Beyond Pairwise Reasoning in Multi-Agent Path Finding

Bojie Shen[1], **Zhe Chen**[1], **Jiaoyang Li**[2], **Muhammad Aamir Cheema**[1], **Daniel D. Harabor**[1]
Peter J. Stuckey[1]

[1] Faculty of Information Technology, Monash University, Australia
[2] Robotics Institute, Carnegie Mellon University, USA
{bojie.shen, zhe.chen, aamir.cheema, daniel.harabor, peter.stuckey}@monash.edu, jiaoyangli@cmu.edu

Abstract

In Multi-Agent Path Finding (MAPF), we are asked to plan collision-free paths for teams of moving agents. Among the leading methods for optimal MAPF is Conflict-Based Search (CBS), an algorithmic family which has received intense attention in recent years and for which large advancements in efficiency and effectiveness have been reported. Yet all of the recent CBS gains come from reasoning over pairs of agents only. In this paper, we show how to further improve CBS by reasoning about more than two agents at the same time. Our new *cluster reasoning* techniques allow us to generate stronger bounds for CBS and to identify more bypasses (alternative cost-equivalent paths), which reduce the number of nodes in the CBS conflict tree.

Introduction

Multi-Agent Path Finding (MAPF) is a combinatorial problem that asks us to find coordinated and collision-free plans for a team of moving agents. It is NP-Hard to find optimal MAPF solutions (Yu and LaValle 2013) under a variety of objective functions, such as sum-of-(individual)-costs. Yet effective solutions to MAPF problems are necessary for a variety of real-world applications, such as automated warehousing (Wurman, D'Andrea, and Mountz 2008), drone swarm coordination (Hönig et al. 2018), and team navigation in computer games (Silver 2005).

Among the state-of-the-art optimal algorithms for MAPF, Conflict-Based Search (CBS) (Sharon et al. 2012) is a best-first search algorithm that routes each agent independently and then resolves conflicts afterwards. In recent years, there have been massive advancements in the efficiency and scalability of CBS. These gains have been achieved by: (a) taking into account symmetries that result in the conflicts between two agents (Li et al. 2021; Zhang et al. 2022); (b) generating complex admissible heuristics (Felner et al. 2018; Li et al. 2019); and (c) introducing *bypasses* (Boyarski et al. 2015a) to reduce the number of subproblems that CBS must tackle. Yet, even on modest size problems with dozens of agents, CBS timeout failures are not uncommon.

Thus far, CBS only reasons about incompatibility between at most two agents at a time. It lazily detects conflicts between pairs of agents, resolves those conflicts by adding pairwise constraints, and generates heuristics by combining information about the interactions of pairs of agents. Other optimal MAPF algorithms, such as ICTS (Sharon et al. 2013), can detect incompatibility between more than two agents. But ICTS has limited scalability compared to CBS because of its large computational overheads. Recently, Mogali, van Hoeve, and Smith (2020) proposed three-agent heuristics for improving the performance of CBS. But their approach is limited to reasoning only at the root node. In this paper, we extend CBS heuristics, applicable at every node, to more than two agents. We do this by exploiting *mutex propagation* (Zhang et al. 2022), a successful pairwise reasoning technique, which we extend to clusters of more than two agents. We derive stronger bounds for CBS and also generate new kinds of *bypasses*, where the assigned paths of some agents are replaced to reduce the number of conflicts. Experiments show substantial improvements for CBS, especially on dense maps.

Preliminaries

Following Stern et al. (2019), a MAPF instance consists of (i) an input grid map, where each cell connects to only orthogonal neighbours, and (ii) a set of k agents $A = \{a_1, \ldots, a_k\}$. We represent the grid map as an undirected and unweighted graph $G = (V, E)$ with nodes V and edges $E \subseteq V \times V$. Each agent $a_i \in A$ has a unique source ($s_i \in V$) and goal ($g_i \in V$). Time is discretised into unit-sized timesteps, and, at each timestep, agents are allowed to move to an adjacent vertex or else wait at their current location. A *path* of agent a_i is a sequence of vertices $p = \langle s_i, \ldots, g_i \rangle$, indicating the location of a_i at each timestep. An agent has *reached* its goal if it permanently waits at its goal location and never has to move off to make way for another agent. The *cost* of a path p is the number of timesteps (i.e., $|p| - 1$) required for an agent to reach the goal location from its source (ignoring wait-costs after reaching). The paths of two agents a_i and a_j can conflict in two ways: (i) a vertex conflict $\langle a_i, a_j, v, t \rangle$ when agent a_i and a_j reach the same vertex $v \in V$ at the same timestep t, and (ii) an edge conflict $\langle a_i, a_j, u, v, t \rangle$ when two agents a_i and a_j traverse the same edge $(u, v) \in E$ from the opposite directions at the same timestep t. A solution is a set of conflict-free paths, one for each agent. Our objective is to find an optimal solution that minimises the *sum of the indi-*

vidual costs (SIC) of the paths.

Conflict-Based Search (CBS)

Conflict-Based Search (CBS) (Sharon et al. 2012) is a state-of-the-art optimal algorithm for solving MAPF. CBS runs a two-level search. The high level of CBS focuses on a pair of agents that have at least one conflict with each other and resolves the conflict by adding constraints. This process involves building a binary tree called Constraint Tree (CT). Each high-level CBS node N is a CT node, which contains:

- a set of constraints $N.constraints$, in which each constraint $\langle a_i, v, t\rangle$ (resp. $\langle a_i, u, v, t\rangle$) prohibits agent a_i from visiting vertex v (resp. edge (u, v)) at timestep t;
- a set of paths $N.\mathcal{P}$ (one for each agent), in which each path $N.\mathcal{P}(a_i)$ is a cost-minimal path for agent a_i that satisfies $N.constraints$ without considering other agents;
- a set of conflicts $N.conflicts$, where each conflict is either a vertex $(\langle a_i, a_j, v, t\rangle)$ or an edge conflict $(\langle a_i, a_j, u, v, t\rangle)$ between $N.\mathcal{P}(a_i)$ and $N.\mathcal{P}(a_j)$; and
- a cost $N.cost$, which is the SIC of $N.\mathcal{P}$.

