# Conflict-Based Search for Multi Agent Path Finding with Asynchronous Actions

Xuemian Wu
Shanghai Jiao Tong University
Shanghai, China
xuemian.wu@sjtu.edu.cn

Shizhe Zhao
Shanghai Jiao Tong University
Shanghai, China
shizhe.zhao@sjtu.edu.cn

Zhongqiang Ren†
Shanghai Jiao Tong University
Shanghai, China
zhongqiang.ren@sjtu.edu.cn

## ABSTRACT

Multi-Agent Path Finding (MAPF) seeks collision-free paths for multiple agents from their respective start locations to their respective goal locations while minimizing path costs. Most existing MAPF algorithms rely on a common assumption of synchronized actions, where the actions of all agents start at the same time and always take a time unit, which may limit the use of MAPF planners in practice. To get rid of this assumption, Continuous-time Conflict-Based Search (CCBS) is a popular approach that can find optimal solutions for MAPF with asynchronous actions (MAPF-AA). However, CCBS has recently been identified to be incomplete due to an uncountably infinite state space created by continuous wait durations. This paper proposes a new method, Conflict-Based Search with Asynchronous Actions (CBS-AA), which bypasses this theoretical issue and can solve MAPF-AA with completeness and solution optimality guarantees. Based on CBS-AA, we also develop conflict resolution techniques to improve the scalability of CBS-AA further. Our test results show that our method can reduce the number of branches by up to 90%.

## KEYWORDS

Conflict-based Search; Multi Agent Path Finding; Asynchronous Actions

**ACM Reference Format:**
Xuemian Wu, Shizhe Zhao, and Zhongqiang Ren†. 2026. Conflict-Based Search for Multi Agent Path Finding with Asynchronous Actions. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages.

## 1 INTRODUCTION

Multi-Agent Path Finding (MAPF) seeks collision-free paths for multiple agents from their respective start locations to their respective goal locations while minimizing path costs. The environment is often represented by a graph, where vertices represent the locations that the agent can reach, and edges represent actions that transit the agent between two locations. MAPF is NP-hard to solve optimally [19], and a variety of MAPF planners were developed, ranging from optimal planners [14, 16], bounded sub-optimal planners [2, 7], to unbounded sub-optimal planners [4, 9]. A common underlying assumption in these planners is that each action of an agent, either waiting in place or moving to an adjacent vertex, takes the same duration, i.e., a time unit, and the actions of all agents are synchronized, i.e., the action of each agent starts at the same discrete time step. This assumption limits the application of MAPF planners, especially when the agent speeds are different or an agent has to vary its speed when going through different edges (Fig. 1).

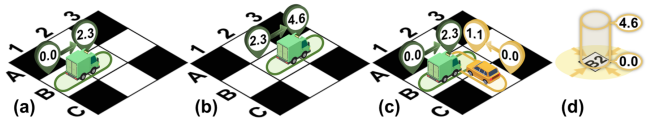


**Figure 1: A motivating example of MAPF-AA where the yellow car moves fast and the green truck moves slowly in continuous time. The circled numbers show the time points: e.g., in (a), the truck moves from B1 to B2 during the time range $[0.0, 2.3]$. This work considers the agent to occupy both ends of an edge when the agent goes through it. As a result, a constraint (as shown in (d)) at B2 with time range $[0.0, 4.6]$ is imposed on the yellow car to avoid collision as shown in (c).**

To bypass this synchronous action assumption, MAPF variants such as Continuous-Time MAPF [1], MAPF with Asynchronous Actions (MAPF-AA) [12], MAPF$_R$ [17] were proposed. The major idea in those variants is that, the actions of agents can take different amounts of time, and as a result, the agents may not start and end each of their actions at the same discrete time steps. Among the exact algorithms that can find optimal solutions, Continuous-time Conflict-Based Search (CCBS) [1] is a leading approach that extends CBS to handle various action durations.

This paper focuses on exact algorithms for MAPF-AA, which considers duration conflict [10, 12] in the sense that when an agent traverses an edge within a time range, the agent occupies both end vertices of the edge during that time range and no two agents can occupy the same vertex at the same time. CCBS [1] can be applied to solve MAPF-AA. However, a naive application can lead to incompleteness due to the infinite number of possible durations for the wait action [5], and may not be able to find a solution even if the instance is solvable. Although the reason for such incompleteness is analyzed [5], we are not aware of any fix in the literature yet, and this paper provides a possible fix by developing a new exact algorithm called Conflict-Based Search with Asynchronous Actions (CBS-AA) for MAPF-AA. Besides CCBS [1], our prior work has studied scalable yet unbounded sub-optimal algorithms for MAPF-AA [20, 21]. In terms of exact algorithms, our prior work developed an A*/M*-based exact algorithm for MAPF-AA called Loosely Synchronized M* (LS-M*) [12], which often runs slower than CCBS [1].

Additionally, in MAPF-AA, it takes a different amount of time when different agents go through the same edge, or the same agent

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

goes through different edges. Such heterogeneous duration tends to complicate the collision avoidance among the agents, and disables many conflict resolution techniques in CBS for MAPF, which limits the scalability of the exact algorithms for MAPF-AA. To improve scalability, we develop constraint propagation techniques based on the agents' action duration within CBS-AA. The intuition behind these techniques is that when two agents collide, we add constraints to the agents to forbid as many actions as possible, and forbid each action as long as possible, so that CBS-AA can resolve collisions in fewer iterations.

We compare different variants of our CBS-AA with the existing CCBS. The results show that with the proposed constraint propagation techniques, the success rate of CBS-AA is significantly higher than that of CCBS [1], and the number of iterations in high-level is reduced by up to 90%. We also compare our CBS-AA with LS-M* [12]. The results show that CBS-AA can find the optimal solution faster, and the costs of the optimal solutions are the same as those of LS-M*.

## 2 PROBLEM FORMULATION

Let $I = \{1, 2, \ldots, N\}$ be the index set of $N$ agents, where each $i \in I$ corresponds to a specific agent. The workspace is represented by an undirected graph $G = (V, E)$, where $V$ is the set of traversable vertices, and $E \subset V \times V$. Each edge $e = (u, v) \in E$ represents an action that moves an agent from $u$ to an adjacent vertex $v$. The travel time of each edge may vary for each agent. Let $\tau^i(u, v) \in \mathbb{R}_{\geq 0}$ denote the travel time for agent $i$ moving from $u$ to $v$. All agents can wait at a vertex for an arbitrary amount of time.

Let $s^i = (v, t)$ denote the space-time state of agent $i$, and let $A^i = (s^i_1, s^i_2)$ denote a state transition from $s^i_1$ to $s^i_2$. Given $A^i = ((v^i_1, t^i_1), (v^i_2, t^i_2))$, let $v_f(A^i), t_f(A^i)$ and $v_t(A^i), t_t(A^i)$ denote the space and time components for $s^i_1$ and $s^i_2$, respectively, and let $\tau(A^i)$ denote the duration of $A^i$. For any action, $t^i_2 = t^i_1 + \tau(A^i)$. For a *move* action, $v^i_2$ is adjacent to $v^i_1$, and $\tau(A^i) = \tau^i(v^i_1, v^i_2)$ is given by input. For a *wait* action, we have $v^i_1 = v^i_2$, and the duration $\tau(A^i) \in \mathbb{R}_{\geq 0}$ is determined by the planner. We use the same conflict model from existing work [10, 12], as illustrated below.

