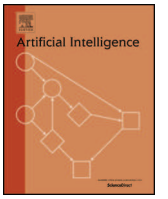
Artificial Intelligence 311 (2022) 103766

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Multi-agent path finding with mutex propagation ☆

Han Zhang [a], Jiaoyang Li [a], Pavel Surynek [b,*], T.K. Satish Kumar [a], Sven Koenig [a]

[a] University of Southern California, Los Angeles, USA
[b] Czech Technical University, Prague, Czechia

ARTICLE INFO

Article history:
Received 17 April 2021
Received in revised form 6 July 2022
Accepted 10 July 2022
Available online 16 July 2022

Keywords:
Heuristic search
Multi-agent path finding
Satisfiability solving

ABSTRACT

Mutex propagation is a form of efficient constraint propagation popularly used in AI planning to tightly approximate the reachable states from a given state. We utilize this idea in the context of Multi-Agent Path Finding (MAPF). When adapted to MAPF, mutex propagation provides stronger constraints for conflict resolution in CBS, a popular optimal search-based MAPF algorithm, as well as in MDD-SAT, an optimal satisfiability-based MAPF algorithm. Mutex propagation provides CBS with the ability to break symmetries in MAPF and provides MDD-SAT with the ability to make stronger inferences than unit propagation. While existing work identifies a limited form of symmetries and requires the manual design of symmetry-breaking constraints, mutex propagation is more general and allows for the automated design of symmetry-breaking constraints. Our experimental results show that CBS with mutex propagation is capable of outperforming CBSH-RCT, a state-of-the-art variant of CBS, with respect to the success rate. We also show that MDD-SAT with mutex propagation often performs better than MDD-SAT with respect to the success rate.

1. Introduction

The Multi-Agent Path Finding (MAPF) problem is a generalization of the single-agent path finding problem to multiple agents. Each agent is required to move from a given start vertex to a given goal vertex on a given graph while avoiding conflicts (collisions) with other agents. A conflict happens when two agents stay at the same vertex or traverse the same edge in opposite directions at the same time. Common objectives for the MAPF problem include minimizing the sum of the path costs [1] and minimizing the makespan [2]. Under both objectives, the MAPF problem is NP-hard [3,4] and arises in many real-world application domains, including automated warehouse robots [5] and aircraft-towing vehicles [6].

Conflict-Based Search (CBS) [7] is a popular algorithm for solving the MAPF problem optimally for both objectives. It is a two-level MAPF algorithm that starts with a minimum-cost path for each agent which might conflict with the other paths. On the high level, CBS maintains a Constraint Tree (CT) and resolves conflicts between pairs of agents by adding spatio-temporal constraints to prohibit one of the agents from occupying the conflicting vertex or traversing the conflicting edge at the conflicting timestep. On the low level, CBS finds individual minimum-cost paths for each agent satisfying the

☆ This paper is an invited revision of a paper which first appeared at the 2020 International Conference on Automated Planning and Scheduling (ICAPS-20).

* Corresponding author.
E-mail addresses: zhan645@usc.edu (H. Zhang), jiaoyanl@usc.edu (J. Li), pavel.surynek@fit.cvut.cz (P. Surynek), tkskwork@gmail.com (T.K. Satish Kumar), skoenig@usc.edu (S. Koenig).

https://doi.org/10.1016/j.artint.2022.103766
0004-3702/

spatio-temporal constraints specified by the high-level CT node. CBS expands high-level CT nodes in a best-first order and returns a set of paths as solution when they are conflict-free. Many improvements to CBS have been made, such as adding conflict-selection strategies [8] and heuristic guidance [9]. Recent work has demonstrated significant improvement on CBS by identifying specific cases where CBS expands a large number of CT nodes to resolve all conflicts between a pair of agents and developing specific techniques to handle each of these cases efficiently. Such cases are referred to as "symmetries", and the techniques for handling them are called symmetry-breaking techniques.

An important alternative to search-based MAPF algorithms is the compilation-based paradigm, where the MAPF problem is first reduced to a target problem, such as *Boolean SATisfiability* (SAT) [10,11], *Constraint Satisfaction* [12] or *Answer Set Programming* [13], and then solved by an existing solver for the target problem, which often implements multiple techniques to find a solution (or rule out its existence) efficiently. In case of SAT as the target problem, complete SAT solvers, such as MDD-SAT, implement Boolean constraint propagation (unit propagation), backjumping, clause learning and even techniques from local search, such as restarts and randomization [14–16]. However, the reduction must be done with care to allow the SAT solver to find a solution efficiently.

In this article, we utilize a well-known technique, called *mutex propagation*, from AI planning.[1] It is a form of constraint propagation corresponding to directed 3-consistency, which in turn is a truncated form of path consistency [17]. Like all constraint propagation techniques, it makes implicit constraints explicit, and it does so efficiently. In AI planning, mutex propagation is applied to the planning graph [18] to tightly approximate the set of all reachable states from a given state in polynomial time [17]. It has successfully been used to design reachability heuristics for state-space planners [19], design heuristics for plan-space planners that make them competitive with state-space planners [20] and improve SAT-based planners [21].

Elements of the planning graph idea have reappeared in MAPF research in the form of Multi-valued Decision Diagrams (MDDs) [8]. MDDs are constructed for each agent individually and essentially capture reachability information for them. However, they do not capture reachability information for groups of agents. On the other hand, building joint MDDs for groups of agents is computationally prohibitive because the joint space grows exponentially in the number of agents. Knowing that mutex propagation alleviates this dilemma in AI planning, we seek to transfer this technique to MAPF, particularly in the CBS- and SAT-based frameworks.

We show that mutex propagation provides stronger constraints for conflict resolution in CBS as well as in MDD-SAT, an optimal SAT-based MAPF algorithm. Mutex propagation provides CBS with the ability to break symmetries in MAPF, and it provides MDD-SAT with the ability to make stronger inferences compared to unit propagation [11]. While existing work identifies a limited form of symmetries and requires the manual design of symmetry-breaking constraints [22,23], mutex propagation is more general and allows for the automated design of symmetry-breaking constraints. Our experimental results show that CBS with mutex propagation is capable of outperforming CBSH-RCT, a state-of-the-art variant of CBS, with respect to the success rate. We also show that MDD-SAT with mutex propagation often achieves higher success rate than MDD-SAT.

This paper extends our previous work presented in [24] and [25], both published in conference proceedings. These extensions include:

1. Generalization of the mutex propagation technique to semi-cardinal conflict reasoning and a new mutex-based technique for conflict selection.
2. A more thorough discussion of the theoretical properties of mutex propagation in MAPF.
3. Extended experimental evaluation with a larger set of benchmarks. Moreover, the same set of benchmarks is used for CBS with mutex propagation and MDD-SAT with mutex propagation.

2. Preliminaries

In this section, we provide background material on MAPF, CBS, MDDs and MDD-SAT.

2.1. MAPF

The MAPF problem has many variants [26]. In this article, we focus on the variant that (1) considers vertex and edge conflicts, (2) uses the "stay at goal vertex" assumption and (3) optimizes the sum of costs. Formally, we define the MAPF problem by an undirected graph $G = (V, E)$ and a set of m agents $\{a_1 \dots a_m\}$. Each agent has a start vertex $s_i \in V$ and a goal vertex (or, synonymously, target) $g_i \in V$. In each timestep, an agent either moves to a neighboring vertex or waits at its current vertex. When an agent is at its goal vertex, it can *terminally wait* there, which means that the agent waits there forever. Both move and wait actions have unit cost, while terminally waiting at the goal vertex has zero cost. A *path* for an agent is a sequence of move and wait actions that leads the agent from its start vertex to terminally waiting at its goal vertex. A *sub-path* for an agent is a sequence of actions that leads the agent from one vertex at a specific timestep to another vertex at a specific timestep. The *cost* of a path is the accumulated cost of all actions in it. A *vertex conflict* happens

[1] Mutex is short for mutual exclusion.

Algorithm 1: CBS: Solve a MAPF instance.

```
   Input  : A MAPF instance.
   Output: A minimum-cost solution of the MAPF instance.
 1 Root.constraints ← ∅;
 2 Root.paths ← individual minimum-cost paths for all agents;
 3 Root.cost ← SoC(Root.paths);
 4 OPEN := {Root};
 5 while OPEN is not empty do
 6 |   n ← a CT node in OPEN with minimum cost;
 7 |   if n.paths have no conflict then
 8 |   |   return n.paths;
 9 |   end
10 |   c ← choose a conflict in n.paths;
11 |   foreach agent a_i in c do
12 |   |   Child ← copy of n;
13 |   |   Add all constraints from the constraint set for agent a_i to Child.constraints;
14 |   |   Update the path for agent a_i in Child.paths via a call to the low level of CBS;
15 |   |   Child.cost ← SoC(Child.paths);
16 |   |   if Child.cost < +∞ then
17 |   |   |   Add Child to OPEN;
18 |   |   end
19 |   end
20 end
21 return "no solution exists";
```

when two agents stay at the same vertex simultaneously, and an *edge conflict* happens when two agents traverse the same edge simultaneously in opposite directions. A *solution* is a set of conflict-free paths for all agents. The *Sum of path Costs* (SoC) is the sum of costs of the paths for all agents. An *optimal solution* is a solution with the minimum SoC.

2.2. CBS

CBS is a two-level optimal MAPF algorithm. Algorithm 1 presents the pseudocode for it. On the high level, CBS maintains a *Constraint Tree* (CT). Each CT node contains a set of constraints and a set of paths, one for each agent, satisfying all these constraints. The root CT node contains no constraints. The cost of a CT node is the SoC of its paths. On the low level, for each CT node, CBS finds an *individual minimum-cost path* for each agent (Lines 2 and 14), that is, a path that has the smallest cost among all paths satisfying all constraints of the CT node (but might conflict with the other paths). CBS uses the space-time A* algorithm to find individual minimum-cost paths. The *individual minimum cost* $l_i^\star$ of agent a_i in a CT node is the cost of its path in the CT node. When expanding a CT node, CBS returns its paths as a solution if they are conflict-free (Lines 7-9). Otherwise, CBS picks a conflict and splits the CT node into two child CT nodes, one for each conflicting agent, which inherit all constraints of their parent CT node (Line 12). CBS calculates a constraint set for each agent and then adds all constraints from the constraint set to the corresponding child CT node to prohibit either one or the other of the two agents from using the conflicting vertex or edge at the conflicting timestep (Line 13). On the high level, CBS expands CT nodes in a best-first order. Therefore, the paths of the first expanded CT node with conflict-free paths form an optimal solution.

Constraints: A constraint is a spatio-temporal restriction introduced by CBS to resolve conflicts. A vertex constraint $\langle a_i, t, v\rangle$ prohibits agent a_i from occupying vertex v at timestep t, and an edge constraint $\langle a_i, t, v, v'\rangle$ prohibits agent a_i from moving from vertex v to vertex v', that is, traversing edge $\langle v, v'\rangle$, between timesteps t and $t+1$.

Cardinal and semi-cardinal conflicts: Two agents have a *cardinal conflict* in a CT node iff there does not exist a pair of conflict-free individual minimum-cost paths for both agents (that, by definition, satisfy all constraints of the CT node). CBS cannot resolve all cardinal conflicts efficiently (examples are in the four cases in Fig. 1) since it needs to check all combinations of paths whose SoC is less than the SoC of an optimal solution, which can necessitate many CT node expansions.

We extend the definition of cardinal conflicts as follows: Agents a_i and a_j have a *cardinal conflict within costs* (l_i, l_j) in a CT node iff there does not exist a pair of conflict-free paths for these two agents with costs l_i and l_j, respectively, satisfying all constraints of the CT node. Cardinal conflicts are then identical to cardinal conflicts within costs $(l_i^\star, l_j^\star)$. The larger the values of $l_i - l_i^\star$ and $l_j - l_j^\star$, the more the path costs of agents need to increase, and the more time-consuming it is for CBS to resolve all the conflicts between the two agents. This extended notion of cardinal conflicts allows us to guide the high-level search of CBS using mutex propagation.

Agent a_i has a *semi-cardinal conflict* with agent a_j in a CT node iff agents a_i and a_j do not have a cardinal conflict and there does not exist an individual minimum-cost path for agent a_i (that, by definition, satisfies all constraints of the CT node) that is conflict-free with the path of agent a_j in the CT node.

