Ben-Gurion University of the Negev
Faculty of Engineering Science
Department of Information Systems Engineering

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

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE M.Sc DEGREE

By: Amir Maliah

Supervised By: Prof. Ariel Felner and Dr. Dor Atzmon

October 2025

Ben-Gurion University of the Negev
Faculty of Engineering Science
Department of Information Systems Engineering

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

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE M.Sc DEGREE

By: Amir Maliah

Supervised By: Prof. Ariel Felner and Dr. Dor Atzmon

Author: Amir Maliah	Date: 05.10.2025
Supervisor: Ariel Felner	Date: 09.10.2025
Supervisor: Dor Atzmon	Date: 09.10.2025
Chairman of Graduate Studies Committee:	Date:

October 2025

Abstract

The Multi-Agent Path Finding (MAPF) problem seeks to compute collision-free paths for multiple agents navigating in a shared space. Conflict-Based Search (CBS) is a prominent two-level framework for solving MAPF problems optimally, especially for minimizing the Sum-of-Costs (SOC) – the sum of the lengths of the paths of all agents. However, many real-world applications require minimizing the Makespan (MKS) – the time until the last agent reaches its goal. In this thesis, we propose a principled redesign of CBS for the MKS objective, with innovations in both its high-level and low-level search components. We introduce the Extended Bounded-Cost Search (EBC) problem as a more appropriate objective for the low-level of CBS for MKS. We also introduce an algorithm to solve EBC problems, EBC*. We demonstrate that solving CBS for MKS using EBC* as the low-level algorithm not only retains optimality but also significantly improves efficiency. We also propose various variants of CBS using EBC* to further enhance performance. In addition, we examine heuristics for MKS using CBS. Most of this work is based on our paper published in AAMAS 2025 [10], and this thesis further extends that work.

Keywords

Multi-agent path finding

Makespan

Sum-of-costs

Conflict-based search

High-level

Low-level

Bounded-cost search

Extended bounded-cost search

Heuristic

Root node

Acknowledgements

The research was supported by the Binational Science Foundation (BSF) under grant #2021643, by the Israel Science Foundation (ISF) under grant #909/23, and by Israel's Ministry of Innovation, Science and Technology (MOST) under grant #6908. In addition, I would like to express my appreciations to my supervisors, Prof. Ariel Felner and Dr. Dor Atzmon, for the guidance and support, ideas and brainstorming, they have given me all along this thesis.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

SAPF	**S**ingle-**A**gent **P**ath **F**inding
MAPF	**M**ulti-**A**gent **P**ath **F**inding
MKS	**Mak**e**s**pan
SOC	**S**um-**O**f-**C**osts
CBS	**C**onflict-**B**ased **S**earch
LC	**L**owest **C**ost
BCS	**B**ounded-**C**ost **S**earch
EBC	**E**xtended **B**ounded-**C**ost **S**earch
CT	**C**onstraint **T**ree
PC	**P**rioritizing **C**onflicts
DS	**D**isjoint **S**plitting
SAT	Boolean **Sat**isfiability Problem
LaCAM	**La**zy **C**onflict-**A**voidance with **M**atching
GBFS	**G**reedy **B**est-**F**irst **S**earch
PS	**P**otential **S**earch
MC	**M**inimal Number of **C**onflicts
ASH	**A**gent **S**ubset **H**euristic
I	**I**ncremental
R	**R**erun
H	**H**euristic
MD	**M**anhattan **D**istance
RMKS	**R**ecursive **Mak**e**s**pan
MS	**M**akespan as first objective and **S**um-of-costs as secondary objective

1 Introduction

The Multi-Agent Path Finding (MAPF) problem [36] is concerned with planning conflict-free paths for multiple agents from their respective start to goal locations on a shared map while avoiding collisions. It arises in a variety of real-world applications, such as warehouse robotics, fleet management, automated traffic control, video games, and navigation for drones, robots, and vehicles. Two principal objective functions are commonly used to evaluate solutions: Makespan (MKS), defined as the time step at which the last agent reaches its destination (i.e., the longest individual path), and Sum-of-Costs (SOC), which accumulates the total path costs of all agents. Finding optimal solutions for both objectives, MKS and SOC, is NP-hard [37, 46]; nevertheless, due to MAPF's wide applicability, efficient optimal algorithms have been developed [9, 13, 31].

Conflict-Based Search (CBS) [31] is a prominent two-level algorithm for solving MAPF. At the high-level, each node represents paths for all agents under a set of constraints, while the low-level individually computes paths, typically returning a lowest-cost path (denoted LC) that satisfies these constraints. CBS was originally designed for minimizing SOC, as most MAPF solvers focus on this objective. However, minimizing MKS is crucial in scenarios where synchronization or overall completion time is required. CBS can be adapted to optimize either SOC or MKS by modifying the high-level node prioritization according to the chosen objective. In practice, running CBS for MKS is achieved by prioritizing high-level nodes with lower MKS, while the low-level remains unchanged and continues to compute LC paths [11, 31, 39]. This mismatch often leads to inefficiencies when minimizing MKS. We denote the standard SOC-oriented version as CBSs(LC) and the MKS-oriented version as CBSm(LC).

This thesis redefines CBS as a general framework that can accommodate different optimization objectives by independently adjusting both its high-level node prioritization and its low-level search engine. However, relying on LC in the low-level, as in CBSm(LC), is inefficient for MKS optimization. To address this, we introduce **Extended Bounded-Cost Search (EBC)**, a novel low-level search setting specifically tailored for MKS. In the standard Bounded-Cost Search (BCS) [35], given a bound B,

the task is to find a path of cost ≤ B or to declare that no such path exists. EBC extends this setting by returning a path of minimal cost when all feasible paths exceed B. We provide a general algorithm, **EBC***, for solving EBC and several instantiations of it. Building on this, we develop **CBSm(EBC)**, a new CBS variant that employs EBC in its low-level rather than LC. We further present theoretical justifications, algorithmic design, and prioritized variants of this approach.

Our empirical evaluation demonstrates that CBSm(EBC) significantly outperforms CBSm(LC), as well as a SAT-based MAPF solver designed for MKS [38]. We also compare CBSm(EBC) against LaCAM* [27], a recent MAPF solver that converges to optimal solutions, and show that CBSm(EBC) achieves superior performance in many cases, particularly on dense maps.

We also extend the framework to address two MKS-related objectives. The first extension **minimizes SOC as a secondary criterion**, and the second **recursively minimizes MKS for subsets of agents**. For each of these extensions, we introduce CBS variants and analyze their performance.

The work presented above is based on our paper published in AAMAS 2025 [10], while the following paragraph introduces additional work beyond the scope of that publication.

We present several new optional **high-level root generation modifications** for CBSm(EBC). Our experiments show that, in some scenarios, using these modifications improves the runtime. We also compared using CBSm(EBC) with these modifications to LaCAM*, and the results demonstrate that using them expanded the range of cases in which CBS outperforms LaCAM*. We also examine heuristics for the high-level nodes of CBS for MKS. We show and explain that, in contrast to the case of SOC, heuristics are not beneficial for MKS.

2 Background

2.1 Single-Agent Path Finding Problem

Single-Agent Path Finding (SAPF) [49] is a common problem which is related to finding paths over different domains for a single agent. In this problem the goal is to find a path between two states in a graph. This problem can be found in GPS navigation, robot routing, planning, network routing and many combinatorial problems. Solving path-finding problems is commonly done by a best-first search algorithm (see pseudo code below). In best-first search, we start by inserting the start state into a list (called OPEN). Until we find the goal state we repeat pulling out a state from OPEN, by some ordering function, push its neighbors into OPEN, and push the extracted state to a different list (called CLOSED). A known best-first search is called A* [12] guided by $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the shortest known path from the start state to n and $h(n)$ is a heuristic function estimating the cost from n to the nearest goal state. If the heuristic function h is admissible, meaning that it never overestimates the cost of the shortest path from n to the goal state, then A* is guaranteed to find an optimal path from the start state to a goal state, if one exists [12].

Algorithm: Best-First Search Pseudo Code

Input: start state, goal state, graph, function f

1. CLOSED = [];
2. OPEN = [start state];
3. while OPEN is not empty do:
    a. n = state with lowest value in OPEN according to f.
    b. if n is a goal state then:
        i. return the path from start to n;
    c. foreach n', directed neighbor of n do:
        i. if (n' is not in CLOSED) and ((n' is not in OPEN) or (n' is in OPEN but with higher value according to f)) then:
            1. add n' to OPEN or replace it if already there;
            2. set n as the parent of n';
    d. delete n from OPEN;
    e. add n to CLOSED;

2.2 Multi-Agent Path Finding Problem

The Multi-Agent Path Finding (MAPF) [36] problem is a generalization of the SAPF problem for $k \geq 1$ agents and defined by the tuple $\langle \mathcal{G}, A, S, G \rangle$, where $\mathcal{G} = (V, E)$ 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. The task is to find paths for all agents from their start states to their goal states, under the constraint that agents cannot collide during their movements. In its general form MAPF is NP-complete [46]

In graph $\mathcal{G}$, two vertices v_1 and v_2 are **neighbors** if there is an edge between them: $(v_1, v_2) \in E$. A **path** π_i for agent a_i is a list of neighboring vertices from start vertex s_i to goal vertex g_i. Time is discretized into timesteps. 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 $\forall t \in (0, \ldots, |\pi_i| - 2) : (\pi_i(t), \pi_i(t+1)) \in E$. 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 that all edges have a uniform cost of 1. A **plan** $\Pi = (\pi_1, \ldots, \pi_n)$ is a list of paths for the agents, which is a solution to the MAPF problem.

A **conflict** in MAPF is a collision between multiple agents. In every environment / domain we decide which conflicts are allowed and which are forbidden. In this thesis we consider vertex and swapping conflicts as the only forbidden conflicts. For any two paths π_i and π_j of agents a_i and a_j ($i \neq j$): (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 $t + 1$ ($\exists t: \pi_i(t) = \pi_j(t+1) \wedge \pi_j(t) = \pi_i(t+1) \wedge (\pi_i(t), \pi_i(t+1)) = e$).

There are many other types of conflicts between agents, the common ones are:

1) Edge conflict: two agents traverse the same edge in the same direction at the same time.
2) Following conflict: one agent moves to the location of another agent while that other agent moves to a third location.

3) Cycle conflict: a sequence of following conflicts involving at least three agents that forms a closed cycle in the agents' movements.

We say that a solution is **valid** if it is conflict-free according to the forbidden conflicts.