To find a conflict-free solution that minimises the SIC, CBS searches in a best-first-search manner and maintains a queue to prioritise the CT nodes using their costs $N.cost$. Initially, the priority queue contains a root CT node with an empty set of constraints, and each path $p \in N.\mathcal{P}$ is an optimal path while ignoring other agents. Whenever CBS expands a CT node N, it selects a conflict between a_i and a_j from $N.conflicts$ and resolves it by splitting N into two child CT nodes. In each of the child CT nodes, CBS adds an additional constraint that prohibits one of the agents from visiting the contested vertex or edge at timestep t. Since the path of a_i (or a_j) no longer satisfies the constraints of the child CT node, CBS calls a low-level solver to replan the path by using a time-space A* search (Silver 2005). Once replanned, the conflicts of the child CT node are updated, and all other paths in $\mathcal{P}$ remain the same. The search continues by inserting the child CT nodes to the queue and terminates when it expands a CT node N that has no conflicts (i.e., $N.conflicts$ $= \emptyset$). The current $N.\mathcal{P}$ is a cost-minimal solution as CBS guarantees to explore both ways of resolving each conflict.

Bypassing Conflicts: Boyarski et al. (2015b) improved the search to *bypass conflicts* by modifying the path of one of the agents involved in the chosen conflict. Given a CT node N and its constraints $N.constraints$, a path p_i is a valid bypass for agent a_i, iff (i) p_i has the same source and goal of a_i; (ii) p_i is a cost-equivalent path of $N.\mathcal{P}(a_i)$ which satisfies $N.constraints$; and (iii) replacing $N.\mathcal{P}(a_i)$ with p_i reduces the total number of conflicts (i.e.,$|N.conflicts|$) of N. CBS finds bypasses when generating child CT nodes. Recall that CBS selects a conflict between two agents a_i and a_j, and each child CT node replans a path p_i (resp. p_j) for agent a_i (resp. a_j) to resolve the conflict. If the replanned path p_i (or p_j) is a valid bypass, we replace the path of a_i (i.e., $N.\mathcal{P}(a_i)$) with p_i and remove the generated child CT nodes without splitting N. Identifying a bypass can resolve a conflict without branching, which reduces the size of CT.

High-Level Heuristic (WDG): So far, CBS prioritises the CT nodes using $N.cost$. However, like many other A* searches, the performance of CBS can be significantly improved by using an admissible heuristic h, which prioritises CT nodes based on $f = N.cost + h$. The first high-level heuristic of CBS is introduced by Felner et al. (2018) which focused on the pairs of agents with cardinal conflicts (i.e., resolving such conflicts must increase the costs of child CT nodes). Later, Li et al. (2019) improved and extended this heuristic considering all pairs of agents that are in conflict. Among many heuristics proposed in (Li et al. 2019), we explain the leading heuristic, Weighted Pairwise Dependence Graph (WDG) heuristic.

In order to compute the heuristic for a CT node N, WDG considers all pairs of agents that are currently in conflict. For each such pair of agents (a_i, a_j), WDG takes the paths and constraints of a_i and a_j in a CT node N and runs a sub-CBS search to solve them as a sub-instance. Completely solving the sub-instance may be costly and easily dominates runtime. Therefore, each sub-CBS solver is set to a node limit $\mathcal{L}$, which only allows the solver to expand at most $\mathcal{L}$ CT nodes. When the sub-CBS search concludes, it is easy to see that the increase of the minimal f-value in the open list Δ_{ij} is a valid lower bound for agent pairs (a_i, a_j). To further consider the intersection of pairs of agents, WDG builds a weighted pairwise dependency graph $G_D = (V_D, E_D, W_D)$ for these agent pairs whose $\Delta_{ij} > 0$. Each vertex $v_i \in V_D$ indexes an agent a_i, each edge $(v_i, v_j) \in E_D$ corresponds to an agent pair (a_i, a_j), and $W_D : E \to \mathcal{D}$ is a weight function that maps each edge $(v_i, v_j) \in E_D$ to $\Delta_{ij} \in \mathcal{D}$ as edge weight. The graph G_D is used to create an integer program to minimise $\sum_i x_i$ subject to $\wedge_{ij} x_i + x_j \geq \Delta_{ij}$, where each x_i represents the increase in length of the current path for agent a_i. The optimal value of this integer program is an admissible heuristic for CT node N. Although computing WDG requires building G_D for each node expanded, most of the edges in G_D can be inherited from the parent CT node.

Though fast and effective, WDG as well as other existing heuristics (Felner et al. 2018; Boyarski et al. 2021) (except for the one introduced below) compute the heuristics only by considering the pairs of agents that are in conflict. Recently, Mogali, van Hoeve, and Smith (2020) proposed a Lagrangian-Relax-and-cut-based (LR) heuristic that reasons about conflicts among groups of three agents. It shows promise that reasoning beyond pairs of agents can generate better heuristics. However, due to the large runtime overhead of the LR heuristic, (i) they apply the LR heuristic only at the root CT node, (ii) they have to limit the maximum cost of the paths, and (iii) the overall speedup is very limited (e.g., within the given runtime limit, they do not solve more instances than the existing algorithm).

MDDs and Mutex Propagation

A Multi-value Decision Diagram (MDD) (Sharon et al. 2013) MDD_i for an agent a_i in a CT node N is a Directed Acyclic Graph (DAG) which compactly stores all cost-minimal paths that satisfy the constraints $N.constraints$. Let us assume the cost of $N.\mathcal{P}(a_i)$ is l_i, the MDD_i has $l_i + 1$ levels. For each level t, MDD_i contains nodes that correspond to all possible locations of agent a_i at timestep t when agent a_i follows a path of cost l_i that satisfies $N.constraints$. The source and goal nodes are single nodes, indicating that agent a_i occupies the vertex s_i and g_i at timestep 0 and l_i, respectively. Building MDD_i for an agent a_i is simple. We run a breadth-first search from the source s_i to explore the nodes that satisfy the constraints $N.constraints$ within cost l_i. Once the search is finished, MDD_i only records the partial DAG that reaches goal g_i. This auxiliary data structure has been widely used to improve the CBS search, e.g., prioritising conflicts (Boyarski et al. 2015b) and symmetry reasoning (Li et al. 2021).