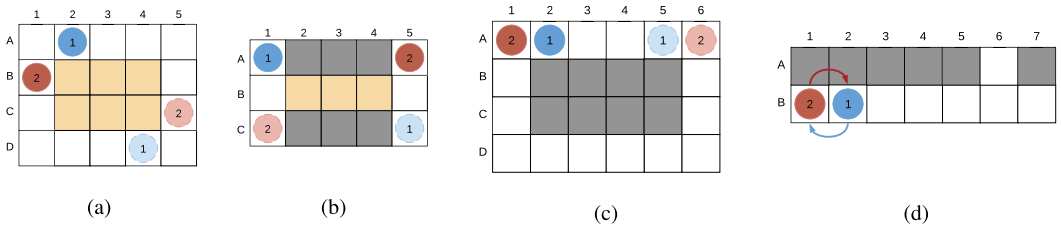


Fig. 1. Shows some MAPF instances with cardinal conflicts in the root CT node. The white and yellow cells indicate free cells, while the gray cells indicate obstacles. Yellow cells highlight interesting areas that we will refer to in the text. The start vertices of the agents are marked with solid-line circles, and their goal vertices are marked with dashed-line circles. (a) shows a cardinal rectangle conflict instance, where one of the two agents needs to wait for one timestep in every optimal solution [22]. (b) shows a cardinal corridor conflict instance, where one of the two agents needs to wait until the other agent exits the corridor [27,28,23]. (c) shows a cardinal target conflict instance, where agent a_2 needs to take the long path in the optimal solution. (d) shows a cardinal switching agents conflict instance, where both agents need to move to the rightmost side of the corridor in order to switch their vertices [29]. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

We extend the definition of semi-cardinal conflicts as follows: Agent a_i has a *semi-cardinal conflict within cost* l_i with agent a_j in a CT node iff agents a_i and a_j do not have a cardinal conflict and there does not exist a path for agent a_i with cost l_i satisfying all constraints of the CT node that is conflict-free with the path of agent a_j in the CT node. Semi-cardinal conflicts for agent a_i are then identical to semi-cardinal conflicts within cost $l_i^\star$. This extended definition of semi-cardinal conflicts will be used later in this article.

Two agents have a *non-cardinal conflict* iff their conflict is not cardinal or semi-cardinal.

Our definitions of cardinal, semi-cardinal and non-cardinal conflicts are different from those used in previous literature [8]. While our definitions are with respect to agents, those in previous literature are with respect to a specific vertex or edge conflict. Despite these differences, the intuitions are similar. In fact, previous literature [8] shows that prioritizing conflicts when choosing a conflict on Line 10 in Algorithm 1 improves the performance of CBS, where the first priority is given to cardinal conflicts, followed by semi-cardinal conflicts, and then to non-cardinal conflicts.

Cardinal rectangle conflicts and barrier constraints: In four-neighbor grid maps, two agents have a cardinal rectangle conflict in a CT node iff (1) all individual minimum-cost paths of both agents cross the same rectangular area; (2) the earliest possible timesteps of both agents reaching each vertex inside the rectangular area are equal; and (3) the directions of both agents moving through the rectangular area are same in both dimensions. Then, there does not exist a pair of conflict-free individual minimum-cost paths for both agents. A barrier constraint is a set of vertex constraints that prohibits one or the other of the two agents from leaving the rectangular area on an individual minimum-cost path. Adding barrier constraints significantly improves the performance of CBS [22]. Moreover, the authors of [22] prove that using barrier constraints as constraint sets to resolve cardinal rectangle conflicts also preserves the optimality and completeness of CBS.

Example 1. Consider the cardinal rectangle conflict instance in Fig. 1a. There does not exist a pair of conflict-free individual minimum-cost paths for agents a_1 and a_2 since any pair of individual minimum-cost paths for both agents has at least one vertex conflict in the yellow rectangular area. In any optimal solution, one or the other of the two agents needs to wait for one timestep, and the optimal SoC is 11. The barrier constraint for agent a_1 is $\{\langle a_1, 2, C2\rangle, \langle a_1, 3, C3\rangle, \langle a_1, 4, C4\rangle\}$, and the barrier constraint for agent a_2 is $\{\langle a_2, 3, B4\rangle, \langle a_2, 4, C4\rangle\}$.

Cardinal corridor and target conflicts: A corridor is a chain of connected vertices, each of degree two. Two agents have a cardinal corridor conflict when all of their individual minimum-cost paths move through the same narrow corridor in opposite directions but collide inside the corridor. Fig. 1b shows an example of a corridor conflict, where the corridor is highlighted in yellow. Two agents have a target conflict when the individual minimum-cost path of one agent passes through the goal vertex of the other agent when the latter agent is terminally waiting at it (see Fig. 1c for example). The authors of [23] propose two symmetry-breaking techniques to detect these two classes of conflicts and resolve them efficiently with specific constraints.

2.3. MDDs

An MDD [29,7] MDD_i^l for agent a_i in a CT node is a $(l+1)$-level directed acyclic graph that consists of all paths of cost l for agent a_i satisfying all constraints of the CT node. We assume that $l \geq l_i^\star$. The MDD nodes of MDD_i^l at level t correspond to all possible vertices of agent a_i at timestep t in these paths. At level 0, MDD_i^l has a single *source MDD node* corresponding to agent a_i occupying its start vertex s_i at timestep 0. At level l, MDD_i^l has a single *sink MDD node* corresponding to agent a_i occupying its goal vertex g_i at timestep l. For an MDD node $n \in MDD_i^l$, we define $n.level$ as the timestep of n and $n.loc$ as the vertex of n. We define the *constraint on MDD node* n as the vertex constraint $\langle a_i, n.level, n.loc\rangle$. For an MDD edge $e = \langle n, n'\rangle \in MDD_i^l$, we define $e.level = n.level$, $e.from = n$ and $e.to = n'$. We define the *constraint on MDD edge* e as the edge constraint $\langle a_i, n.level, n.loc, n'.loc\rangle$.

2.4. MDD-SAT

MDD-SAT [11] is a SAT-based MAPF solver that pioneered the use of MDDs in the compilation-based paradigm. MDD-SAT is also the first SAT-based MAPF solver that minimizes the SoC.[2]

We first describe how to construct a Boolean formula $\mathcal{F}(x)$ that is satisfiable iff a given MAPF instance has a solution with a SoC of at most x.

MDD-SAT first builds MDDs for individual agents and a given number of levels. The number of levels l for the MDDs is $l = l_0 + (x - x_0)$ [11], where l_0 is a lower bound on the makespan, calculated as the largest individual minimum cost for the agents, and x_0 is a lower bound on the SoC, calculated as the sum of the individual minimum costs.

A propositional variable is introduced for each MDD node n_i and MDD edge $\langle n_i, n_i' \rangle \in MDD_i^l$. We define $\mathcal{X}_{n_i}$ as the variables corresponding to MDD nodes and $\mathcal{E}_{n_i,n_i'}$ as the variables corresponding to MDD edges. $\mathcal{X}_{n_i}$ is *TRUE* iff agent a_i is in vertex $n_i.loc \in V$ at timestep $n_i.level$, and $\mathcal{E}_{n_i,n_i'}$ is *TRUE* (that is, the corresponding MDD edge is selected) iff agent a_i moves from vertex $n_i.loc$ to vertex $n_i'.loc$ between timesteps $n_i.level$ and $n_i.level + 1$.

The MAPF movement rules are encoded as clauses using these variables. Satisfying truth-value assignments to the variables correspond to paths in MDDs due to the following constraints for all MDD nodes $n_i \in MDD_i^{l_i}$ and MDD edges $\langle n_i, n_i' \rangle \in MDD_i^{l_i}$:

$$\mathcal{X}_{n_i} \Rightarrow \bigvee_{n_i' \mid \langle n_i, n_i' \rangle \in MDD_i^{l_i}} \mathcal{E}_{n_i,n_i'} \tag{1}$$

$$\sum_{n_i' \mid \langle n_i, n_i' \rangle \in MDD_i^{l_i}} \mathcal{E}_{n_i,n_i'} \leq 1 \tag{2}$$

$$\mathcal{X}_{n_i'} \Rightarrow \bigvee_{n_i \mid \langle n_i, n_i' \rangle \in MDD_i^{l_i}} \mathcal{E}_{n_i,n_i'} \tag{3}$$

$$\sum_{n_i \mid \langle n_i, n_i' \rangle \in MDD_i^{l_i}} \mathcal{E}_{n_i,n_i'} \leq 1 \tag{4}$$

$$\mathcal{E}_{n_i,n_i'} \Rightarrow \mathcal{X}_{n_i} \wedge \mathcal{X}_{n_i'}. \tag{5}$$

Constraints (1)-(5) ensure that the selected MDD edges form paths in the MDDs. They state that, if agent a_i is in a vertex $n_i.loc$ at timestep $n_i.level$, it must leave it via exactly one outgoing MDD edge $\langle n_i.loc, n_i'.loc \rangle$ and appear in vertex $n_i'.loc$ at timestep $n_i.level + 1$. In particular, Constraints (1) and (2) state that, if an agent is in a vertex, it must leave that vertex via exactly one outgoing MDD edge. Similarly, Constraints (3) and (4) state that, if an agent is in a vertex, then it must arrive there via exactly one incoming MDD edge. If agent a_i traverses edge $\langle n_i.loc, n_i'.loc \rangle$ then it must first enter the edge at timestep $n_i.level$ and leave it at timestep $n_i'.level$ (5).

Constraints that ensure vertex conflict avoidance state that two or more agents cannot be in vertex v at a timestep t:

$$\sum_{a_i \in A \mid \exists n_i \in MDD_i^{l_i} : (n_i.loc = v \wedge n_i.level = t)} \mathcal{X}_{n_i} \leq 1. \tag{6}$$

All pseudo-Boolean *at-most-one* constraints of the form in Constraint (6) can be expressed as clauses. One possible representation is to use a clique of binary clauses that forbids agents a_i and a_j to be in identical vertices $n_i.loc$ and $n_j.loc$ at timestep t: $\neg\mathcal{X}_{n_i} \vee \neg\mathcal{X}_{n_j}$.

Constraints that ensure edge conflict avoidance state that it cannot be that agent a_i traverses edge $\langle u, v \rangle$ between timesteps t and $t + 1$ and agent a_j traverses edge $\langle v, u \rangle$ between timesteps t and $t + 1$. One possible representation of one of these constraints is to use the following clause for every pair of MDD edges $\langle n_i, n_i' \rangle \in MDD_i^{l_i}$ and $\langle n_j, n_j' \rangle \in MDD_j^{l_j}$ with $n_i.loc = n_j'.loc$ and $n_i'.loc = n_j.loc$:

$$\neg\mathcal{E}_{n_i,n_i'} \vee \neg\mathcal{E}_{n_j,n_j'}. \tag{7}$$

Finally, the number of variables corresponding to MDD edges set to *TRUE* must be at most x to ensure that the SoC is at most x, which can be done with a cardinality constraint [31–33]. There are multiple approaches for encoding the cardinality constraint. MDD-SAT uses a sequential unary counter [31], a method inspired by Boolean circuit design; see [11] for details.

This concludes our description of how to construct a Boolean formula $\mathcal{F}(x)$ that is satisfiable iff a given MAPF instance has a solution with an SoC of at most x. Due to the construction of $\mathcal{F}(x)$, a solution can be read off from a satisfying

[2] The previous optimal SAT-based MAPF solvers [30,2] minimize the makespan.

truth-value assignment of $\mathcal{F}(x)$, which can be found with an off-the-shelf SAT solver [15]. We construct and solve the Boolean formulas $\mathcal{F}(x_0)$, $\mathcal{F}(x_0+1)$, ... in sequence. An optimal solution to the given MAPF instance then corresponds to the truth-value assignment of the first satisfiable Boolean formula in the sequence since the satisfiability of $\mathcal{F}(x)$ is monotonic in x.

3. Mutexes and mutex propagation

In this section, we explain the original idea of mutex propagation on planning graphs and generalize it to mutex propagation on MDDs for MAPF.

