Minimizing Makespan with Conflict-Based Search for Optimal Multi-Agent Path Finding

Amir Maliah
Ben-Gurion University, Israel
amirmal@post.bgu.ac.il

Dor Atzmon
Bar-Ilan University, Israel
dor.atzmon@biu.ac.il

Ariel Felner
Ben-Gurion University, Israel
felner@bgu.ac.il

ABSTRACT

Conflict-based search (CBS) is a prominent algorithm that optimally solves the *Multi-Agent Path Finding* problem (MAPF). There are two common objective functions for MAPF: *Makespan* (MKS), the time elapsed until the task ends, and *Sum-Of-Costs* (SOC), the sum of costs of all paths. Most existing MAPF algorithms, including CBS, were not designed particularly for minimizing MKS. In this paper, we show that CBS can be redefined as a framework that can be fine-tuned to different objectives. We introduce an instantiation of this framework for minimizing MKS. Its low-level solves a new search setting which we call *Extended Bounded-Cost Search*. Our experiments show that our new algorithm can significantly outperform previous algorithms for MKS. Moreover, we discuss two extensions of MKS which are also members of our general framework.

KEYWORDS

Makespan, Conflict-Based Search, Multi-Agent Path Finding

ACM Reference Format:
Amir Maliah, Dor Atzmon, and Ariel Felner. 2025. Minimizing Makespan with Conflict-Based Search for Optimal Multi-Agent Path Finding. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

The task in the *Multi-Agent Path Finding* problem (MAPF) is to find conflict-free paths for multiple agents [29]. 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: *Makespan* (MKS) and *Sum-of-Costs* (SOC). MKS is the highest-cost (longest) path among all agents' paths and can be seen as the time it takes the agents to complete the task. SOC is the sum of costs of all paths and can be seen as the energy waste by all agents. It is NP-hard to optimally solve MAPF for both MKS [30] and SOC [39]. Still, since MAPF has wide applicability, efficient optimal algorithms were proposed for MAPF [5, 8, 24].

Conflict-Based Search (CBS) [24] is a prominent two-level algorithm. On its high level, each node contains paths for all agents under a set of constraints. Each such path is individually planned by CBS's low-level search, which usually finds the *lowest-cost* path (denoted LC hereafter) that satisfy the imposed constraints.

CBS was originally designed for SOC. Nevertheless, minimizing MKS is important for different applications. For example, consider packing several items for delivery by robots operating in an automated warehouse. The package can only be shipped when containing all items. Notably CBS can optimally solve MAPF for minimizing either SOC or MKS (or other objectives as well). This is done by modifying the prioritization of nodes in the high level according to the required objective. Thus, the common way to run CBS for MKS is to modify the priority function in the high level to prefer nodes with low MKS. Nevertheless, the low level remains untouched and searches for the lowest-cost (LC) path for the given agent [7, 24, 32]. We label the CBS variants with this low level of LC by CBSs(LC) for a high level designed to minimize SOC (this is the common version), and by CBSm(LC) for a high-level designed for MKS.

In this paper, we redefine CBS as a general framework for different objectives, which better treats the selected objective by separately adjusting the priority functions of the low and high levels for any given objective. CBSs(LC) and CBSm(LC) are two members of this framework. We propose other members that are better suited for MKS.

Additionally, we introduce a new search setting that extends the *Bounded-Cost Search* setting (BCS) [28]. In BCS, we are given a bound B and the task is to find a path with cost $\leq B$ or to declare that such a path does not exist. Our extended setting is called *Extended Bounded-Cost Search* (EBC). If a path with cost $\leq B$ does not exist then, in EBC, we seek a path with minimal cost (which is $> B$). We present a general algorithm for solving EBC and provide several instantiations for this algorithm. We then show that using LC as the low level, as done by CBSm(LC), is not efficient. For MKS, the task in the low level is, in fact, solving EBC. Thus, we introduce a novel CBS for MKS, called CBSm(EBC), that solves EBC in its low level (instead of LC) and discuss several instantiations of it.

Our experiments show that CBSm(EBC) significantly outperforms CBSm(LC), as well as a SAT-based MAPF solver designed for MKS. We also compared CBSm(EBC) to LaCAM*, a recent MAPF solver that converges to finding optimal solutions and show the relative advantages of each of these algorithms. In many cases, especially in small maps, CBSm(EBC) significantly outperforms LaCAM* and was able to solve problem instances with the maximum number of agents possible in some maps.

Finally, we discuss two extensions of the MKS objective. The first extension aims to minimize MKS but ties are broken in favor of low SOC as a secondary objective. The second extension recursively minimizes MKS for subsets of agents. We introduce variants of CBS for these settings and study their performance.

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowe (eds.), May 19 – 23, 2025, Detroit, Michigan, USA.

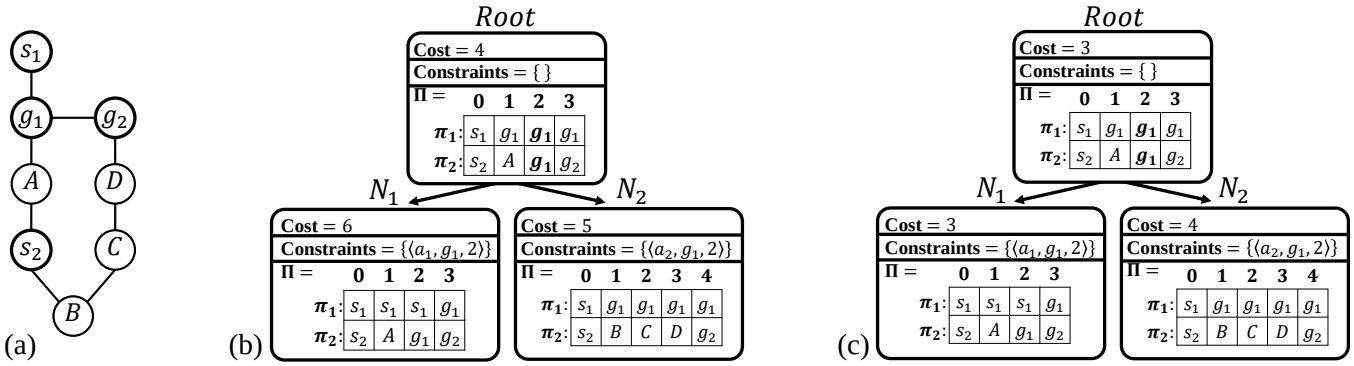

Figure 1: (a) MAPF problem instance. (b) CT for SOC. (c) CT for MKS.