Example 1. *Figure 1b shows MDDs of three agents, where we assume the constraints $N.constraints = \emptyset$. MDD_1, MDD_2 and MDD_3 correspond to the cost-minimal paths for each agent, with costs of 4, 6 and 4, respectively.*

Mutex[1] propagation is a popular technique used in AI planning, such as planning graph (Blum and Furst 1997), state-space planner (Nguyen and Kambhampati 2000), and improving SAT-based planner (Kautz and Selman 1996). Recently, Zhang et al. (2022); Surynek et al. (2020) extended mutex propagation to identify and resolve pairwise symmetries for MAPF problems. Like many constraint propagation techniques, mutex propagation finds incompatible nodes between the MDDs of two agents. Given MDDs for two agents, mutex propagation finds two types of mutexes:

- Initial mutexes: a pair of MDD nodes/edges is an initial mutex iff these two MDD nodes/edges correspond to a vertex/edge conflict at the same level t.
- Propagated mutexes: a pair of MDD nodes (resp. edges) is a propagated mutex iff they are at the same level t and all pairs of their parent MDD edges (resp. nodes) are either initial mutex or propagated mutex.

A pair of MDD nodes is *mutex* if either initial or propagated mutex. In general, the initial mutexes are detected first and then propagated through MDD to find the propagated mutexes. Many existing algorithms (Mackworth 1977; Zhang et al. 2022) can detect mutexes between MDDs. We omit the details of such algorithms.

Property 1. *Iff two nodes from different MDDs at the same level are mutex, there exists no pair of conflict-free paths that traverse through the two nodes and reach their goal locations on their individual minimum cost (Zhang et al. 2022).*

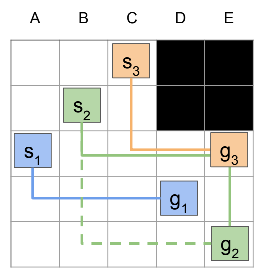
Example 2. *Figure 1b shows an example of mutex propagation of MDD_2 with MDD_1 and MDD_3.*

Our Approach

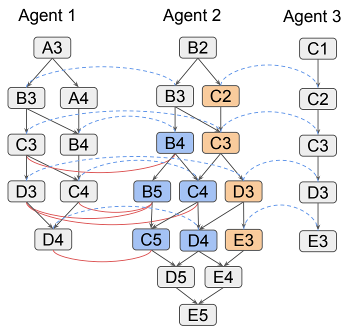
While the best heuristic for CBS is quite sophisticated, it only ever reasons about the interactions of pairs of agents. In this work, we detect and make use of interactions between three or more agents to improve heuristics and find bypasses.

Definition 1 (Conflict Cluster)**.** *Given a CT node N, a conflict cluster C is a set of agents such that, considering every agent $a \in C$ with a set of cost-minimal paths that satisfy $N.constraints$, there exist no conflict-free assignments of paths for these agents.*

[1]Mutex is a short term for mutual exclusion.



(a) MAPF problem.



(b) MDDs and Mutex Propagation.

Figure 1: (a) A MAPF instance with three agents. (b) Examples of MDDs for three agents and the results of mutex propagation between agent a_2 with agent a_1 and a_3. The initial and propagated mutexes are shown in dashed blue arcs and solid red arcs, respectively. The incompatible nodes between a_2 and a_1 (resp. a_3) are coloured in blue (resp. orange). All paths of a_2 have incompatible nodes and thus must collide with either a_1 or a_3.

Example 3. *Figure 1 shows a conflict cluster containing three agents, where the current paths (shown in solid lines) of a_2 and a_3 collide at $C3$. Although switching a_2 to another cost-minimal path (e.g., the path shown in the dashed line) avoids the conflict with a_3, it conflicts with another agent a_1. In fact, there exists at least one conflict between two or more agents no matter what cost-minimal paths the agents choose. Thus, the optimal solution requires at least one agent to wait for at least 1 timestep.*

The critically important feature of conflict clusters is that, if a CT node N has a conflict cluster C, then the SIC of any collision-free paths that satisfies $N.constraints$ is guaranteed to be larger than the cost of N because some pair of agents in C must conflict, i.e., the cost must increase by at least 1. However, the WDG heuristic fails to capture this case, since conflict-free paths exist for any pair of agents while ignoring the other agents.

Computing Heuristics and Bypasses

Our approach iteratively detects the conflict clusters for a CT node N. Whenever our algorithm finds a conflict cluster, we increment the heuristic value by one and exclude the agents in this cluster to ensure the clusters detected are independent of each other. As a byproduct, our approach may also explore a bypass, a cost-equivalent path that satisfies the constraints $N.constraints$ and reduces the total number of conflicts $N.conflicts$. Here, we explain the high-level idea of computing an admissible heuristic by integrating the best pairwise heuristic WDG and the heuristic value of conflict clusters found, as well as adapting the bypasses based on the CBS framework. The details of detecting conflict clusters and bypasses will be explained later.

Algorithm 1: Compute Heuristic and Bypass

```
Input: $N$: a current CT node of CBS.
Output: Heuristic value $h$ for a CT node.
1  $h_p, EA_p \leftarrow$ computeWDGHeuristic($N$);
2  $h_c, EA_c \leftarrow$ inheritClusterFromParent($N$);
3  $EA \leftarrow EA_p \cup EA_c$;
4  $SG \leftarrow$ buildConflictStateGraph($N$);
5  while $a_m \leftarrow$ getMaxConflictAgent($SG, EA$) do
6      $R \leftarrow$ findClusterOrBypass($a_m, EA, N$);
7      if $R \equiv$ ConflictCluster(C) then
8          appendCluster($N, C$);
9          $EA \leftarrow EA \cup C$;
10         $h_c$ ++;
11     else if $R \equiv$ Bypass($p_m$) then
12         updatePathAndConflict($N, a_m, p_m$);
13         updateConflictStateGraph($N, SG$);
14 return $h \leftarrow h_p + h_c$;
```

The pseudo-code of our approach for computing an improved heuristic or bypass is shown in Algorithm 1. Similar to CBS, given a CT node N, our algorithm first computes the WDG heuristic following (Li et al. 2019) and returns the pairwise heuristic value h_p and a set of agents EA_p considered in WDG (i.e., agents in the dependency graph G_D) (line 1). Our algorithm uses $N.clusters$ to store a set of conflict clusters detected in N. Before detecting the new conflict clusters, our algorithm inherits the conflict clusters EA_c from the parent CT node of N and their heuristic value h_c (explained later in the section) in order to avoid recomputation (line 2). Both EA_p and EA_c are appended to a set EA, which maintains the set of excluded agents (line 3). This is to ensure that the clusters detected are independent of each other and with the agents used in the WDG heuristic.