3.1. Mutex propagation on planning graphs

Planning graphs [17] are directed and leveled data structures containing two types of nodes, *proposition nodes* and *action nodes*, that are arranged into levels. Even-numbered levels contain only proposition nodes, while odd-numbered levels contain only action nodes. The zeroth level represents the start state. A planning graph edge connects a proposition node to an action node at the next level iff the proposition is a precondition of that action. A planning graph edge also connects an action node to a proposition node at the next level iff the proposition is made true by that action. The planning graph represents the effects of parallel actions, but it does so very loosely. To approximate the set of reachable states better, one uses mutex propagation on the planning graph with the following rules:

- Two action nodes at level i are mutex iff (1) the effect of one action is the negation of the effect of the other action; (2) one action deletes the precondition of the other action; or (3) there exists a precondition of one action and a precondition of the other action that are mutex at level $i-1$.
- Two proposition nodes at level i are mutex iff (1) one proposition is the negation of the other proposition; or (2) all actions at level $i-1$ that achieve one proposition are pairwise mutex with all actions at level $i-1$ that achieve the other proposition.

3.2. Mutex propagation on MDDs

In the context of MAPF, MDDs are directed and leveled data structures that resemble planning graphs. However, each action has a single precondition because each agent at each timestep can either wait at its current vertex u or traverse some edge $\langle u, v\rangle$, both with the single precondition of the agent being in vertex u at that timestep. Therefore, it is not necessary to represent the action layers explicitly, and a collection of MDDs built individually for each agent can be seen as a special case of a planning graph. In addition, MDD nodes carry information about reachability both from the start vertex and to the goal vertex. Therefore, unlike planning graphs, MDDs don't represent non-goal propositions in the last level. Similarly, the mutex propagation rules can also be simplified in the case of MDDs, as explained later, resulting in the following semantics: If two MDD nodes $n_i \in MDD_i^{l_i}$ and $n_j \in MDD_j^{l_j}$ at level t are mutex, then there do not exist conflict-free sub-paths that move agents a_i and a_j from their start vertices at timestep 0 to the vertices $n_i.loc$ and $n_j.loc$ at timestep t. Since mutex propagation can be done in polynomial time, the set of reachable vertices can be efficiently and tightly approximated, from which useful information can be derived for symmetry breaking and guiding the high-level search of CBS.

We define two types of *initial mutexes* corresponding to vertex and edge conflicts in MAPF, respectively:

- Two MDD nodes n_i and n_j are initial mutex iff they are of MDDs for different agents, $n_i.level = n_j.level$ and $n_i.loc = n_j.loc$.
- Two MDD edges $e_i = \langle n_i, n_i'\rangle$ and $e_j = \langle n_j, n_j'\rangle$ are initial mutex iff they are of MDDs for different agents, $e_i.level = e_j.level$, $n_i.loc = n_j'.loc$ and $n_j.loc = n_i'.loc$.

Example 2. Fig. 2 shows the MDDs for agents a_1 and a_2 on the cardinal rectangle conflict instance in Fig. 1a. The label of each MDD node is its vertex. Initial mutexes are represented with blue dashed arcs.

At level 1, MDD nodes B2 $\in MDD_1^5$ and B2 $\in MDD_2^5$ are initial mutex and thus connected by a blue dashed arc because both agents a_1 and a_2 staying in vertex B2 at timestep 1 causes a vertex conflict.

We define two types of *propagated mutexes* expressing our mutex propagation rules:

1. *Forward propagation for MDD nodes:* Two MDD nodes n_i and n_j are propagated mutex iff they are of MDDs for different agents, $n_i.level = n_j.level$ and all pairs of MDD edges e_i and e_j with $e_i.to = n_i$ and $e_j.to = n_j$ are either initial mutex or propagated mutex.
2. *Forward propagation for MDD edges:* Two MDD edges e_i and e_j are propagated mutex iff they are of MDDs for different agents, $e_i.level = e_j.level$ and MDD nodes $e_i.from$ and $e_j.from$ are either initial mutex or propagated mutex.

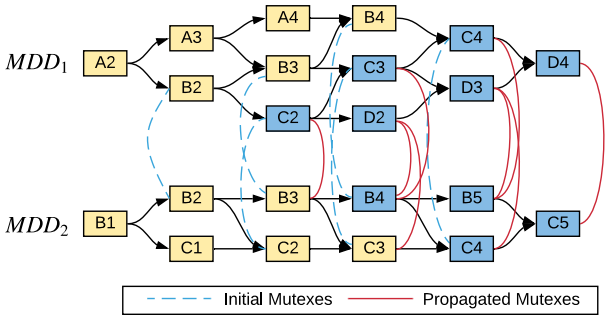


Fig. 2. Shows the MDDs for agents a_1 and a_2 and 6 levels each on the cardinal rectangle conflict instance in Fig. 1a along with the mutexes between their MDD nodes. Initial mutexes are represented with blue dashed arcs, and propagated mutexes are represented with red solid arcs. Edge mutexes are not shown here.

Algorithm 2: GENERATE-MUTEX: Determine all mutexes between two MDDs.

Input : Two MDDs $MDD_i^{l_i}$ and $MDD_j^{l_j}$.
Output: A set of mutexes M.

```
1  queue ← all initial mutexes between MDD_i^{l_i} and MDD_j^{l_j};
2  M ← ∅;
3  while queue is not empty do
4      m ← pop a mutex from queue with the smallest level, breaking ties in favor of mutexes between MDD nodes;
5      Add m to M;
6      if m is a mutex between MDD nodes then
7          ⟨n_i, n_j⟩ ← m;
8          foreach e_i such that e_i.from = n_i do
9              foreach e_j such that e_j.from = n_j do
10                 Add ⟨e_i, e_j⟩ to queue;
11             end
12         end
13     else // m is a mutex between MDD edges
14         ⟨e_i, e_j⟩ ← m;
15         n_i ← e_i.to;
16         n_j ← e_j.to;
17         is_propagated_mutex ← True;
18         foreach e'_i such that e'_i.to = n_i do
19             foreach e'_j such that e'_j.to = n_j do
20                 if ⟨e'_i, e'_j⟩ is not in M then
21                     is_propagated_mutex ← False;
22                 end
23             end
24         end
25         if is_propagated_mutex then
26             Add ⟨n_i, n_j⟩ to queue;
27         end
28     end
29 end
30 return M;
```

Example 3. Propagated mutexes between MDD nodes in Fig. 2 are represented with red solid arcs. Propagated mutexes between MDD edges are not shown. At Level 1, the MDD edges from B2 to C2 of MDD_1^5 and from B2 to B3 of MDD_2^5 are propagated mutex since B2 $\in MDD_1^5$ and B2 $\in MDD_2^5$ are initial mutex. At level 2, C2 $\in MDD_1^5$ and B3 $\in MDD_2^5$ have only one incoming MDD edge each, namely the MDD edges from B2 to C2 of MDD_1^5 and from B2 to B3 of MDD_2^5. Thus, these MDD nodes are propagated mutex and connected by a red solid arc.

We define two MDD nodes or two MDD edges to be *mutex* iff they are initial mutex or propagated mutex. We use Algorithm 2 to find all mutexes between two MDDs. The algorithm is similar to AC-3 [34]. The pseudocode is only for illustrating the general idea and not intended to be efficient. We add all initial mutexes to a queue (Line 1) and check all mutexes in the order of their levels to determine whether the mutex can be propagated. The propagated mutexes are then added to the queue (Lines 10 and 26). For mutexes at the same level, we break ties in favor of mutexes between MDD

nodes (Line 4). Therefore, at the same level, we first check mutexes between MDD nodes and then mutexes between MDD edges.

Property 1. *If two MDD nodes $n_i \in MDD_i^{l_i}$ and $n_j \in MDD_j^{l_j}$ with $n_i.level = n_j.level = l$ are mutex, then there does not exist a pair of conflict-free sub-paths p_i and p_j for agents a_i and a_j, respectively, such that sub-path p_i begins at start vertex s_i at timestep 0 and reaches vertex $n_i.loc$ at timestep l and sub-path p_j begins at start vertex s_j at timestep 0 and reaches vertex $n_j.loc$ at timestep l.*

Proof. The property is trivially true if MDD nodes n_i and n_j are initial mutex. For a proof of the property by contradiction if MDD nodes n_i and n_j are propagated mutex, assume that there exist two such sub-paths p_i and p_j that are conflict-free. Define $n_{i,t}$ as the MDD node corresponding to the vertex of agent a_i at timestep t when it follows sub-path p_i. Similarly, define $n_{j,t}$ as the MDD node corresponding to the vertex of agent a_j at timestep t when it follows sub-path p_j. By definition, $n_{i,0}.loc = s_i$, $n_{j,0}.loc = s_j$, $n_{i,l} = n_i$ and $n_{j,l} = n_j$. Using induction, we now prove the contradiction that MDD nodes n_i and n_j are not propagated mutex. In the base case, MDD nodes $n_{i,0}$ and $n_{j,0}$ are not mutex because $s_i \neq s_j$. Assume that MDD nodes $n_{i,t}$ and $n_{j,t}$ are not propagated mutex for timestep $t < l$. MDD nodes $n_{i,t}$ and $n_{j,t}$ are not initial mutex because sub-paths p_i and p_j are conflict-free. We define MDD edge e_i as $\langle n_{i,t}, n_{i,t+1}\rangle$ and MDD edge e_j as $\langle n_{j,t}, n_{j,t+1}\rangle$. MDD edges e_i and e_j are not initial mutex because sub-paths p_i and p_j are conflict-free. MDD edges e_i and e_j are not propagated mutex either because their source MDD nodes $n_{i,t}$ and $n_{j,t}$ are not mutex. This implies that MDD edges $n_{i,t+1}$ and $n_{j,t+1}$ are not propagated mutex. By induction, MDD nodes n_i and n_j are not propagated mutex, which contradicts the assumption. □

Property 2. *If two MDD edges $\langle n_i, n_i'\rangle \in MDD_i^{l_i}$ and $\langle n_j, n_j'\rangle \in MDD_j^{l_j}$ with $n_i.level = n_j.level = l$ are mutex, then there does not exist a pair of conflict-free sub-paths p_i and p_j for agents a_i and a_j, respectively, such that sub-path p_i begins at start vertex s_i at timestep 0 and traverses edge $\langle n_i.loc, n_i'.loc\rangle$ between timesteps l and $l+1$ and sub-path p_j begins at start vertex s_j at timestep 0 and traverses edge $\langle n_j.loc, n_j'.loc\rangle$ between timesteps l and $l+1$.*

Proof. If the MDD edges $\langle n_i, n_i'\rangle$ and $\langle n_j, n_j'\rangle$ are initial mutex, then there must exist an edge conflict between sub-paths p_i and p_j between timesteps l and $l+1$. If the MDD edges $\langle n_i, n_i'\rangle$ and $\langle n_j, n_j'\rangle$ are propagated mutex, then, by definition, MDD nodes n_i and n_j are mutex. From Property 1, there must exist a vertex or edge conflict between sub-paths p_i and p_j at or before timestep l. □

Property 3. *If two MDD nodes $n_i \in MDD_i^{l_i}$ and $n_j \in MDD_j^{l_j}$ with $n_i.level = n_j.level = l$ are not mutex, then there exists a pair of conflict-free sub-paths p_i and p_j for agents a_i and a_j, respectively, such that sub-path p_i begins at start vertex s_i at timestep 0 and reaches vertex $n_i.loc$ at timestep l and sub-path p_j begins at start vertex s_j at timestep 0 and reaches vertex $n_j.loc$ at timestep l.*

Proof. Because MDD nodes n_i and n_j are not mutex, there exists a pair of their incoming MDD edges that are not mutex. Therefore, the source MDD nodes of these two MDD edges are not mutex either. Continuing this backward induction, we can construct the desired conflict-free sub-paths. □

Theorem 1 combines Properties 1 and 3, and in doing so, it characterizes the joint-reachable states of two agents.

Theorem 1. *Two MDD nodes $n_i \in MDD_i^{l_i}$ and $n_j \in MDD_j^{l_j}$ with $n_i.level = n_j.level = l$ are not mutex iff there exists a pair of conflict-free sub-paths p_i and p_j for agents a_i and a_j, respectively, such that sub-path p_i begins at start vertex s_i at timestep 0 and reaches vertex $n_i.loc$ at timestep l and sub-path p_j begins at start vertex s_j at timestep 0 and reaches vertex $n_j.loc$ at timestep l.*

For Theorem 1 to hold, l_i and l_j do not have to be the individual minimum costs of agents a_i and a_j, respectively.