2 BACKGROUND AND PROBLEM DEFINITION

The *Multi-Agent Path Finding* problem (MAPF) [29] 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. In graph $\mathcal{G}$, two vertices v_1 and v_2 are neighbors if there is an edge between them: $(v_1, v_2) \in \mathcal{E}$. Time is discretized into timesteps, and a *path* π_i for agent a_i is a list of neighboring vertices from start vertex s_i to goal vertex g_i. Let $\pi_i(t)$ denote the vertex of agent a_i at timestep t according to path π_i. Therefore, $\pi_i(0) = s_i$, $\pi_i(|\pi_i| - 1) = g_i$, and $\forall t \in (0, \ldots, |\pi_i| - 2) : (\pi_i(t), \pi_i(t+1)) \in \mathcal{E}$. A path represents 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 $\pi_i(t) = \pi_i(t+1)$). The cost $c(\pi_i)$ of path π_i is the number of edge traversals it contains ($= |\pi_i| - 1$). We thus assume a graph where all edges have a uniform cost of 1.

A *plan* $\Pi = (\pi_1, \ldots, \pi_n)$ is a list of paths for the agents. A *solution* to MAPF is a *conflict-free* plan Π; that is, any two paths in 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 simultaneously traverse edge e in opposite directions between timesteps t and $t+1$ ($\exists t : \pi_i(t) = \pi_j(t+1) \wedge \pi_j(t) = \pi_i(t+1) \wedge (\pi_i(t), \pi_i(t+1)) = e$). 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 $C_{MKS}(\Pi) = \max_{\pi_i \in \Pi} c(\pi_i)$.

Figure 1(a) depicts 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, respectively. Here, the optimal SOC solution $\Pi = (\pi_1, \pi_2)$ is $\pi_1 = (s_1, g_1)$ and $\pi_2 = (s_2, B, C, D, g_2)$ with $C_{SOC}(\Pi) = 5$ and $C_{MKS}(\Pi) = 4$. Note that agent a_2 had to take the detour via B in order to not conflict with agent a_1 at vertex g_1. The optimal MKS solution $\Pi = (\pi_1, \pi_2)$ is $\pi_1 = (s_1, s_1, s_1, g_1)$ and $\pi_2 = (s_2, A, g_1, g_2)$ with $C_{SOC}(\Pi) = 6$ and $C_{MKS}(\Pi) = 3$. Here, agent a_1 waits at its start vertex s_1 to allow agent a_2 pass through vertex g_1 and achieve MKS of 3.

2.1 Conflict-Based Search (CBS)

Conflict-Based Search (CBS) [24] 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 $t+1$. 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 (presented in Algorithm 1) performs a best-first search 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 (lines 2-6). Then, repeatedly, CBS extracts the lowest-cost CT node N from Open (lines 7-8). If $N.\Pi$ is conflict-free, it is returned as a solution (lines 9-10). Otherwise, a conflict $\langle a_i, a_j, x, t \rangle$ is chosen (line 11). 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 (lines 12-13 and 17-22). The new CT nodes N_i and N_j are inserted into Open (line 14). The only difference between minimizing SOC and MKS is the way the cost of each CT node is determined (line 21): if the cost $N.cost$ of each CT node N is $C_{SOC}(N.\Pi)$ then the optimal SOC solution is returned, and if the cost is $C_{MKS}(N.\Pi)$ then the optimal MKS solution is returned.

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 (line 20) to replan the lowest-cost path for agent a_i that satisfies the new set of constraints in N_i. We denote a lowest-cost low-level search by LC. LC can be implemented by (Temporal-)A* [27], which executes A* but must satisfy the constraints. It prioritizes nodes n by $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of reaching n and $h(n)$ is a heuristic estimation for reaching the goal from n.

We denote CBS for minimizing SOC and MKS, with low level of LC, by CBSs(LC) and CBSm(LC), respectively.

Example: Figure 1(b) presents the CT created by CBSs(LC) for the problem instance in Figure 1(a). At the Root, the cost (SOC)

Algorithm 1: High Level of CBS

```
1  HighLevel(G, A, S, G)
2      Init Open, Root ; Root.constraints = {}
3      foreach a_i ∈ A do
4          Plan path Root.Π.π_i // Low Level
5      Root.cost = C(N.Π)
6      Insert Root into Open
7      while Open is not empty do
8          Extract N from Open // High-Level's Priority
9          if N.Π is conflict-free then
10             return N.Π
11         ⟨a_i, a_j, x, t⟩ = GetConflict(N)
12         N_i = GenerateChild(N, ⟨a_i, x, t⟩)
13         N_j = GenerateChild(N, ⟨a_j, x, t⟩)
14         Insert N_i and N_j into Open
15     return No Solution
16 GenerateChild(N, ⟨a_i, x, t⟩)
17     Init N'
18     N'.constraints = N.constraints ∪ {⟨a_i, x, t⟩}
19     N'.Π = N.Π
20     Replan path Π.π_i under N'.constraints // Low Level
21     N'.cost = C(N'.Π)
22     return N'
```

is 4. We identify the vertex conflict $\langle a_1, a_2, g_1, 2\rangle$ and, thus, the Root is not a goal. To resolve the conflict, two child CT nodes N_1 and N_2 are created with the additional constraints $\langle a_1, g_1, 2\rangle$ and $\langle a_2, g_1, 2\rangle$, respectively. The low-level search, then, replans for each conflicting agent in each of the two new CT nodes. At N_1, the path of agent a_1 becomes two timesteps longer (a_1 waits two timesteps at its start vertex s_1) and the cost increases to 6. At N_2, the path of agent a_2 is one timestep longer (from vertices B, C, and D) and the cost increases to 5. As both CT nodes contain conflict-free plans (solutions), these CT nodes are goals and the optimal SOC solution is found at N_2 with a cost of 5. Similarly, Figure 1(c) presents the CT created by CBSm(LC). At the Root, the cost (MKS) is 3. Here, at N_1, while the path of agent a_1 becomes two timesteps longer, the cost remains 3. As at N_2 the cost increases to 4, the optimal MKS solution is found at N_1 with a cost of 3.