The algorithm then begins to compute our cluster heuristic and bypass by building a conflict state graph SG (line 4). This graph is a simple undirected graph that maintains an edge between every pair of conflicting agents in N. We use this graph to efficiently track the conflicts in the current plan $N.\mathcal{P}$. The algorithm then calls getMaxConflictAgent which iteratively accesses the agents in SG that have not been excluded and returns the agent a_m (line 5) that has the maximum number of conflicts with other non-excluded agents a_i (i.e., $a_i \notin EA$). We choose the agent a_m with the maximum number of conflicts because such an agent is more likely to find a smaller conflict cluster, thus potentially leading to a better heuristic value. The function getMaxConflictAgent does not consider an agent that was returned earlier and returns null when all agents are either excluded or were returned earlier (in which case the while loop terminates). The algorithm then calls the function findClusterOrBypass (to be detailed later) which returns either a detected conflict cluster involving a_m or a bypass for agent a_m (line 6). Based on the returned result R, the algorithm proceeds as follows.

- If R is a conflict cluster (line 7-10), the algorithm appends the conflict cluster C to $N.clusters$. All agents in C are marked as excluded agents, and the cluster heuristic h_c is increased by one, because resolving the conflict in a cluster must increase SIC by at least one (see Definition 1 and the following example).
- If R is a bypass (line 11-13), the algorithm takes the bypass path p_m and updates N by changing the path of a_m to p_m. The conflicts of the old path are also removed and replaced with new conflicts of p_m. The conflict state graph SG is also updated accordingly.

When the while loop terminates, the algorithm returns the heuristic value (i.e., $h_p + h_c$). Note that our algorithm could work without applying WDG heuristic, h_p. However, WDG is a relatively cheap yet effective heuristic and helps improve the performance overall.

Inherit Clusters from Parent Node Since CBS only constrains a portion of agents (mostly one agent only) when generating child CT nodes, the incompatibility among other agents, excluding constrained agents, is not changed. Hence, we inherit the information (similar to the WDG heuristic) to avoid recomputation. To inherit a conflict cluster C from parent CT node Pr (at line 2), we need to ensure two conditions: (i) the path cost of every agent $a_i \in C$ of the current CT node N and its parent CT node are exactly the same (i.e., $|N.\mathcal{P}(a_i)| = |Pr.\mathcal{P}(a_i)|$) and (ii) every agent $a_i \in C$ is a non-excluded agent (i.e., $a_i \notin EA$). To ensure (i) and (ii), we iteratively scan through $Pr.clusters$ and filter out the clusters if $|N.\mathcal{P}(a_i)| \neq |Pr.\mathcal{P}(a_i)|$ or $a_i \in EA$. For each inherited cluster, we mark these agents as excluded and increase the cluster heuristic h_c by one (line 2).

Theorem 1. *Given a CT node N, the heuristic $h = h_p + h_c$ computed by Algorithm 1 is admissible.*

Proof. The pairwise heuristic h_p is computed by considering a subset of agents $A_p \subseteq A$, and h_p is an admissible heuristic of CT node N as shown by Li et al. (2019). Algorithm 1 excludes these agents and computes the cluster heuristic h_c by detecting the conflict clusters from agents $A_c = A \setminus A_p$, thus h_p and h_c are disjoint. By Definition 1, each conflict cluster must increase the cost of CT node N by at least one. Thus, h_c is also admissible as each conflict cluster $C \in A_c$ detected is independent of other clusters. Therefore, $h = h_p + h_c$ is an admissible heuristic. □

Finding Conflict Cluster or Bypass

To find a conflict cluster or bypass for an agent a_m, one can incrementally join the MDD of a_m with other non-excluded agent a_c (i.e., $a_c \notin EA$) and remove MDD nodes if there is a pair of agents in conflict. We find a conflict cluster if the joint MDD contains no feasible paths for each agent to reach its goal. Alternatively, we may explore a bypass of a_m from these feasible paths in the joint MDD. However, this naive approach has two drawbacks: (i) joining the MDDs exponentially increases the size of the joint MDD; and (ii) exhaustively checking all non-excluded agents may be time-consuming. In this work, we consider a more sophisticated algorithm to identify the cluster and bypass using mutex propagation. The key idea is to find incompatible nodes between a pair of MDDs.

Algorithm 2: Find Cluster or Bypass

```
Input: $a_m$: a selected agent, $EA$: excluded agents, $N$: a
       current CT node.
Output: a conflict cluster $C$ or a bypass $p_m$ for agent $a_m$
Initialisation: $C \leftarrow \{a_m\}$; $PA \leftarrow \emptyset$
1  $p_m \leftarrow$ GETPATH$(a_m, N)$;
2  $MDD_m \leftarrow$ GETMDD$(a_m, N)$;
3  $CA \leftarrow$ GETCONFLICTAGENTS$(a_m, p_m, N, EA)$;
4  for each $a_c \in CA \setminus PA$ do
5      $PA \leftarrow PA \cup \{a_c\}$;
6      $MDD_c \leftarrow$ GETMDD$(a_c, N)$;
7      $M \leftarrow$ MUTEXPROPAGATION$(MDD_m, MDD_c)$;
8      if $M \neq \emptyset$ then
9          $C \leftarrow C \cup \{a_c\}$;
10         DELETENODES$(MDD_m, M)$;
11         if $MDD_m = \emptyset$ then
12             return conflict cluster $C$ ;
13         if $p_m \notin MDD_m$ then
14             $p_m \leftarrow$ GETMINCONFLICTPATH$(MDD_m)$;
15             goto line 3;
16 return $|CA|$ reduced ? Bypass$(p_m)$ : null;
```

Definition 2 (Incompatible Node). *Given a pair of MDDs MDD_i and MDD_j for agents a_i and a_j, a MDD node n_i at level t from MDD_i is incompatible with MDD_j iff n_i is* mutex *with all MDD nodes at level t from MDD_j.*

According to Property 1, if an MDD node n_i from MDD_i is incompatible with MDD_j, all possible cost-minimised paths of a_i using n_i have conflicts with all cost-minimised paths of a_j. Our algorithm maintains a path p_m for a_m and uses it as guidance to detect a conflict cluster or bypass, by incrementally removing nodes of MDD_m that are incompatible with the MDDs of the agents whose paths conflict with p_m. Next, we explain the details of our algorithm.