Example 4. In Fig. 2, MDD nodes D4 $\in MDD_1^5$ and C5 $\in MDD_2^5$ at level 5 are propagated mutex. From Theorem 1, there does not exist a pair of conflict-free sub-paths for agents a_1 and a_2 such that both agents arrive at their respective goal vertices at timestep 5, which means that there does not exist a pair of conflict-free paths of cost 5 for the agents. Therefore, by definition, the conflict between them is cardinal.

4. Mutex propagation in CBS

In this section, we show how mutex propagation can be integrated into CBS. We show how mutex propagation can be used to identify and resolve cardinal and semi-cardinal conflicts, and we describe the resulting variant of CBS.

4.1. Identifying and resolving cardinal conflicts

In this section, we present algorithms for identifying and resolving cardinal conflicts and an algorithm for determining the number of levels of the MDDs needed to identify and resolve cardinal conflicts.

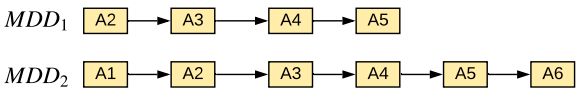


Fig. 3. Shows the MDDs for agents a_1 and a_2, with 4 and 6 levels, respectively, on the cardinal target conflict instance in Fig. 1c. There are no mutexes between the two MDDs.

Algorithm 3: CLASSIFY-CONFLICT: Determine whether agents a_i and a_j have a cardinal conflict within costs (l_i, l_j).

Input : Two MDDs $MDD_i^{l_i}$ and $MDD_j^{l_j}$ with $l_i \leq l_j$.
Output: The conflict type of agents a_i and a_j, which is PC, AC or NC.

1 $N_j^{'} \leftarrow$ MDD nodes at level l_i of $MDD_j^{l_j}$ that are not mutex with the sink MDD node of $MDD_i^{l_i}$;
2 **if** $N_j^{'} = \emptyset$ **then**
3 | **return** *PC*;
4 **end**
5 **foreach** $n_j \in N_j^{'}$ **do**
6 | **if** *there exists a sub-path in* $MDD_j^{l_j}$ *from* n_j *to its sink MDD node that does not traverse any MDD node n with* $n.loc = g_i$ *and* $n.level > l_i$ **then**
7 | | **return** *NC*;
8 | **end**
9 **end**
10 **return** *AC*;

4.1.1. Identifying cardinal conflicts

In Example 4, we show how mutex propagation can be used to identify cardinal conflicts between two MDDs with the same number of levels. Here, we show how to generalize this technique to identify cardinal conflicts between any two MDDs. This generalization requires us to handle the corner case where an agent can terminally wait at its goal vertex: Two agents with different individual minimum costs can have vertex conflicts after one agent terminally waits at its goal vertex, as shown in Fig. 3. These conflicts are not captured by mutexes. Algorithm 2 does not generate any mutexes for these two MDDs, but there exists a cardinal conflict because the individual minimum-cost path of agent a_2 traverses the goal vertex of agent a_1 after agent a_1 terminally waits at it. To be able to identify and resolve such cardinal conflicts as well, we distinguish two classes of cardinal conflicts:

Pre-goal cardinal conflict (PC) within costs (l_i, l_j)*:* There does not exist a pair of conflict-free paths for agents a_1 and a_2 with costs l_i and l_j, respectively, satisfying all constraints of the CT node even if we do not consider conflicts that happen after one agent terminally waits at its goal vertex.

After-goal cardinal conflict (AC) within costs (l_i, l_j)*:* There exists at least one pair of conflict-free paths for agents a_1 and a_2 with costs l_i and l_j, respectively, satisfying all constraints of the CT node if we do not consider conflicts that happen after one agent terminally waits at its goal vertex. However, for every such pair of paths, one agent traverses the goal vertex of the other agent after the other agent terminally waits at its goal vertex.

Given two agents a_i and a_j and their MDDs $MDD_i^{l_i}$ and $MDD_j^{l_j}$, we use Algorithm 3 to determine whether these agents have a cardinal conflict within costs (l_i, l_j). Algorithm 3 returns PC, AC or NC ("Not a Cardinal conflict"). Without loss of generality, we assume that $l_i \leq l_j$ throughout this article. Algorithm 3 first checks whether all MDD nodes of $MDD_j^{l_j}$ at level l_i are mutex with the sink MDD node of $MDD_i^{l_i}$. If so, then it classifies the conflict as a PC. Otherwise, it checks whether there exists an MDD node $n_j \in MDD_j^{l_j}$ at level l_i that is not mutex with the sink MDD node of $MDD_i^{l_i}$ and a sub-path in $MDD_j^{l_j}$ from MDD node n_j to its sink MDD node that does not traverse any MDD node with vertex g_i, that is, the goal vertex of agent a_i (lines 5-6). If so, it classifies the conflict as an NC. If such an MDD node and sub-path do not exist, then it classifies the conflict as an AC.

Theorem 2. *Algorithm 3 returns* NC *for given* $MDD_i^{l_i}$ *and* $MDD_j^{l_j}$ *iff there exists a pair of conflict-free paths* p_i *and* p_j *for agents* a_i *and* a_j *with costs* l_i *and* l_j*, respectively.*

Proof. First, assume that there exist such conflict-free paths p_i and p_j with costs l_i and l_j, respectively. From Theorem 1, the MDD node $n_j \in MDD_j^{l_j}$ corresponding to the vertex of agent a_j at timestep l_i is not mutex with the sink MDD node of $MDD_i^{l_i}$. Therefore, MDD node set $N_j^{'}$ in Algorithm 3 is not empty. Since paths p_i and p_j are conflict-free, agent a_j following

Algorithm 4: GENERATE-CONSTRAINTS-PC: Generate constraints for PCs.

Input : Two MDDs $MDD_i^{l_i}$ and $MDD_j^{l_j}$.
Output: Constraint set C_i for agent a_i and constraint set C_j for agent a_j.

1 $C_i \leftarrow$ constraints on every MDD node of $MDD_i^{l_i}$ that is mutex with all MDD nodes of $MDD_j^{l_j}$ at the same level;
2 $C_j \leftarrow$ constraints on every MDD node of $MDD_j^{l_j}$ that is mutex with all MDD nodes of $MDD_i^{l_i}$ at the same level;
3 **return** $\langle C_i, C_j \rangle$;

p_j does not traverse vertex g_i at or after timestep l_i. Thus, there exists a sub-path in $MDD_j^{l_j}$ from MDD node n_j to its sink MDD node that does not traverse any MDD node with vertex g_i, and Algorithm 3 returns NC.

Now assume that Algorithm 3 returns NC. From Line 6 of Algorithm 3, there exists a sub-path p in $MDD_j^{l_j}$ from an MDD node $n_j \in N'_j$ to its sink MDD node that does not traverse any MDD node with vertex g_i. From Line 1 of Algorithm 3, MDD node n_j is not mutex with the sink MDD node of $MDD_i^{l_i}$. From Theorem 1, there exists a pair of conflict-free sub-paths p_i and p_j for agents a_i and a_j, respectively, such that sub-path p_i begins at start vertex s_i at timestep 0 and reaches goal vertex g_i at timestep l_i and sub-path p_j begins at start vertex s_j at timestep 0 and reaches vertex $n_j.loc$ at timestep l_i. If agent a_i follows sub-path p_i until timestep l_i and then terminally waits at goal vertex g_i and agent a_j follows sub-path p_j until timestep l_i, then follows sub-path p and terminally waits at goal vertex g_j, then these two paths are conflict-free and of costs l_i and l_j, respectively. □

For Theorem 2 to hold, l_i and l_j do not have to be the individual minimum costs of agents a_i and a_j, respectively.

Corollary 1. *Algorithm 3 returns* PC *or* AC *for given* $MDD_i^{l_i^\star}$ *and* $MDD_j^{l_j^\star}$ *iff agents* a_i *and* a_j *have a cardinal conflict, where* $l_i^\star$ *and* $l_j^\star$ *are the individual minimum costs of agents* a_i *and* a_j*, respectively.*

Proof. Algorithm 3 returns either *PC*, *NC*, or *AC*. From Theorem 2, Algorithm 3 returns *NC* for given $MDD_i^{l_i^\star}$ and $MDD_j^{l_j^\star}$ iff there exists a pair of conflict-free paths p_i and p_j for agents a_i and a_j with costs $l_i^\star$ and $l_j^\star$, respectively. Therefore, Algorithm 3 returns *PC* or *AC* for given $MDD_i^{l_i^\star}$ and $MDD_j^{l_j^\star}$ iff such paths p_i and p_j do not exist, and thus, by definition, agents a_i and a_j have a cardinal conflict. □

4.1.2. Resolving cardinal conflicts

For given $MDD_i^{l_i}$ and $MDD_j^{l_j}$ for which CLASSIFY-CONFLICT returns PC, we use Algorithm 4 to generate the constraint sets C_i and C_j for agents a_i and a_j, respectively. Constraint set C_i contains the constraints on every MDD node of $MDD_i^{l_i}$ that is mutex with all MDD nodes of $MDD_j^{l_j}$ at the same level. Similarly, constraint set C_j contains the constraints on every MDD node of $MDD_j^{l_j}$ that is mutex with all MDD nodes of $MDD_i^{l_i}$ at the same level.

Property 4. *In case Algorithm 3 returns* PC *for given* $MDD_i^{l_i}$ *and* $MDD_j^{l_j}$*, for constraint sets* C_i *and* C_j *generated by Algorithm 4, if the path* p_i *of agent* a_i *violates a constraint in* C_i*, and the path* p_j *of agent* a_j *violates a constraint in* C_j*, then the two paths are not conflict-free.*

Proof. Define $\langle a_i, t_i, v_i \rangle$ and $\langle a_j, t_j, v_j \rangle$ as the two constraints violated by paths p_i and p_j, respectively. If $t_i \leq t_j$, define $n_j \in MDD_j^{l_j}$ as the MDD node corresponding to the vertex of p_j at timestep t_i. From Line 1 of Algorithm 4, the MDD node $n_i \in MDD_i^{l_i}$ with $n_i.loc = v_i$ and $n_i.level = t_i$ is mutex with MDD node n_j. From Theorem 1, paths p_i and p_j are not conflict-free. A similar proof works for $t_i \geq t_j$. □

For given $MDD_i^{l_i}$ and $MDD_j^{l_j}$ for which CLASSIFY-CONFLICT returns AC, we use Algorithm 5 to generate the constraint sets C_i and C_j for agents a_i and a_j, respectively, making use of the following two *cost constraints* [23]:

1. $\underline{\langle a_i, l \rangle}$ forces the path cost of agent a_i to be larger than l.
2. $\langle a_i, l \rangle$ forces the path cost of agent a_i to be less than or equal to l.

These two cost constraints can be implemented easily by changing the termination condition in the A* algorithm that CBS uses to find individual minimum-cost paths: If constraint $\langle a_i, l \rangle$ is provided, the low-level search for a_i terminates only if it expands a node that is associated with location g_i and its timestep is larger than l; If constraint $\underline{\langle a_i, l \rangle}$ is provided, the

Algorithm 5: GENERATE-CONSTRAINTS-AC: Generate constraints for ACs.

Input : Two MDDs $MDD_i^{l_i}$ and $MDD_j^{l_j}$ with $l_i \leq l_j$.
Output: Constraint set C_i for agent a_i and constraint set C_j for agent a_j.
1 $C_i \leftarrow$ {cost constraint $\langle a_i, l_i \rangle$};
2 $N_j \leftarrow$ MDD nodes of $MDD_j^{l_j}$ at level l_i that are mutex with the sink MDD node of $MDD_i^{l_i}$;
3 $N_{AC} \leftarrow$ MDD nodes $n \in MDD_j^{l_j}$ with $n.loc = g_i$ and $n.level > l_i$;
4 $C_j \leftarrow$ {cost constraint $\overline{\langle a_i, l_i \rangle}$} $\cup$ {constraints on all MDD nodes in $N_j \cup N_{AC}$};
5 **return** $\langle C_i, C_j \rangle$;

low-level search for a_i prunes any node that has f-value larger than l (since all paths represented by such a node only reach g_i after timestep l). Constraint set C_i contains only the cost constraint $\langle a_i, l_i \rangle$. Constraint set C_j contains the cost constraint $\overline{\langle a_i, l_i \rangle}$, the constraints on all MDD nodes of $MDD_j^{l_j}$ at level l_i that are mutex with the sink MDD node of $MDD_i^{l_i}$ and the constraints on all MDD nodes $n \in MDD_j^{l_j}$ with $n.loc = g_i$ and $n.level > l_i$. Cost constraint $\overline{\langle a_i, l_i \rangle}$ implies that no other agent can use g_i after timestep l, and, therefore it also applies to agent a_j.

