, maintained in each high-level node, where each tree level considers a different agent and every low-level successor imposes a positive constraint on the agent. Therefore, each branch represents a specific movement for some of the agents. For a specific branch, the configuration generator completes the movement for the agents that have no constraints. The configuration generator can be implemented by different possible methods. It has been experimentally demonstrated that using the *PIBT* algorithm [36] for the configuration generator outperforms other methods. PIBT is a fast MAPF algorithm that moves the agents towards their goals while preferring agents with a higher priority. When needed, PIBT lets a lower-priority agent inherit a higher priority.

Algorithm 1: LaCAM

```
1  LaCAM (MAPF instance)
2      init Open, Closed; N^init ← ⟨S, [[C^init]], null⟩
3      Open.insert(N^init); Closed[S] = N^init
4      while Open is not empty do
5          N ← Open.peek()
6          if N.config = G then return N
7          if N.tree = ∅ then Open.extract(); continue
8          C ← low_level_search(N)
9          Q^new ← configuration_generator(N, C)
10         if Q^new = null then continue
11         if Closed[Q^new] ≠ null then continue
12         N^new ← ⟨Q^new, [[C^init]], N⟩
13         Open.insert(N^new); Closed[Q^new] ≠ N^new
14     return No Solution
```

Pseudo-code. The pseudo-code of LaCAM's high-level can be found in Algorithm 1. Each node represents $\langle config, tree, parent \rangle$, a tuple of a configuration ($config$), low-level tree that is used to track the low-level constraints ($tree$), and a parent-pointer ($parent$). First, LaCAM initialize Open and Closed with a root node $\mathcal{N}^{init}$ (Lines 2-3). C^{init} means "no constraints". While Open is not empty, an expansion cycle is performed as follows (Lines 4-13). The next node $\mathcal{N}$ from Open is examined (Line 5). If it has the goal configuration G, then $\mathcal{N}$ is returned (Line 6), and the solution can be easily constructed using the parent-pointers. Otherwise, if all constraint combinations are complete, then $\mathcal{N}$ is discarded and a new expansion cycle begins (Line 7). If they are not complete, the next constraint is set and a new configuration Q^{new} is generated (Lines 8-9). A new node $\mathcal{N}^{new}$ is created based on the new configuration Q^{new} and inserted into Open and Closed (Lines 12-13) only if the configuration Q^{new} could have been generated (it is possible to generate a new configuration under the low-level constraints) and the configuration Q^{new} is not already explored (Lines 10-11).

2.3.1 LaCAM. LaCAM** [33] is a successor of the suboptimal MAPF algorithm *LaCAM* [34], capable of finding optimal solutions. The main difference between LaCAM and LaCAM* is that LaCAM performs a DFS (where Open is a stack) and halts when a solution is chosen for expansion and LaCAM* performs a DFBnB, prunes nodes with an f-value that is *higher than or equal to* the cost of the lowest-cost solution found, and halts only when Open is empty, and the optimal solution is found. For calculating the f-value of nodes, LaCAM* uses a heuristic function that estimates the cost of reaching a goal for each agent separately, assuming a free space, i.e., ignoring all other agents. Trivially, for SOC, the heuristic value is equal to the sum of all these estimates and, for MKS, the heuristic value is equal to the maximum value of these estimates. For example, consider again the problem instance presented in Figure 1. A heuristic function h with a free-space assumption estimates, for agent a_1, $h(s_1, g_1) = 1$ and, for agent a_2, $h(s_2, g_2) = 3$. At the root node of LaCAM*, we have the initial configuration of the start vertices of the agents (s_1, s_2). Therefore, for SOC, the heuristic value is equal to 4 and, for MKS, the heuristic value is equal to 3. We can see that, in this example, the heuristic value for MKS is accurate and equal to the cost of the optimal MKS solution. However, the heuristic value for SOC is inaccurate. As we explain below, the heuristic value for MKS is often accurate, making it suitable for finding optimal MKS solutions.

*2.3.2 LaCAM*2.* Recently, an improved version of LaCAM* was introduced [35]. We denote it as LaCAM*2, which has five main modifications, as follows.

(1) Non-deterministic node extraction. To prevent progressing toward the goal configuration in the wrong direction during the search, in some cases, stochastically, LaCAM*2 chooses a random node from Open and continues the search from that randomly selected node.

(2) Space utilization optimization (SUO/Scatter). When choosing a configuration using the configuration generator, the agents may eventually become congested in narrow spaces. To prevent multiple agents from reaching such areas, this modification also prefers configurations that better scatter the agents.