A MAPF solver is **sound** if it always outputs a valid solution.

A MAPF solution is **optimal** if it is a valid solution with the best cost (minimal or maximal cost, depending on the objective function) across all existing valid solutions.

This thesis will focus on minimizing the MKS objective function – the maximum, over all agents, of the number of timesteps required to reach the goal for the last time and never leave it again. The MKS of plan Π is $C_{MKS}(\Pi) = \max_{\pi_i \in \Pi} c(\pi_i)$.

2.3 Conflict-Based Search Algorithm

Conflict-Based Search (CBS) [31] is an optimal search-based MAPF algorithm. The main idea of the algorithm is to turn a solution into an optimal solution by solving its conflicts. In order to do so, we start by finding an admissible (not overestimate) solution and then we choose a conflict, which we solve by adding a constraint on one of the agents that has been involved in the conflict so it will avoid the collision. 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 $t+1$. An agent's path that satisfies all its constraints is called a **consistent path**, and a solution composed of only consistent paths is called a **consistent solution**.

The algorithm is working in a two-level structure:

The high-level is a constraint tree (CT) in which each node (N) contains a set of constraints of the agents (N.constraints), a plan that satisfies N.constraints ($N.\Pi$) (a consistent solution) and the cost of $N.\Pi$ (N.cost). The root has no constraints. If conflict between two agents has been found in $N.\Pi$, generate two children for N that one has a constraint on agent 1 to avoid the conflict and the second has a constraint on agent 2 to avoid the conflict, else, the node is declared as a goal node, and the $N.\Pi$ is returned. Each node inherits the constraints of its parent. We use best-first search on the tree by lowest N.cost. N.cost can be either $C_{SOC}(\Pi)$ or $C_{MKS}(\Pi)$ depending on whether the aim is to minimize SOC or MKS. In order to solve a chosen conflict in the current node we find new

consistent paths for the constrained agents in the child nodes. The path of each agent a_i in plan $N.\Pi$ ($N.\Pi.\pi_i$), satisfying N.constraints, is planned by CBS's low-level search.

The low-level is a SAPF solver (e.g. A*). The solver's task is to find a new consistent path for the agent for which a constraint has been added at the current high-level node.

High-Level of CBS Pseudo Code

Input: $\langle \mathcal{G}, A, S, G \rangle$, cost function C, low-level search algorithm L

1. Init OPEN = [], Root
2. Root.constraints = {}
3. Foreach $a_i \in A$
    a. Plan path Root.$\Pi.\pi_i$ with L // Low-Level
4. Root.cost = $C(\text{Root}.\Pi)$
5. Insert Root into OPEN
6. while OPEN is not empty do:
    a. Extract N from OPEN (best node in OPEN according to C)
    b. If $N.\Pi$ is valid (conflict-free) then:
        i. Return $N.\Pi$;
    c. $\langle a_i, a_j, x, t \rangle$ = choose a conflict from $N.\Pi$
    d. $N_i = \text{GenerateChild}(N, \langle a_i, x, t \rangle)$
    e. $N_j = \text{GenerateChild}(N, \langle a_j, x, t \rangle)$
    f. Insert N_i and N_j into OPEN
7. Return No Solution

$\text{GenerateChild}(N, \langle a, x, t \rangle)$:

1. Init N'
2. $N'.\text{constraints} = N.\text{constraints} \cup \{\langle a, x, t \rangle\}$
3. $N'.\Pi = N.\Pi$
4. Replan path $N.\Pi.\pi$ of agent a under $N'.\text{constraints}$ with L // Low-Level
5. $N'.\text{cost} = C(N'.\Pi)$
6. Return N'

Figure 1(a): MAPF problem instance

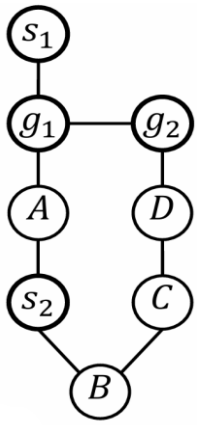


Figure 1(a) depicts a MAPF problem instance containing two agents a_1 and a_2 with start vertices s_1 and s_2, and goal vertices g_1 and g_2, respectively. Here, an optimal SOC solution $\Pi = (\pi_1, \pi_2)$ is $\pi_1 = (s_1, g_1)$ and $\pi_2 = (s_2, B, C, D, g_2)$ with $C_{SOC}(\Pi) = 5$ and $C_{MKS}(\Pi) = 4$. 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. An 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 to pass through vertex g_1 and achieve a MKS of 3. MKS of 2 (or below) is not possible, because the shortest path from s_2 to g_2 is 3.

We denote a lowest-cost low-level search by LC. LC can be implemented by Temporal-A* [34], 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.

Figure 1(b): CT for SOC

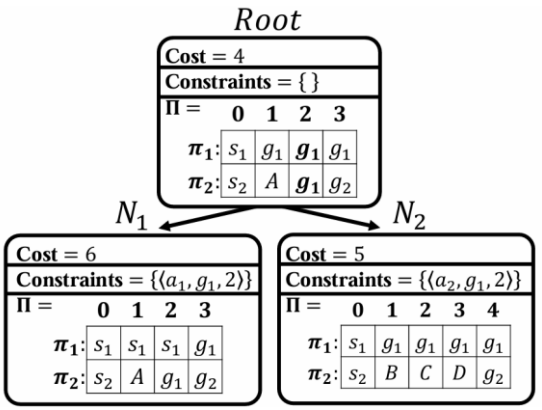


Figure 1(c): CT for MKS

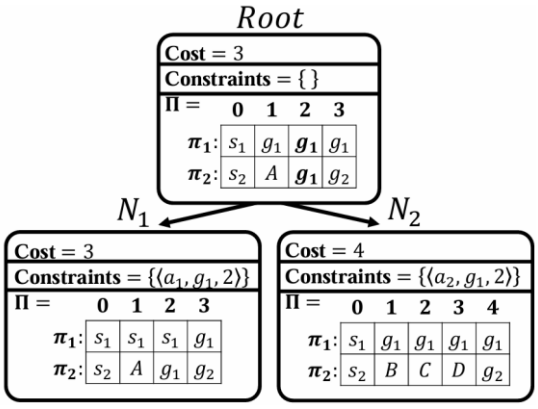


Figure 1(b) presents the CT created by CBSs(LC) for the problem instance in Figure 1(a). At the Root, the cost (SOC) is 4. We identify the vertex conflict $\langle a_1, a_2, g_1, 2\rangle$ and, thus, the Root is not a goal. To resolve the conflict, two child CT nodes N_1 and N_2 are created with the 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 (goes through 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 an optimal SOC solution is found at N_2 with a cost of 5.

Similarly, **Figure 1(c)** presents the CT created by CBSm(LC) for the problem instance in Figure 1(a). 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, an optimal MKS solution is found at N_1 with a cost of 3.

The optimality of the algorithm requires an admissible cost function (a minimization cost function, e.g. SOC and MKS), which means that for every CT node N, N.cost is at most the minimum cost over all consistent valid solutions according to N.constraints.

As for the completeness, in order to find an optimal solution if one exists the cost-function requires only non-zero-cost actions (SOC and MKS are examples of that, fuel is an example for a cost-function with a zero-cost action – the wait action). CBS cannot guarantee to identify an unsolvable problem.

2.3.1 CBS Improvements

Bypass [5] – Sometimes we can solve conflicts without splitting the high-level node into 2 children but by generating only 1 child, which can exponentially improve the running time and space. The 2 children of every node that have been generated by a chosen conflict can both/one/none of them have a higher cost than the cost of their parent node which we labeled as cardinal/semi-cardinal/non-cardinal conflict respectively. In many cases, the conflicts are not cardinal which means that at least one of the conflicting agents has an alternative path of the same cost that avoids the conflict (also called a valid bypass). If there exists such a path such that the agent has fewer

conflicts than before (also called a helpful bypass), then we adopt this alternative path instead of the previous path in the only child node.

Prioritizing Conflicts (PC) [5] – As we know, both children that has been generated from a cardinal conflict has higher costs than their parent node. Therefore, if a high-level node N has a cardinal conflict, then it is guaranteed that the cost of an optimal solution generated from N will be higher than N.cost. As a result, choosing to solve cardinal conflicts over semi-/non-cardinal conflicts may reduce the number of nodes generated by the CT, and therefore improve the running time and space of the algorithm.

Disjoint Splitting (DS) [18] - 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.

Reasoning Techniques – Impose a larger set of constraints to resolve conflicts instead of setting a single constraint on each agent. Pairwise Reasoning [17] 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 [47] generalizes Pairwise Reasoning and automatically locates pairs of agents in a graph that have multiple conflicting paths and generates constraints. Cluster Reasoning [33] extends the reasoning from pairs of agents to clusters.

High-Level-Heuristics (CBS-H) [7, 16] – As explained in PC, the algorithm is able to use cardinal conflicts to know whether all optimal solutions under a high-level node have a higher cost than this node. Therefore, we can use this fact to add a heuristic value to the cost of a high-level node N, such that this value is the number of cardinal conflicts between disjoint pairs of agents (also called disjoint conflicts) in $N.\Pi$. In order to find this value we can create a cardinal conflict graph in which each node is an agent and each edge between two nodes is a cardinal conflict between the two agents the two nodes represent. Then the heuristics can be the maximum matching or the minimum vertex cover of the graph (we can also find a lower bound to these values to reduce computational time, but it may affect the high-level performances). The graph is fully generated only once, at the root node of the CT, and its changes between 2 adjacent high-level nodes are minimal (only the edges of 1 vertex).

3 Related Work

In this section, we present previous work on MAPF related to this thesis to provide a broader perspective on key aspects of MAPF research. We divide it into three parts: MAPF cost functions, MAPF variants and MAPF algorithms.

3.1 Cost Functions

Cost functions (or objective functions) are functions to evaluate a solution of path finding problems. In SAPF a solution cost is evaluated unambiguously: the number of actions/timesteps in the agent's path (notice that in SAPF there is no use for wait actions). However, in MAPF, an evaluation of the solution can be defined in various ways. Here are the definitions of some of the common cost functions, besides MKS, which is already defined:

Sum-of-Costs (flowtime): The summation, over all agents, of the number of timesteps required to reach the goal for the last time and never leave it again. The SOC of plan Π is $C_{SOC}(\Pi) = \Sigma_{\pi_i \in \Pi}\, c(\pi_i)$.