Definition 1 (Duration Occupancy). *When agent i performs an action $((v^i_1, t^i_1), (v^i_2, t^i_2))$, both $v^i_1$ and $v^i_2$ are occupied by i during the action, which is called Duration Occupancy (DO). Specifically, $v^i_1$ is occupied at time $t^i_1$, $v^i_2$ is occupied at time $t^i_2$, and both $v^i_1, v^i_2$ are occupied during $(t^i_1, t^i_2)$. At any time point, a vertex can only be occupied by at most one agent. Multiple agents are in conflict if they both occupy the same vertex for a non-empty time interval, which is referred to as Duration Conflict (DC).*

Let $\pi^i(v_s, v_g) = (a^i_0 = (v_s, 0), a^i_1, \ldots, a^i_k = (v_g, t_k))$ denote a path from $v_s$ to $v_g$. The cost of $\pi^i$ is $t_k$, denoted as $g(\pi^i)$, which is the time point it reaches $v_g$ and can permanently stay after $t_k$ without conflict. Let $V_s, V_g$ denote the set of start and goal locations of all agents, and $v^i_s \in V_s, v^i_g \in V_g$ denote the start and goal location of agent $i$ respectively. We assume there is no conflict when all agents stay at their start or goal locations. Multi Agent Path Finding with Asynchronous Action (MAPF-AA) $P = \langle V, E, V_s, V_g \rangle$ seeks to find a set of conflict-free paths such that (1) each agent $i \in I$ starts at $v^i_s$ and ends at $v^i_g$; and (2) minimize the sum of costs (SoC) of all agents' paths, i.e., $\min \sum_{i \in I} g(\pi^i)$.

## 3 PRELIMINARIES

*CBS.* Conflict-Based Search (CBS) [14] is a two-level search algorithm that finds an optimal joint path for MAPF. At the high-level, CBS constructs a search tree and starts with a root node which consists of all agents' individually optimal path ignoring any conflict. And then CBS selects a specific node that has the smallest $g$-value (sum of costs), and detects conflicts along the paths of any pair of agents. According to the detected conflict, two constraints are generated corresponding to the two sub-trees. For each of those two constraints, CBS runs a low-level search to find a new path satisfying the constraints. At the low-level, a single-agent planner is invoked to plan an optimal path that satisfies all constraints related to a specific agent. CBS guarantees finding a conflict-free joint path with the minimal sum of costs.

*CCBS.* Continue Conflict-Based Search (CCBS) [1] is a CBS-based method and can solve the $\text{MAPF}_\text{R}$ problem to optimality. It assumes the travel time for an edge is real-valued and non-uniform, and different edges have different travel times. Therefore, the time is continue and the action of agents is asynchronous. To detect conflicts, CCBS assumes that all agents move in a straight line at a constant speed and detects collisions based on agents' geometry, such as circles. To resolve conflicts, CCBS adds constraints over pairs of actions and time ranges, instead of location-time pairs.

Specifically, for a conflict $\langle (a^i, t^i), (a^j, t^j) \rangle$, which means that agent $i$ performs action $a^i$ at $t^i$ and agent $j$ performs action $a^j$ at $t^j$, and they collide, CCBS computes for each action an unsafe intervals w.r.t the other's action. The unsafe interval $[t^i, t^i_u)$ of $(a^i, t^i)$ w.r.t. $(a^j, t^j)$ is the maximal time interval starting from $t^i$ in which if agent $i$ performs $a^i$ then it is in conflict with the action $(a^j, t^j)$. CCBS adds to agent $i$ the constraint $\langle i, a^i, [t^i, t^i_u) \rangle$, which means agent $i$ cannot perform action $a^i$ in the range $[t^i, t^i_u))$, and adds to agent $j$ the constraint $\langle j, a^j, [t^j, t^j_u) \rangle$.

The low-level solver of CCBS is Constrained Safe Interval Path Planning (CSIPP), where safe intervals for a vertex are computed based on CCBS constraints. In detail, a CCBS constraint $\langle i, a^i_w, [t^i, t^i_u) \rangle$ about a wait action $a^i_w$ of vertex $v$ will divide the safe interval of $v$ to two parts: one that ends at $t^i$ and another that starts at $t^i_u$. If the action $a^i_m$ is a move action related to edge $e = (v, v')$, CCBS replaces the action $a^i_m$ with action $a^i_{m'}$ that starts by waiting in $v$ for duration $t^i_u - t^i$ before moving to $v'$.

CCBS has issues as reported in [5]. CCBS fails to resolve conflicts for wait actions, since the time is continue, there are infinitely many wait actions with duration time $\tau_w \in \mathbb{R}_{\geq 0}$. The constraint added by branching each time in CCBS only focuses on a specific duration of the wait action, which can lead to infinite number of branching when resolving conflict caused by wait actions. For example, wait action $a^i_w = (B, B, 2.0)$ means waiting at vertex $B$ for a duration of 2.0, and CCBS adds a constraint $\langle i, a^i_w, [t^i, t^i_u) \rangle$ to prohibit agent $i$ from performing $a^i_w$ at time $t \in [t^i, t^i_u)$. But agent $i$ still can preform wait action $a^i_{w1} = (B, B, 2.01), a^i_{w2} = (B, B, 2.001), a^i_{w3} = (B, B, 2.0001)$... at time $t \in [t^i, t^i_u)$. So, CCBS may not terminate when there is a wait action.

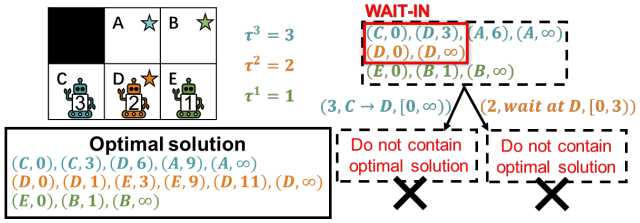


**Figure 2: Toy example for the issues of CCBS.**

In the open-sourced implementation[1], CCBS makes a change when transferring constraints about wait action to low-level solver. For the previous constraint $\langle i, a^i_w, [t^i, t^i_u)\rangle$, CSIPP divides the safe interval of $B$ to two parts: $[0, t^i)$ and $[t^i_u, \infty)$, which means that agent $i$ can not perform any wait action $a = (B, B, \tau_w), \tau_w \in \mathbb{R}_{\geq 0}$ in $[t^i, t^i_u)$. However, "not perform any wait action $a = (B, B, \tau_w), \tau_w \in \mathbb{R}_{\geq 0}$ in $[t^i, t^i_u)$" is not equivalent to "not perform wait action $a^i_w = (B, B, 2.0)$ in $[t^i, t^i_u)$". This inconsistency may miss feasible solutions during branching and can not guarantee completeness and optimality. For the rest of the paper, we refer to this implemented version as CCBS.