(3) Monte-Carlo configuration generation. The low-level search and the configuration generator have a stochastic nature for choosing a configuration for the successor. Therefore, in LaCAM*2, a few random configurations are sampled and examined, and the best one is ultimately selected.

(4) Dynamic incorporation of alternative solutions (iterative). A solution that is found by the algorithm is in the form of a path in the composite graph. When such a solution is found, LaCAM*2 tries to improve it by iteratively choosing a single agent and two vertices along its path. Then, LaCAM*2 aims to find a shorter path between the two vertices, which does not conflict with all other paths. If such a path is found, the best solution found is updated.

(5) Recursive use of LaCAM.* Similar to Modification 4 above, after a solution is found, LaCAM*2 tries to improve it. This is done by LaCAM*2 recursively calling itself between two composite nodes along the solution found. While Modification 4 is aimed at finding a shorter sub-path for a single agent, here, the aim is to find a better sub-plan for all agents.

2.4 Conflict-Based Search (CBS)

Conflict-Based Search (CBS) [39] is an optimal MAPF algorithm for either SOC or MKS. In CBS, a constraint $\langle a_i, x, t \rangle$ (x is either a vertex or an edge) prohibits agent a_i from occupying vertex x at timestep t or from traversing edge x between timesteps t and $t+1$. Note that, in contrast to the constraint used in LaCAM, which is *positive* and enforces the agent to perform a specific action, the constraint in CBS is *negative*. CBS is a two-level algorithm. On its high level, CBS contracts a *Constraint Tree* (CT). Each CT node N contains: (1) a set of constraints, denoted $N.constraints$; (2) a plan $N.\Pi$ that satisfies $N.constraints$; and (3) the cost $N.cost$ of plan $N.\Pi$. $N.cost$ can be either $C_{SOC}(\Pi)$ or $C_{MKS}(\Pi)$ depending on whether the aim is to minimize SOC or MKS. The path of each agent a_i in plan $N.\Pi$ (denoted $N.\Pi.\pi_i$) satisfying $N.constraints$ is planned by CBS's low-level search.

The high level performs a BFS over the CT nodes by prioritizing CT nodes according to their costs. It starts from initializing a *Root* CT node containing an empty set of constraints and inserting it into Open. Then, repeatedly, CBS extracts the lowest-cost CT node N from Open. If $N.\Pi$ is conflict-free, it is returned as a solution. Otherwise, a conflict $\langle a_i, a_j, x, t \rangle$ is chosen. To resolve the conflict, two new child CT nodes N_i and N_j are created for node N with the constraints $N.constraints$ and the additional constraints $\langle a_i, x, t \rangle$ and $\langle a_j, x, t \rangle$ are added to N_i and N_j, respectively. The new CT nodes N_i and N_j are inserted into Open.

When a new constraint is added to agent a_i in a new CT node N_i (to resolve a conflict), the low level is called to replan the lowest-cost path for agent a_i that satisfies the new set of constraints in N_i. The lowest-cost low-level search can be implemented by (Temporal-)A* [42], which executes A* but must satisfy the constraints.

CBSm. Many improvements for CBS were introduced over the years [6, 7, 11, 22–24, 41, 52]. Recently, a CBS version specifically adapted for minimizing MKS was proposed [30]. We denote this version of CBS by *CBSm*. Instead of calling, in CT node N, a low level that finds the lowest-cost path, it uses a low level that finds a bounded-cost path, which is bounded by the cost of the longest path (MKS) currently existing in CT node N. If no such path exists, then it calls a lowest-cost low-level search. To the best of our knowledge, CBSm is the state-of-the-art MAPF solver for MKS.

3 RELATED WORK

Due to its applicative nature, MAPF has been solved by different algorithms and extended in various directions, as follows.

3.1 Solvers

There are two main categories of MAPF solvers: search-based and reduction-based. Search-based algorithms systematically explore different paths (or sub-paths) until a solution can be determined. Beside, LaCAM* and CBS, search-based algorithms that optimally solve MAPF include *Cooperative A** (CA*) [42], *Operator Decomposition* (OD) [43], *Independence Detection* (ID) [43], *M** [48], *Increasing Cost Tree Search* (ICTS) [40], and *Branch and Cut and Price* (BCP) [20]. Reduction-based algorithms compile MAPF into another known problem that has mature and effective solvers. Previous studies on reduction-based approaches include reducing MAPF to *Integer-Linear Program* (ILP) [50], *Answer Set Programming* (ASP) [10, 32], *SAT* [5, 46], and more. In this paper, we only consider CBSm in our experiments because it experimentally outperformed other search-based and reduction-based algorithms for optimally solving MAPF for MKS.