Property 5. *In case Algorithm 3 returns* AC *for given $MDD_i^{l_i}$ and $MDD_j^{l_j}$, for constraint sets C_i and C_j generated by Algorithm 5, if the path p_i of agent a_i violates a constraint in C_i, and the path p_j of agent a_j violates a constraint in C_j, then the two paths are not conflict-free.*

Proof. Constraint set C_i contains the cost constraint that the path cost of agent a_i should be larger than l_i. If agent a_i violates this cost constraint, then agent a_i occupies goal vertex g_i at timestep l_i and afterward. Constraint set C_j contains the vertex constraints on all MDD nodes in MDD node set $N_j \cup N_{AC}$. MDD node set N_{AC} contains all those MDD nodes of $MDD_j^{l_j}$ at levels larger than l_i whose vertices are the goal vertex g_i. If agent a_j violates the constraint on an MDD node $n \in N_{AC}$, it must have a conflict with agent a_i because agent a_i occupies goal vertex g_i at timestep $n.level$. MDD node set N_j contains all those MDD nodes of $MDD_j^{l_j}$ that are mutex with the sink MDD node of $MDD_i^{l_i}$. If agent a_j violates the constraint on an MDD node $n \in N_j$, it must have a conflict with agent a_i at or before timestep l_i because agent a_i occupies goal vertex g_i at timestep l_i, according to Theorem 1. □

Properties 4 and 5 ensure that the constraint sets generated by Algorithms 4 and 5 do not rule out any pairs of conflict-free paths for their agents. For them to hold, l_i and l_j do not have to be the individual minimum costs of agents a_i and a_j, respectively.

Property 6. *Algorithms 4 and 5 generate constraint sets that increase the individual minimum costs of their agents.*

Proof. If Algorithm 3 outputs PC, then constraint set $C_k, k \in \{i, j\}$, contains constraints on all MDD nodes of $MDD_k^{l_k}$ at level l_i because the sink MDD node of $MDD_i^{l_i}$ is mutex with all MDD nodes of $MDD_j^{l_j}$ at level l_i. Therefore, any path of agent a_k satisfying the constraints of constraint set C_k must have a cost of at least $l_k + 1$.

If Algorithm 3 outputs AC, then constraint set C_i contains only the cost constraint $\langle a_i, l_i \rangle$. Therefore, any path of agent a_i satisfying the constraints of constraint set C_i must have a cost of at least $l_i + 1$. For agent a_j, constraint set C_j contains constraints on all MDD nodes in MDD node sets N_j and N_{AC}. We prove by contradiction that there does not exist a path for agent a_j of cost less than or equal to l_j. Assume that such a path p exists. We define N'_j as the set of MDD nodes of $MDD_j^{l_j}$ at level l_i that are not mutex with the sink MDD node of $MDD_i^{l_i}$. Path p must traverse the vertex of an MDD node in MDD node set N'_j at timestep l_i because the constraint set for agent a_j contains vertex constraints for the vertices of the other MDD nodes of $MDD_j^{l_j}$ at timestep l_i. Because Algorithm 3 outputs AC, from Line 6, all sub-paths in $MDD_j^{l_j}$ from any MDD node in MDD node set N'_j to the sink MDD node of $MDD_j^{l_j}$ traverse some MDD node in MDD node set N_{AC}, but the constraint set for agent a_j contains a vertex constraint for the vertex of each MDD node in MDD node set N_{AC} at the timestep that corresponds to its level. Therefore, such a path p does not exist. □

Property 6 ensures that, in every child CT node generated with the new constraints in the constraint sets from Algorithms 4 or 5, the individual minimum cost of at least one agent increases. All other individual minimum costs stay the same. Therefore, the SoC of that child CT node is larger than the SoC of its parent CT node.

Algorithm 6: GENERATE-CONSTRAINTS-C: Generate constraints for cardinal conflicts.

Input : Two agents a_i and a_j with $l_i^\star \leq l_j^\star$ and CLASSIFY-CONFLICT($MDD_i^{l_i^\star}$, $MDD_j^{l_j^\star}$) $\neq$ NC.
Output: Constraint set C_i for agent a_i and constraint set C_j for agent a_j.
1 $d_i \leftarrow 0$;
2 $d_j \leftarrow 0$;
3 **while** CLASSIFY-CONFLICT($MDD_i^{l_i^\star+d_i+1}$, $MDD_j^{l_j^\star+d_j+1}$) $\neq NC$ **do**
4 $\quad d_i \leftarrow d_i + 1$;
5 $\quad d_j \leftarrow d_j + 1$;
6 **end**
7 **while** CLASSIFY-CONFLICT($MDD_i^{l_i^\star+d_i+1}$, $MDD_j^{l_j^\star+d_j}$) $\neq NC$ **do**
8 $\quad d_i \leftarrow d_i + 1$;
9 **end**
10 **if** CLASSIFY-CONFLICT($MDD_i^{l_i^\star+d_i}$, $MDD_j^{l_j^\star+d_j}$) = PC **then**
11 $\quad$ **return** GENERATE-CONSTRAINTS-PC($MDD_i^{l_i^\star+d_i}$, $MDD_j^{l_j^\star+d_j}$);
12 **else** // CLASSIFY-CONFLICT returns AC
13 $\quad$ **return** GENERATE-CONSTRAINTS-AC($MDD_i^{l_i^\star+d_i}$, $MDD_j^{l_j^\star+d_j}$);
14 **end**

Moreover, Properties 4-6 do not rely on l_i and l_j being the individual minimum costs of agents a_i and a_j, respectively. Therefore, we can pick any l_i and l_j as long as agents a_i and a_j have a cardinal conflict within costs (l_i, l_j), that is, Algorithm 3 returns AC or PC for given $MDD_i^{l_i}$ and $MDD_j^{l_j}$.

In practice, to keep the sizes of the constraint sets C_i and C_j small (which could reduce the runtime of the low-level search of CBS), we remove the constraints on all such MDD nodes n from the constraint set C_k, $k \in \{i, j\}$, if the constraints on all predecessors of MDD node n in the MDD are also in constraint set C_k. Such constraints are redundant because the agent cannot reach vertex $n.loc$ at timestep $n.level$.

Example 5. In Fig. 2, the constraints generated by Algorithm 4 are those of the MDD nodes which are filled with solid blue. After removing redundancies, the constraint set C_1 for agent a_1 contains constraints $\langle a_1, 2, C2\rangle$, $\langle a_1, 3, C3\rangle$, and $\langle a_1, 4, C4\rangle$, while the constraint set C_2 for agent a_2 contains constraints $\langle a_2, 3, B4\rangle$ and $\langle a_2, 4, C4\rangle$. These two constraint sets are exactly the barrier constraints for this cardinal rectangle conflict.

Overall, our way of generating constraints for cardinal conflicts guarantees the optimality and completeness of CBS, since the proof of Theorem 2 in [22] applies.

4.1.3. Determining the numbers of levels of MDDs

For some cardinal conflicts, the minimum SoC of conflict-free paths for the two conflicting agents is much larger than the sum of their individual minimum costs. If we generate constraints using MDDs for the conflicting agents whose numbers of levels are the respective individual minimum costs, CBS still needs to expand multiple CT nodes to resolve all conflicts between the conflicting agents. An example is a cardinal corridor conflict where, in any optimal solution, one agent needs to wait for a certain number k of timesteps to allow the other agent to traverse the corridor. If Algorithms 4 and 5 use MDDs whose numbers of levels are the respective individual minimum costs, the constraints generated by them can increase the cost of either agent by only one. Therefore, CBS needs to increase the CT to a depth of at least k to find an optimal solution. Without heuristic guidance, CBS thus needs to expand $\omega(2^k)$ CT nodes.

To resolve cardinal conflicts efficiently when the agents need to increase the sum of their individual minimum costs by more than one, we aggressively increase the numbers of levels of their MDDs before using them to generate the constraint sets. We can use Algorithms 4 and 5 as long as agents a_i and a_j have a cardinal conflict within costs (l_i, l_j), that is, Algorithm 3 returns AC or PC for given $MDD_i^{l_i}$ and $MDD_j^{l_j}$. Algorithm 6 shows a way to determine the numbers of levels which satisfy this requirement. It first tries to increase l_i and l_j simultaneously, and then tries to increase only l_i. As shown in Algorithm 6, we begin with $d_i = d_j = 0$ and increase d_i and d_j simultaneously until CLASSIFY-CONFLICT($MDD_i^{l_i^\star+d_i+1}$, $MDD_j^{l_j^\star+d_j+1}$) returns NC (Lines 3-6). We then increase only d_i until CLASSIFY-CONFLICT($MDD_i^{l_i^\star+d_i+1}$, $MDD_j^{l_j^\star+d_j}$) returns NC (Lines 7-9). Finally, $MDD_i^{l_i^\star+d_i}$ and $MDD_j^{l_j^\star+d_j}$ are used to generate the constraint sets on Lines 10-14.

Example 6. In any optimal solution of the cardinal corridor conflict instance in Fig. 1b, one of the two agents has a cost of 11. In the root CT node, the individual minimum costs are $l_i^\star = l_j^\star = 6$. CLASSIFY-CONFLICT in Algorithm 6 returns *PC* for

up to and including $d_i = 4$ and $d_j = 4$ but not for $d_i = 5$ and $d_j = 4$ or for $d_i = 5$ and $d_j = 5$. Therefore, Algorithm 6 uses MDD_1^{10} and MDD_2^{10} to generate the constraint sets. After removing redundancies, the constraint set for agent a_1 is $C_1 = \{\langle a_1, 5, B5\rangle, \langle a_1, 6, B4\rangle, \langle a_1, 6, B5\rangle, \langle a_1, 7, B3\rangle, \langle a_1, 7, B4\rangle\}$, which prevents the agent from arriving in its goal vertex $C5$ before timestep 11. Similarly, the constraint set for agent a_2 is $C_2 = \{\langle a_2, 5, B1\rangle, \langle a_2, 6, B2\rangle, \langle a_2, 6, B1\rangle, \langle a_2, 7, B3\rangle, \langle a_2, 7, B2\rangle\}$, which prevents the agent from arriving in its goal vertex $C1$ before timestep 11. Conceptually, the constraint set for each agent prevents the agent from leaving the corridor before some timestep. If we use the approach in [23] to resolve this corridor conflict, the constraint set for a_1 would contain vertex constraints on $B5$ at timesteps 5-9. The constraint set generated by mutex reasoning also prevents agent a_1 from reaching $B5$ at timesteps 5-9 indirectly and has some extra vertex constraints inside the corridor.

We also investigated alternative ways of choosing the numbers of MDD levels:

- **A1** We begin with $d_i = d_j = 0$ and, in each iteration, randomly choose d_i or d_j. If d_i is chosen, increase d_i by one if Classify-Conflict($MDD_i^{l_i^\star+d_i+1}, MDD_j^{l_j^\star+d_j}$) returns NC. Otherwise, if d_j is chosen, increase d_j by one if Classify-Conflict($MDD_i^{l_i^\star+d_i}, MDD_j^{l_j^\star+d_j+1}$) returns NC. We repeat this procedure until both Classify-Conflict($MDD_i^{l_i^\star+d_i+1}$, $MDD_j^{l_j^\star+d_j}$) and Classify-Conflict($MDD_i^{l_i^\star+d_i}, MDD_j^{l_j^\star+d_j+1}$) return NC. Finally, $MDD_i^{l_i^\star+d_i}$ and $MDD_j^{l_j^\star+d_j}$ are used to generate the constraint sets.
- **A2** We begin with $d = d_i = d_j = 0$ and increase d until Classify-Conflict($MDD_i^{l_i^\star+d+1}, MDD_j^{l_j^\star+d+1}$) returns NC. We then increase only d_i until Classify-Conflict($MDD_i^{l_i^\star+d+d_i+1}, MDD_j^{l_j^\star+d}$) returns NC. We then increase only d_j until Classify-Conflict($MDD_i^{l_i^\star+d}, MDD_j^{l_j^\star+d+d_j+1}$) returns NC. Finally, if $d_i > d_j$, $MDD_i^{l_i^\star+d+d_i}$ and $MDD_j^{l_j^\star+d}$ are used to generate the constraint sets. Otherwise, $MDD_i^{l_i^\star+d}$ and $MDD_j^{l_j^\star+d+d_j}$ are used to generate the constraint sets.