Example 1. *As shown in Fig. 2, there are three agents $I = \{1, 2, 3\}$. The duration of all move actions of agent 1, 2 and 3 is $\tau^1 = 1$, $\tau^2 = 2$ and $\tau^3 = 3$, respectively. In the root node of high-level, agent 2 and 3 have a conflict $\langle (a^2, t^2 = 0), (a^3, t^3 = 0)\rangle$ where $a^2 = (D, D, \infty)$ is a wait action with $\infty$ duration time, $a^3 = (C, D, 3)$ is a move action from $C$ to $D$. By Def. 1, the unsafe interval of $(a^2, t^2 = 0)$ w.r.t. $(a^3, t^3 = 0)$ is $[0, 3)$ and the unsafe interval of $(a^3, t^3 = 0)$ w.r.t. $(a^2, t^2 = 0)$ is $[0, \infty)$. By the implementation of CCBS, two constraints are generated, $(3, C \to D, [0, \infty))$ forbids agent 3 to perform action $a^3$ in the range $[0, \infty)$ and $(2, wait\ at\ D, [0, 3))$ forbids agent 2 to perform any wait action $a = (D, D, \tau_w), \tau_w \in \mathbb{R}_{\geq 0}$ in the range $[0, 3)$. The sub-trees of both branches do not contain the optimal solution.*

*LS-M*.* Loosely Synchronized M* (LS-M*) [12] solves MAPF-AA by introducing new search states that include both the locations and the action times of the agents. Similar to A*, LSS iteratively selects states from an open list, expands them to generate successors, prunes those that are either conflicting or less promising, and inserts the remaining ones into the open list for future expansion. This process continues until a conflict-free joint path from the start locations to the goal locations is found for all agents. LS-M* further introduce the idea of subdimensional expansion [16] into LSS and can handle more agents than LSS. LS-M* is complete and finds an optimal solution for MAPF-AA, but can only handle a relatively small number of agents.

# 4 METHOD

This section proposes Conflict Based Search with Asynchronous Action (CBS-AA), which finds an optimal solution for MAPF-AA. We first modify CCBS to effectively resolve conflicts and call this modified method Constraint on Single Action (CSA). Then, we use DO to propagate constraints and resolve conflicts efficiently, which we call Constraint on Multiple Actions (CMA).

[1]https://github.com/PathPlanning/Continuous-CBS

**Algorithm 1** CBS-AA

INPUT: $G = (V, E)$
OUTPUT: a conflict-free joint path $\pi$ in $G$.
1: $\Omega_c \leftarrow \emptyset$, $\pi, g \leftarrow LowLevelPlan(\Omega_c)$
2: Add $P_{root,1} = (\pi, g, \Omega_c)$ to OPEN
3: **while** OPEN $\neq \emptyset$ **do**
4: $P = (\pi, g, \Omega_c) \leftarrow$ OPEN.pop()
5: $cft \leftarrow DetectConflict(\pi)$
6: **if** $cft = NULL$ **then return** $\pi$
7: $\Omega \leftarrow GenerateConstraints(cft)$
8: **for all** $\omega^i \in \Omega$ **do**
9: $\Omega' = \Omega_c \cup \{\omega^i\}$
10: $\pi', g' \leftarrow LowLevelPlan(\Omega')$
11: Add $P' = (\pi', g', \Omega')$ to OPEN
12: **end for**
13: **end while**
14: **return** failure

*Overview.* CBS-AA (Alg. 1) is similar to CBS with three processes modified: *LowLevelPlan*, *DetectConflict* and *GenerateConstraints*. *LowLevelPlan* adapts Safe Interval Path Planning (SIPP) [11] to handle continuous-time. The numbers associated with safe intervals and constraints are all positive real numbers. We cut safe intervals into continuous time intervals according to the constraints, rather than a set of discrete time steps. *DetectConflict* detects conflicts in the continuous time range. When there is overlap in the time intervals for two agents to occupy a same vertex, a conflict is returned. In *GenerateConstraints*, we propose two different conflict resolution methods for MAPF-AA, CSA and CMA, as detailed later.

## 4.1 Conflict Detection and Classification

For a vertex $v$, there are three types of actions:

$$\begin{cases} \texttt{IN}: & \{A^i | v_t(A^i) = v\} \\ \texttt{OUT}: & \{A^i | v_f(A^i) = v\} \\ \texttt{WAIT}: & \{A^i | v_f(A^i) = v_t(A^i) = v\} \end{cases} \tag{1}$$

If agent $i$ wants to go through $v$, it must perform these three actions $A^i_I \in$ IN, $A^i_W \in$ WAIT and $A^i_O \in$ OUT at $v$ in sequence. Let $\tau(A^i, v)$ denote the time interval during which the transition of $i$ occupies vertex $v$ based on Def. 1. If agent $i$ performs $A^i_I$ at $t$, $\tau(A^i_I, v) = (t, t + \tau(A^i_I))$, $\tau(A^i_W, v) = [t + \tau(A^i_I), t + \tau(A^i_I) + \tau(A^i_W))$ and $\tau(A^i_O, v) = [t + \tau(A^i_I) + \tau(A^i_W), t + \tau(A^i_I) + \tau(A^i_W) + \tau(A^i_O))$. If $i$ does not need to wait at $v$, the duration $\tau(A^i_W)$ is 0 and the wait action $A^i_W$ occupies $v$ only at one time point $t + \tau(A^i_I)$.

Two agent $i$ and $j$ are in conflict if there is a $v$ such that $\tau(A^i, v) \cap \tau(A^j, v) \neq \emptyset$. Let $\langle A^i, A^j, v\rangle$ denote a duration conflict between agents $i$ and $j$ that both occupy the same vertex $v$ during their actions $A^i, A^j$. For two agent $i$ and $j$, we always detect and resolve the earliest conflict between them. We classify all conflicts to be resolved into three types (Fig. 3):

- IN-IN: $\langle A^i, A^j, v\rangle_I$, where $v = v_t(A^i) = v_t(A^j)$;
- OUT-IN: $\langle A^i, A^j, v\rangle_O$, where $v = v_t(A^i) = v_f(A^j)$;
- WAIT-IN: $\langle A^i, A^j, v\rangle_W$, where $v = v_t(A^i) = v_f(A^j) = v_t(A^j)$.

While there are nine possible combinations of two agents' actions {IN,OUT,WAIT} × {IN,OUT,WAIT}, we only need to consider the combinations that involve IN. Since for any other conflicts

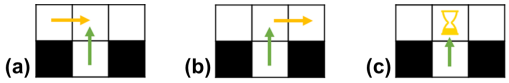


**Figure 3: Three Conflict Types. (a): IN-IN; (b): OUT-IN; (c): WAIT-IN**

{OUT,WAIT} × {OUT,WAIT}, an IN must be involved prior to WAIT or OUT and cause a conflict at the same vertex due to DO. From now on, for any conflict between $i$ and $j$, let $i$ be the agent who performs the IN action, and $j$ may or may not perform IN.