3.2 Extensions

The standard MAPF may not capture all constraints that exist in real-life applications. Therefore, many studies extended the classical definition of the problem. These include cases where agents have different sizes and may occupy multiple locations at a single timestep [26]; cases where agents have imperfect execution, and the solutions are designed to withstand delays [2, 3, 38]; cases where goals have deadlines [12, 29]; online/lifelong scenarios [27, 28, 31, 47, 49], where new agents/tasks arrive over time; cases where time is continuous [1]; and MAPF with other objective functions, such as *Fuel* [14, 19], which takes into account only move actions. While this paper studies the standard MAPF problem, our methods can be generalized for other MAPF extensions, such as the ones above.

4 VARYING LACAM*'S SEARCH STRATEGY

As mentioned, LaCAM performs a DFS on its high level. The main reason for using this particular search is that the algorithm was developed to find a (suboptimal) solution as fast as possible (the original paper was titled *"LaCAM: Search-Based Algorithm for Quick Multi-Agent Pathfinding"*) [34]. Following the same direction, LaCAM* (titled *"Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding"*) [33] performs a DFBnB, which improves the quality of the solution found by continuing the search until the optimal solution is reached. LaCAM* aimed to be scalable (the author mentions that *"it [LaCAM*] suboptimally solved 99% of the instances retrieved from the MAPF benchmark"*) and it only eventually converges to the optimal solution (*"the convergence speed was slow in large instances with many agents"*). However, as we show, LaCAM* can optimally solve many large instances containing many agents when MKS is minimized.

Standard LaCAM* performs a DFBnB on its high level and, therefore, we denote it as *LaCAM*-DFBnB*. Indeed, DFBnB is an anytime algorithm that converges to the optimal solution. However, in this paper, we are only interested in finding optimal solutions. Hence, different known search strategies can also be employed while maintaining the optimality of the returned solution. We also considered the following search strategies for LaCAM*: A* (LaCAM*-A*) and IDA* (LaCAM*-IDA*). Likewise, for LaCAM*2, we consider LaCAM*2-A* and LaCAM*2-IDA*. We note that, some of the modifications made for LaCAM*2-DFBnB are not relevant to LaCAM*2-A* and LaCAM*2-IDA*. In particular, Modifications 4 and 5 (mentioned in Section 2.3) are used to quickly refine the quality of the solution found. However, the first solution found by LaCAM*2-A* and LaCAM*2-IDA* is optimal, and the algorithms immediately halt. Therefore, these two modifications are not relevant to them. In our experiments below, we also added a version of LaCAM*2-DFBnB that as well does not include these two refiners (Modifications 4 and 5), denoted as LaCAM*2-DFBnBnr.

4.1 Node Expansion and Runtime

A common classification divides nodes expanded by any search algorithm into three groups [9]: *must-expand*, *maybe-expand*, and *never-expand*. Let C^* denote the cost of the optimal solution. A must-expand node is every node with $f < C^*$; it must be expanded to guarantee optimality. A maybe-expand node is every node with $f = C^*$; it may or may not be expanded. A never-expand node is every node with $f > C^*$; it can be prevented from being expanded. Therefore, when an algorithm expands never-expand nodes, they are often referred to as *surplus nodes*, as it was not required to expand them in order to determine the optimality of the solution.

Example. Consider the search tree illustrated in Figure 2, in which there are three goal nodes in depths 2, 3, and 4. Assume a unit-edge cost, a zero heuristic value for all nodes, and a DFS that prefers the left successor first. Here, $C^* = 2$. The black nodes in depths 0 and 1 are must-expand nodes, the dark-gray nodes are maybe-expand nodes, and the light-gray nodes are never-expand nodes, and can be

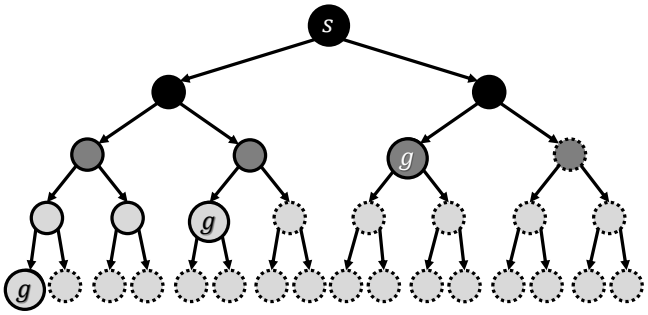

Figure 2: Nodes classification. Black: Must-expand nodes; Dark gray: Maybe-expand nodes; Light gray: Never-expand nodes (surplus nodes). Solid-line circle: Expanded by DFBnB.

avoided from expansion (surplus). In this example, DFBnB gradually improves its solution and expands all nodes represented by a solid-line circle. Here, DFBnB only expands four surplus nodes, however, on larger tree, it may expand many such nodes. In contrast, A* and IDA* never expand these nodes.