2.2 CBS's Improvements

Over the years, many improvements were proposed to enhance CBS. We summarize the main ones next.

(1) *Prioritizing Conflicts* (PC) [2] categorizes conflicts into three types: *cardinal conflicts*, where resolving the conflict results in an increase in the cost in the two child CT nodes; *semi-cardinal conflict*, where the cost increases in only one child CT node; and *non-cardinal conflicts*, where the cost remains the same. Solving cardinal conflicts first, then semi-cardinal conflicts, and finally non-cardinal conflicts often reduces the size of the CT.

(2) *CBS with heuristics* (CBSH) [4, 11] adds an admissible heuristic value to the costs of CT nodes, estimating the remaining cost that will be added below a CT node. Adding heuristics increases the costs of CT nodes and reduces the search effort.

(3) *Disjoint Splitting* (DS) [13] resolves conflicts by imposing negative and positive constraints on one conflicting agent, instead of two negative constraints on both conflicting agents. A positive constraint forces the agent to obey the constraint and prohibits all other agents from violating it and, thus, many conflicts may be resolved altogether, which also often reduces the size of the CT.

(4) Instead of setting a single constraint on each agent to resolve conflicts, reasoning techniques impose a larger set of constraints. *Pairwise Reasoning* (PR) [12] detects pairs of agents in a grid that have multiple paths conflict in a rectangle (GR), in a corridor (GC), or at a goal vertex (T). *Mutex Propagation* (MP) [40] generalizes PR and automatically locates pairs of agents in a graph that have multiple conflicting paths and generates constraints. *Cluster Reasoning* (CR) [26] extends the reasoning from pairs of agents to groups (clusters) of agents.

3 BOUNDED-COST SEARCH AND EXTENSIONS

Typical search algorithms are designed to minimize the cost of the solution path. An important search setting is *Bounded-Cost Search* (BCS) [28]. In BCS, a bound B is given and the task is to find a solution (path) with cost $\leq B$ as quickly as possible. The returned path is not necessarily the optimal solution whose cost is denoted by C^*. According to the common definition [28], algorithms that solve BCS must either return such a path or return failure which means that such a path does not exist (when $C^* > B$). Below, we introduce the *Extended Bounded-Cost Search* setting and a corresponding algorithm, which are used for the low level of CBS for MKS.

3.1 Bounded-Cost Search Algorithms

A general framework for solving BCS runs a (best-first) search from the start vertex. But, whenever a node n is generated with $f(n) > B$, it is pruned. Such a general framework was mentioned by Gilon et al. (2017) and was referred to as *reasonable* because it is reasonable to perform a best-first search and to prune nodes with cost above the bound. The algorithm halts in the following two possible scenarios. **(1) Success:** a goal node with cost $\leq B$ is reached and returned. **(2) Failure:** no such path exists.

Variants of this framework differ in how nodes are prioritized in the best-first process. While any priority function will work, prominent variants are the following:

(i) A* within the BCS framework returns the lowest cost (LC) path if it is smaller than B, otherwise it returns failure. While A* is a member of this framework, it might not be efficient, as it strives to return the optimal path even if another (not optimal) solution within the bound can be found faster.

(ii) Greedy Best-first Search (GBFS) (sometimes called *pure heuristic search*) is a simple, yet efficient, heuristic search algorithm which prioritizes nodes n based only on $h(n)$. GBFS aims to reach the goal fast and is usually faster than A* as it returns a path without proving its optimality.

(iii) Potential Search (PS) [28] prioritizes nodes n according to their *potential* (pt) for quickly leading to a bounded-cost solution. pt is defined by $pt(n) = h(n)/(B - g(n))$, where a lower value indicates a better potential. For various bounds B, it was shown that PS is able to find a bounded-cost path faster than A*.

3.2 Extended BCS - When BCS Fails

We now introduce a new search setting that further extends BCS to, arguably, a more realistic scenario. As defined above, if there is no solution within the bound then BCS algorithms return *failure*. A realistic question that arises is: "OK, now what?" This depends on the user. Of course, if the user gives up because there is no solution within his/her budget then that is fine. However, in many realistic scenarios, the user might be willing to increase the budget.

In our new setting, which we call *Extended BCS* (EBC), when a BCS algorithm returns failure (no solution with cost $\leq B$), then we want a solution with the minimal cost (which is above B). Indeed, anyone can think of real-world cases where EBC is relevant and the user is willing to exceed his/her initial budget. In fact, we will see below that CBS's low level for MKS needs to solve exactly the EBC problem.

The EBC* algorithm. We now present an algorithm (EBC*) that solves EBC and uses the BCS framework as an internal component. EBC* receives a bound B and solves EBC in two steps. **Step 1:** EBC* aims to find a path with cost $\leq B$ as quickly as possible. If such a path is found, EBC* halts. **Step 2:** After proving that such a path does not exist, it falls back to A* and seeks the lowest-cost path (with cost $> B$). EBC* is a member of the general *Focal Search* family, as presented by Gilon et al. (2017). EBC* maintains two lists: Open and Focal. Every generated node is added to Open. Focal is a subset of Open, where Focal $= \{n \in \text{Open} | f(n) \leq B\}$. In step 1, EBC* searches only in Focal. In that step it actually activates the BCS framework with any possible internal priority function such as the ones listed above. When Focal becomes empty, it is a proof that no solution with cost $\leq B$ exists and EBC* moves to step 2. In step 2, EBC* changes its policy and continues to search in Open but in a best-first search order according to $f = g + h$ in order to find the shortest path as fast as possible. So, in step 1, any possible priority function for searching in Focal is acceptable. But, in step 2, only LC polices (e.g., $f = g + h$) are valid.

There are many possible implementations of EBC* including many possible data structures to maintain Open and Focal. The straightforward implementation is that Open is a priority queue sorted according to $f = g + h$ but Focal is another priority queue sorted according to any of the BCS framework priority functions. In our implementation, we also broke ties in Open according to the priority function of Focal. Below, we propose to use EBC* for CBS's low-level for MKS (denoted CBSm(EBC)).