Fuel: The summation, over all agents, of the number of move actions required to reach the goal for the last time and never leave it again**.**

Weighted Sum-of-Costs: The weighted summation, over all agents, of the number of timesteps required to reach the goal for the last time and never leave it again, where each agent's contribution is multiplied by its assigned weight. The weighted SOC of plan Π is $C_{WSOC}(\Pi) = \Sigma_{\pi_i \in \Pi}\, w_i * c(\pi_i)$.

The most common cost functions are the SOC and MKS functions. Finding an optimal solution to MAPF problem has been shown to be NP-hard for both objectives [37, 46]. The SOC objective is efficient to use when each agent's action is costly. The MKS objective is efficient to use when only the time of the latest finishing agent matters.

3.2 MAPF Variants

MAPF can be defined in many different ways, depending on the specific variant and parameter choices [36]:

- Offline vs online MAPF (agents can appear and disappear, agents may have new tasks after completing their current tasks).
- Behavior at goal: Agents stay vs disappear once they have reached their goal. Staying at the goal leads to deciding whether wait actions at goal cost some positive value vs doesn't cost at all vs cost only if the agent will leave its goal later.
- Actions costs: Actions have a uniform cost vs different costs.
- Goal assignment: How goals are allocated to agents – one goal per agent, all agents share the same set of goals, agents are divided into groups each with its own set of goals.
- Which types of conflicts are allowed and which are forbidden.
- Which cost function to use.

This thesis is applied to a variant of offline MAPF in which agents stay at goal and wait actions at goals will cost as long as the agent will leave it later, actions have uniform cost of 1, each agent has its own unique goal, forbid vertex and swapping conflicts, and using the MKS objective function – even though, this research may be modified easily to be applied to many other variants. Further research may explore these modifications.

3.3 MAPF Algorithms

A wide range of algorithms have been proposed to solve MAPF. These algorithms differ in their approach, underlying assumptions, and computational trade-offs, and can be broadly grouped into several classes:

- **Search based algorithms**: Some MAPF algorithms are based on search methods, including M* [50], RM* [50], Enhanced Partial Expansion A* (EPEA*) [51], ODA* [52], Increasing-Cost-Tree Search (ICTS) [32] and CBS [31].

- **Reduction solvers**: Much work was done reducing MAPF to known problems, such as Constraint-Satisfaction Problems (CSP) [29], SAT [4, 38, 40], Inductive Logic Programming (ILP) [45] and Answer Set Programming (ASP) [6].
- **Rule-based algorithms**: These algorithms rely on specific movement rules tailored to different scenarios and typically avoid extensive search. Agents plan their paths according to predefined rules, prioritizing computational efficiency and completeness over solution optimality [53, 54].

Each of these classes includes both **optimal** and **sub-optimal solvers**, depending on whether the algorithm guarantees optimal-cost solutions. In this thesis, the main focus is on the CBS algorithm, an optimal search-based MAPF solver. Optimal search algorithms can be further categorized into two types:

1. **Adaptations of classical single-agent search methods:** These algorithms build on algorithms originally designed for SAPF, such as A*, and include M* [50], RM* [50], EPEA* [51], and ODA* [52].
2. **Algorithms specifically designed for MAPF:** These methods, including ICTS [32] and CBS [31], are developed with multi-agent interactions in mind and often exploit problem structure for efficiency.

Understanding these algorithmic categories is essential for the subsequent discussion of CBS.

4 Methodology and Proposed Algorithms

4.1 Generalize CBS

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

Standard CBS was originally designed for SOC. Likewise, its improvements 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, CBS-H [7, 16] uses admissible heuristic values representing an increase in SOC. Designing heuristics for MKS is being explored further later in this thesis.

We evaluated the impact of some of these improvements on CBS for both SOC and MKS. We compared the baseline CBS variants (CBSs(LC) and CBSm(LC)) with variants that include these enhancements, PC+GR+GC+T+DS (defined above), which are denoted **CBSs(LC)+ and CBSm(LC)+**.

Table 1: Success rate and average cost on 32x32 grid

k	SOC		MKS	
	CBSs(LC)	CBSs(LC)+	CBSм(LC)	CBSм(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 presents the success rate of these algorithms for 25 random problem instances with a time limit of 60 seconds on 32×32 4-connected grid 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).

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 the cost of a path 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 in line with a similar observation made by Surynek et al. when investigating SAT solvers for MAPF [41]. 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 costs of optimal solutions 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 is guaranteed to be increased. By contrast, the MKS cost only increases if the cost of the highest-cost path increases.

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

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

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

Proof. Let N be a CT node and let N' be its child CT node. N' first inherits the plan $N.\Pi$ plus the constraints $N.constraints$. To resolve a conflict in N, a constraint is added on a single agent a_i in 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) maintains Property 1 by a low-level that replans a 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 a lowest-cost path. For many other objectives, Property 1 is also maintained with a similar low-level search (which replans a lowest-cost path). For MKS, for instance, the plan, where each agent has a lowest-cost path, certainly has the minimal MKS cost. However, Property 1 can also be maintained for MKS without a low-level search returning a 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 $a2$ had any path with cost ≤ 10.

The original paper on CBS designed it for SOC [31] and suggested a low-level that finds a lowest-cost path. That paper mentions that, for MKS, it is sufficient to modify only the high-level to prioritize CT nodes by their MKS. While this is true, it may not be efficient. Based on this, for optimally and efficiently solving MAPF for other objectives, we propose to adjust not only the high-level 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 maintained. 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.

4.2 Bounded-Cost Search

Typical search algorithms are designed to minimize the cost of the solution path. An important search setting is **Bounded-Cost Search** (BCS) [35]. In BCS, a bound B is given and the task is to find a solution with cost $\leq B$ as quickly as possible. The returned path is not necessarily an optimal solution whose cost is denoted by C^*. According to

the common definition, algorithms that solve BCS must either return such path or return failure which means that such path does not exist (when $C^* > B$).

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 makes sense 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 a path is 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*** [12] within the BCS framework returns a lowest cost (LC) path if it is smaller or equal to B, otherwise it returns failure. While A* is a member of this framework, it might not be efficient, as it strives to return the shortest path even if another (longer) solution within the bound can be found faster.

(ii) **Greedy Best-first Search** (GBFS) [55] (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) [35] prioritizes nodes n according to their potential (pt) for quickly leading to a bounded-cost solution. Potential of node n is defined by $pt(n) = \frac{h(n)}{B-g(n)}$, 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*.

4.3 Extended Bounded-Cost Search

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. In such cases, the user gives up because there is no solution within his/her budget. 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).

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 a lowest-cost path (with cost $> B$). EBC* is a member of the general Focal Search family [56]. EBC* maintains two lists: Open and Focal. Every generated node is added to Open. Focal is a subset of Open, where $\text{Focal} = \{n \in \text{Open} \mid f(n) \leq B\}$. 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 policies (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 using EBC* for CBS's low-level for MKS.

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 whose cost is $\leq B_i$. In EBC* $B_{i+1} = B_i + 1$ and $m = \infty$. Other extensions are also possible.

4.4 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 a lowest-cost path, is more restrictive. Instead, we suggest running 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, a more suitable low-level procedure for CBSm 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 must be increased. But, to minimize MKS, the new path for a_i (which now is the agent with the highest-cost path) should be minimized (albeit larger than the previous MKS). This is exactly the EBC setting 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 it has no parent node, therefore, no bound exists. To solve this problem, for R, we execute LC (various alternatives to this solution will be presented later).

Theorem 1. CBSm(EBC) maintains Property 1.

Proof. To prove that CBSm(EBC) maintains 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 a lowest-cost path. In both cases, the cost of N' is the same for CBSm(EBC) and CBSm(LC).

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 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 prioritizing 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 prioritizes 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 **CBSm(EBC,MC)**.

Table 2: CBS high- and low-levels variants

Algorithm	High-Level Priority	Low-Level Problem	Low-Level Priority
CBSs(LC)	SOC	LC	Minimum f
CBSm(LC)	MKS	LC	Minimum f
CBSm(EBC,GBFS)	MKS	EBC	Minimum h
CBSm(EBC,PS)	MKS	EBC	Minimum pt
CBSm(EBC,MC)	MKS	EBC	MC
CBSms(LC)	MKS then SOC	LC	Minimum f
CBSrm(LC)	RMKS	LC	Minimum f

Table 2 summarizes our new understandings on CBS. CBS is, in fact, a general framework which any instantiation of it needs to specify the prioritization function in the high-level and the prioritization function in the low-level, as well as the problem the low-level solves. All the algorithms mentioned in this paper are listed in the table. The variants in the two last rows are described later in this thesis.

We note that the idea of favoring nodes with minimal MC was used before, but in other contexts. These include: the SIPPS [15] algorithm in lifelong MAPF, where new tasks arrive over time; the bCOA* [21] algorithm in bounded-suboptimal MAPF, where the demand for optimality is relaxed; and the CBS-M [19] algorithm 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 thesis.

CBS is, in fact, a general framework which any instantiation of it needs to specify the prioritization function in the high-level (e.g. SOC/MKS) as well as the problem the low-level solves (e.g. LC/EBC), and finally the exact algorithm used for the low-level (e.g. f/GBFS/PS/MC). In our experiments below, all CBS-based algorithms had the additional high-level tie-breaking strategy of minimal conflicts.

4.5 High-Level Heuristics for CBSm

As explained above, conflicts can be divided into 3 groups by their cardinality – cardinal conflicts, semi-cardinal conflicts and non-cardinal conflicts. CBS-H [7, 16] is a variant of the CBS algorithm which uses a high-level heuristics for the SOC objective function by utilizing cardinal conflicts between disjoint pairs of agents. According to CBS-H the heuristic value of node N is the number of disjoint conflicts in $N.\Pi$. This heuristic is only applied for the SOC objective but not for MKS, because if all agents should increase their path's length by 1, the SOC will increase by k but the MKS will increase by 1. In order to modify it for MKS we can use a heuristic value of 1, instead of the number of disjoint conflicts, when a node has at least 1 cardinal conflict.

A cardinal conflict for the MKS objective occurs differently for the SOC objective – For the SOC objective, a cardinal conflict occurs when in order to resolve a conflict at least one of the agents must increase its path length. For the MKS objective, a cardinal conflict occurs when in order to resolve a conflict at least one of the agents must increase its path length enough to exceed the current MKS value. Therefore, for MKS, a cardinal conflict occurs less frequently than for SOC, making heuristics based on cardinal conflicts less effective for MKS.