## 4.2 Constraint and Low-level Planner

Safe Interval Path Planning (SIPP) [11] is often used as the low-level planner in CBS. It constructs a search space with states defined by their vertex and safe time interval, resulting in a graph that generally only has a few states per vertex. SIPP is more efficient than A* in the presence of wait durations and finds an optimal path that avoids any unsafe time intervals. We adapt SIPP to MAPF-AA setting by using the following constraints for an agent $i$, let $A^i = ((v_1^i, t_1^i), (v_2^i, t_2^i))$:

- Motion Constraint (MC) $\langle i, u \rightarrow v, [l, r)\rangle_m$: forbids all move actions $A^i$ where $v_1^i = u, v_2^i = v$ and $t_1^i \in [l, r)$;
- Wait Constraint (WC) $\langle i, v, [l, r)\rangle_w$: forbids all wait actions $A^i$ where $v_1^i = v$ and $\tau(A^i, v) \cap [l, r) \neq \emptyset$
- Occupancy Constraint (OC) $\langle i, v, t\rangle_o$: forbids all actions (IN, OUT and WAIT) $A^i$ where $t \in \tau(A^i, v)$;

Fig. 4 illustrates how the constraints affect the search space of the low-level planner.

In the low-level planner of CBS [14], tie-breaking is a useful method to find conflict-free solutions faster. It can find an optimal path for agent $i$ that satisfies the constraints added by high-level and has fewer conflicts with the planned paths of other agents. SIPPS [6] extends SIPP to consider other paths as soft constraints and breaks ties by preferring the path that has fewer soft conflicts (i.e., conflicts with soft constraints). But SIPPS ignores the cases where an agent may encounter multiple soft conflicts if it waits within a safe interval.

To consider these cases, we propose SIPPS with Waiting Conflict (SIPPS-WC) to consider the soft conflicts when waiting. Specifically, a state $s = (v, t, t_h, c_v^w)$ in SIPPS-WC consists of a vertex $v$, an arrival time $t$, an end time of the corresponding safe interval $t_h$, and an integer number $c_v^w$ indicating the number of soft conflicts if waiting at $v$ from $t$ to $t_h$. As shown in Fig. 5, $c_v^w$ can help distinguish between a path that moves from $v$ to $v'$ and then waits at $v'$ and another path that waits at $v$ and then moves to $v'$. To prune states, if two states $s_1$ and $s_2$ have the same $v$, $t_h$ and $c_v^w$, then $s_1$ dominants $s_2$ if the arrival time $s_1.t \leq s_2.t$ and the number of soft conflicts along the path from the start vertex to $v$ $c(s_1) \leq c(s_2)$. Our low-level planner adapts SIPPS-WC to continuous time and the three types of constraints as aforementioned (Fig. 5).

## 4.3 Constraints on Single Action (CSA)

Let $A^i = ((v_1^i, t_1^i), (v_2^i, t_2^i))$ and $A^j = ((v_1^j, t_1^j), (v_2^j, t_2^j))$. In an IN-IN conflict $\langle A^i, A^j, v\rangle_I$, if we permit $j$'s current action, then $i$ cannot

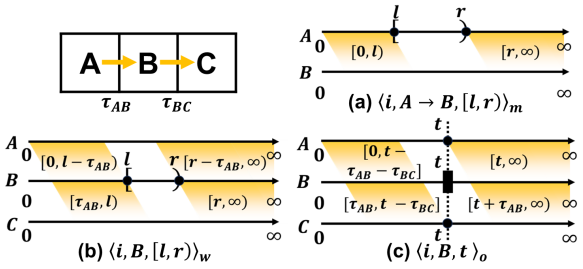


**Figure 4: Changes in the search space of low-level after adding constraints. Duration time from $A$ to $B$ and from $B$ to $C$ are abbreviated as $\tau_{AB}$ and $\tau_{BC}$. (a) MC $\langle i, A \rightarrow B, [l, r)\rangle_m$: before time $l$, the interval at which moving from $A$ to $B$ can be started is $[0, l)$; after time $r$, the interval is $[r, \infty)$. (b) WC $\langle i, B, [l, r)\rangle_w$: before time $l$, the interval at which an IN in $B$ can be started is $[0, l - \tau_{AB})$ and the interval at which an OUT in $B$ can be started is $[\tau_{AB}, l)$; after time $r$, the interval about IN in $B$ is $[r - \tau_{AB}, \infty)$ and the interval about OUT in $B$ is $[r, \infty)$; the safe interval of $B$ is $[0, l)$ and $[r, \infty)$. (c) OC $\langle i, B, t\rangle_o$: before time $t$, the interval at which an IN in $B$ can be started is $[0, t - \tau_{AB} - \tau_{BC}]$ (by Def. 1, $t - \tau_{AB} - \tau_{BC}$ is included) and the interval at which an OUT in $B$ can be started is $[\tau_{AB}, t - \tau_{BC}]$; after time $t$, the interval about IN in $B$ is $[t, \infty)$ and the interval about OUT in $B$ is $[t + \tau_{AB}, \infty)$; the safe interval of $B$ is $[0, t - \tau_{BC})$ and $[t + \tau_{AB}, \infty)$.**

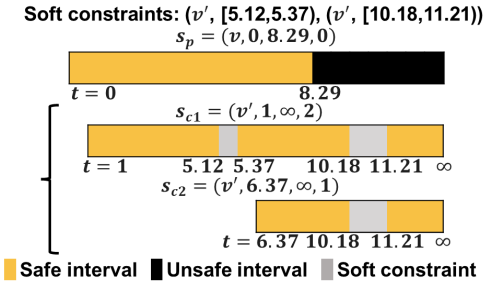


**Figure 5: Expanding states in SIPPS-WC. The parent state $s_p$ at vertex $v$ with safe interval $[0, 8.29)$ can get two child states $s_{c1}$ and $s_{c2}$ at vertex $v'$. State $s_{c1}$ has safe interval $[1, \infty)$ (starts moving at $t = 0$, arrives at $v'$ at $t = 1$) and $s_{c1}.c_v^w = 2$. State $s_{c2}$ has safe interval $[6.37, \infty)$ (waits at $v$, starts moving at $t = 5.37$, arrives at $v'$ at $t = 6.37$) and $s_{c2}.c_v^w = 1$**

perform its action during the time interval $[l^i = t_1^i, r^i = t_2^j)$. Here, setting $l^i = t_1^i$ eliminates the current action of $i$, as the conflict has been detected. Let $r^i = t_2^j$, since $j$ occupies $v$ in $(t_1^j, t_2^j]$ and $i$ occupies $v$ in $(t_2^j, t_2^j + \tau(A^i)]$ (Def. 1). Similar reasoning applies to an OUT-IN conflict. Constraints for both IN-IN and OUT-IN conflicts are denoted as Eq. 2.