Algorithm 1 calls the function FINDCLUSTERORBYPASS to find a conflict cluster or bypass for input agent a_m. The pseudo-code of this algorithm is shown in Algorithm 2. To begin, the algorithm initialises the conflict cluster C to contain the agent a_m and initialises the processed agents PA to be empty. It retrieves the current path p_m of a_m (line 1). The MDD of a_m, denoted as MDD_m, is then built which satisfies all constraints in N (line 2). The algorithm then calls GETCONFLICTAGENTS, which considers all non-excluded agents in N and returns the set of non-excluded agents CA that conflict with a_m (line 3).

In each iteration, the algorithm iteratively accesses the agents in CA that have not been processed before. For each such agent $a_c \in CA \setminus PA$, we build the MDD of a_c, denoted as MDD_c (line 6). The algorithm performs mutex propagation between the MDD_m and MDD_c and returns the incompatible nodes M of MDD_m which are mutex with every MDD node of MDD_c in the same level (line 7). If M is not empty, we append the agent a_c into the conflict cluster C (line 9) and recursively delete every incompatible node $n \in M$ (and the connected edges) from MDD_m (line 10). After deleting the incompatible nodes of MDD_m, it is possible that MDD_m becomes empty or the current path p_m is not valid in MDD_m as some of the nodes have been deleted. We handle each case as follows.

- If the MDD_m is empty, this implies that C is a conflict cluster which is returned (line 11-12).
- If the $p_m \notin MDD_m$, the algorithm finds an alternative path from MDD_m that has the minimal number of conflicts with the other agents (lines 13 and 14). Since path p_m is updated, there may be new agents that are in conflict with this new path. So, the algorithm goes to line 3 and re-computes CA. Since CA is changed, the algorithm continues to iteratively process the agents in $CA \setminus PA$ (line 4 onwards).

When the algorithm has processed all agents in $CA \setminus PA$, it terminates (line 16) by returning the bypass p_m if this bypass has fewer conflicts than the original path $N.\mathcal{P}(a_m)$. Otherwise, it returns null, indicating that no cluster or bypass is detected. The bypass p_m returned by Algorithm 2 is an alternative path retrieved from MDD_m, which avoids traversing through incompatible nodes. Since the MDD_m satisfies every constraint on a_m and has the same cost as $N.\mathcal{P}(a_m)$, p_m is a valid bypass.

To find the minimum-conflict path (line 14) and update the conflict agents (line 3), we must repeatedly detect conflicts between a_m and other agents, which can be time-consuming. Therefore, we use a labelling method that labels the conflict agents on each node and edge of MDD_m. Every time the algorithm extracts the path, we run a breadth-first search from source to goal of MDD_m and compute the minimum number of conflicts and its predecessor on each node visited. The minimum-conflict path and its conflict agents can be easily retrieved from a backward extraction following the predecessor node. Note that we only label MDD_m in Algorithm 2 once (when the algorithm reaches line 14 for the first time).

Theorem 2. *The cluster C returned by Algorithm 2 is a conflict cluster, according to Definition 1.*

Proof. Mutex propagation of MDD_c and MDD_m removes from MDD_m only the nodes which are incompatible with all paths in MDD_c. So unless the agent a_c increases its path length, the paths removed for a_m from MDD_m must conflict with a_c. If MDD_m becomes empty, then clearly all paths of the current path length of a_m must conflict with some other agents in the cluster. Hence, at least one agent in the cluster must increase its path length by one to avoid conflicts. □

Example 4. *Consider the example from Figure 1. Assume the paths in $N.\mathcal{P}$ for a_1, a_2 and a_3 are the solid blue, green and orange lines in Figure 1a, respectively. Algorithm 2 starts with $a_m = a_2$ and initialises $C = \{a_2\}$. The algorithm finds the set of conflicting agents $CA = \{a_3\}$ because a_2 and a_3 conflict. It then processes a_3 and performs mutex propagation between MDD_2 and MDD_3. The incompatible nodes (e.g., coloured orange in Figure 1b) of MDD_2 are removed and a_3 is appended to C. Since the path of a_2 no longer exists in MDD_2, the algorithm then updates its path in $N.\mathcal{P}$ to be the minimal conflicts path (e.g., the dashed green line). This new path collides with a_1. The algorithm returns to line 3 and finds conflicting agents $CA = \{a_1\}$.*

Agent a_1 is processed and appended to C. MDD_2 becomes empty after removing incompatible nodes (e.g., coloured blue in Figure 1b) from MDD_2. The algorithm returns the conflict cluster $C = \{a_1, a_2, a_3\}$.

Optimisation

In this section, we introduce optimisation techniques that improve the cluster heuristic h_c and speed up our algorithm.

Solving the Cluster Recall that Algorithm 1 increases the cluster heuristic h_c by one (line 10) whenever it detects a conflict cluster C. However, to get a better heuristic value, we can solve the cluster as a sub-instance to improve the lower bound of C. Therefore, we take the paths and constraints of agent $a_i \in C$ from the current CT node N, and run a sub-CBS search to solve C. To restrict the computation cost of this optimisation, when solving a cluster, we also set a limit $\mathcal{L}$ on the number of CT nodes expanded by the sub-CBS. By default, we use the same setting (i.e., $\mathcal{L} = 10$) as used in WDG heuristic (Li et al. 2019). Let Δ_C be the increase of the minimal f-value in the open list after running sub-CBS for this cluster. We increment the heuristic by Δ_C (i.e., Algorithm 1 line 10 : h_c += Δ_C). It is easy to see that the correctness of Theorem 1 is preserved.