Other BCS Extensions. Besides EBC, other realistic BCS extensions exist. For example, when there is no solution within B, the user may increase the budget to B'. The same BCS algorithm can be called but now with B'. This can be further extended and the user can give a sequence of bounds $\{B_1, B_2, \ldots, B_m\}$. Then, in iteration i, the aim is to find a solution which is $\leq B_i$. Other extensions are also possible.

k	SOC		MKS	
	CBSs(LC)	CBSs(LC)+	CBSm(LC)	CBSm(LC)+
5	100%	100%	100%	100%
10	100%	100%	100%	100%
20	100%	100%	100%	100%
50	0%	88%	100%	100%
100	0%	0%	100%	100%
150	0%	0%	100%	88%
200	0%	0%	36%	12%
250	0%	0%	0%	0%
	Cost			
5	118		38	
10	225		40	
20	449		43	
50	1,136		47	

Table 1: Success rate and average cost on 32×32 grids.

4 GENERALIZING CBS FOR OTHER OBJECTIVES

In this section, we generalize CBS to better suit optimal MAPF solving for different objective functions. We begin by discussing existing CBS's improvements.

4.1 Comparing Improvements for SOC and MKS

Standard CBS [24] was originally designed for SOC. Likewise, its improvements (mentioned in Section 2) were originally developed for (and tested on) MAPF for SOC. Some of them do not even directly apply to CBS for other objectives, such as MKS. For instance, CBSH uses admissible heuristic values representing an increase in SOC. Designing and creating heuristics for other objectives require further research.

We evaluated the impact of these improvements on CBS for both SOC and MKS. We compared the baseline CBS variants (CBSs(LC) and CBSm(LC)) with variants that also added these enhancements (PC+GR+GC+T+DS, defined above) which are denoted CBSs(LC)+ and CBSm(LC)+. Again, they all use a low level that searches for the lowest-cost path (LC). Table 1 presents the success rate of these algorithms for 25 random problem instances with a time limit of 60 seconds on 32×32 4-connected grids with 20% obstacles. As can be seen, for SOC, these improvements significantly improved the algorithm, and CBSs(LC)+ solved 88% of the problem instances with 50 agents while CBSs(LC) was not able to solve any of them. However, these improvements did not help CBS for minimizing MKS and, in fact, CBSm(LC)+ performed worse than CBSm(LC). For example for 200 agents CBSm(LC)+ only solved a third of the instances solved by CBSm(LC).[1]

The explanation for this phenomenon is as follows. The tested improvements attempt to provide higher costs for CT nodes, which, in turn, prunes parts of the CT and results in lower runtime. The cost of a CT node can be increased only when a cost of a path

[1]We observed similar trends also on other maps as well as when we examined subsets of improvements or when PC categorized conflicts by their increase in MKS instead of SOC.

becomes higher. When minimizing SOC, any path whose cost becomes higher increases the cost of the CT node. However, when minimizing MKS, the cost is determined only by the highest-cost path. Thus, these improvements do not influence the cost of the CT node when the cost of another (not the highest-cost) path becomes higher. By contrast, these improvements always consume runtime to compute them. Therefore, they were not effective for MKS and even worsened the performance in many cases, e.g., for 150 and 200 agents.

In general, Table 1 shows that solving MAPF for MKS is easier than for SOC, and CBS can solve instances with many more agents for MKS than for SOC. This is inline with a similar observation made by Surynek et al. [34] when investigating SAT solvers for MAPF. The main reason is that, there are often many more optimal solutions when MKS is minimized than when SOC is minimized. This is due to the fact that MKS is only influenced by the highest-cost path while SOC is influenced by all paths. The bottom part of Table 1 compares the cost of the optimal paths for SOC and MKS. When more agents exist, the optimal SOC increases more significantly than the optimal MKS. When more agents are added to a problem instance, the SOC immediately increases. By contrast, the MKS only increases if the cost of the highest-cost path increases.

4.2 The Optimality of CBS

A main property that makes CBS optimal for SOC, MKS, and other objective functions is the following.

Property 1. *The cost $N.cost$ of any CT node N equals the lowest cost plan that satisfies $N.constraints$.*

This property ensures that when CBS (that runs best-first search) finds a solution, it is guaranteed to be the optimal solution as all other solutions must have higher costs.

Let N be a CT node and let N' be its child CT node. The child N' first inherits the plan $N.\Pi$ plus the constraints $N.constraints$ (lines 18-19 in Algorithm 1). To resolve a conflict in CT node N, a constraint is added on a single agent a_i in the child CT node N'. Therefore, to satisfy $N'.constraints$, we only need to replan for a_i (the paths of all other agents already satisfy the constraints).

For minimizing SOC, CBSs(LC) obtains Property 1 by a low level that replans the lowest-cost path that satisfies the constraints (LC). If the low level were to replan a path that does not have the lowest cost, Property 1 would have been violated, and the algorithm might have found a suboptimal solution. Therefore, for minimizing SOC, the low level *must* return the lowest-cost path. For many other objectives, Property 1 is also obtained with a similar low-level search (which replans the lowest-cost path). For MKS, for instance, the plan, where each agent has the lowest-cost path, certainly has the minimal MKS. However, Property 1 can also be obtained without a low-level search returning the lowest-cost path. Consider a plan $\Pi = (\pi_1, \pi_2)$ of two agents, where $c(\pi_1) = 10$ and $c(\pi_2) = 5$, and these paths are of lowest-cost. Here, $C_{MKS}(\Pi) = 10$. However, for MKS, the cost remains 10 if agent a_2 had any path with cost ≤ 10.

The original paper on CBS [24] designed it for SOC and suggested a low level that finds the lowest cost path. That paper mentions that, for optimizing MKS, it is sufficient to modify the high level to prioritize CT nodes by their MKS. While this is true, it may not be efficient.

Based all on this, for optimally and efficiently solving MAPF for other objectives, we propose to adjust not only the high but *also* the low level. For a given objective, the low level should be able to return any path as long as Property 1 is obtained. By doing this, the low level may expand fewer nodes or find a path that conflicts less with other agents and, thus, results in fewer high-level expansions. Next, we demonstrate a more suitable low level for MKS.