In a WAIT-IN conflict $\langle A^i, A^j, v\rangle_W$, we simply add constraints to forbid agents to occupy $v$ at a time point $t_r = \min(t_2^i, t_2^j)$. All

constraints are denoted as Eq. 3.

$$\begin{cases} \langle i, v_1^i \rightarrow v_2^i, [t_1^i, t_2^j) \rangle_m \\ \langle j, v_1^j \rightarrow v_2^j, [t_1^j, t_2^i) \rangle_m \end{cases} \tag{2}$$

$$\begin{cases} \langle i, v, t_r \rangle_o \\ \langle j, v, t_r \rangle_o \end{cases} \tag{3}$$

Remark 1. *The difference between CSA and CCBS lies in the constraints on* WAIT *actions. CCBS adds a constraint to a specific* WAIT *action, while CSA adds a constraint to a vertex $v$, which forbids all actions that occupy $v$. The constraints in CSA avoid the problem about infinitely many wait actions.*

## 4.4 Constraints on Multiple Actions (CMA)

For a specific vertex $v$, $e_1 = (u_1, v) \in E$ and $e_2 = (u_2, v) \in E$, the travel time for agent $i$, $\tau^i(u_1, v)$, can be different from $\tau^i(u_2, v)$. Let $\tau^i_{in}(v)$, $\tau^i_{out}(v)$ denote the minimum travel time of $i$'s IN and OUT at $v$, i.e., $\tau^i_{in}(v) = \min_{e=(u,v)\in E}(\tau^i(u, v))$ and $\tau^i_{out}(v) = \min_{e=(v,u)\in E}(\tau^i(v, u))$. If agent $i$ starts to perform an IN in $v$ at $t$, it occupies $v$ at least $(t, t + \tau^i_{in}(v) + \tau^i_{out}(v))$. If agent $i$ starts to perform a WAIT or OUT in $v$ at $t$, it occupies $v$ at least $(t - \tau^i_{in}(v), t + \tau^i_{out}(v))$. CMA uses such DO to propagate constraints to multiple actions and time intervals.

*Resolve IN(j)-IN(i).* To permit $j$'s action, we add a constraint on $i$ to forbid all IN actions starting within the time interval $[t_1^i, t_1^j + \tau^j_{in} + \tau^j_{out})$, where $t_1^j + \tau^j_{in} + \tau^j_{out}$ represents the earliest time point for $j$ to leave $v$ if starting to perform an IN in $v$ at $t_1^j$. The same strategy is applicable for $i$:

$$\begin{cases} \langle i, * \rightarrow v_2^i, [t_1^i, t_1^j + \tau^j_{in} + \tau^j_{out}) \rangle_m \\ \langle j, * \rightarrow v_2^j, [t_1^j, t_1^i + \tau^i_{in} + \tau^i_{out}) \rangle_m \end{cases} \tag{4}$$

*Resolve OUT(j)-IN(i).* To permit $j$'s action, we add a constraint on $i$ to forbid all IN actions starting within the time interval $[t_1^i, t_1^j + \tau^j_{out})$. To permit $i$'s action, multiple constraints on $j$ are added to forbid all OUT actions starting within the time interval $T = [t_1^j, t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in})$, and forbid WAIT actions that occupy $v$ at any time point in $T$.

$$\langle i, * \rightarrow v, [t_1^i, t_1^j + \tau^j_{out}) \rangle_m \tag{5}$$

$$\begin{aligned} &\langle j, v, [t_1^j, t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}) \rangle_w \quad \cup \\ &\langle j, v \rightarrow *, [t_1^j, t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}) \rangle_m \end{aligned} \tag{6}$$

Here, $[t_1^i, t_1^j + \tau^j_{out})$ allows $i$ to either start the IN earlier than $t_1^i$, or after the earliest time when $j$'s OUT finishes ($\geq t_1^j + \tau^j_{out}$). $t_1^i + \tau^i_{in} + \tau^i_{out}$ is the earliest time point that $i$ finishes an OUT action to leave $v$, and $t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$ is the earliest time point that $j$ finishes the IN to reach $v$, so that $j$ can start to perform a WAIT or OUT afterwards.

*Resolve WAIT(j)-IN(i).* Due to the existence of WAIT action and the variable duration of WAIT action, this case requires extra care on the time interval of constraints. We first show the form of constraints, then we discuss how to properly set the time interval of constraints.

Overall, we add a constraint on $i$ to forbid all IN actions starting within the time interval $T^i = [l^i, r^i)$, or to forbid $j$'s WAIT to occupy $v$ for at any time point in $T^j = [l^j, r^j)$, where $T^i, T^j$ depend on the duration of $j$'s WAIT action.

Similar to Eq. 6, to permit $i$'s action, $t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$ is a critical time point. $t_1^i + \tau^i_{in} + \tau^i_{out}$ is the earliest time point at which agent $i$ leaves the conflict vertex $v$. Then agent $j$ can start the IN action, arriving at $t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$, and performs a WAIT. Therefore, $t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$ is the earliest time point that $j$ can start to perform a WAIT without conflicting with $i$. We set $r^j = t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$. To permit $j$'s action, $t_2^j + \tau^j_{out}$ is a critical time point. $t_2^j + \tau^j_{out}$ is the earliest time point at which agent $j$ leaves the conflict vertex $v$. Therefore, $t_2^j + \tau^j_{out}$ is the earliest time point that $i$ can start to perform a IN without conflicting with $j$. $r^i$ should be equal to $t_2^j + \tau^j_{out}$. As illustrated in Fig. 4, we can set $l^i = t_1^i$ and $l^j \in [t_1^j, t_2^j]$ to make the constraints affect the relevant actions. To resolve conflicts efficiently without eliminating potential solutions and impacting completeness, we set $l^j = t_2^j$.

Now, we have $l^j = t_2^j$, $r^j = t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$, $l^i = t_1^i$ and $r^i = t_2^j + \tau^j_{out}$, where $t_1^i$ is the start time of the IN action and $t_2^j$ is the end time of the WAIT action. However, due to the uncertain duration of WAIT action, the end time point $t_2^j$ may be very large, which can result in $l^j > r^j$ and thus invalidate the interval $T^j$. When $t_2^j$ is large, special care is needed.

If $t_2^j \geq t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$, $j$ performs a long WAIT. The long wait action is divided into two consecutive short WAIT actions. The start and end times of the first short WAIT action are $t_1^{j'} = t_1^j$ and $t_2^{j'} = t_1^i + \tau^i_{in} + \tau^i_{out}$ respectively. The start and end times of the second short WAIT action are $t_1^{j''} = t_1^i + \tau^i_{in} + \tau^i_{out}$ and $t_2^{j''} = t_2^j$ respectively. We first resolve the conflict between $j$'s first short WAIT action and $i$'s IN action, using the above idea. Therefore, we have $l^j = t_2^{j'}$, $r^j = t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}$, $l^i = t_1^i$ and $r^i = t_2^{j'} + \tau^j_{out}$, where $t_1^i$ is the start time of the IN action and $t_2^{j'}$ is the end time of the first short WAIT action.

In summary, there are two cases:

Case 1 (Fig. 6 (a)): $t_2^j < r^j$

$$\begin{cases} \langle i, * \rightarrow v, [t_1^i, t_2^j + \tau^j_{out}) \rangle_m \\ \langle j, v, [t_2^j, t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}) \rangle_w \end{cases} \tag{7}$$

Case 2 (Fig. 6 (b)): $t_2^j \geq r^j$

$$\begin{cases} \langle i, * \rightarrow v, [t_1^i, t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{out}) \rangle_m \\ \langle j, v, [t_1^i + \tau^i_{in} + \tau^i_{out}, t_1^i + \tau^i_{in} + \tau^i_{out} + \tau^j_{in}) \rangle_w \end{cases} \tag{8}$$

Note that conflict resolution in Case 2 needs multiple iterations to permit $j$'s long WAIT action. In the first iteration, after adding the constraint of case 2, the start time of $i$'s IN action is delayed to permit $j$'s first short WAIT action. Then in the second iteration, we have a conflict between $j$'s second short WAIT action and $i$'s delayed IN action. After a finite number of iterations, case 2 will return to case 1, and finally the entire long WAIT action of $j$ is permitted.