Agent Subset Heuristic (ASH) for CBSm's High-Level: Instead of modifying heuristics of SOC to fit MKS, we can suggest heuristics designed specifically for MKS. Notice that the SOC value is determined by the paths of all agents, but the MKS value is determined by the path of an individual agent and by the paths of some other agents that affect its path. Hence, computing the MKS cost for any subset of agents provides an admissible heuristic for the MKS objective.

Given a high-level node N let the optimal solution cost of N be N.OSC (the minimum cost over all consistent valid solutions according to N.constraints). Let N' be the same node as N but for a subset of agents A' (N' has the same constraints and same paths as

in N for the agents of A'), then $N'.\,OSC \leq N.\,OSC$. Therefore, $\mathrm{Max}(0, N'.\,OSC - N.\,cost)$ is an admissible heuristic for node N. In general, this can be done for more than one group of agents such that an admissible heuristic for node N is $\mathrm{Max}(0, N_1'.\,OSC - N.\,cost, N_2'.\,OSC - N.\,cost, N_3'.\,OSC - N.\,cost \ldots)$.

In our experiments we experimented with solving every possible pair of agents (denoted ASH-pairs). Further research may explore different variants of this method (different subsets).

4.6 High-Level Root Behavior Modifications for CBSm

In CBSm(EBC) and in CBSs(LC) the root node generation procedure is identical. That is because we can't use EBC* in the low-level of the root because no bound has been set yet. But is that so? When generating the root we call the low-level k times, once for each agent. Let $\Pi_n(N)$ be the plan of agents 1,2,3,…,n of high-level node N. After the n'th low-level call at the root node, R, we have paths for agents 1,2,3,…,n. At this point there is a guarantee that the solution cost will be at least the MKS of plan $\Pi_n(R)$, which is: $c(\Pi_n(R)) = \max_{\pi_i \in (\Pi_n(R))} c(\pi_i)$. Therefore, we can use EBC* for the next low-level call with this cost as the bound. In this way we can use EBC* as the low-level of the high-level root node as well (for the first agent we will use LC which is the same as EBC* with $B = 0$). We denote that algorithm as **CBSm(EBC)(I)** (I for incremental).

Let L be the agent with the longest shortest path across all agents, so after generating the high-level root node the length of L's path will determine the MKS of the node, R.cost. Let's assume that all agents 1 to L-1 have shorter shortest path than agent L, then during the root generation process, agents 1 to L-1 will execute EBC* with $\mathrm{bound} < R.\,\mathrm{cost}$, which is more restrictive than executing EBC* with $\mathrm{bound} = R.\,\mathrm{cost}$. A more restrictive search may lead to more conflicts because less optional paths are available. As a solution, we can sort the agents by some heuristics such as Manhattan distance or by the agents' lowest cost paths lengths in a decreasing order. The two algorithms that use these methods are denoted as $\mathbf{CBSm(EBC)(IH_{MD})}$ and $\mathbf{CBSm(EBC)(IH_{LC})}$ respectively (both can be called as **CBSm(EBC)(IH)**) (H for heuristic).

By using **I** or **IH**, each agent takes into consideration minimizing the conflicts with the agents' paths calculated before. But what about the agents' paths that have not been calculated yet? We cannot avoid conflicting with them because their paths still have not been found. Therefore, we can replan each agent with respect to all other agents after finding their initial paths. In this way, all agents' paths take into consideration the paths of all other agents during the generation of the high-level root node. This method can be used alone, or after the **I** method or after the **IH** methods, denoted as **CBSm(EBC)(R)**, **CBSm(EBC)(IR)**, $\mathbf{CBSm(EBC)(IH_{MD}R)/(IH_{LC}R)/(IHR)}$ respectively (R for rerun).

Note that multiple reruns can be executed in order to reduce the number of conflicts, but the trade-off between the number of conflicts and the computation time of each rerun should be taken into consideration.

High-Level of CBSm(EBC) (with Optional Root Adjustments) Pseudo Code

Input: $\langle G,A,S,G \rangle$

1 Init OPEN = [], Root
2 Root.constraints = {}
3 Sort A by Manhattan distances / Lowest cost paths lengths in decreasing order // IH_{MD}/IH_{LC}
4 Foreach $a_i \in A$
    b. Plan path Root.$\Pi.\pi_i$ by LC // no I and no IH
    or
    c. Plan path Root.$\Pi.\pi_i$ by EBC // I or IH
5 Foreach $a_i \in A$
    a. Plan path Root.$\Pi.\pi_i$ by EBC // R
6 Root.cost = $C_{MKS}(Root.\Pi)$
7 Insert Root into OPEN
8 While OPEN is not empty do:
    a. Extract N with minimum MKS from OPEN
    b. If N.Π is valid (conflict-free) then:
        i. Return N.Π;
    c. $\langle a_i, a_j, x, t \rangle$ = choose a conflict from N.Π
    d. $N_i = GenerateChild(N, \langle a_i, x, t \rangle)$

e. $N_j = \text{GenerateChild}(N, \langle a_j, x, t \rangle)$

f. Insert N_i and N_j into OPEN

9 Return No Solution

$\text{GenerateChild}(N, \langle a, x, t \rangle)$:

1 Init N'

2 $N'.\text{constraints} = N.\text{constraints} \cup \{\langle a, x, t \rangle\}$

3 $N'.\Pi = N.\Pi$

4 Replan path $N.\Pi.\pi$ of agent a under $N'.\text{constraints}$ by EBC

5 $N'.\text{cost} = C_{MKS}(N'.\Pi)$

6 Return N'

In line 3 we can choose to sort the agents by some ordering function according to the chosen heuristic (Manhattan distance or lowest cost paths individual costs).

In line 4b, when calculating a path for agent a_i using EBC, the bound is the current MKS of the root: $B = C_{MKS}(\text{Root}.\Pi_i) = \max_{1 \le j < i} \left(c\left(\text{Root}.\Pi.\pi_j\right) \right)$

In line 5, we replan the paths for all agents while considering the paths of every other agent, so that the new paths aim to avoid as many conflicts as possible with all agents, rather than only with those preceding them in the agent sequence.

All of these lines can be used together in any combination to create different variations.

Figure 2: MAPF problem instance

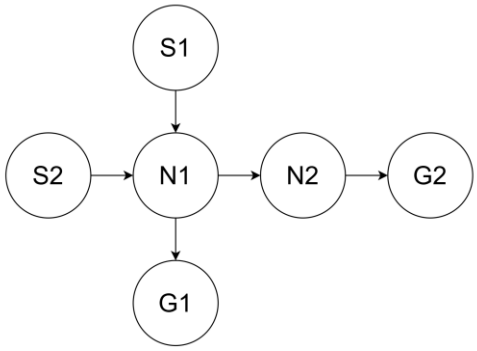


Figure 2 depicts a MAPF problem instance containing two agents a_1 and a_2 with start vertices S1 and S2, and goal vertices G1 and G2, respectively. The optimal MKS cost is 3, when agent a_2 goes to N1,N2,G2 and agent a_1 waits at S1, and then to N1,G1.

CBSm, without any of the suggested root adjustments, will generate the root node by finding lowest cost paths for agent a_1 and for agent a_2 which will be S1,N1,G1 and S2,N1,N2,G2 respectively. Then a conflict occurs ($\langle a_1, a_2, N1, 1\rangle$), and the CT will expand the root.

By using the **I** method the root will find a path for agent a_1 using EBC with B=0, results in lowest cost path (S1,N1,G1), and then find path for agent a_2 using EBC with B=2, results in lowest cost path (S2,N1,N2,G2), which are still colliding.

By using the $\mathbf{IH_{MD}}$ method the agents' order will flip such that the path for agent a_2 is being calculated first (because its Manhattan distance is greater) by using EBC for agent a_2 with B=0, which will result in lowest cost path (S2,N1,N2,G2). Only then the path of agent a_1 will be calculated by using EBC with B=3 (using MC), which now results in path (S1,S1,N1,G1), and in a conflict-free plan without any CT expansions. The same is applied for $\mathbf{IH_{LC}}$ in this case.

Lastly, by using **R** (and without using IH), after line 4, the plan will be (S1,N1,G1) and (S2,N1,N2,G2) with the conflict $\langle a_1, a_2, N1, 1\rangle$. Then in line 5, without constraints being added, the path of agent a_1 is being replanned, using EBC with B=3, and now the plan is (S1,S1,N1,G1) and (S2,N1,N2,G2) which is conflict-free (replanning a_2 now is redundant as the plan is already valid), and again, no CT expansions.

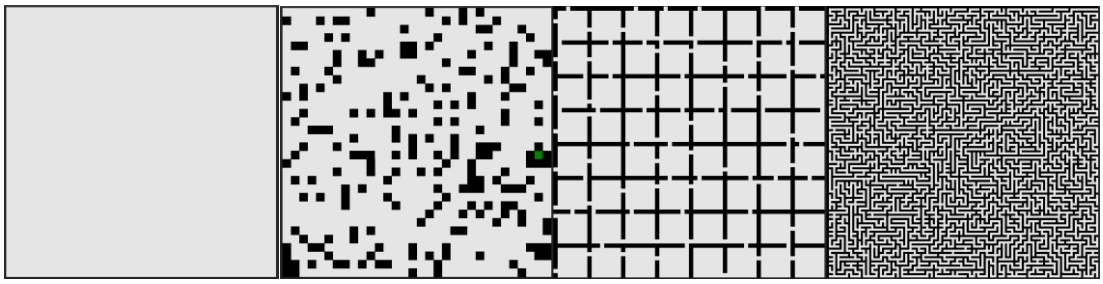
As can be seen from this running example, IH and R can reduce the number of high-level node expansions compared to not using them, and as for I, it depends on the random order of the agents according to their indices (swap the indices of a_1 and a_2, and now the case of I is identical to IH). The next section also includes results of experiments comparing these methods and their combinations to not using them and to LaCAM*.

5 Experiments

In this section, we empirically compare the various MKS solvers. We experimented on eight grid maps (also used by Li et al.), from the Moving AI repository - http://movingai.com/benchmarks/mapf:

1. empty-32-32 (Empty) (32*32 grid without obstacles)
2. random-32-32-20 (Random) (32*32 grid with 20% random obstacles)
3. room-64-64-8 (Room) (64*64 grid, 8*8 rooms, each room is 8*8)
4. maze-128-128-1 (Maze) (128*128 maze, corridors width is 1)
5. warehouse-10-20-10-2-1 (Warehouse) (10*20 racks, each of size 10*2 such that the aisles between them are of width 1)
6. den520d (Game1) (256*257 game grid map)
7. brc202d (Game2) (530*481 game grid map)
8. Paris-1-256 (City) (256*256 city grid map)

(1) (2) (3) (4)



(5) (6) (7) (8)

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 and requires downloading the Boost library. Our implementation is available at https://github.com/maliahamir/CBS-Makespan-Versions.