Memoisation The algorithms have two operations that can be repetitively performed in the same or different branches of a CT tree: (i) computing heuristic for the same conflict clusters using sub-CBS described above; and (ii) performing the mutex propagation between the same pair of MDDs (Algorithm 2 - line 7). We say that two conflict clusters (resp. MDDs) are the same if the two clusters (resp. MDDs) have the same agents with exactly the same constraints for each agent. In order to speed up the search, we apply memoisation by maintaining a centralised database in CBS. To avoid (i), we simply maintain a hash table to cache the increased cost Δ_C of a conflict cluster C by hashing all constraints $\in N.constraints$ of agents in C as a key. However, avoiding (ii) needs some modifications detailed below.

Algorithm 2 takes the MDD of a_m and performs mutex propagation with the MDDs of the conflicting agents $a_c \in CA$. In each iteration, the incompatible MDD nodes of MDD_m are removed which results in a smaller MDD_m. Therefore, we cannot cache the results of mutex propagation between MDD_m and MDD_c as MDD_m changes after each iteration. To overcome this issue, we propose to apply a reusable version of mutex propagation. This reusable mutex propagation does not consider the updated MDD_m, but only considers the original unmodified MDD_m and MDD_c from the CT node N. Let us denote the unmodified MDD_m as MDD'_m. The algorithm begins with MDD_m of agent a_m. Every time the algorithm performs mutex propagation between a_m and a_c, it performs the reusable mutex propagation and returns the incompatible nodes of MDD'_m to MDD'_c. We use this result to remove MDD nodes from MDD_m until MDD_m becomes empty or there is no other valid conflicting agent a_c. Although this lazy strategy weakens mutex propagation (i.e., the reusable mutex propagation may be able to detect only a subset of incompatible nodes), we can now cache the incompatible nodes between a_m and a_c based on the constraints of the two agents. In the experiments, we show that the reusable mutex propagation almost always leads to a speedup as it does not lose too much mutex information and can reuse many mutex calculations.

Experiments

In this section, we compare our algorithm against the state-of-the-art variation of CBS (Li et al. 2021) taken from the repository[2] of the authors. This algorithm applies all leading optimisation techniques including: (i) high-level heuristics: weighted pairwise dependence graph (WDG) (Li et al. 2019); (ii) symmetry reasoning techniques: target reasoning, generalised rectangle and corridor reasoning (Li et al. 2021); and (iii) prioritising and bypassing conflicts (Boyarski et al. 2015b,a). We use WDG to refer to this algorithm. Our algorithm is built on top of WDG and, in addition, uses cluster heuristic and bypass (CHBP). It is shown as WDG+CHBP in the experiments. We also compare the algorithm when only cluster heuristic is used and the bypass is ignored (i.e., WDG+CH) or when only the bypass returned by Algorithm 2 is used but the cluster heuristic is ignored (shown as WDG+BP). We do not compare our algorithm against the LR heuristic (Mogali, van Hoeve, and Smith 2020) because it requires us to modify the definition of MAPF by limiting the maximum cost of the paths.

Benchmarks We conduct experiments on four diverse maps taken from the widely used 4-connected grid map benchmarks[3], described by Stern et al. (2019). These maps cover different real-life scenarios.

- Random map (random-32-32-20): a 32×32 grid map with 20% random blocked cells. The number of agents on the map is set to 20, 30, ..., 70.
- Empty map (empty-32-32): an empty 32×32 grid map. The number of agents in the map is set to 50, 70, ..., 150.
- Warehouse map (warehouse-10-20-10-2-1): a 161×63 grid map which simulates the warehouse environment with 10×20 stacks. Each stack has 10×2 grids. The number of agents is set to 30, 50, ..., 130.
- Game map (den520d): a 256×257 grid map from a video game. The number of agents is set to 40, 60, ..., 140.

The benchmark contains, for each map setting, two sets of instances each containing 25 instances: the first set generates agents with randomly selected start and goal locations; the second set generates agents with an even mix of short and long distances between their start and goal locations. We run every instance for 1 minute and report the overall performance. The instances that cannot be solved in 1 minute by an algorithm are considered unsolved. All algorithms are implemented in C++ and compiled with -O3 flag. We conduct all experiments on a Nectar research cloud with 128GB of RAM running Ubuntu 18.04.4 LTS (Bionic Beaver). For reproducibility, our implementation is available online.[4]