Example 2. *Following Ex. 1, Fig. 7 shows the search process of CMA. CMA can eventually find an optimal solution. Four key conflicts need to be resolved. The first conflict is $\langle A^i, A^j, v \rangle_W$, where $i = 3$, $j = 2$, $A^i = ((C, 0), (D, 3))$ and $A^j = ((D, 0), (D, \infty))$. The second*

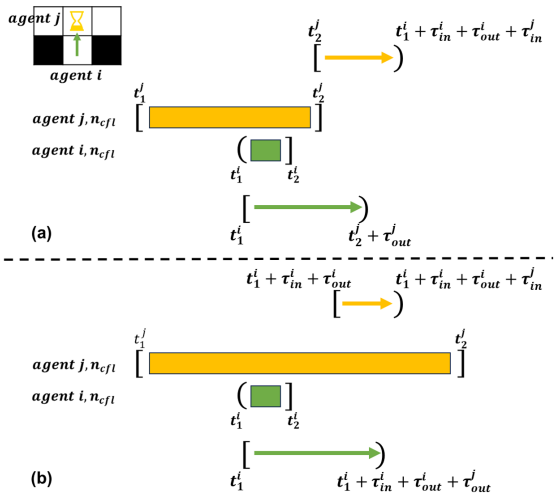


**Figure 6: Resolve WAIT($j$)-IN($i$). Rectangular strips are conflicts. Arrows are corresponding constraints. (a) Case 1: $t_2^j < r^j$. (b) Case 2: $t_2^j \geq r^j$.**

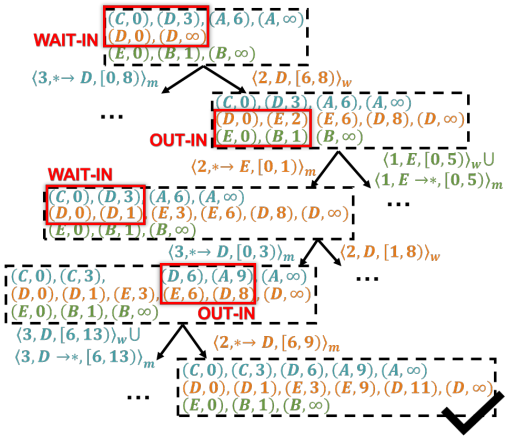


**Figure 7: Search process of CMA for Ex. 1**

*conflict is* $\langle A^i, A^j, v\rangle_O$*, where* $i = 2$*,* $j = 1$*,* $A^i = ((D,0),(E,2))$ *and* $A^j = ((E,0),(B,1))$*. The third conflict is* $\langle A^i, A^j, v\rangle_W$*, where* $i = 3$*,* $j = 2$*,* $A^i = ((C,0),(D,3))$ *and* $A^j = ((D,0),(D,1))$*. The fourth conflict is* $\langle A^i, A^j, v\rangle_O$*, where* $i = 2$*,* $j = 3$*,* $A^i = ((E,6),(D,8))$ *and* $A^j = ((D,6),(A,9))$*.*

## 4.5 Discussion on CMA and CSA

For each constraint generated in CMA and CSA, the end time $r$ of the time interval in that constraint must be no smaller than the start time $l$ of the time interval. Otherwise, the interval and the constraint are invalid. It is easy to verify that $r$ is not smaller than $l$ in the constraints of CSA. But Eq. 4, Eq. 5 and Eq. 6 of CMA may result in $r$ being smaller than $l$ due to the differences in travel time.

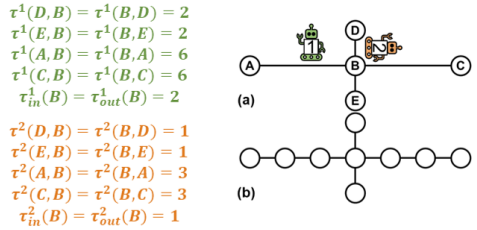


**Figure 8: (a) An example where CMA generates an invalid constraint. (b) By inserting intermediate vertices into the longer edges, the constraint generated by CMA can be valid.**

Example 3. *As shown in Fig. 8 (a), the conflict is* $\langle A^i, A^j, v\rangle_I$*, where* $i = 1$*,* $j = 2$*,* $A^i = ((A,5),(B,11))$ *and* $A^j = ((D,10.5),(B,11.5))$*. According to Eq. 4, the constraint added to agent* $j$ *is* $\langle j, * \rightarrow B, [10.5, 9)\rangle_m$*, which is invalid.*

If the following assumption holds, then $r$ is always greater than $l$ for any generated constraint in CMA.

Assumption 1. *For each agent* $i \in I$*, the duration for* $i$ *to traverse any edge* $e \in E$ *is the same constant real number. This constant can be different for different agents.*

In practice, Assumption 1 can be satisfied by inserting intermediate vertexes into the longer edges (as shown in Fig. 8 (b)) and Keeping agents moving at constant speeds. The grid world used for path planning in an automated warehouse is a common example. By Assumption 1, all edges linked to $v$ have the same travel time for agent $i$. The time intervals in Eq. 4 are changed to $[t_1^i, t_2^j + \tau^j)$ and $[t_1^j, t_2^i + \tau^i)$ where $\tau^i$, $\tau^j$ denote the travel time of agent $i$, $j$. It is easy to verify that $r$ is not smaller than $l$ in these new time intervals. The same is true for Eq. 5 and Eq. 6.

## 4.6 Analysis

In this subsection, we show that CBS-AA is optimal and complete. We begin by showing that the constraints introduced by CBS-AA are mutually disjunctive, a property essential for proving optimality. Next, we demonstrate that CBS-AA always terminates within a finite number of steps. Finally, we prove the optimality and completeness of CBS-AA.

Definition 2. *Two constraints are Mutually Disjunctive (MD) [8] if a set of conflict-free paths cannot simultaneously violate them. In other words, if a set of paths simultaneously violates these two constraints, then it must have a conflict.*

Lemma 1. *Constraints in CSA (Eq.* (2), (3)*) are MD.*

Proof. In Eq (2), the end time $r$ of the time interval added to $i$ ($j$) is $t_t(A^j)$ ($t_t(A^i)$). Therefore, simultaneously violating these two constraints necessarily leads to a conflict. In Eq (3), violating both constraints will obviously result in a conflict at time point $t_r$. □

Assumption 1 allows us to use $\tau^i, \tau^j$ to denote the travel time of agent $i, j \in I$ for any edge in $G$, which greatly simplifies the notation. This section thus uses this assumption to show the ideas in the proofs. The proof can be extended to the general case without

relying on Assumption 1 by using $\tau^i_{in}(v), \tau^i_{out}(v)$. Given $\langle A^i, A^j, v\rangle$, let $A^i = ((v^i_1, t^i_1), (v^i_2, t^i_2))$ and $A^j = ((v^j_1, t^j_1), (v^j_2, t^j_2))$.