An important note (disclaimer): The implementation of MC for all the following experiments does not find paths with the absolute minimum number of conflicts, because once a goal node is reached (at the low-level) the algorithm returns the path found. In this way the algorithm ignores target conflicts when searching for a path with minimal number of conflicts. We tested an implementation where we do find paths that take into consideration target conflicts, but the results were worse. This can be explained by the fact that in order to avoid target conflicts at the low-level, the low-level search tree needs to grow deeper (search more timesteps). Further research may find an efficient way to implement this absolute MC logic.

5.1 Comparing CBS-Based Algorithms

We start by comparing our four CBS-based algorithms CBSm(LC), CBSm(EBC,GBFS), CBSm(EBC,PS), and CBSm(EBC,MC).

Table 3(a): Average number of low-level expansions per call from the high-level on Empty, Random and City maps

k	Empty				Random				City			
	CBSm (LC)	CBSm(EBC)			CBSm (LC)	CBSm(EBC)			CBSm (LC)	CBSm (EBC)		
		GBFS	PS	MC		GBFS	PS	MC		GBFS	PS	MC
5	**21**	**21**	**21**	22	27	**26**	37	**26**	198	**179**	188	186
10	**23**	**23**	**23**	**23**	30	**26**	37	29	207	**187**	193	191
20	29	**24**	31	27	34	**28**	38	36	268	**187**	202	200
50	40	**26**	40	34	49	**35**	50	64	348	**200**	235	241
100	68	**28**	47	46	89	**45**	163	152	452	**216**	278	313
150	82	**31**	64	61	141	**73**	140	372	543	**208**	286	349

Table 3(b): Average number of high-level expansions on Empty, Random and City maps

k	Empty		Random		City	
	CBSm (LC)	CBSm(EBC)	CBSm (LC)	CBSm(EBC)	CBSm (LC)	CBSm (EBC)

		GBFS	PS	MC		GBFS	PS	MC		GBFS	PS	MC
5	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**
10	**0**	1	1	**0**	2	9	6	**1**	**0**	**0**	**0**	**0**
20	3	4	4	**2**	5	20	14	**3**	2	1	1	**0**
50	38	30	25	**8**	38	126	135	**15**	7	22	17	**3**
100	153	134	12K	**27**	269	22K	34K	**45**	27	61	47	**9**
150	437	441	24K	**53**	3K	52K	80K	**86**	87	140	96	**18**

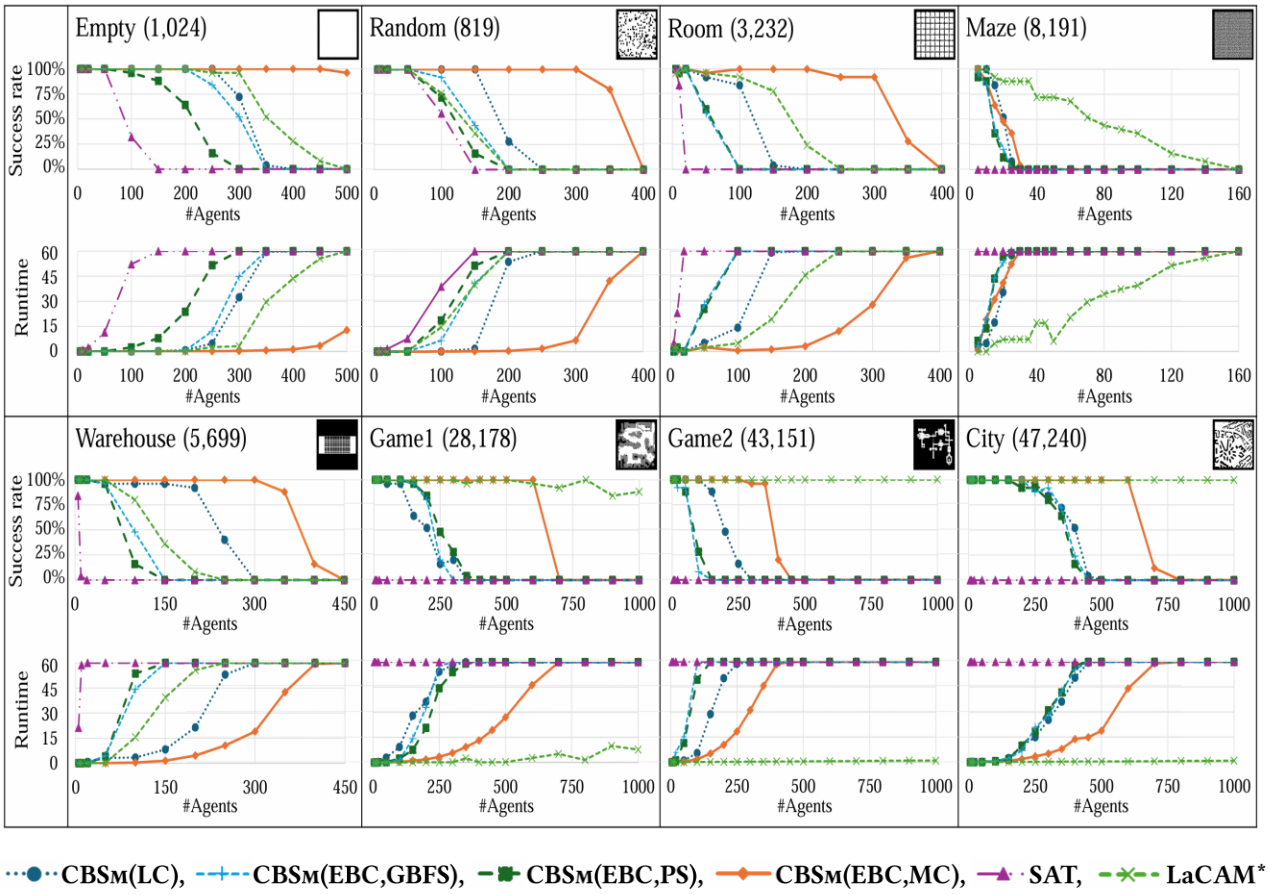
Table 3(a) and **Table 3(b)** show the average number of low-level expansions per call from the high-level and average number of high-level expansions, respectively, on three representative maps: Empty, Random, and City, and up to 150 agents. As expected, in most cases, CBSm(EBC,GBFS) and CBSm(EBC,PS) 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 agents conflict more with each other, and both algorithms expanded more high-level nodes. CBSm(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 CBSm(EBC,MC), to a small extent, expanded more nodes on each of its low-level executions on average, it expanded significantly less high-level nodes which, as we show next, resulted in a superb performance.

5.2 Comparing CBS-Based Algorithms with SAT and LaCAM*

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 [4, 38, 40]; Integer-Linear Program (ILP) [45]; Answer Set Programming (ASP) [6]; and CSP [29]. For our experiments, we used a reduction to SAT implemented in Picat [48], which is commonly used in MAPF, we denote it by SAT (We note that other compilation-based approaches, e.g., by Surynek et al. [42], 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). In addition, LaCAM* [27] is a recent anytime solver, which extends the suboptimal algorithm LaCAM [28] and converges to an 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.

Figure 3: The success rate and average planning runtime



We experimented with the following six algorithms: CBSm(LC), CBSm(EBC,GBFS), CBSm(EBC,PS), CBSm(EBC,MC), SAT, and LaCAM* on all eight grid maps. **Figure 3** shows the success rate, for timeout of 60s, and average planning runtime (in seconds). The runtime for unsolved instances was set to 60s. Clearly, CBSm(EBC,MC) significantly outperforms SAT as well as any other CBS variant and, in particular, it outperforms the common CBSm(LC). Additionally, as can be seen in the success rate graphs, for some maps, CBSm(EBC,MC) solved scenarios that no other algorithm solved in the experiments. Some of these scenarios, as part of the experiments, were resolved for the first time within the time limit of 60 seconds. For example, in Random,

while CBSm(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. CBSm(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 CBSm(EBC,MC) in the other maps. A general trend is that CBSm(EBC,MC) outperformed LaCAM* in dense maps with fewer empty cells while LaCAM* outperformed CBSm(EBC,MC) in sparse 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 others. CBSm(EBC,MC) is thus more robust across all domains tested.

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

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 presents the average MKS for the experiments. Here, we only present averages for cases where all 25 problem instances were solved by the tested algorithms. This shows that, during the experiments, many scenarios with hundreds of agents were solved across different maps for the first time for MKS, and according to the results in Figure 3, the credit can be given to CBSm(EBC,MC) and LaCAM*. For example, in Empty, CBSm(EBC,MC) solved 96% of the problem instances with 500 agents, which is the maximum number of agents possible for this map.

5.3 CBSm(EBC,MC) with and without Heuristics

As explained in the previous section about the ASH heuristics for CBSm, we experimented with the ASH-pairs variant. The following figure shows the runtime of CBSm(EBC,MC) without any high-level heuristics vs with a high-level heuristic of ASH-pairs, on 2 maps – Warehouse and maze-32-32-2 (32*32 maze, corridors width is 2) (denoted Maze2)

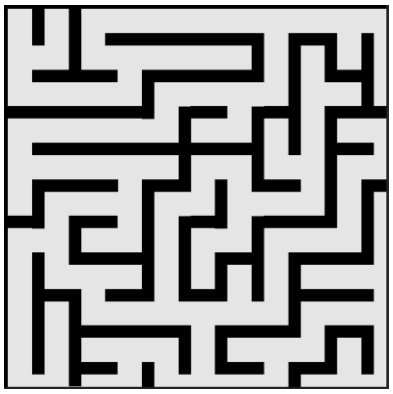


Figure 4: Runtime of CBSm(EBC,MC) with and without High-level Heuristic

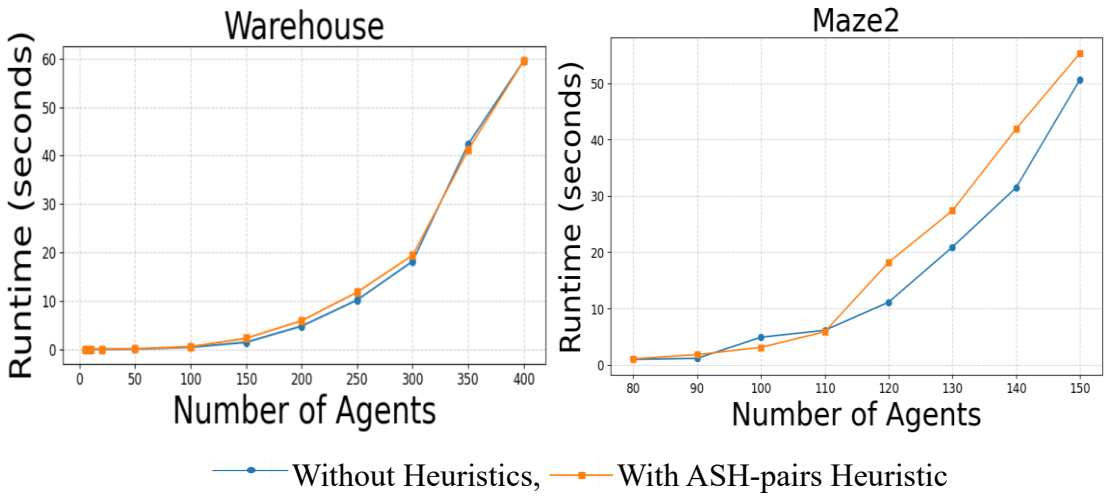


Figure 4 shows that using ASH-pairs as the high-level heuristic for CBS with MKS is not efficient and even worsen the performances. The following table will address the reason behind these results.