[2]https://github.com/Jiaoyang-Li/CBSH2-RTC

[3]https://movingai.com/benchmarks/mapf

[4]https://github.com/bshen95/CBSH2-RTC-CHBP

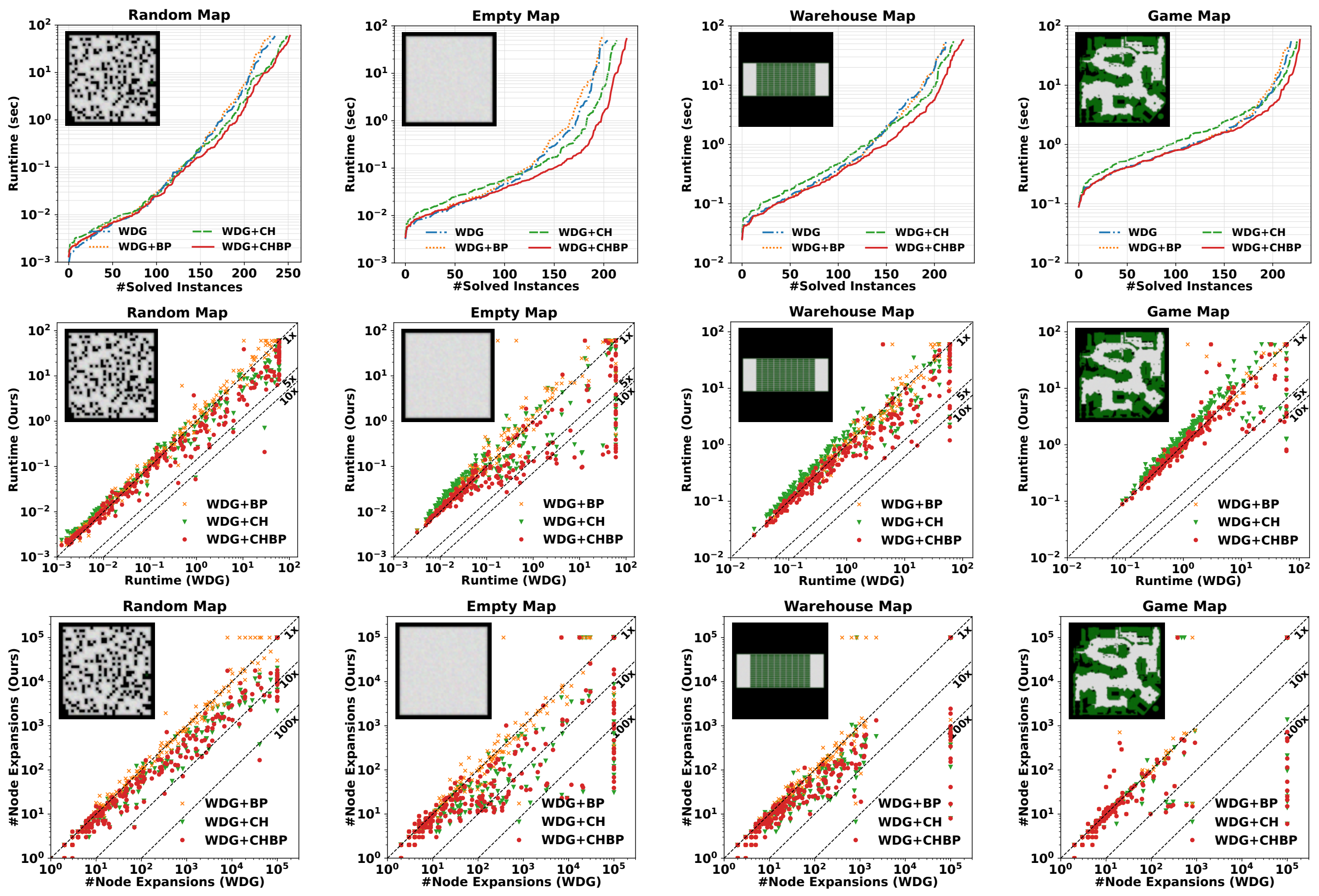


Figure 2: Cactus plots for runtime in seconds (top row), scatter plots for runtime in seconds (middle row), and scatter plots for CT node expansions (bottom row). If an approach fails to solve an instance in 60 seconds (i.e., unsolved instance), its runtime in the figure is shown as 60 seconds and its number of node expansions is shown to be 10^5 (all solved instances have runtime less than 60 and node expansions less than 10^5).

Runtime and CT Node Expansions

Top row in Figure 2 shows the cactus plots for runtime (sec) of different algorithms. Using cluster heuristic and bypass (WDG+CHBP) significantly improves the performance on hard instances (note the log scale on the y-axis). While the cluster heuristic (WDG+CH) leads to improvements over WDG, bypassing alone (WDG+BP) does not help solve hard instances. This is because CHBP mainly benefits from increasing heuristic value whereas bypassing itself neither considers heuristic value nor excludes the agents of detected clusters which results in degraded performance.

The scatter plots in the middle row (Figure 2) show detailed runtime comparisons versus the baseline WDG. The three diagonal lines show the performance improvement compared to WDG (1x, 5x or 10x), i.e., a point under the diagonal line 5x indicates that our algorithm is more than 5 times faster than WDG on that instance. The scatter plots show that our methods improve upon the baseline for most of the instances and rarely show significantly worse runtime. Importantly, our methods are able to solve many instances that are unsolved by the baseline (the instances shown at 60 seconds on the x-axis). There are some instances which we fail to solve but the baseline can solve (illustrated by the points at 60 seconds on the y-axis). Note that the number of such instances is much smaller than the instances that the baseline cannot solve but our algorithm can solve. Overall WDG, WDG+BP, WDG+CH and WDG+CHBP solve 875, 859, 896 and 937 instances, respectively.

The scatter plots on the bottom row in Figure 2 show detailed comparisons of # CT node expansions of our algorithms with WDG. Again, our methods almost always lead to fewer CT node expansions.

Ablation Study on Optimisation Techniques

Figure 3 shows the scatter plots for the runtime of our final algorithm (WDG+CHBP) versus two modified versions of the algorithm without applying optimisation techniques. The diagonal lines show how slow the two versions are compared to our final algorithm WDG+CHBP, i.e., a point above the diagonal 2x shows an instance where the algorithm is more than 2 times slower than WDG+CHBP. Clearly, both versions show worse performance than our final algorithm,

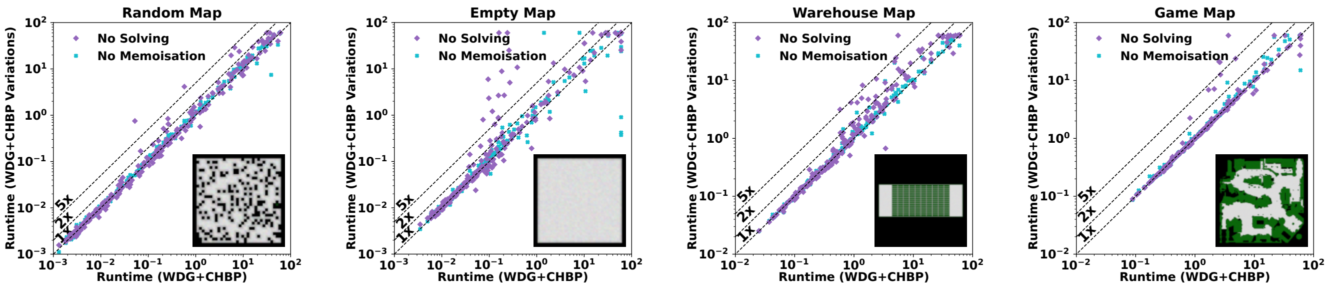


Figure 3: Effect of optimisation techniques on runtime (sec) of our final algorithm (WDG+CHBP). No Solving is when the optimisation to solve the cluster is not applied and No Memoisation is when the memoisation is not applied.