DFBnB, A, and IDA**. When search algorithms are executed, the number of expansions is not the only factor that impacts their runtime. Consider DFBnB, A*, and IDA*. Each has scenarios where it performs worse and better due to its own weaknesses and strengths. Weaknesses: DFBnB deepens the search and may reach and expand surplus nodes, and it halts only when Open is empty; A* is required to order Open, which incurs runtime; IDA* may perform multiple iterations until the optimal solution is found. Strengths: A* and IDA* do not expand surplus nodes and halt when a solution is expanded; DFBnB and IDA* are not required to order Open; DFBnB and A* only perform a single iteration. Next, we evaluate experimentally LaCAM* and LaCAM*2 with the different search strategies.

5 EXPERIMENTS

In this section, we conduct an extensive empirical study on standard benchmark maps from the *MovingAI* repository [44]. As the number of agents for each map in the benchmark is limited (usually up to 1,000 agents), we generated problem instances using the *MAPF-LNS2* [21] instance generator using the publicly available implementation.[2] For each map, 25 problem instances were created. We experimented with CBSm, LaCAM*-DFBnB, LaCAM*-A*, LaCAM*-IDA*, LaCAM*2-DFBnB, LaCAM*2-A*, LaCAM*2-DFBnBnr, and LaCAM*2-IDA*. We set the time limit to 60 seconds. We will make our implementation publicly available upon acceptance.

5.1 Comparing SOC vs. MKS

We first examined the difference in finding suboptimal and optimal solutions for minimizing SOC and MKS, for LaCAM*2-DFBnB. We evaluated LaCAM*2-DFBnB's performance on the warehouse map *warehouse-20-40-10-2-1* (denoted Warehouse), with 100, 200, ..., 1,000 agents. Table 1 presents the success rate (percentage of solved instances within the time limit) of this experiment.

[2] https://github.com/Jiaoyang-Li/MAPF-LNS2

#Agents	Suboptimal		Optimal	
	SOC	MKS	SOC	MKS
100	100%	100%	0%	100%
200	100%	100%	0%	100%
300	100%	100%	0%	100%
400	100%	100%	0%	92%
500	100%	100%	0%	96%
600	100%	100%	0%	92%
700	100%	100%	0%	96%
800	100%	100%	100%	100%
900	100%	100%	100%	88%
1000	100%	100%	0%	100%

Table 1: Success rate of LaCAM*2-DFBnB on Warehouse for finding a suboptimal and optimal solution for SOC or MKS.

As was already known, LaCAM*2-DFBnB excels in finding a first suboptimal solution and was able to suboptimally solve all problem instances for both SOC and MKS. Indeed, which was expected, LaCAM*2-DFBnB could not optimally solve any problem instance for SOC. However, many problem instances were optimally solved for MKS; LaCAM*2-DFBnB optimally solved all 25 problem instances with 1,000 agents (some problem instances were not solved for fewer agents due to LaCAM*2's stochastic behavior). Importantly, LaCAM*2-DFBnB is executed in a state space where each node contains a configuration (the vertices of all agents). Given an underlying graph $\mathcal{G}$ with $|\mathcal{V}| = n$ vertices and k agents, there are $\approx n^k$ possible nodes for LaCAM*2-DFBnB. Specifically, the map Warehouse, used for the above experiment, contains 22,599 vertices. Therefore, the size of the state space, for 1,000 agents, is equal to $\approx 22,599^{1,000}$, which is a huge number. These results and findings inspired us to use LaCAM*2-DFBnB as an optimal algorithm for MKS. This raises the question: *How is it possible that such a large problem instance was optimally solved?*

As mentioned, in SOC, the heuristic value at each node is equal to the sum of all the estimates of all agents to reach their goals from their current vertices. In MKS, the heuristic value is equal to the maximum of these estimates. A deeper look at the optimally solved problem instances for MKS revealed that the initial heuristic estimate at LaCAM*2-DFBnB's root node was perfect for *all* of them, which means that the heuristic value was precisely equal to the makespan of the optimal solution. The reason for having a perfect heuristic for MKS is that, often, to avoid conflicts, agents with non-maximal cost paths can extend their paths without increasing the cost of the solution (as it is only affected by the maximal one). In the case of a perfect heuristic, there are no nodes to be classified as must-expand nodes, where $f < C^*$. Therefore, when the optimal solution is found, all nodes in Open have $f \geq C^*$ and can be immediately pruned. In fact, the only modification we added to LaCAM*2-DFBnB (and LaCAM*-DFBnB) is that, when the cost of the best solution found is equal to the heuristic value of the root, the algorithm immediately halts (this was not part of the original algorithm). In *all* the experiments below, we only aim to find optimal MKS solutions.