Table 5: High-Level Root Node Perfect Heuristic Value (The difference between the Optimal Solution MKS Cost and the High-Level Root Node MKS Cost)

#Agents	Empty	Random	Room	Warehouse	Game1	Game2	City
5	0	0	0.04	0	0	0	0
10	0	0	0.04	0	0	0	0
20	0	0	0	0	0	0	0
50	0	0	0.08	0	0	0	0
100	0	0	0	0	0	0	0
150	0	0	0	0	0	0	0
200	0	0	0	0	0	0	0
250	0	0	-	0	0	0	0
300	0	0	-	0	0	0	0
350	0	-	-	0	0	0	0
400	0	-	-	0	0	0	0
450	0	-	-	0	0	0	0
500	0	-	-	-	0	0	0
600	-	-	-	-	0	0	0
700	-	-	-	-	0	-	0
800	-	-	-	-	0	-	0
900	-	-	-	-	0	-	-

Table 5 shows that the differences between the MKS in the high-level root nodes and the MKS of optimal solutions for the experimented instances are almost always 0. A value is placed in the table only where all 25 instances succeeded. The value of 0.04 in Room with 5 agents means that one instance, out of the 25, had an optimal solution cost of 1 higher than the root node cost. The same thing with 10 agents, the value is 0.04 for the same specific instance but with 5 more agents. As for 20 agents this instance is now with value of 0, because the 10 new agents had higher MKS than the other 10 agents. And as for 50 agents, that has a value of 0.08, this is again only one instance but with value of 2.

In other words, in cases where this value is 0, the perfect heuristic of any high-level node during the search is also 0, which means that **calculating any admissible heuristic is redundant for MKS in most cases.**

On the other hand, we can observe a specific example of an instance where the perfect heuristic of the high-level root node is greater than 0 – In Maze2 map, random instance 14, with 100 agents, the perfect heuristic value of the high-level root node is 2, and while CBSm(EBC,MC) without any high-level heuristic failed to find a solution within 60 seconds, the algorithm with ASH-pairs heuristic found a solution in 1.8 seconds. The heuristic value of the root node, using ASH-pairs heuristic, was 2 (perfect), and we can see in **Figure 4**, in this specific example (Maze2 with 100 agents) that the average runtime was better with the heuristic. This shows us that when the perfect heuristic of the root node is greater than 0, then a high-level heuristic function might be very helpful, but cases like this are very rare (at least in the maps and instances we tested from the MAPF benchmark dataset).

This also tells us that when the MKS value must be increased during the search, the algorithm had to put a lot of effort in order to discover it. This can be explained by the fact that in order to guarantee an increment of the MKS value during the CBSm search, an agent must increase its path length, such that it is larger than the current MKS and that no other option is possible without increasing it. We attempted to address this problem by applying the ASH-pairs heuristic to instances where the algorithm failed without it; however, this approach did not yield any results. Further investigation into more effective variants/subsets or alternative types of heuristics for MKS may reveal that the failure of harder instances arises from the fact that these values are greater than zero.

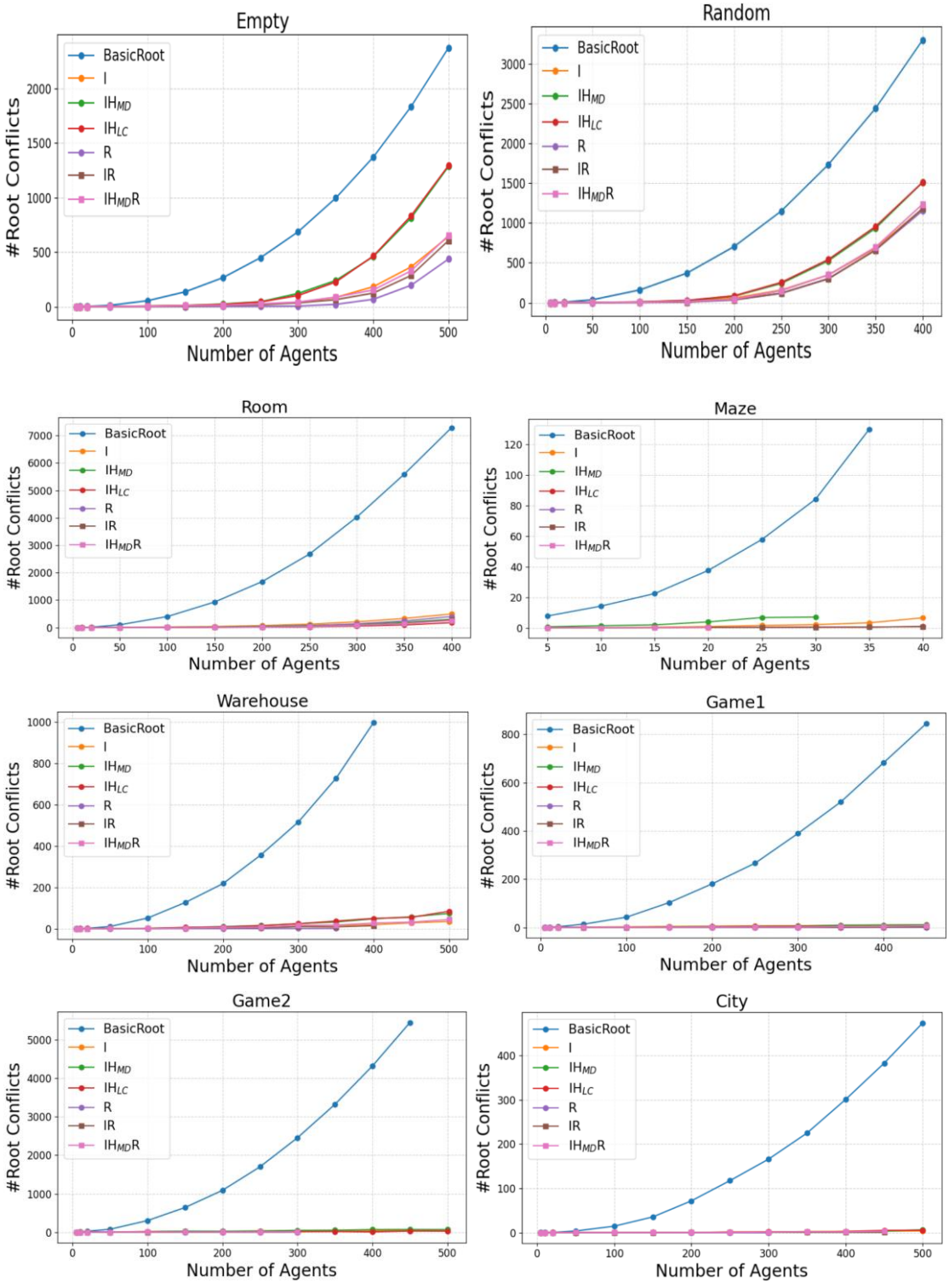
In addition, this insight tells us, that when the perfect heuristic value of the high-level root node is 0, a solution can be found without adding any constraints at all – without any high-level nodes expansions. Therefore, adjusting the high-level root node generation behavior might help significantly.

5.4 CBSm(EBC,MC) with and without Root Modifications

The goal of adjusting the high-level root node generation behavior is reducing the runtime of the algorithm by attempting to reduce the number of conflicts before any