A1 is an alternative way to which we add randomness, and A2 is similar to Algorithm 6 except that, after simultaneously increasing l_i and l_j, it chooses and increases the one of l_i and l_j which allows larger increasing of MDD lengths. Our experimental results show that all three ways of choosing the numbers of MDD levels perform similarly across different MAPF instances. Therefore, we use Algorithm 6 exclusively in this article.

4.2. Identifying and resolving semi-cardinal conflicts

In this section, we present the algorithm for identifying and resolving semi-cardinal conflicts and an algorithm for determining the needed number of levels of the MDDs. Since our proposed algorithm for doing so is straightforward, we omit presenting it in pseudo-code but describe it using text. Let p_j denote the path of agent a_j in the CT node. Our algorithm identifies the following cases of semi-cardinal conflicts: (a) There does not exist a path p_i for agent a_i that is conflict-free with p_j since p_j traverses g_i at timestep $t \geq l_i$; (b) There does not exist a path p_i for agent a_i that is conflict-free with p_j since p_j traverses a specific vertex at timestep $t \leq l_i$; and (c) There does not exist a path p_i for agent a_i that is conflict-free with p_j since p_j traverses a specific edge at timestep t. Our algorithm uses $MDD_i^{l_i}$ and $MDD_j^{l_j^\star}$ to identify and resolve semi-cardinal conflicts between agents a_i and a_j within cost l_i as follows:

1. If p_j traverses goal vertex g_i at timestep $t > l_i$, the constraint sets C_i and C_j for agents a_i and a_j, respectively, are the same as the constraint sets for target conflicts in goal vertex g_i at timestep t [23].
2. If p_j traverses a vertex at timestep $t \leq l_i$ and the corresponding MDD node of $MDD_j^{l_j^\star}$ at level t is mutex with all MDD nodes of $MDD_i^{l_i}$ at level t, the constraint set C_i for agent a_i is the set of constraints on all MDD nodes of $MDD_i^{l_i}$ at level t. The constraint set C_j for agent a_j is the set of constraints on all MDD nodes of $MDD_j^{l_j^\star}$ at level t that are mutex with all MDD nodes of $MDD_i^{l_i}$ at level t.
3. If p_j traverses an edge between timesteps $t < l_i$ and $t+1$ and the corresponding MDD edge of $MDD_i^{l_j^\star}$ at level t is mutex with all MDD edges of $MDD_i^{l_i}$ at level t, the constraint set C_i for agent a_i is the set of constraints on all MDD edges of $MDD_i^{l_i}$ at level t. The constraint set C_j for agent a_j is the set of constraints on all MDD edges of $MDD_j^{l_j^\star}$ at level t that are mutex with all MDD edges of $MDD_i^{l_i}$ at level t.

Property 7. *In case our algorithm identifies a semi-cardinal conflict for given $MDD_i^{l_i}$ and $MDD_j^{l_j}$, for constraint sets C_i and C_j generated by our algorithm, if the path p_i of agent a_i violates a constraint in C_i and the path p_j of agent a_j violates a constraint in C_j, then the two paths are not conflict-free.*

Proof. For Case 1, the proof of Property 5 applies, except that the path of agent a_j is now fixed. For Case 2, the proof of Property 4 applies under the same conditions. For Case 3, define $\langle a_i, t, v_i, v_i' \rangle$ and $\langle a_j, t, v_j, v_j' \rangle$ as the two constraints violated by paths p_i and p_j, respectively. From Property 2, paths p_i and p_j are not conflict-free. □

Property 7 ensures that the constraint sets generated by our algorithm don't rule out any pairs of conflict-free paths for the corresponding agents. Moreover, the property does not rely on l_i being the individual minimum cost of agent a_i. Therefore, we can pick any l_i as long as agent a_i has a semi-cardinal conflict within cost l_i with agent a_j. Thus, we begin with $d = 0$ and increase d until our algorithm no longer identifies a semi-cardinal conflict within cost $l_i^\star + d + 1$. Finally, $MDD_i^{l_i^\star+d}$ and $MDD_j^{l_j^\star}$ are used to generate the constraint sets.

Overall, our way of generating constraint sets for semi-cardinal conflicts guarantees the optimality and completeness of CBS, since the proof of Theorem 2 in [22] applies.

4.3. CBS with mutex propagation

In this section, we describe how to incorporate our mutex propagation techniques into CBS, as shown in Algorithm 1.

Before choosing a conflict on Line 10, CBS now uses our techniques to identify cardinal and semi-cardinal conflicts. As before, the first priority is given to cardinal conflicts, followed by semi-cardinal conflicts, and then to non-cardinal conflicts. It then uses our techniques to determine the constraint sets to be used on Line 13.

While generating the constraint sets for resolving the conflicts, CBS automatically determines the new individual minimum costs of the corresponding agents: If CBS chooses a cardinal conflict between agents a_i and a_j within costs $\langle l_i, l_j \rangle$, then the individual minimum cost of agent a_i changes from $l_i^\star$ to $l_i + 1$ in one child CT node and the individual minimum cost of agent a_j changes from $l_j^\star$ to $l_j + 1$ in the other child CT node. If CBS chooses a semi-cardinal conflict between agents a_i and a_j within cost l_j, then the individual minimum cost of agent a_j changes from $l_j^\star$ to $l_j + 1$ in one child CT node. In both cases, all other individual minimum costs stay the same, that is, the increase in the individual minimum cost of the affected agent is identical to the increase in the SoC from the parent CT node to the child CT node. Larger increases might allow the high-level search of CBS to terminate earlier. Thus, in the high-level search, if CBS chooses a cardinal conflict, it chooses one with the largest $min(l_i - l_i^\star, l_j - l_j^\star)$ among all cardinal conflicts, breaking ties in favor of the largest $max(l_i - l_i^\star, l_j - l_j^\star)$. If CBS chooses a semi-cardinal conflict, it chooses one with the largest $l_j - l_j^\star$.

Our mutex propagation techniques incur a computational overhead per CT node expansion. To keep this overhead small, we implemented CBS to cache the MDDs and the constraints for cardinal conflicts in two hash tables, using the indices of the agents and the sets of all constraints imposed on them as keys. We didn't implement CBS to cache the constraints for semi-cardinal conflicts since this requires including the path of one of the two agents in the keys. Therefore, the cached semi-cardinal constraints are reused less often than the cached cardinal constraints.

5. Mutex propagation in MDD-SAT

In this section, we describe how mutex propagation can be incorporated into MDD-SAT and the advantages that result thereof.

MDD-SAT already uses MDDs to construct the Boolean formula $\mathcal{F}(x)$ in conjunctive normal form for which an off-the-shelf SAT solver determines the satisfiability. Thus, it is easy to integrate knowledge about pairs of MDD nodes that are mutex with each other into the construction of $\mathcal{F}(x)$. If MDD nodes n_i and n_j in the MDDs for agents a_i and a_j are mutex at level t, we simply add the binary clause $(\neg\mathcal{X}_{n_i} \vee \neg\mathcal{X}_{n_j})$ to $\mathcal{F}(x)$, which states that the two agents cannot be in vertices $n_i.loc$ and $n_j.loc$, respectively, at timestep t.

5.1. Mutex propagation versus unit propagation in MDD-SAT

In this section, we study the power of mutex propagation in MDD-SAT by comparing it to unit propagation (UP) [35], a standard technique implemented in SAT solvers. UP is a form of resolution inference that is used by SAT solvers to extend a partial consistent assignment of truth values to Boolean variables, as follows: If there exists a clause in which all literals but one are *FALSE* in the partial assignment, then the last literal must be *TRUE* for the clause as well as $\mathcal{F}(x)$ to be *TRUE*.

Mutexes are expressed as binary clauses, such as $(\neg\mathcal{X}_{n_i} \vee \neg\mathcal{X}_{n_j})$. Setting $\mathcal{X}_{n_i}$ to *TRUE* enables UP to set $\mathcal{X}_{n_j}$ to *FALSE*. However, UP can sometimes make the same inference without the binary clause expressing the mutex, as shown in the following example.

Fig. 4 (left) shows a corridor consisting of three vertices u, v and w and two agents a_1 and a_2 that need to traverse it in opposite directions. Fig. 5 shows the MDDs for agents a_1 and a_2, both with 3 levels, in which nodes corresponding to vertices u, v and w for agent a_1 are denoted n_1^u, n_1^v, and n_1^w, respectively, and nodes corresponding to u, v and w for agent a_2 are denoted n_2^u, n_2^v, and n_2^w, respectively.

Agents a_1 and a_2 cannot both be in vertex v at timestep 1 due to a vertex conflict. Mutex propagation hence discovers that agent a_1 cannot be in vertex w at timestep 2 if agent a_2 is in vertex u at timestep 2, resulting in the binary clause $(\neg\mathcal{X}_{n_1^w} \vee \neg\mathcal{X}_{n_2^u})$ expressing the mutex. Setting $\mathcal{X}_{n_1^w}$ to *TRUE* then enables UP to set $\mathcal{X}_{n_2^u}$ to *FALSE*. However, UP can make

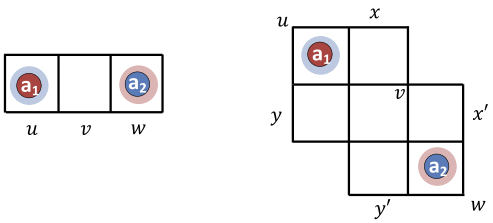


Fig. 4. Shows two simple MAPF instances, where agent a_1 has to move from vertex u to vertex w and agent a_2 has to move from vertex w to vertex u.

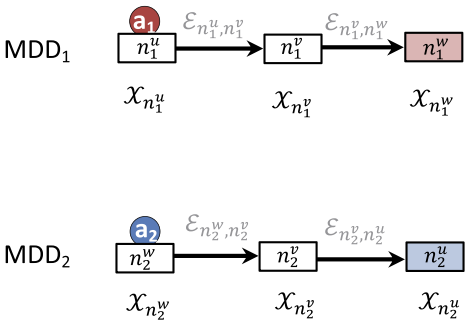


Fig. 5. Shows the MDDs for agents a_1 and a_2 and three levels each on the MAPF instance in Fig. 4 (left). Corresponding Boolean variables are shown.

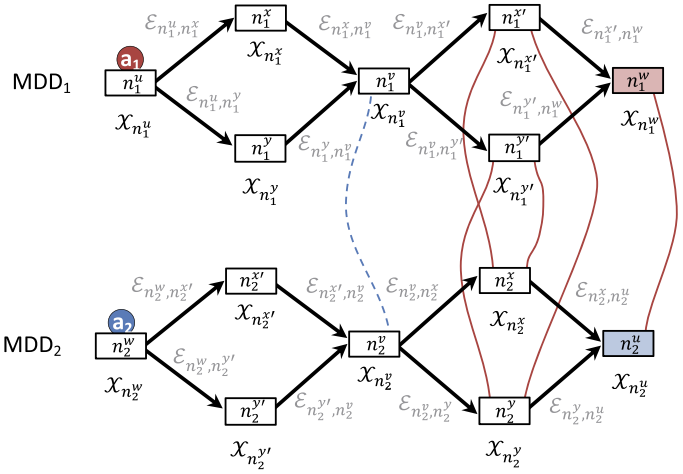


Fig. 6. Shows the MDDs for agents a_1 and a_2 and five levels each on the MAPF instance in Fig. 4 (right).

the same inference in several ways without the binary clause expressing the mutex, one of which is as follows: Setting $\mathcal{X}_{n_1^w}$ to *TRUE* enables UP to set $\mathcal{E}_{n_1^v,n_1^w}$ to *TRUE* due to the binary clause $(\neg\mathcal{X}_{n_1^w} \vee \mathcal{E}_{n_1^v,n_1^w})$ corresponding to Constraint (3) in Section 2.4. Next, UP can set $\mathcal{X}_{n_1^v}$ to *TRUE* due to the binary clause $(\neg\mathcal{E}_{n_1^v,n_1^w} \vee \mathcal{X}_{n_1^v})$ corresponding to Constraint (5), $\mathcal{X}_{n_2^v}$ to *FALSE* due to the binary clause $(\neg\mathcal{X}_{n_1^v} \vee \mathcal{X}_{n_2^v})$ corresponding to Constraint (6), and $\mathcal{E}_{n_2^v,n_2^u}$ to *FALSE* due to the binary clause $(\neg\mathcal{E}_{n_2^v,n_2^u} \vee \mathcal{X}_{n_2^v})$ corresponding to Constraint (5). Finally, UP can set $\mathcal{X}_{n_2^u}$ to *FALSE* due to the binary clause $(\neg\mathcal{X}_{n_2^u} \vee \mathcal{E}_{n_2^v,n_2^u})$ corresponding to Constraint (3).