4.3 Adjusting CBS for MKS

When MKS is minimized, the cost of each CT node N is determined solely according to the cost of the highest-cost path, rather than the sum of costs of all paths (as in SOC). Therefore, the traditional low level of CBSm(LC), which searches for the lowest-cost path, is more restrictive. Instead, we suggest to run a low-level that aims to satisfy the new constraint for the conflicting agent a_i but with the objective of not increasing $N.cost$ which is the cost of the longest path. Thus, the low level of our new CBS variant, denoted CBSm(EBC), solves a BCS problem for agent a_i where the bound B is the cost of the current plan (its MKS). Of course, since that cost is not yet known (as the path of a_i is now being calculated), we inherit the cost from N's parent for the bound B. If there is no such path for a_i with cost $\leq B$ then B (=MKS) must increase. But, to minimize MKS, the new path for a_i (which now becomes the agent with the highest-cost path) should be minimized (albeit larger than the previous MKS). This is exactly the EBC problem as defined above. We thus run EBC* for the low level of CBS for MKS and denote it by CBSm(EBC).

The Root CT node R should be treated slightly differently. At R, when calling the low level to plan a path for each agent, $R.cost$ is yet to be determined and no bound exists. Therefore, for R, we execute LC.[2]

Theorem 1. *CBSm(EBC) obtains Property 1.*

Proof. To prove that CBSm(EBC) obtains Property 1, we need to show that for every CT node N', the cost $N'.cost$ when using CBSm(EBC) is identical to the one when using CBSm(LC). Let π_i be the replanned path returned by either LC or EBC in CT node N'. If there exists a path with cost $\leq$ the MKS of the parent CT node N, then both LC and EBC* return such a path. If there is no such path, both LC and EBC* return the lowest-cost path. In both cases, the cost of N' is the same for CBSm(EBC) and CBSm(LC). □

4.4 Prioritizing Focal

For the low level of CBSm(EBC), implementations of EBC* differ in how they prioritize nodes in Focal in step 1 of the algorithm (in step 2, they all execute variants of A*). We consider the following prioritizations for step 1.

(1+2) GBFS and PS. Using these two BCS algorithms to prioritize nodes in step 1 of the low level can quickly find a bounded-cost path, faster than using LC (e.g., A*). They are expected to reduce the number of expansions of each low-level search. We denote CBSm(EBC) with a low level of EBC that prioritizes Focal according to GBFS and PS by CBSm(EBC,GBFS) and CBSm(EBC,PS), respectively.

(3) Minimal Number of Conflicts (MC). While the above prioritizations might expand fewer low-level nodes, they may still find a path that has numerous conflicts with other agents. Thus, while

[2] Other low-level searches at the Root were not effective in our trials.

k	High-Level Expansions												Low-Level Expansions per High-Level Call											
	Empty				Random				City				Empty				Random				City			
	CBSм	CBSм(EBC)			CBSм	CBSм(EBC)			CBSм	CBSм(EBC)			CBSм	CBSм(EBC)			CBSм	CBSм(EBC)			CBSм	CBSм(EBC)		
	(LC)	GBFS	PS	MC	(LC)	GBFS	PS	MC	(LC)	GBFS	PS	MC	(LC)	GBFS	PS	MC	(LC)	GBFS	PS	MC	(LC)	GBFS	PS	MC
5	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**21**	**21**	**21**	22	27	**26**	37	**26**	198	**179**	188	186
10	**0**	1	1	**0**	2	9	6	**1**	**0**	**0**	**0**	**0**	**23**	**23**	**23**	**23**	30	**26**	37	29	207	**187**	193	191
20	3	4	4	**2**	5	20	14	**3**	2	1	1	**0**	29	**24**	31	27	34	**28**	38	36	268	**187**	202	200
50	38	30	25	**8**	38	126	135	**15**	7	22	17	**3**	40	**26**	40	34	49	**35**	50	64	348	**200**	235	241
100	153	134	12K	**27**	269	22K	34K	**45**	27	61	47	**9**	68	**28**	47	46	89	**45**	163	152	452	**216**	278	313
150	437	441	24K	**53**	3K	52K	80K	**86**	87	140	96	**18**	82	**31**	64	61	141	**73**	140	372	543	**208**	286	349

Table 2: Average number of high-level expansions (left) and low-level expansions per high-level call (right), on Empty, Random, and City.

Algorithm	High level	Low level	
	Priority	Type	Priority
CBSs(LC)	SOC	LC	f
CBSм(LC)	MKS	LC	f
CBSм(EBC,GBFS)	MKS	EBC	h
CBSм(EBC,PS)	MKS	EBC	$h/(B-g)$
CBSм(EBC,MC)	MKS	EBC	#conflicts
CBSмs(LC)	MKS,SOC	LC	f
CBSRм(LC)	RMKS	LC	f

Table 3: CBS's high and low levels.

the number of low-level nodes may decrease, the number of high-level CT nodes (the size of the CT) may still be large. Therefore, we also propose to prioritize nodes in Focal by the number of conflicts with other agents, which we call *Minimal Number of Conflicts* (MC). That is, in step 1, MC prioritises nodes with the lowest number of conflicts and aims to decrease the number of high-level nodes. For implementing MC, the paths of all agents are also passed to the low level and each low-level node maintains the accumulated number of conflicts along the search tree. We denote this variant by CBSм(EBC,MC).

We note that the idea of favoring nodes with minimal MC was used before, but in other contexts. These include: the *SIPPS* algorithm [10] in lifelong MAPF, where new tasks arrive over time; the *bCOA** algorithm [16] in bounded-suboptimal MAPF, where the demand for optimality is relaxed; and the *CBS-M* algorithm [14] in the problem of moving agents in formation. All these algorithms also prioritize nodes according to the minimum number of conflicts, but they do it to solve different problems from the plain MAPF that we focus on in this paper.