Lemma 2. IN-IN *constraints (Eq.* (4)*) in CMA are MD.*

Proof. Assume agent $i$ performs IN at time $x$, occupying $v$ over $\tau^i_x = (x, x + 2\tau^i)$, and agent $j$ performs IN at time $y$, occupying $v$ over $\tau^j_y = (y, y + 2\tau^j)$. If both agents violate the constraint, then $x \in [t^i_1, t^j_2 + \tau^j)$ and $y \in [t^j_1, t^i_2 + \tau^i)$. We show the intervals must intersect. By contradiction, suppose they do not. (i) If $x \geq y + 2\tau^j$, then $x < t^j_2 + \tau^j$ while $y + 2\tau^j \geq t^j_1 + 2\tau^j = t^j_2 + \tau^j$, a contradiction. (ii) If $y \geq x + 2\tau^i$, then $y < t^i_2 + \tau^i$ while $x + 2\tau^i \geq t^i_1 + 2\tau^i = t^i_2 + \tau^i$, a contradiction. Hence, the intervals intersect, implying a conflict. Without Assumption 1, replacing $2\tau$ by $\tau_{in} + \tau_{out}$ yields the same conclusion. Therefore, constraints in Eq. (4) are MD. □

With the same idea, it can be proved that OUT-IN constraints (Eq. (5), (6)) and WAIT-IN constraints (Eq. (7) and (8)) are MD.

Lemma 3. *CBS-AA can terminate within a finite number of steps on a solvable MAPF-AA problem.*

Proof. Due to space limits, we sketch the main idea, similar to the one in [3]. If the method does not terminate, there exists an infinite sequence of high-level nodes whose costs are all below the optimal solution and all contain conflicts.

Let a trajectory $\sigma$ be a finite sequence of move actions with cost $g(\sigma)$ equal to the sum of their durations. If a path $\pi$ contains all actions of $\sigma$ in order, we write $\pi \sim \sigma$. For any finite $c \in \mathbb{R}$, the set $\{\sigma \mid g(\sigma) < c\}$ is finite. Hence, infinitely many constraints must be added to paths mapping to a specific trajectory $\sigma^\infty$.

Non-termination requires that, despite infinitely many constraints, there still exists a path $\pi' \sim \sigma^\infty$ satisfying all constraints with $g(\pi') < c$. However, each constraint introduced by CBS-AA reduces the feasible execution time of actions in $\sigma^\infty$ by a non-zero amount. After finitely many branchings, no such $\pi'$ can exist. Therefore, CBS-AA must terminate in finite steps. □

Theorem 1. *CBS-AA is optimal and solution complete.*

Proof. Because all constraints added by CBS-AA are MD (Lem. 1 and Lem. 2), all solutions are reachable from the root of search tree. And CBS-AA always expands the high-level node with minimum objective value. Thus, CBS-AA is optimal. If a feasible solution exists, CBS-AA is solution complete because it can terminate within a finite number of steps (Lem. 3). □

# 5 EXPERIMENTAL RESULTS

In the previous section, we proposed CSA to correct the errors of CCBS, and further proposed CMA to resolve conflicts efficiently. In order to find conflict-free solutions faster, we proposed a new low-level planner, SIPPS-WC. In this section, we compare our CSA, CMA and CMAS (CMA with SIPPS-WC) with CCBS [1], to show the gradual improvement of the methods. Among them, CSA and CMA use SIPP at the low-level, CMAS uses SIPPS-WC at the low-level. Besides, to verify the optimality, we compare CMAS with LS-M* [12].

We use four maps of different sizes from an online data set [15]: "empty-32-32", "random-32-32-20", "den312d" and "warehouse-10-20-10-2-2". We test the algorithms by varying the number of agents $N$ from 10 to 50.For testing purposes, we consider that the edges in each map have fixed-length edges of a unit, and each agent has a random speed between 1 and 20. In other words, it takes the same amount of time to traverse any edge for the same agent, and it takes different amount of time to traverse the same edge for different agents. When agents reach their goals, they will stay there permanently. The runtime limit of each instance is 30 seconds. We implement all algorithms in C++ and run all tests on a computer with an Intel Core i7-11800H CPU.

## 5.1 Comparison with CCBS

*Success Rates.* Fig. 9 (a)-(d) show the success rates of the methods. We observe that CSA has a slightly higher success rate than CCBS, mainly due to the changes in the wait action. Different from CCBS, the constraint added by CSA for wait actions (Fig. 4 (c)) may affect all of IN, OUT, WAIT actions, and therefore resolve conflicts more efficiently and improve the success rates a bit. Since CCBS's constraints on wait actions are not mutually disjunctive, it is possible that CCBS overlooks feasible solutions and fails within the time limit, resulting in a lower success rate. Additionally, the success rates of CMA are obviously higher than CSA and CCBS, especially when the number of agents increases. The reason is that, CMA add constraints that affect multiple actions and add constraints at a time interval for WAIT actions, which makes CMA resolve conflicts more efficiently than CSA and CCBS. Finally, SIPPS-WC can help find an optimal path and avoid collisions with other agents as much as possible, especially for cases involving waiting at a safe interval. With the help of SIPPS-WC, the success rates of CMAS are further improved and can handle up to 50 agents in the warehouse map, which is otherwise hard to achieve by the other three algorithms.

*Number of Expansions.* Fig. 9 (e)-(f) shows the number of high-level nodes expanded. CMA has the least number of expansions, which means it can resolve more conflicts with one branching, which reduces the number of expansions to find a solution. Compared with CSA, CMA reduces the number of expansions by up to 90%: When the number of agents is 25 in the empty map, the average number of expansions in CSA is 8286 while the one for CMA is 617. By considering paths of other agents in the low-level planner, CMAS can find an optimal solution with even fewer expansions, which thus enhances the success rates.

*Runtime.* We further extend the runtime limit to 120 seconds and analyze the runtime of different algorithms in empty $32 \times 32$. As shown in Fig. 10, when the number of agents $N \geq 25$, the success rates of CCBS and CSA are low, while the success rate of CMA drops rapidly. However, the success rate of CMAS remains above 60%. When the number of agents $N = 30$, CMAS can find solutions in 68% of instances within 60 seconds.

*SIPP v.s. SIPPS-WC.* We compare the average runtime per call of SIPP in CMA with that of SIPPS-WC in CMAS in empty $32 \times 32$. As shown in Table. 1, the runtime of SIPPS-WC increases with the number of agents. The increase in the number of agents leads to an increase in the number of soft constraints contained in SIPPS-WC.