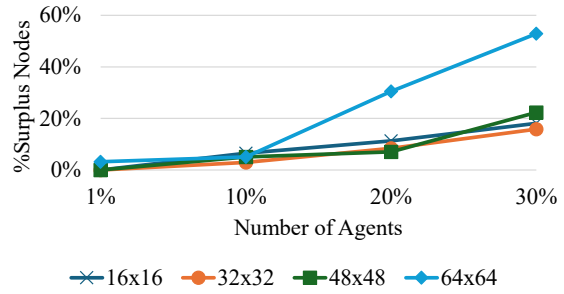

Figure 3: Percentage of surplus nodes on empty grids.

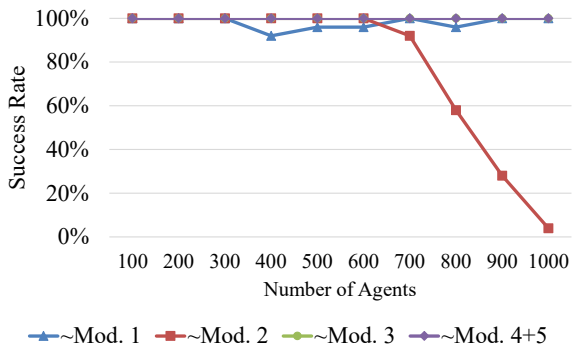

Figure 4: Success rate of LaCAM*2-DFBnB when each modification is removed.

5.2 Counting Surplus Nodes

In this paper, we focus on finding optimal solutions for MKS whose cost is C^*. To evaluate the impact of surplus nodes (with $f > C^*$) expanded by LaCAM*2-DFBnB for minimizing MKS, we experimented on empty grids of sizes 16×16, 32×32, 48×48, and 64×64. The results are presented in Figure 3, where every curve represents a different grid size. Each empty grid was tested with agents that occupy 1%, 10%, 20%, and 30% of the grid (x axis). We measured the average percentage of surplus nodes expanded by LaCAM*2-DFBnB out of the total expanded nodes (all problem instances were solved).

In general, increasing the number of agents also increases the percentage of surplus nodes; when more agents are present, it is more likely for LaCAM*2-DFBnB to reach a solution that is farther than the optimal one, resulting in many surplus expansions. Notably, the percentage of surplus nodes can be very high; in the 64×64 grid with agents occupying 30% of the grid, on average, more than 50% of expansions were of surplus nodes. We also observed specific cases where more than 80% of the expansions were of surplus nodes. These results motivate the use of different search strategies, as proposed in this paper, which do not expand any surplus nodes.

5.3 Considering LaCAM*2's Improvements

To evaluate the impact of the five modifications of LaCAM*2 (described in Section 2.3.2), we executed LaCAM*2-DFBnB on Warehouse with 100, 200, ..., 1,000 agents (as in the experiment of Table 1) and removed each modification in turn (we consider Modifications 4 and 5 together, as done in the original LaCAM*2's paper). Figure 4 shows the result of this experiment, where each curve represents LaCAM*2 without a single modification. For instance, Mod. 1 represents LaCAM*2 without Modification 1 of "Non-deterministic node extraction". Removing any of the modification, did not have a large impact on the performance of LaCAM*2, besides Modification 2. This means that, Modification 2 has the largest impact on the algorithm and significantly improves its performance.

5.4 Evaluating Overall Performance

Next, we evaluate the performance of all eight algorithms (CBSм, LaCAM*-DFBnB, LaCAM*-A*, LaCAM*-IDA*, LaCAM*2-DFBnB, LaCAM*2-A*, LaCAM*2-DFBnBnr, and LaCAM*2-IDA*) on eight various benchmark maps: *empty-32-32* (denoted Empty), *room-64-64-16* (Room), *maze-128-128-10* (Maze), *warehouse-20-40-10-2-1* (Warehouse), *den520d* (Game1), *brc202d* (Game2), *Berlin_1_256* (City1), and *Boston_0_256* (City2). For each map, we created problem instances with 100, 200, ..., 1,000, 1,500, ..., 5,000 agents (with gaps of 100 agents for up to 1,000 agents, and with gaps of 500 agents for more than 1,000 agents). We measured the success rate and the average runtime. The runtime was set to 60 seconds for an unsolved instance within the time limit. Figure 5 shows the results of this experiment. The number that appears next to each map name represents the number of available vertices in the map.