Table 3 summarizes our new understandings on CBS. CBS is, in fact, a general framework which any instantiation of it needs to specify both the prioritization function in the high-level as well as the problem the low level solves (the *type* column) and finally the exact algorithm used for the low-level (the *priority* column). All the algorithms mentioned in this paper are listed in the table. The variants in the two last rows will be described later in the paper. In our experiments below, all CBS-based algorithms had the additional high-level tie-breaking strategy of minimal conflicts.

5 EMPIRICAL EVALUATION

In this section, we empirically compare the various MKS solvers. We experimented on eight grid maps (also used by Li et al. [12]), from the *MovingAI* repository [29]: *empty-32-32* (denoted Empty), *random-32-32-20* (Random), *room-64-64-8* (Room), *maze-128-128-1* (Maze), *warehouse-10-20-10-2-1* (Warehouse), *den520d* (Game1), *brc202d* (Game2), and *Paris_1_256* (City). For each map and for each k agents, we experimented on the 25 problem instances that exist in the repository, where a problem instance was created using the first k lines in each scenario file. Our machine was Intel® Core I7-1065G7 with 16GB of RAM. We will publish our implementation and results file upon acceptance.

We start by comparing our four CBS-based algorithms CBSм(LC), CBSм(EBC,GBFS), CBSм(EBC,PS), and CBSм(EBC,MC). Table 2 shows the average number of high-level expansions (left) and low-level expansions per high-level call (right) on three representative maps Empty, Random, and City, and up to 150 agents. As expected, in most cases, CBSм(EBC,GBFS) and CBSм(EBC,PS) expanded the lowest number of low-level nodes (their low-level search was the fastest). This is because both aim to quickly find a solution without minimizing either the path's cost or the number of conflicts. However, as a result, the paths are longer and, thus, conflict more with other agents, and both algorithms expanded more high-level nodes. CBSм(EBC,MC) solved problem instances with many agents while only expanding several dozen high-level nodes. This is because MC finds a path that has the lowest number of conflicts which prevents future CT nodes. Overall, while CBSм(EBC,MC), to a small extent, expanded more nodes on each of its low level executions, it expanded significantly less high-level nodes which, as we show next, resulted in a superb performance.

Next, we include in our experimental evaluation existing MAPF algorithms. Many existing MAPF algorithms specifically designed for minimizing MKS are compilation-based. They optimally solve MAPF by compiling it into a known NP-hard problem that has mature and effective off-the-shelf solvers. This includes compiling MAPF into SAT [31, 33], Integer-Linear Program (ILP) [38]; Answer Set Programming (ASP) [3, 20]; and CSP [23]. For our experiments, we used a reduction to SAT implemented in Picat [41], which is commonly used in MAPF [1].[3] We denote it by *SAT*. In addition,

[3]We note that other compilation-based approaches, e.g., by Surynek et al. [35], consider a slightly different MAPF variant where agents cannot move to a currently occupied vertex, even if the former agent moves away in another direction.

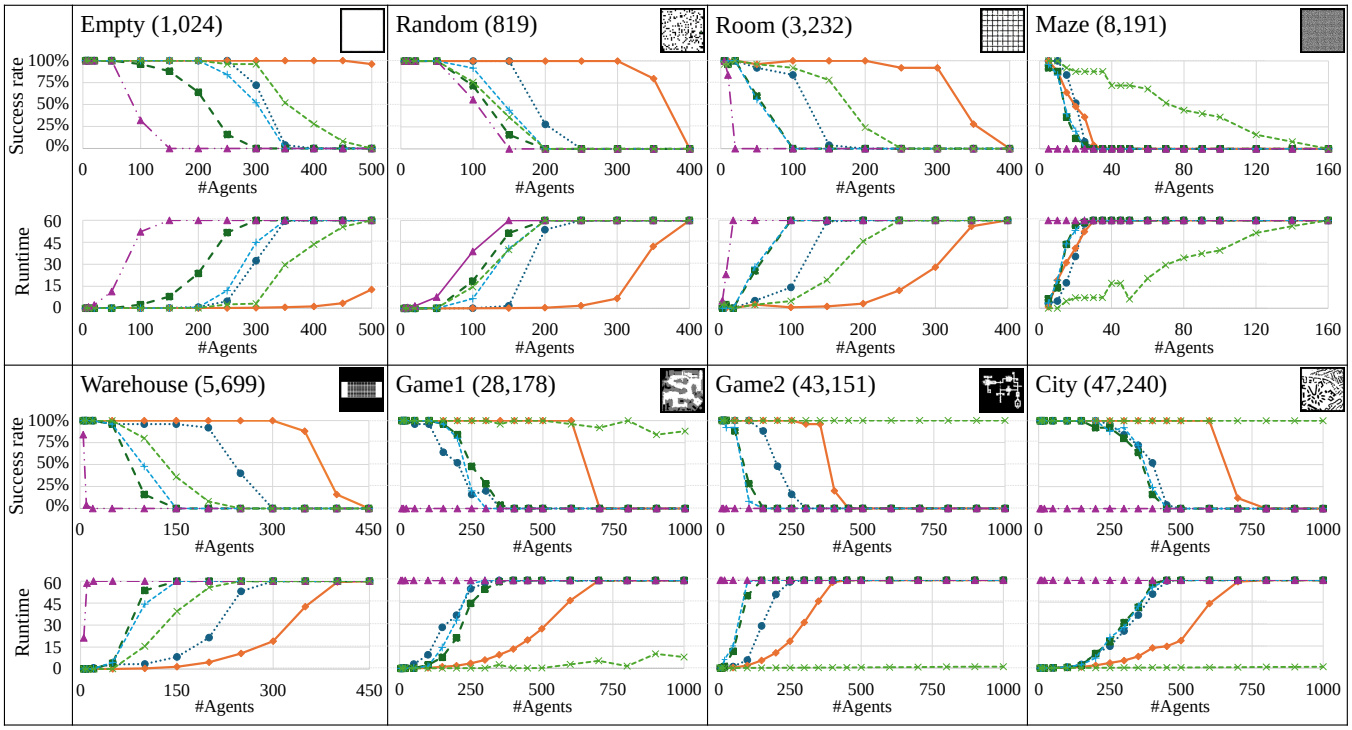

Figure 2: Results for CBSм(LC), CBSм(EBC,GBFS), CBSм(EBC,PS), CBSм(EBC,MC), SAT, LaCAM*.