In the above example, all relevant clauses are binary and benefit UP. In general, however, mutex propagation is strictly more powerful than UP.

Property 8. *Formula $\mathcal{F}(x)$ with clauses expressing the mutexes allows for strictly stronger Boolean constraint propagation compared to the same formula without such clauses.*

Proof. Fig. 4 (right) shows a corridor that is similar to the one in Fig. 4 (left), except that the agents can now bypass each other between vertices u and v and between vertices v and w. Fig. 6 shows the corresponding two MDDs, both with 5 levels. Agents a_1 and a_2 cannot both be in vertex v at timestep 2 due to a vertex conflict. Mutex propagation then discovers mutexes between agent a_1 being in vertices x' or y' at timestep 3 and agent a_2 being in vertices x or y at timestep 3, and

Table 1
Shows the number of CT node expansions on different cardinal conflict instances. The "Size" and "Length" of the cardinal rectangle and cardinal corridor conflict instances are the size and length of the yellow areas in Figs. 1a and 1b, respectively. The "Size" of the cardinal target conflict instances is the map size. The "Width" of the cardinal switching agents conflict instances is the map width.

Cardinal Rectangle Conflict Instance				
Size	5×5	6×6	7×7	8×8
CBSH	225	1,316	8,524	52,042
CBSH-RCT	1	1	1	1
CBSH-M	1	1	1	1
CBSH-MS	1	1	1	1
Cardinal Corridor Conflict Instance				
Length	12	14	16	18
CBSH	9, 851	45, 260	$>$ 52, 938	$>$ 49, 717
CBSH-RCT	1	1	1	1
CBSH-M	1	1	1	1
CBSH-MS	1	1	1	1
Cardinal Target Conflict Instance				
Size	5×5	6×6	7×7	8×8
CBSH	22, 134	$>$ 148, 966	$>$ 170, 132	$>$ 129, 534
CBSH-RCT	37, 031	$>$ 169, 869	$>$ 153, 794	$>$ 99, 235
CBSH-M	1	1	1	1
CBSH-MS	1	1	1	1
Cardinal Switching Agents Conflict Instance				
Width	7	8	9	10
CBSH	$>$ 183, 832	$>$ 154, 385	$>$ 213, 224	$>$ 221, 836
CBSH-RCT	$>$ 349, 461	$>$ 359, 559	$>$ 329, 361	$>$ 355, 826
CBSH-M	19	32	130	32
CBSH-MS	10	27	71	104

finally, a mutex between agent a_1 being in vertex w at timestep 4 and agent a_2 being in vertex u at timestep 4, expressed as the binary clause $(\neg\mathcal{X}_{n_1^w} \vee \neg\mathcal{X}_{n_2^u})$. Setting $\mathcal{X}_{n_1^w}$ to *TRUE* then enables UP to set $\mathcal{X}_{n_2^u}$ to *FALSE*.

UP cannot make the same inference without the binary clauses expressing the mutexes: Setting $\mathcal{X}_{n_1^w}$ to *TRUE* does not enable UP to set the truth value of any other variable since none of the clauses containing $\mathcal{X}_{n_1^w}$, namely, $(\neg\mathcal{X}_{n_1^w} \vee \mathcal{E}_{n_1^{x'},n_1^w} \vee \mathcal{E}_{n_1^{y'},n_1^w})$ corresponding to Constraint (3), $(\neg\mathcal{E}_{n_1^{x'},n_1^w} \vee \mathcal{X}_{n_1^w})$ corresponding to Constraint (5), and $(\neg\mathcal{E}_{n_1^{y'},n_1^w} \vee \mathcal{X}_{n_1^w})$ corresponding to Constraint (5), becomes a unit clause. □

6. Experimental results

In this section, we report experimental results for CBS and MDD-SAT with mutex propagation. We implemented all MAPF solvers in C++ and ran all experiments on t2.large AWS EC2 instances with 8 GB of memory.

6.1. Evaluation of mutex propagation in CBS

In this section, we report on experiments with different variants of CBSH, a state-of-the-art CBS-based solver with heuristic guidance [36]. We use CBSH and CBSH-RCT [23], a variant of CBSH with three symmetry-breaking techniques, namely, for rectangle, corridor, and target conflicts, as baselines and compare them to variants of CBSH with mutex propagation. The variants of CBSH with mutex propagation that we consider are: CBSH-M, CBSH-MS, CBSH-M-L and CBSH-MS-L, where M stands for mutex reasoning for only cardinal conflicts, MS stands for mutex reasoning for cardinal and semi-cardinal conflicts, and L stands for selecting the conflicts to be resolved based on path costs. All MAPF solvers share the same codebase, except for conflict classification and constraint generation. In Section 6.1.1, we compare CBSH with mutex propagation against the baselines. In Section 6.1.2, we compare different variants of CBSH with mutex propagation and different ways of determining the numbers of levels of MDDs.

6.1.1. Comparison of CBSH with mutex propagation against the baseline algorithms

Cardinal conflict instances: We use the cardinal conflict instances in Fig. 1. Table 1 shows the number of CT node expansions of CBSH, CBSH-RCT, CBSH-M and CBSH-MS. We omit CBSH-M-L and CBSH-MS-L since they yield the same results as CBSH-M and CBSH-MS, respectively. The Prefix "$>$" in an entry means that the MAPF solver doesn't solve the instance within a runtime limit of 5 minutes, and the number after "$>$" is the number of CT node expansions when the runtime limit is reached. The number of CT node expansions is large for CBSH on all instances. The number of CT node expansions is

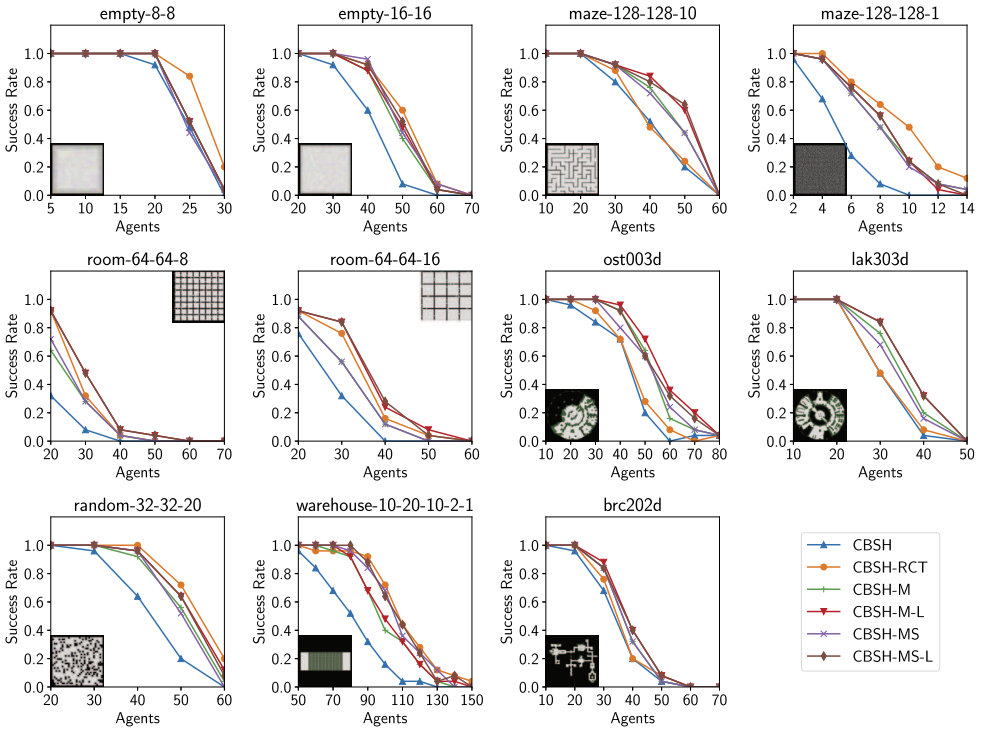


Fig. 7. Shows the success rates of various CBS-based MAPF solvers on several different maps with varying numbers of agents.

1 for CBSH-RCT on the cardinal rectangle conflict and cardinal corridor conflict instances because it uses symmetry-breaking techniques. CBSH-RCT solves these instances faster than CBSH-M and CBSH-MS because the rectangle and corridor reasoning techniques have a smaller runtime overhead than mutex propagation. However, the number of CT node expansions is 1 for CBSH-M and CBSH-MS not only on these instances but also on the cardinal target conflict instances. The numbers of CT node expansions for CBSH-M and CBSH-MS are smaller than those for CBSH and CBSH-RCT on the cardinal switching agents conflict instances as well, where CBSH-M and CBSH-MS expands only 3 CT nodes with cardinal conflicts at the bottom of the CT and only CT nodes with semi-cardinal conflicts and non-cardinal conflicts in the rest of the CT. Overall, CBSH-M and CBSH-MS solve all cardinal instances with a small number of CT node expansions and with a runtime of less than 1 second.

Benchmark instances: We use four-neighbor grid map instances from the MAPF benchmark set in [26]. These include 5 small maps: 2 empty maps of sizes 8×8 (empty-8-8) and 16×16 (empty-16-16), respectively, a 32×32 map with random blocked grid cells (random-32-32-20), and 2 64×64 maps divided into regular rooms of sizes 8×8 (room-64-64-8) and 16×16 (room-64-64-16), respectively, interconnected via doors. We also use 6 large maps: 2 128×128 maze maps with corridor widths 1 (maze-128-128-1) and 10 (maze-128-128-10), respectively, a 161×63 warehouse map with corridor width 1 (warehouse-10-20-10-2-1), and 3 game maps ('ost003d', 'lak303d', and 'brc202d').

We vary the number of agents and, for each number of agents, average over 25 "even scenarios"[3] from the benchmark set. Fig. 7 shows the success rates of CBSH, CBSH-RCT, CBSH-M, CBSH-MS, CBSH-M-L and CBSH-MS-L, which specify the percentage of instances solved by each algorithm within the runtime limit of 2 minutes. All variants of CBSH with mutex propagation outperform CBSH in terms of success rate on all maps. This shows that the runtime overhead of mutex reasoning is outweighed by its benefits in improving the total runtime. The addition of conflict selection (L) improves the success rate of both CBSH-M and CBSH-MS on all maps. On empty-8-8, empty-16-16, maze-128-128-1, and random-32-32-20, CBSH-RCT has higher success rate than all mutex propagation variants because its symmetry-breaking techniques run faster than mutex propagation. On maze-128-128-10, room-64-64-8, room-64-64-16, and game maps, both CBSH-M-L and CBSH-MS-L outperform CBSH-RCT because they identify conflicts that CBSH-RCT does not identify. CBSH-M(-L) outperforms or has a similar success rate as CBSH-MS(-L) because CBSH-MS(-L) does not identify many semi-cardinal conflicts but runs slower than CBSH-M(-L).

The average runtime ratios of mutex reasoning, that is, the fraction of runtime expended on mutex propagation and constraint generation, in CBSH-M-L and CBSH-MS-L are shown in Fig. 8. The runtime ratios differ on different maps. In empty-16-16 and random-32-32-20, we observe that the addition of semi-cardinal conflict reasoning increases the runtime ratio of mutex reasoning. In empty-16-16, CBSH-M-L incurs the smallest runtime ratio of mutex reasoning, which implies

[3] a category of instances described in [26].