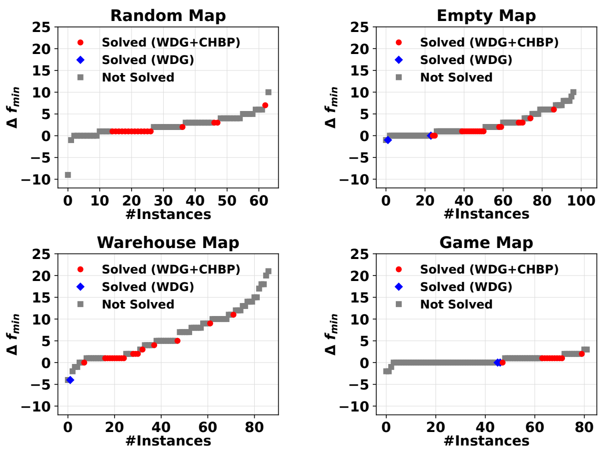


Figure 4: Δf_{min}=f_{min}(WDG+CHBP) $-$ f_{min}(WDG). Instances solved by both WDG and WDG+CHBP are removed as they have $\Delta f_{min} = 0$. We show instances solved by only WDG (blue), solved by only WDG+CHBP (red) and unsolved by both (grey).

which demonstrates the effectiveness of our proposed optimisations. In addition, No Solving is significantly worse than WDG+CHBP on many instances. This shows that solving the cluster is the most important enhancement as it significantly increases the heuristic value for some of the clusters detected. Although not as significant, memoisation also plays an important role by avoiding repeatedly solve the same clusters and perform mutex propagation between the same pair of MDDs.

Effect of Heuristic Value and Insights

In Figure 4, we show Δf_{min}=f_{min}(WDG+CHBP) $-$ f_{min}(WDG) where $f_{min}(X)$ is the minimum f-value ($f = N.cost + h$) in the open list when the algorithm X terminates. Δf_{min} shows the difference in heuristic values of the two algorithms indicating how much the cluster heuristic is able to improve the search progress compared to WDG. Note that while it cannot make the heuristic worse at a CT node, it does change the CBS search tree which may lead to a smaller f_{min} for WDG+CHBP for some instances compared to WDG. Figure 4 shows that Δf_{min} is mostly positive and there is typically a significant increase in f_{min} for WDG+CHBP compared to WDG, especially on the empty and warehouse maps. Also, note that WDG+CHBP solves many instances that WDG cannot solve. On the other hand, there are very few instances that only WDG can solve.

Map	CBS Search		Compute WDG		Compute CHBP	
	Total(s)	PH(ms)	Total(s)	PH(ms)	Total(s)	PH(ms)
Random	1094.39	0.54	1158.83	0.57	1452.25	0.71
Empty	1061.40	0.50	1342.00	0.63	2513.68	1.23
Warehouse	2826.13	18.53	932.69	6.12	1334.28	8.75
Game	2972.42	32.93	627.17	6.95	1084.17	12.01

Table 1: Performance breakdown of WDG+CHBP. We show the total runtime (Total) and the average runtime per heuristic calculated (PH) for each component in WDG+CHBP.

Table 1 shows the average runtime per heuristic calculated of the various components: CBS search, that is everything else than heuristics and bypass calculation; Compute WDG, the time to compute the WDG heuristic; and Compute CHBP, the time to compute our heuristic and bypasses. Clearly, the more complex heuristics are more expensive on average than the WDG heuristic, but never more than 2 times more expensive. They take less time than the remaining components on the larger maps. Overall of course the computation cost of this heuristic almost always pays off in terms of reduced high-level search.

Conclusion

In this work, we propose new techniques to compute heuristics by reasoning incompatibility beyond two agents. Our approach dynamically finds conflict clusters and bypasses at the same time. We substantially improve CBS by solving more instances in limited time and reducing the CT node expansion and runtime to solve problems. For instances with a timeout failure, we push the lower-bound (i.e., f_{min}) to a significantly higher value. We show that reasoning for conflict clusters is essential to solving larger MAPF problems. Future works include capturing more complex clusters, integrating conflict cluster heuristics to the integer program of the WDG heuristic, and designing strong methods to efficiently resolve all conflicts in clusters.

Acknowledgements

This research was partially supported by the Australian Research Council under grants FT180100140, DP190100013, DP200100025, DP230100081 and also by a gift from Amazon.

References

Blum, A.; and Furst, M. L. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1-2): 281–300.

Boyarski, E.; Felner, A.; Le Bodic, P.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021. f-Aware Conflict Prioritization & Improved Heuristics for Conflict-Based Search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 12241–12248.

Boyarski, E.; Felner, A.; Sharon, G.; and Stern, R. 2015a. Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS)*, 47–51.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, S. E. 2015b. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 740–746.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS)*, 83–87.

Hönig, W.; Preiss, J. A.; Kumar, T. S.; Sukhatme, G. S.; and Ayanian, N. 2018. Trajectory Planning for Quadrotor Swarms. *IEEE Transactions on Robotics*, 34(4): 856–869.

Kautz, H. A.; and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, 1194–1201.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, 442–449.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; Gange, G.; and Koenig, S. 2021. Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search. *Artificial Intelligence*, 301: 103574.

Mackworth, A. K. 1977. Consistency in Networks of Relations. *Aritificial Intelligence*, 8(1): 99–118.

Mogali, J. K.; van Hoeve, W.; and Smith, S. F. 2020. Template Matching and Decision Diagrams for Multi-Agent Path Finding. In *Proceedings of the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, 347–363.

Nguyen, X.; and Kambhampati, S. 2000. Extracting Effective and Admissible State Space Heuristics from the Planning Graph. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, 798–805.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-Based Search For Optimal Multi-Agent Path Finding. *Artificial Intelligence*, 219: 40–66.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence*, 195: 470–495.

Silver, D. 2005. Cooperative Pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Path Finding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 151–159.

Surynek, P.; Li, J.; Zhang, H.; Kumar, T. K. S.; and Koenig, S. 2020. Mutex Propagation for SAT-based Multi-Agent Path Finding. In *Proceedings of the International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, 248–258.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI magazine*, 29(1): 9–9.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI)*, 1443–1449.

Zhang, H.; Li, J.; Surynek, P.; Kumar, T. S.; and Koenig, S. 2022. Multi-Agent Path Finding with Mutex Propagation. *Artificial Intelligence*, 311: 103766.