k	Em	Rn	Rm	Mz	Wh	G1	G2	Ct
5	34	38	96	978	155	283	769	291
10	41	40	104	980	160	309	857	330
20	45	43	114	-	172	342	923	368
50	49	47	125	-	177	361	967	437
100	51	49	133	-	187	375	1,014	474
150	52	52	135	-	189	385	1,024	488
200	53	53	137	-	193	389	1,031	494
250	53	54	-	-	198	394	1,034	494
300	53	54	-	-	199	395	1,036	497
350	54	54	-	-	199	398	1,050	501
400	54	-	-	-	-	399	1,054	503
450	54	-	-	-	-	402	1,055	507
500	55	-	-	-	-	403	1,057	508
600	-	-	-	-	-	405	1,060	514
700	-	-	-	-	-	-	1,061	517
800	-	-	-	-	-	407	1,069	518
900	-	-	-	-	-	-	1,074	518
1,000	-	-	-	-	-	-	1,080	521

Table 4: Average cost (MKS) on eight benchmark maps.

*LaCAM** [21] is a recent anytime solver, which extends the suboptimal algorithm *LaCAM* [22] and converges to the optimal MKS solution. Both LaCAM and LaCAM* search in a configuration space where each node represents vertices for the entire set of agents and, therefore, the start node contains S and the goal node contains G.

We experimented with the following six algorithms CBSм(LC), CBSм(EBC,GBFS), CBSм(EBC,PS), CBSм(EBC,MC), SAT, and LaCAM* on all eight grid maps. Figure 2 shows the success rate, for timeout of 60s, and average planning runtime (in secs). The runtime for unsolved instances was set to 60s. Clearly, CBSм(EBC,MC) significantly outperforms SAT as well as any other CBS variants and, in particular, it outperforms the common CBSм(LC). For example, in Random, while CBSм(EBC,MC) solved all 25 problem instances with 300 agents, all other algorithms did not solve any problem instance with 250 agents. We note that LaCAM* was also best in many cases. CBSм(EBC,MC) performed best in Empty, Random, Room, and Warehouse, and LaCAM* performed best in Maze, Game1, Game2, and City but was much worse than CBSм(EBC,MC) in the other maps. A general trend is that CBSм(EBC,MC) outperformed LaCAM* in more dense maps with fewer empty cells while LaCAM* outperformed CBSм(EBC,MC) in more sparser maps (the number of empty cells in each map is presented in parentheses in the figure). Thus, LaCAM* excels in some domains but is relatively poor in other domains. CBSм(EBC,MC) is thus more robust across all domains tested.

Table 4 presents the average MKS for the experiment. Here, we only present averages for cases where all 25 problem instances were solved by the tested algorithms. This shows that CBSм(EBC,MC) allows solving many problem instances with hundreds agents across the different maps. For example, in Empty, CBSм(EBC,MC) solved 96% of the problem instances with 500 agents, which is the maximum number of agents possible for this map; many of which were optimally solved for the first time (!) for MKS due to CBSм(EBC,MC).

k	CBSм(LC)	CBSм(EBC)			CBSмs(LC)
		GBFS	PS	MC	
5	**118**	119	**118**	121	**118**
10	228	229	229	241	**225**
20	460	467	465	508	**449**
50	1,193	1,251	1,230	1,445	**1,136**

Table 5: Average SOC on Random.

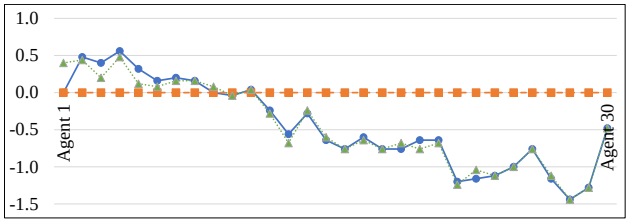

Figure 3: Minimizing SOC, MS, RMKS.

6 BEYOND MKS AS A PRIMARY OBJECTIVE

When optimizing MKS, the cost is determined only based on the highest-cost path. Consequently, all other paths may have a high cost for no particular reason. While the main aim of this paper is to find the optimal MKS solution, it may be desired to have other additional objectives. In this section, we discuss two such cases.

6.1 SOC as a Secondary Objective

Often, one may prefer a solution that minimizes the SOC as a secondary objective, among all optimal MKS solutions. Such objective, denoted *MS*, was considered, for example, by Liu et al. [17] for the *Multi-Agent Pickup-and-Delivery* problem, where each agent has to visit two ordered locations (pickup location and delivery location), and by Lam et al. [9] for the *Multi-Agent Collective Construction* problem, where multiple agents are required to construct a given three-dimensional structure by repositioning blocks.

All CBS algorithms in this paper that prioritize high-level nodes with lower MKS return the optimal MKS solution. However, as CBSм(LC) uses a low level that plans the lowest-cost paths for the agents, its plan has a relatively low SOC. Yet, it does not guarantee the minimal SOC (as a second objective). Therefore, we also propose a simple CBS-based algorithm, called CBSмs(LC), which returns the minimal SOC solution among all minimal MKS solutions. To do so, CBSмs(LC) prioritizes high-level nodes with low MKS and breaks ties by prioritizing low SOC. Notably, CBSмs(LC) must use LC as a low level, and not EBC, to obtain the new objective. This is because, while returning a bounded-cost path may minimize the first objective, MKS, it may not minimize the second objective, SOC.

Table 5 shows the average SOC of the returned plan by CBSм(LC), CBSм(EBC,GBFS), CBSм(EBC,PS), CBSм(EBC,MC), and CBSмs(LC) on Random. Interestingly, often, the optimal solution for this objective (which was found by CBSмs(LC) in Table 5) has the same SOC as the optimal SOC solution (Table 1), and the averages in our experiments were identical. CBSм(LC) finds solutions with close to optimal SOC. On the other hand, CBSм(EBC,MC) exploits the leeway between lowest-cost paths and bounded-cost paths, and finds solutions with higher SOC. As CBSмs(LC) could not benefit from using EBC (as opposed to CBSм(EBC,MC)), it did not solve many problem instances with more than 50 agents (100 and more). Therefore, it requires further research to improve CBSмs(LC).