As also observed by Maliah et al. (2025), CBSм performs best in small maps, and outperformed all other solvers in Empty and Room. However, for any larger map, CBSм achieved relatively poor results compared to other solvers. In almost all maps, LaCAM*2-DFBnBnr outperformed LaCAM*2-DFBnB. This means that the refiners of Modifications 4 and 5 do not improve the algorithm and even make it perform worse. The reason is that these modifications try to find better paths for any of the agents, while only a path of a single agent (the highest-cost path) impacts the cost of the solution. The LaCAM*'s algorithms (LaCAM*-DFBnB, LaCAM*-A*, and LaCAM*-IDA*) presented a similar performance in most cases, which implies that the selected search strategy does not significantly affect LaCAM*. In contrast, LaCAM*2-A* and LaCAM*2-IDA* performed better than LaCAM*2-DFBnB (and LaCAM*2-DFBnBnr) and also outperformed all three LaCAM*'s solvers (LaCAM*-DFBnB, LaCAM*-A*, and LaCAM*-IDA*). Notably, problem instances that contain thousands of agents were optimally solved for MKS. For instance, in City1 with 3,000 agents, LaCAM*2-A* reached a 100% success rate (!).

The LaCAM*2's algorithms improve the LaCAM*'s algorithms mainly due to their ability to choose better configurations, such as ones that scatter the agents (Modification 2). On one hand, when the agents are scattered, the agents may be led to sparse areas and a solution will be quickly found. On the other hand, this may extend the time it takes to converge to the optimal MKS solution. For LaCAM*2-DFBnB, when scattering the agents, the f-value of successors often increase, resulting in a longer runtime for finding the optimal solutions. However, LaCAM*2-A* and LaCAM*2-IDA* gradually only consider nodes with minimal f-values. Therefore, when a successor of a higher f-value is created, these two algorithms enforce the nodes to generate more successors until one

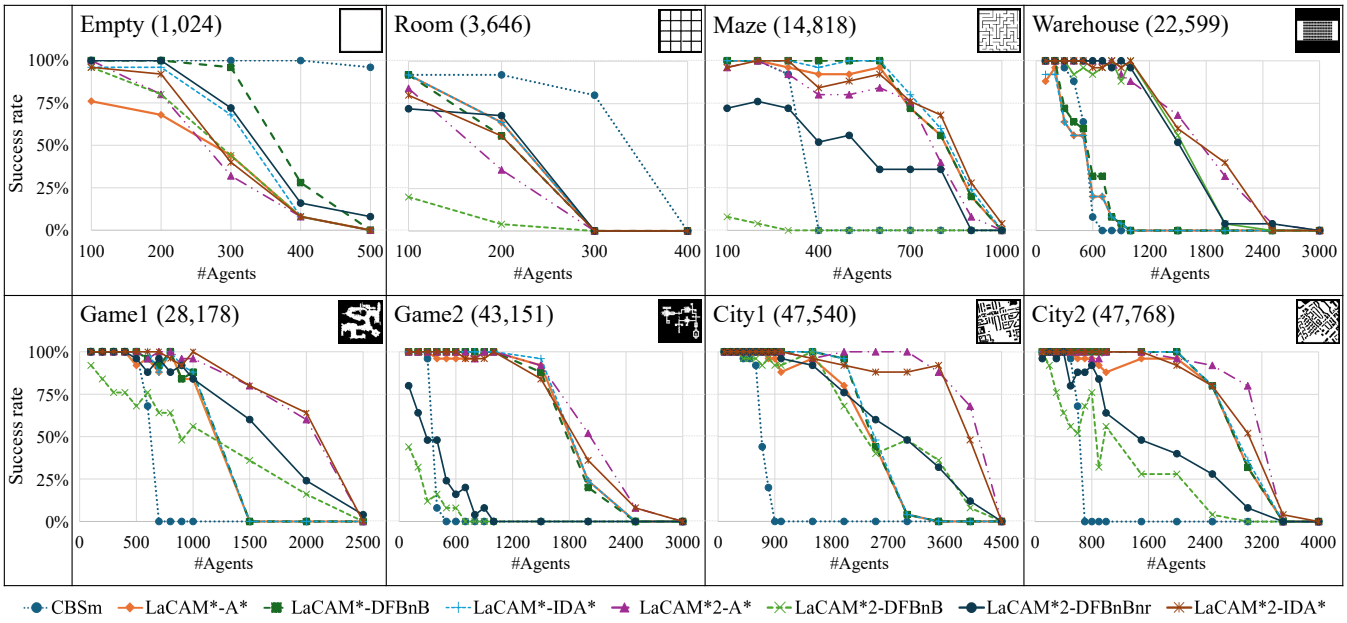

Figure 5: Success rate and runtime (in seconds) on eight benchmark maps.

with the minimal f-value is generated. As a result, they both scatter the agents and restrain this behavior by choosing a node of a low cost.