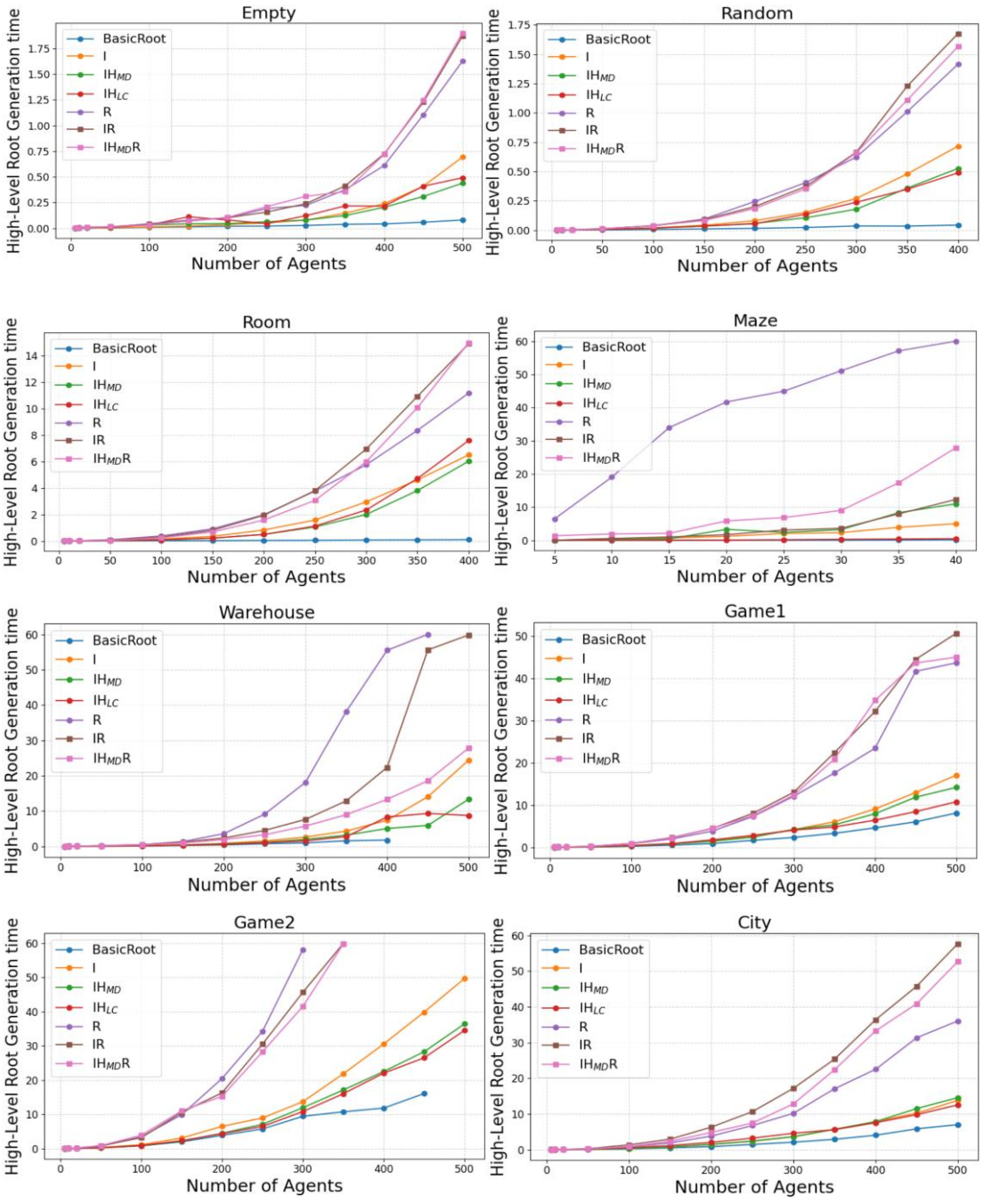
high-level node expansion. The following figure shows the number of conflicts of the high-level root plan after generating it (denoted as root conflicts), across different maps, with and without root adjustments. Denote the basic root generation as BasicRoot.

Figure 5: Number of root conflicts across different maps for different algorithms



As we can see in **Figure 5**, all of the root adjustments reduced the number of conflicts significantly compared to BasicRoot. Now we should investigate the main thing that might worsen the overall runtime performance, which is the high-level root node generation time (calculated by the time passed from creating the root node instance to finding its final plan, which is mainly the low-level runs total runtime during the high-level root node generation). The following figure presents the time took to generate the high-level root node in different variants of CBSm(EBC,MC) across different maps. Results that exceeded the 60-second time limit were set to 60 seconds.

Figure 6: High-level root generation time across different maps for different algorithms



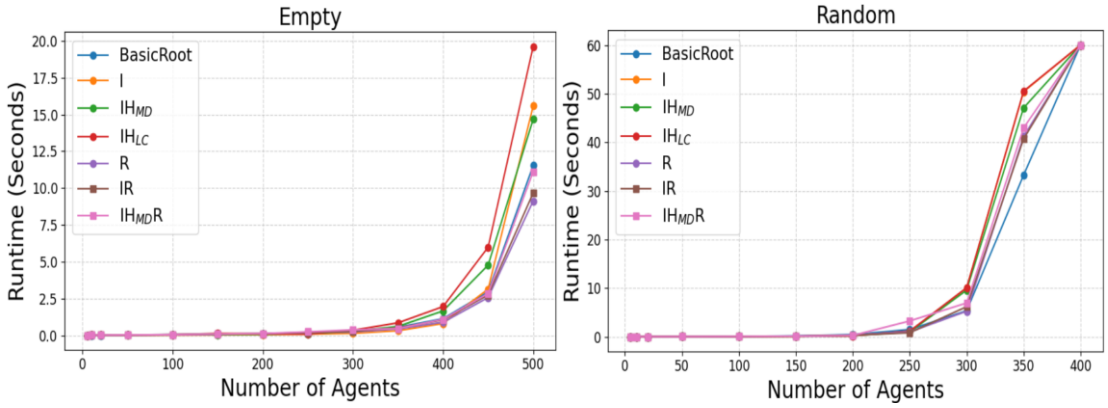
The main trend in **Figure 6** is that the root generation time, with the basic root generation behavior, is the smallest compared to the other versions, to the point where it is negligible in some maps (Empty, Random, Room, Maze, Warehouse).

Another important thing to notice is that the generation time for each map is acting different for each algorithm. For example, in Empty and Random all the algorithm, up to 400 agents, have a generation time of less than 2 seconds, which is negligible compared to the time limit of 60 seconds (0%-3%). In Room, the algorithms with the worst generation time were the 3 R algorithms, which for 400 agents had root generation times of 11-15 second, which compared to 60 seconds is 18%-23%. And in all 5 other maps the R algorithm reached generation times of (or close to) 60 seconds, which is already a failed run, even before finish generating the root.

In addition, the I algorithms that don't use R always had better performances compared to the R algorithms. This is because the R method is doing k additional EBC searches with the highest bound and with the need to avoid as many conflicts as possible with all k-1 other agents. The difference between this and IH_{LC}, is that in IH_{LC} we also compute k additional EBC (actually k-1 EBC and 1 LC) searches with the highest bound but each search, i, needs to avoid as many conflicts as possible with only i-1 agents.

After addressing these 2 factors, in regard to the overall runtime, the following figure shows the comparison in runtime between CBSm(EBC,MC) with and without root adjustments, and in addition comparing to LaCAM* in the maps where it was better than CBS (Maze, Game1, Game2, City). The success rate of the algorithms is not reported separately, as it is strongly correlated with runtime.

Figure 7: Runtime (second) – Comparing Root variants and LaCAM*



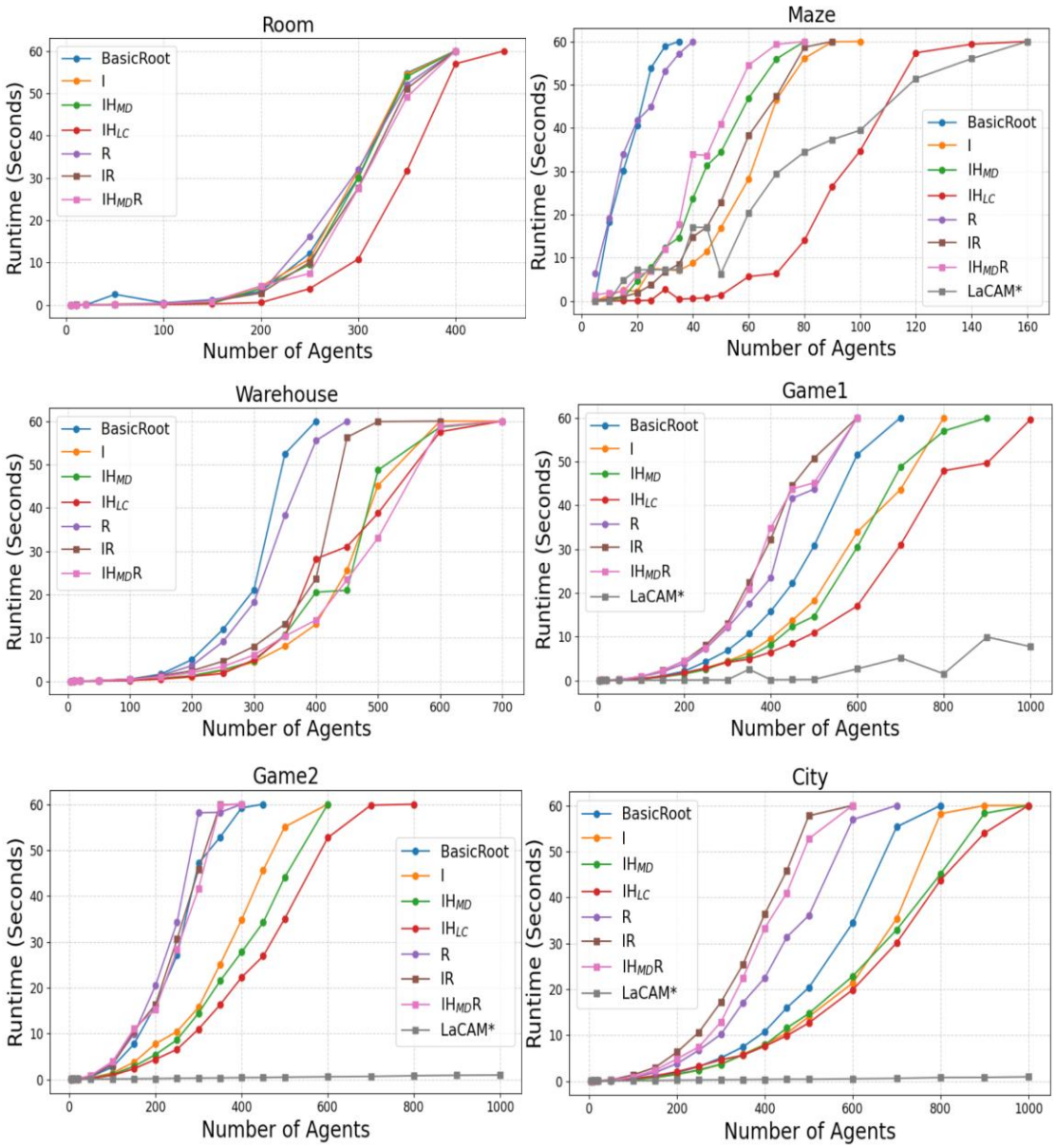


Figure 7 presents the total runtime in seconds (y-axis) of the different algorithms on different maps for different numbers of agents. **In Empty and in Random**, we can see that the R algorithms were slightly better than the I algorithms that don't use R, and that BasicRoot was between them in Empty (noticeable at 500 agents), and better than both in Random (noticeable at 350 agents). **In Room** all algorithms were the same, more or less, except for IH_{LC} which had the best runtime performances, for example with 300 agents, IH_{LC} had an average runtime of 10 seconds while BasicRoot had 30 seconds. **In Warehouse** all of the new methods were better than BasicRoot, while the 5 I algorithms had the best performances, for example, with 350 agents, the I algorithms, all had approximately 10 seconds runtime, while R had almost 40 seconds and BasicRoot had more than 50 seconds. **In Maze**, LaCAM*, which is far better than BasicRoot, had

worse runtime performances than IH_{LC} (except for 120 and 140 agents where it has 10%-15% better performances), for example, with 70 agents, IH_{LC} had an average runtime performances of 6.3 seconds while LaCAM* had 30 seconds. **As for City, Game1 and Game2**, LaCAM* is still the best but as for CBSm algorithms, IH_{LC} is the absolute best. Additionally, in those maps, the R algorithms are the worst and the I algorithms without R are the best (not including LaCAM*), while BasicRoot is between them, except for Game2 where BasicRoot is close to the R algorithms.