6.2 Recursive MKS

While, in MKS, the highest-cost path is minimized, in cases where time is crucial, one may prefer to also recursively minimize the next highest-cost path. That is, for two solutions Π_1 and Π_2, we prefer the one with the lower MKS. If their MKS is equal, the highest-cost path is excluded from each solution, and their MKS is compared again. This process recursively continues until a preferable solution is chosen. If the costs of all of their paths are equal, then there is no preferable solution among them. We call this objective function *Recursive MKS* (or *RMKS* in short). For example, consider an instance with three agents with solutions Π_1 with costs 6,4,3 in and $\Pi_2$with costs 6,5,1. When MKS is solely minimized, we treat both plans equally, as $C_{MKS}(\Pi_1) = C_{MKS}(\Pi_2) = 6$. If we consider SOC as a secondary objective (MS), as defined in Section 6.1, then Π_2 is preferred, as $C_{SOC}(\Pi_1) = 13$ and $C_{SOC}(\Pi_2) = 12$. However, if MKS is minimized recursively, then Π_1 is preferred, as two agents already reach their goals at timestep 4 in Π_1 while they reach their goals only at timestep 5 in Π_2. Similar to CBSмs(LC) (Section 6.1), a CBS-based algorithm for RMKS (denoted CBSRм(LC)) uses LC as its low level. CBSRм(LC) must use such a low level because the RMKS objective function is influenced by the paths of all agents and, if one of the paths is not of lowest cost, the solution may be suboptimal. The difference between CBSмs(LC) and CBSRм(LC) is that CBSRм(LC) prioritizes high-level nodes according to RMKS.

To evaluate the different cost functions, we compared three CBS variants: minimizing SOC (with CBSs), MS (CBSмs(LC)), and RMKS (CBSRм(LC)) on Random with 30 agents. We ordered the costs in each solution from the highest-cost path (denoted *Agent 1*) to the lowest-cost path (*Agent 30*) and calculated the average for each path's cost. Figure 3 shows the results where the x-axis is the different agents and the y-axis is the cost of the given agent. Costs are provided with their constant deviations (plus or minus) from RMKS (normalized to $y = 0$). Both RMKS and MS have the same cost at Agent 1 as both minimize MKS. However, for the following agents, RMKS achieves lower costs. As expected, SOC has the smallest area under the curve.

7 CONCLUSION AND FUTURE WORK

We showed that *Conflict-Based Search* (CBS) [24] is more general than originally defined and can be seen as a framework where both its high and low levels can be adjusted for a selected objective. We demonstrate this idea for Makespan (MKS) and propose an *Extended Bounded-Cost Search* (EBC) for its low level. Our experiments show that prioritizing nodes in EBC by the minimal number of conflicts (CBSм(EBC,MC)) significantly outperforms all other CBS-based algorithms for MKS. Many instances were first solved only due to CBSм(EBC,MC). The only competitor is the recently introduced LaCAM* where, in some cases, CBSм(EBC,MC) outperforms LaCAM* and, in others, LaCAM* outperforms CBSм(EBC,MC). We also presented two extensions for MKS and CBS variants for them.

Future work will: **(1)** adjust other MAPF algorithms for MKS, e.g., *Branch and Cut and Price* (BCP) [8] and *Increasing Cost Tree Search* (ICTS) [25]; **(2)** create CBS's admissible heuristics [4, 11] for MKS; **(3)** adapt CBS for other objective functions, e.g., the total traveled distance (fuel) [29]; **(4)** examine CBSм(EBC,MC) in lifelong/online scenarios [15, 18, 19, 36, 37], where new tasks/agents arrive over time, and **(5)** provide more comparison with LaCAM* to further determine the pros and cons of each.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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*.

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

[6] Daniel Gilon, Ariel Felner, and Roni Stern. 2017. Dynamic Potential Search on Weighted Graphs. In *SoCS*. 119–123.

[7] Ofir Gordon, Yuval Filmus, and Oren Salzman. 2021. Revisiting the Complexity Analysis of Conflict-Based Search: New Computational Techniques and Improved Bounds. In *SoCS*. 64–72.

[8] 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.

[9] Edward Lam, Peter J. Stuckey, Sven Koenig, and T. K. Satish Kumar. 2020. Exact Approaches to the Multi-agent Collective Construction Problem. In *CP*. 743–758.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Jiaoyang Li, Kexuan Sun, Hang Ma, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. 2020. Moving Agents in Formation in Congested Environments. In *AAMAS*. 726–734.

[15] 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.

[16] Jae Kyu Lim and Panagiotis Tsiotras. 2022. CBS-Budget (CBSB): A Complete and Bounded Suboptimal Search for Multi-Agent Path Finding. *ArXiv* (2022).

[17] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. 2019. Task and Path Planning for Multi-Agent Pickup and Delivery. In *AAMAS*. 1152–1160.

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

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

[20] 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.

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

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

[23] Malcolm Ryan. 2010. Constraint-based multi-robot path planning. In *ICRA*. 922–928.

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

[25] 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.

[26] 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.

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

[28] Roni Stern, Rami Puzis, and Ariel Felner. 2011. Potential Search: A Bounded-Cost Search Algorithm. In *ICAPS*. 234–241.

[29] 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.

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

[31] Pavel Surynek. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI*. 564–576.

[32] Pavel Surynek. 2017. Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems. *Ann. Math. Artif. Intell.* 81, 3-4 (2017), 329–375.

[33] 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.

[34] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. 2016. An Empirical Comparison of the Hardness of Multi-Agent Path Finding under the Makespan and the Sum of Costs Objectives. In *SoCS*. 145–146.

[35] Pavel Surynek, Roni Stern, Eli Boyarski, and Ariel Felner. 2022. Migrating Techniques from Search-based Multi-Agent Path Finding Solvers to SAT-based Approach. *JAIR* 73 (2022), 553–618.

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

[37] 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.

[38] Jingjin Yu and Steven M. LaValle. 2013. Planning optimal paths for multiple robots on graphs. In *ICRA*. 3612–3617.

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

[40] 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.

[41] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. 2015. *Constraint solving and planning with Picat*. Springer.