6 CONCLUSION AND FUTURE WORK

In this paper, we show that LaCAM* can quickly optimally solve MAPF for minimizing MKS. Standard LaCAM* performs a DFBnB. We consider other search strategies for LaCAM*: BFS and ID. Moreover, we compare both LaCAM* and LaCAM*2 with these search strategies, showing that each performs best and achieves state-of-the-art performance in different scenarios.

This work has many possible directions for future work:

- employing other search methods for LaCAM*, such as *beam search* [37];
- designing an algorithm-selection method that chooses the preferred algorithm for a given scenario, e.g., choosing CBSm or LaCAM*2, and, for LaCAM*2, a preferred search method;
- improving the heuristic function of LaCAM*2 for different objective functions: SOC, MKS, or Fuel;
- using LaCAM*2 for optimally solving related problems, such as *lifelong MAPF*, where multiple tasks need to be accomplished by the agents, or any other extension mentioned in Section 3.2.
- adjusting other MAPF algorithms for MKS, e.g., *Increasing Cost Tree Search* (ICTS) [40] and *Branch and Cut and Price* (BCP) [20].

REFERENCES

[1] Anton Andreychuk, Konstantin S. Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. 2022. Multi-agent pathfinding with continuous time. *Artificial Intelligence* 305 (2022), 103662.

[2] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R. Sturtevant, and Sven Koenig. 2020. Probabilistic Robust Multi-Agent Path Finding. In *ICAPS*. 29–37.

[3] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. 2020. Robust multi-agent path finding and executing. *JAIR* 67 (2020), 549–579.

[4] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. 2014. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In *the Symposium on Combinatorial Search (SoCS)*. 19–27.

[5] Roman Barták and Jirí Svancara. 2019. On SAT-Based Approaches for Multi-Agent Path Finding with the Sum-of-Costs Objective. In *SoCS*. 10–17.

[6] Eli Boyarski, Shao-Hung Chan, Dor Atzmon, Ariel Felner, and Sven Koenig. 2022. On Merging Agents in Multi-Agent Pathfinding Algorithms. In *SoCS*. 11–19.

[7] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *IJCAI*. 740–746.

[8] Shao-Hung Chan, Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. 2022. Flex Distribution for Bounded-Suboptimal Multi-Agent Path Finding. In *the AAAI Conference on Artificial Intelligence (AAAI)*. 9313–9322.

[9] Rina Dechter and Judea Pearl. 1985. Generalized Best-First Search Strategies and the Optimality of A^*. *J. ACM* 32, 3 (1985), 505—-536.

[10] Esra Erdem, Doga G. Kisa, Umut Oztok, and Peter Schueller. 2013. A general formal framework for pathfinding problems with multiple agents. In *AAAI*. 290–296.

[11] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. In *ICAPS*.

[12] Gilad Fine, Dor Atzmon, and Noa Agmon. 2023. Anonymous Multi-Agent Path Finding with Individual Deadlines. In *AAMAS*. 869–877.

[13] Graeme Gange, Daniel Harabor, and Peter J. Stuckey. 2019. Lazy CBS: implicit Conflict-based Search using Lazy Clause Generation. In *ICAPS*. 155–162.

[14] Tzvika Geft and Dan Halperin. 2022. Refined Hardness of Distance-Optimal Multi-Agent Path Finding. In *AAMAS*. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 481–488.

[15] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert C Holte, and Jonathan Schaeffer. 2014. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research* 50 (2014), 141–187.

[16] 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* 4(2) (1968), 100–107.

[17] Richard E. Korf. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27, 1 (1985), 97–109. https://doi.org/10.1016/0004-3702(85)90084-0

[18] Richard E. Korf. 1993. Linear-space best-first search. *Artificial Intelligence* 62, 1 (1993), 41–78. https://doi.org/10.1016/0004-3702(93)90054-D

[19] Daniel Koyfman, Dor Atzmon, Shahaf Shperberg, and Ariel Felner. 2025. Minimizing Fuel in Multi-Agent Pathfinding. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 18. 83–91.

[20] 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* 144 (2022), 105809.