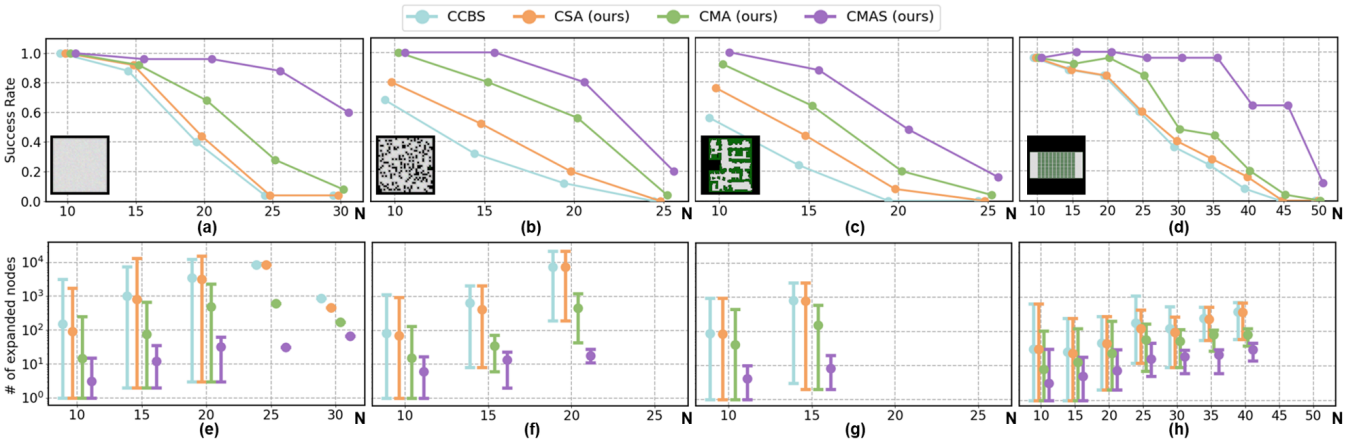


**Figure 9: (a)-(d): Success rates of the algorithms. $N$ is the number of agents. (e)-(h): Min., Avg., and Max. number of high-level nodes expanded by the algorithms, which only counts the cases where all four methods succeed. $N$ is the number of agents.**

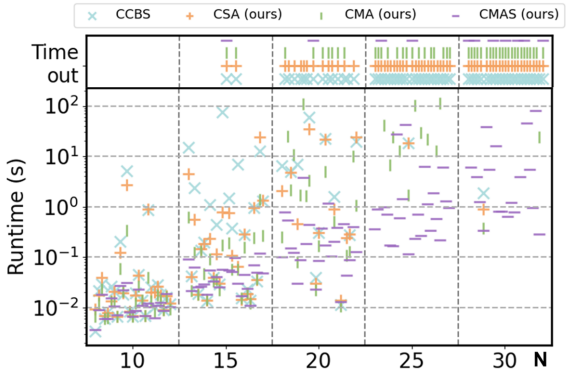


**Figure 10: Runtime in empty $32 \times 32$. $N$ is the number of agents.**

**Table 1: Comparison of SIPP with SIPPS-WC in terms of average runtime per call (ms).**

| N | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|
| SIPP | 0.803 | 0.930 | 0.902 | 0.999 | 1.03 |
| SIPPS-WC | 0.845 | 1.20 | 1.37 | 1.62 | 1.95 |

Multiple child nodes can be generated for the same safe interval (depending on the number of soft constraints involved), thus increasing the number of nodes to explore. However, due to the consideration of other agents' paths, CMAS can find conflict-free solutions faster.

## 5.2 Comparison with LS-M*

LS-M* [12] is an A*-based MAPF planner and can find an optimal solution in MAPF-AA. We compare CMAS with it in order to verify the optimality of CMAS. The experimental setup is the same as above. We fix the map to random-32-32-20 and the number of agents is $N = 2, 4, 6, 8$. And considering that LS-M* needs a long time to search for an optimal solution, we add an additional experiment to extend the runtime limit to 120 seconds. As shown in Table. 2, when the number of agents increases to more than 4, the success rate of LS-M* begins to decline. When the number of agents reaches 8, the success rate of LS-M* is 0%. However, CMAS can always maintain a 100% success rate, and handle more agents than LS-M*. The cost of the solution found by CMAS is the same as LS-M*, while CMAS can find the solution in less run time. The results show that CMAS can find the same optimal solution as LS-M* in a shorter time.

**Table 2: Comparison of CMAS with LS-M* in terms of Success Rate (SR) and Avg. Run Time (RT) under different Time Limit (TL). The data are shown in format (CMAS/ LS-M* under $TL = 30s$ / LS-M* under $TL = 120s$).**

| N | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| SR | 1.0/1.0/1.0 | 1.0/0.96/1.0 | 1.0/0.2/0.4 | 1.0/0.0/0.0 |
| RT (s) | 0.009/0.895/0.895 | 0.002/1.52/4.44 | 0.082/19.0/49.6 | -/- |

## 6 CONCLUSION AND FUTURE WORK

This paper focuses on MAPF-AA, and develops a new exact algorithm CBS-AA for MAPF-AA with solution optimality guarantees, based on the popular CBS framework. CBS-AA introduces new conflict resolution techniques for agents with asynchronous actions, which improves the runtime efficiency of the algorithm. Experimental results demonstrate the advantages of our new approaches in different settings against several baseline methods.

For future work, one can also consider speed and uncertainty in MAPF-AA or combine MAPF-AA with target allocation and sequencing [13, 18].

## ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of China under Grant 62403313, and the Natural Science Foundation of Shanghai under Grant 24ZR1435900.

# REFERENCES


[1] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. 2022. Multi-agent pathfinding with continuous time. *Artificial Intelligence* 305 (2022), 103662. https://doi.org/10.1016/j.artint.2022.103662

[2] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the international symposium on combinatorial Search*, Vol. 5. 19–27.

[3] Alvin Combrink, Sabino Francesco Roselli, and Martin Fabian. 2025. Sound and Solution-Complete CCBS. *arXiv preprint arXiv:2508.16410* (2025).

[4] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. 2013. Push and rotate: cooperative multi-agent path planning. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. 87–94.

[5] Andy Li, Zhe Chen, Danial Harabor, and Mor Vered. 2025. Revisiting Conflict Based Search with Continuous-Time. *arXiv preprint arXiv:2501.07744* (2025).

[6] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. 2022. MAPF-LNS2: Fast repairing for multi-agent path finding via large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 10256–10265.

[7] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. 2021. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 12353–12362.

[8] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, TK Satish Kumar, and Sven Koenig. 2019. Multi-agent path finding for large agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7627–7634.

[9] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. 2022. Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding. *Artificial Intelligence* (2022), 103752. https://doi.org/10.1016/j.artint.2022.103752

[10] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. 2021. Time-Independent Planning for Multiple Moving Agents. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 13 (May 2021), 11299–11307.

[11] Mike Phillips and Maxim Likhachev. 2011. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, 5628–5635.

[12] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. 2021. Loosely synchronized search for multi-agent path finding with asynchronous actions. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 9714–9719.

[13] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. 2023. CBSS: A New Approach for Multiagent Combinatorial Path Finding. *IEEE Transactions on Robotics* 39, 4 (2023), 2669–2683. https://doi.org/10.1109/TRO.2023.3266993

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

[15] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 10. 151–158.

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

[17] Thayne T Walker, Nathan R Sturtevant, and Ariel Felner. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains.. In *IJCAI*. 534–540.

[18] Xuemian Wu and Zhongqiang Ren. 2025. Multi-Agent Combinatorial Path Finding for Tractor-Trailers in Occupancy Grids. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 14124–14131. https://doi.org/10.1109/IROS60139.2025.11246219

[19] Jingjin Yu and Steven LaValle. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 27. 1443–1449.

[20] Shuai Zhou, Shizhe Zhao, and Zhongqiang Ren. 2025. Loosely Synchronized Rule-Based Planning for Multi-Agent Path Finding with Asynchronous Actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 14763–14770.

[21] Shuai Zhou, Shizhe Zhao, and Zhongqiang Ren. 2025. LSRP*: Scalable and Anytime Planning for Multi-Agent Path Finding with Asynchronous Actions (Extended Abstract). In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 18. 275–276. https://doi.org/10.1609/socs.v18i1.36016