In summary, the runtime analysis highlights that no single algorithm consistently dominates across all map types and agent counts. Instead, performance advantages depend on the environment: R variants show slight improvements in smaller maps (Empty, Random, Room, Warehouse), IH_{LC} consistently stands out in structured or constrained maps such as Room, Warehouse, and Maze, and LaCAM* demonstrates superior scalability in larger, more complex maps like City, Game1, and Game2. Meanwhile, BasicRoot tends to perform in the middle range, occasionally surpassing R but generally lagging behind the more advanced I-based approaches. These trends emphasize the importance of map characteristics in determining algorithmic efficiency. Future work may develop a mechanism for fitting an algorithm to a given scenario.

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 thesis is to find an 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. for the Multi-Agent Pickup-and-Delivery problem [22], where each agent has to visit two ordered locations (pickup location and delivery location), and by Lam et al. for the Multi-Agent Collective Construction problem [14], 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 an optimal MKS solution. However, as CBSm(LC) uses a low-level that plans 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 **CBSms(LC)**, which returns the minimal SOC solution among all minimal MKS solutions. To do so, CBSms(LC) prioritizes high-level nodes with low MKS and breaks ties by prioritizing low SOC. Notably, CBSms(LC) must use LC as a low-level, and not EBC, to maintain optimality for 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 6: Average SOC on map Random

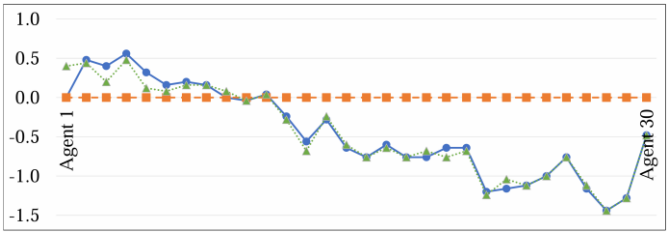
k	CBSm(LC)	CBSm(EBC)			CBSms(LC)
		GBFS	PS	MC	
5	**118**	119	**118**	121	**118**
10	228	229	229	241	**226**
20	460	467	465	508	**449**
50	1,193	1,251	1,230	1,445	**1,136**

Table 6 shows the average SOC of the returned plan by CBSm(LC), CBSm(EBC,GBFS), CBSm(EBC,PS), CBSm(EBC,MC), and CBSms(LC) on Random. Interestingly, often, an optimal solution for this objective (which was found by CBSms(LC) in Table 6) has the same SOC as an optimal SOC solution (Table 1), and the averages in our experiments were almost identical. CBSm(LC) finds solutions with close to optimal SOC. On the other hand, CBSm(EBC,MC) exploits the leeway between lowest-cost paths and bounded-cost paths, and finds solutions with higher SOC. As CBSms(LC) could not benefit from using EBC (as opposed to CBSm(EBC,MC)), it did not solve many problem instances with more than 50 agents (100 and more). Therefore, it requires further research to improve CBSms(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 plans Π_1 with costs 6,4,3 and Π_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), 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 CBSms(LC), a CBS-based algorithm for RMKS (denoted CBSrm(LC)) uses LC as its low-level. CBSrm(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 CBSms(LC) and CBSrm(LC) is that CBSrm(LC) prioritizes high-level nodes according to RMKS.

Figure 8: Average path lengths of different objectives



To evaluate the different cost functions, we compared three CBS variants: minimizing SOC (with CBSs(LC)), minimizing MS (with CBSms(LC)), and minimizing RMKS (with CBSrm(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 8** 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 Conclusions

We showed that CBS 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 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 (CBSm(EBC,MC)) significantly outperforms all other CBS-based algorithms for MKS. Many instances were first solved only due to CBSm(EBC,MC). The only competitor is the recently introduced LaCAM* where, in some cases, CBSm(EBC,MC) outperforms LaCAM* and, in others, LaCAM* outperforms CBSm(EBC,MC). We introduced a new method of applying high-level heuristics for CBSm. Additionally, we proposed several new ways of generating the high-level root node in CBSm(EBC), which, as can be seen in our experiments, can improve the performance of the algorithm significantly in some maps. We also presented two extensions for MKS and CBS variants for them.

Future work will: (1) develop a mechanism for fitting an algorithm to a given scenario; (2) experiment with additional variants for the ASH Heuristics of CBSm; (3) come up with other new methods of generating the high-level root node of CBSm(EBC). (4) adjust other MAPF algorithms for MKS, e.g., BCP [13] and ICTS [32]; (5) adapt CBS for other objective functions, e.g., the total traveled distance (fuel) [8, 36]; (6) examine CBSm(EBC,MC) in lifelong/online scenarios [20, 23, 25, 43, 44], where new tasks/agents arrive over time; (7) extend this study to consider agent delays [1, 2, 3, 24, 30]; (8) provide more comparison with LaCAM* [27] to further determine the pros and cons of each;

Bibliography

[1] Dor Atzmon, Sara Bernardini, Fabio Fagnani, and David Fairbairn. 2023. Exploiting Geometric Constraints in Multi-Agent Pathfinding. In ICAPS. 17–25.

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

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

[6] 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.

[7] 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.

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

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

[10] Amir Maliah, Ariel Felner, Dor Atzmon. 2025. Minimizing Makespan with Conflict-Based Search for Optimal Multi-Agent Path Finding. In AAMAS.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

[24] Hang Ma, T. K. Satish Kumar, and Sven Koenig. 2017. Multi-Agent Path Finding with Delay Probabilities. In AAAI. 3605–3612.

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

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

[33] 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.

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

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

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

[47] 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.

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

[49] Dijkstra, E. W. (1959). A note on two problems in connection with graphs. Numerische Mathematik, 1(1).

[50] Wagner, G., & Choset, H. (2011, September). M: A complete multirobot path planning algorithm with performance bounds. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 3260–3267). San Francisco, CA, USA.

[51] Goldenberg, M., Felner, A., Sturtevant, N., Holte, R. C., & Schaeffer, J. (2013, July). Optimal-generation variants of EPEA. In Proceedings of the 6th Annual Symposium on Combinatorial Search (SoCS) (pp. 89-97). Leavenworth, WA, USA.

[52] Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In Proceedings of the AAAI Conference on Artificial Intelligence (pp. 173–178).

[53] M. Khorshid, R. C. Holte, and N. R. Sturtevant, “A polynomial-time algorithm for non-optimal multi-agent pathfinding,” in Proc. Int. Symp. Combinatorial Search (SoCS), 2011, pp. 76–83.

[54] M. Wilde, M. Mors, and C. Witteveen, “Push-and-rotate: A complete multi-agent pathfinding algorithm,” in Proc. 6th Annual Symp. on Combinatorial Search (SoCS), 2013, pp. 98–104.

[55] N. J. Nilsson, Principles of Artificial Intelligence. Palo Alto, CA: Tioga Publishing, 1980.

[56] D. Gilon, A. Felner, and R. Stern, "Anytime Focal Search with Applications," Proc. 7th Annu. Symp. Combinatorial Search (SoCS), 2017, pp. 1–9.

תקציר

בעיית מציאת מסלולים לסוכנים מרובים (Multi-Agent Path Finding - MAPF) עוסקת בחישוב מסלולים ללא התנגשויות עבור מספר סוכנים הנעים במרחב משותף. חיפוש מבוסס קונפליקטים (Conflict-Based Search - CBS) הוא אלגוריתם בולט בעל שתי רמות (Low- and high-level) לפתרון בעיות מסוג זה בצורה אופטימלית, במיוחד לצורך מזעור סכום העלויות (Sun-of-Costs), שהוא סכום אורכי המסלולים של כל הסוכנים. עם זאת, ישנם שימושים רבים בעולם האמיתי הדורשים מזעור של זמן הסיום הכולל (Makespan) הזמן עד אשר הסוכן האחרון מגיע ליעד שלו. בעבודה זו, אנו מציעים עיצוב מחדש של CBS עבור מטרת ה-Makespan. אנו מציגים את בעיית ה-EBC כמטרה מתאימה יותר ל-low-level של CBS עבור Makespan. בנוסף, אנו מציגים אלגוריתם לפתרון בעיות EBC, הקרוי EBC*. אנו מראים כי פתרון CBS עבור Makespan תוך שימוש ב-EBC* כאלגוריתם ה-low-level לא רק שומר על אופטימליות, אלא גם משפר באופן משמעותי את היעילות בזמן. כמו כן, אנו מציעים גרסאות נוספות של CBS המבוססות על EBC* לשם שיפור ביצועים נוסף. מרבית עבודה זו מבוססת על המאמר שלנו שהתקבל לכנס AAMAS 2025 [10], ותזה זו מרחיבה את העבודה ההיא.

אוניברסיטת בן-גוריון בנגב
הפקולטה למדעי ההנדסה
המחלקה להנדסת מערכות תוכנה ומידע

מזעור Makespan עם חיפוש מבוסס קונפליקטים עבור הבעיה האופטימלית למציאת מסלולים לסוכנים מרובים

חיבור זה מהווה חלק מהדרישות לקבלת התואר מגיסטר בהנדסה

מאת: אמיר מליח

מנחים: פרופ' אריאל פלנר וד"ר דור עצמון

חתימת המחבר: אמיר מליח	תאריך: 05.10.2025
אישור המנחה/ים: אריאל פלנר	תאריך: 09.10.2025
דור עצמון	תאריך: 09.10.2025
אישור יו"ר ועדת תואר שני מחלקתית:	תאריך:

אוניברסיטת בן-גוריון בנגב
הפקולטה למדעי ההנדסה
המחלקה להנדסת מערכות תוכנה ומידע

מזעור Makespan עם חיפוש מבוסס קונפליקטים עבור הבעיה האופטימלית למציאת מסלולים לסוכנים מרובים

חיבור זה מהווה חלק מהדרישות לקבלת התואר מגיסטר בהנדסה

מאת: אמיר מליח

מנחים: פרופ' אריאל פלנר וד"ר דור עצמון

כ"ד בתשרי, תשפ"ה

16.10.2025