[21] 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 *AAAI*. 10256–10265.

[22] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *IJCAI*. 442–449.

[23] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, Graeme Gange, and Sven Koenig. 2021. Pairwise symmetry reasoning for multi-agent path finding search. *AIJ* 301 (2021), 103574.

[24] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. 2019. Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search. In *ICAPS*. 279–283.

[25] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. 2021. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. In *the AAAI Conference on Artificial Intelligence (AAAI)*. 12353–12362.

[26] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, and Sven Koenig. 2019. Multi-Agent Path Finding for Large Agents. In *the AAAI Conference on Artificial Intelligence (AAAI)*. 7627–7634.

[27] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In *AAAI*. 11272–11281.

[28] Hang Ma. 2021. A Competitive Analysis of Online Multi-Agent Path Finding. In *ICAPS*. 234–242.

[29] Hang Ma, Glenn Wagner, Ariel Felner, Jiaoyang Li, TK Kumar, and Sven Koenig. 2018. Multi-agent path finding with deadlines. In *the International Joint Conference on Artificial Intelligence (IJCAI)*. 417–423.

[30] Amir Maliah, Dor Atzmon, and Ariel Felner. 2025. Minimizing Makespan with Conflict-Based Search for Optimal Multi-Agent Path Finding. In *AAMAS*. 1418–1426.

[31] Jonathan Morag, Ariel Felner, Roni Stern, Dor Atzmon, and Eli Boyarski. 2022. Online Multi-Agent Path Finding: New Results. In *SoCS*. 229–233.

[32] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. 2017. Generalized Target Assignment and Path Finding Using Answer Set Programming. In *IJCAI*. 1216–1223.

[33] Keisuke Okumura. 2023. Improving LaCAM for scalable eventually optimal multi-agent pathfinding. In *IJCAI*. 243–251.

[34] Keisuke Okumura. 2023. LaCAM: search-based algorithm for quick multi-agent pathfinding. In *AAAI*. 11655–11662.

[35] Keisuke Okumura. 2024. Engineering LaCAM*: Towards Real-time, Large-scale, and Near-optimal Multi-agent Pathfinding. In *the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1501–1509.

[36] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. 2022. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence* 310 (2022), 103752.

[37] Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach* (3rd ed.). Pearson.

[38] Tomer Shahar, Shashank Shekhar, Dor Atzmon, Abdallah Saffidine, Brendan Juba, and Roni Stern. 2021. Safe Multi-Agent Pathfinding with Time Uncertainty. *JAIR* 70 (2021), 923–954.

[39] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2015. Conflict-based search for optimal multi-agent pathfinding. *AIJ* 219 (2015), 40–66.

[40] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *AIJ* 195 (2013), 470–495.

[41] Bojie Shen, Zhe Che, Jiaoyang Li, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. 2023. Beyond Pairwise Reasoning in Multi-Agent Path Finding. In *ICAPS*. 384–392.

[42] David Silver. 2005. Cooperative Pathfinding. In *AIIDE*. 117–122.

[43] Trevor Standley. 2010. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *the AAAI Conference on Artificial Intelligence (AAAI)*. 173–178.

[44] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Roman Barták, and Eli Boyarski. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *SoCS*. 151–159.

[45] Pavel Surynek. 2010. An Optimization Variant of Multi-Robot Path Planning Is Intractable. In *AAAI*. 1261–1263.

[46] P. Surynek, A. Felner, R. Stern, and E. Boyarski. 2016. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *ECAI*. 810––818.

[47] Jiří Švancara, Marek Vlk, Roni Stern, Dor Atzmon, and Roman Barták. 2019. Online multi-agent pathfinding. In *AAAI*. 7732–7739.

[48] Glenn Wagner and Howie Choset. 2015. Subdimensional expansion for multirobot path planning. *Artificial Intelligence* 219 (2015), 1–24.

[49] Qian Wan, Chonglin Gu, Sankui Sun, Mengxia Chen, Hejiao Huang, and Xiaohua Jia. 2018. Lifelong Multi-Agent Path Finding in A Dynamic Environment. In *ICARCV*. 875–882.

[50] Jingjin Yu and Steven M LaValle. 2013. Multi-agent path planning and network flow. In *Algorithmic foundations of robotics X*. Springer, 157–173.

[51] Jingjin Yu and Steven M. LaValle. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *AAAI*. 1444–149.

[52] Han Zhang, Jiaoyang Li, Pavel Surynek, Sven Koenig, and T. K. Satish Kumar. 2020. Multi-Agent Path Finding with Mutex Propagation. In *ICAPS*. 323–332.