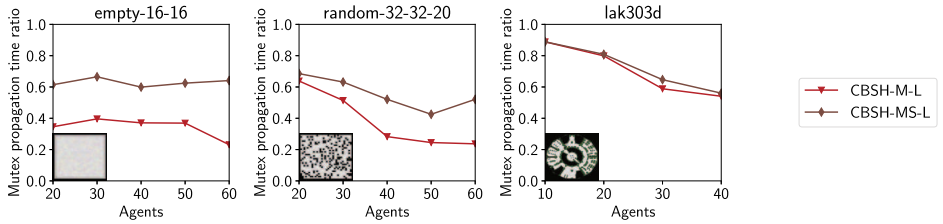


Fig. 8. Shows the average runtime ratios of mutex reasoning in CBSH-M-L and CBSH-MS-L on different representative maps with varying numbers of agents.

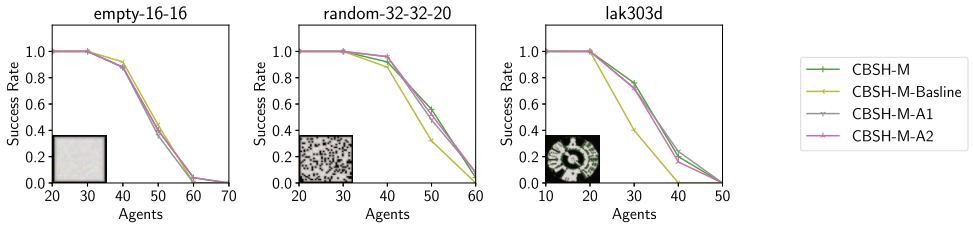


Fig. 9. Shows the success rates of different variants of CBSH-M, implementing different ways of determining the numbers of MDD levels, on different representative maps with varying numbers of agents.

that there are relatively fewer cardinal conflicts encountered by CBSH-M-L. In Fig. 7, we observe that CBSH-M-L is outperformed by CBSH-RCT. On the other hand, in 'lak303d', CBSH-M-L and CBSH-MS-L spend a large portion of time on mutex reasoning, which implies that there are a larger number of cardinal conflicts in these instances. In Fig. 7, we also observe that CBSH-M-L outperforms CBSH-RCT since it is able to identify more cardinal conflicts.

6.1.2. Comparison of different ways of determining the numbers of MDD levels

In this section, we show that it is important for CBSH with mutex propagation to try to increase the numbers of MDD levels before generating the constraint sets. We also show that the performance of the solver is not significantly affected by the exact choice of determining the numbers of MDD levels among the alternatives discussed in Section 4.1.3. We compare CBSH-M against CBSH-M-Baseline, a variant that does not increase the numbers of MDD levels, and CBSH-M-A1 and CBSH-M-A2, two other variants that implement the two alternative ways of determining the numbers of MDD levels. Fig. 9 shows the success rates of these four solvers. CBSH-M, CBSH-M-A1, and CBSH-M-A2 have very similar success rates on all maps. CBSH-M-Baseline has a slightly higher success rate than them on empty-16-16 because most cardinal conflicts on this map can be resolved efficiently without the solver increasing the numbers of MDD levels and CBSH-M-Baseline has a smaller runtime overhead than the other three solvers. However, on random-32-32-20 and lak303d, CBSH-M-Baseline has a lower success rate than the other three solvers because there are some cardinal conflicts that it cannot resolve efficiently while the other three solvers can.

6.2. Evaluation of mutex propagation in MDD-SAT

In this section, we experiment with different variants of MDD-SAT. We use MDD-SAT as a baseline and compare it to MDD-SAT with mutex propagation, namely, MDD-SAT-mutex. Both MAPF solvers share the same codebase written in C++. We use the Glucose 3.0 SAT solver [16].

We use the same 11 four-neighbor grid maps of Section 6.1 while varying the number of agents and, for each number of agents, average over 25 "even scenarios" from the benchmark set. Fig. 10 shows the success rates of MDD-SAT and MDD-SAT-mutex within a runtime limit of 5 minutes.

MDD-SAT-mutex is often faster than MDD-SAT, even though it spends a significant amount of time on mutex propagation. Overall, the runtime advantage of mutex propagation is less pronounced for SAT-based MAPF solvers compared to CBS-based MAPF solvers.

Fig. 11 shows the average runtime ratios of mutex propagation in MDD-SAT-mutex for different maps. Generally, we observe that the runtime ratios of mutex propagation in MDD-SAT-mutex are much lower compared to those in CBS. The runtime ratio is as high as 40% on random-32-32-20 and as low as 10% on 'lak303d'. We also observe that the runtime ratio of mutex propagation increases with the number of agents until a certain threshold is reached. After this, the runtime of the SAT solver becomes the dominating factor in the overall runtime.

It is also important to note that mutex propagation in MDD-SAT-mutex is used only for preprocessing the MDDs, that is, it is executed only once before extracting a valid solution from the MDDs via the SAT solver. This is in contrast to CBS, where mutex propagation is interleaved with CT node expansions. This accounts for the lower runtime ratios of mutex propagation in MDD-SAT-mutex compared to those in CBS.

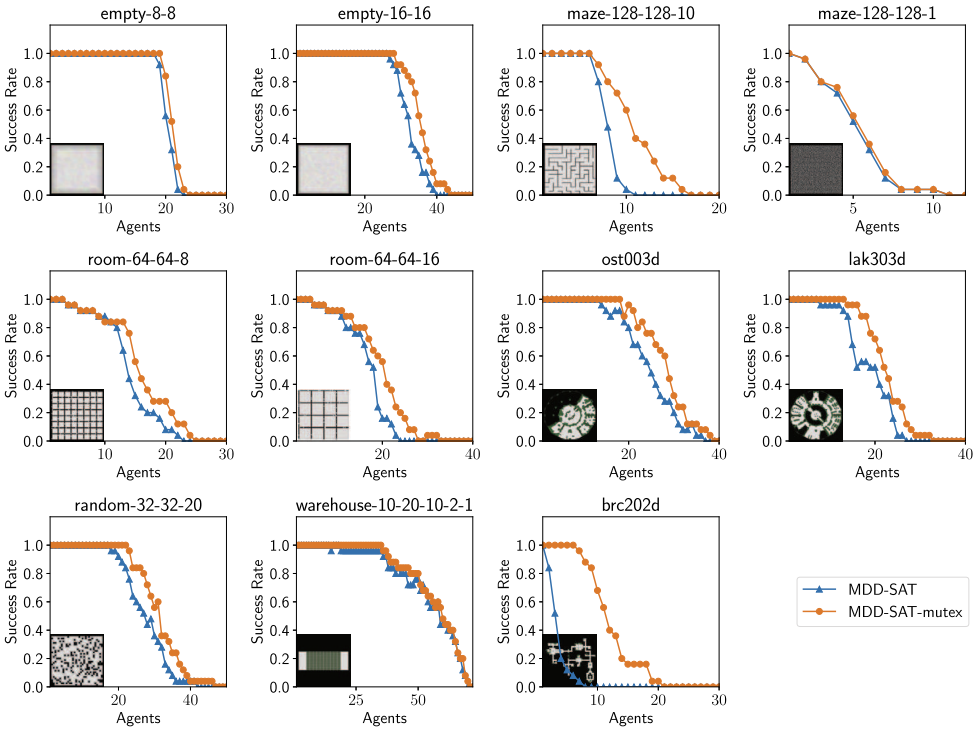


Fig. 10. Shows the success rates of MDD-SAT and MDD-SAT-mutex on several different maps with varying numbers of agents.

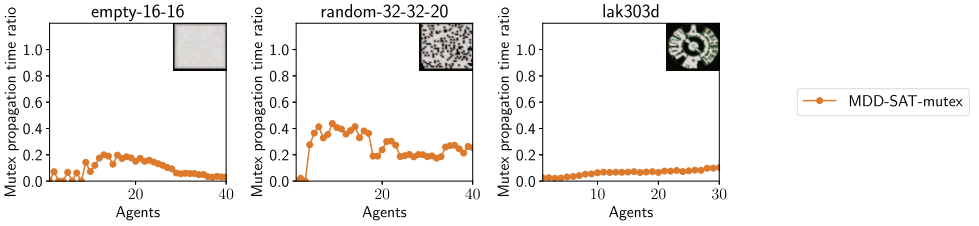


Fig. 11. Shows the average runtime ratio of mutex propagation in MDD-SAT-mutex on different representative maps with varying numbers of agents.

The similarity of success rates of MDD-SAT and MDD-SAT-mutex in some benchmarks (when the corresponding plots are close to each other), or the result that MDD-SAT-mutex solves few more instances within the given time limit means that MDD-SAT-mutex is constant factor faster than MDD-SAT.

Additional results concerning the number of clauses being generated by MDD-SAT and MDD-SAT-mutex are shown in Table 2. We can observe that the number of clauses generated by mutex propagation in MDD-SAT-mutex is substantially smaller than the number of clauses of the original encoding represented by MDD-SAT. However the number of clauses generated by mutex propagation quickly grows for increasing number of agents. This can be explained by more complex agents' interactions when more agents are present.

7. Related work

Our work is related to merging MDDs in [29] since merging MDDs, like mutex propagation, can also be used to determine the existence of conflict-free paths. Different from mutex propagation, merging MDDs finds all reachable states instead of all mutually exclusive states. It is also applicable to more than two agents. However, it becomes very time-consuming as the number of agents increases.

Our work is also related to CBS with improved heuristics (CBSH2) [9] since CBSH2 also uses reasoning about agent pairs to determine the minimum increment of the SoC for pairs of agents. In particular, the heuristic used by CBSH with mutex propagation is equivalent to the DG heuristic of CBSH2 because *a cardinal conflict* defined in this article is equivalent to *a pair of dependent agents* defined in [9]. In addition, the number of levels determined in Section 4.1.3 can also be used to improve the WDG heuristic of CBSH2, which we leave for future work.

Table 2
Number of clauses generated by MDD-SAT and MDD-SAT-mutex. The percentage of clauses generated during mutex propagation is shown. Results per number of agents are aggregated across scenarios out of 25 scenarios solved within the time limit.

Number of agents	empty-16-16		
	MDD-SAT	MDD-SAT-mutex	% of derived
10	14165	14175	0.07
20	89606	89969	0.40
30	3115189	3288143	5.26

Number of agents	random-32-32-20		
	MDD-SAT	MDD-SAT-mutex	% of derived
8	17599	17640	0.23
12	432852	438309	1.25
16	14021125	14625268	4.13

Number of agents	lak303d		
	MDD-SAT	MDD-SAT-mutex	% of derived
5	63670	63675	<0.01
10	1323064	1323202	0.01
15	50505605	50644713	0.27

Our work is also related to mutex reasoning built into the makespan-optimal SAT-based MAPF solver in [37]. Mutexes are computed for each pair of vertices and each pair of agents via search in the joint search space corresponding to configurations of the pair of agents. No mutex propagation is used.

Finally, our work is related to [38], [39], and [40], where graph structures in mutex networks (such as cliques) are detected. After the detection, the mutexes represented by these structures are encoded using representations more sophisticated than pairwise binary mutex clauses. However, these MAPF solvers benefit only marginally from such representations.

8. Conclusions and future work

In this article, we studied mutex propagation in the context of MAPF to efficiently reason about the interactions of pairs of agents and infer applicable constraints from the resulting mutexes. We also proposed a novel algorithmic framework for automatically identifying cardinal conflicts and generating strong constraint sets for guiding the high-level search of CBS while preserving its optimality guarantee. Our experimental results report significant improvements over the state-of-the-art CBS-based and SAT-based MAPF solvers with respect to runtime and success rate.

Interesting directions for future work include: (a) Applying mutex propagation to variants of MAPF problems where conflicts are implicitly represented, such as the robust MAPF problem [41] and the MAPF problem with generalized conflicts [42]; (b) Studying mutex propagation for incomplete Boolean models of MAPF where the formula is built lazily [43]; and (c) Studying efficient Boolean encodings of the large mutex networks that arise in MDDs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We thank Peter J. Stuckey, Daniel Harabor, and Graeme Gange for helpful discussions. The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, and 1837779 as well as a gift from Amazon. The research at the Czech Technical University was supported by GAČR - the Czech Science Foundation under grant number 22-31346S. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

References

[1] G. Sharon, R. Stern, M. Goldenberg, A. Felner, The increasing cost tree search for optimal multi-agent pathfinding, Artif. Intell. 195 (2013) 470–495.
[2] P. Surynek, Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems, Ann. Math. Artif. Intell. 81 (3–4) (2017) 329–375.
[3] J. Yu, S.M. LaValle, Structure and intractability of optimal multi-robot path planning on graphs, in: AAAI, 2013, pp. 1443–1449.

[4] H. Ma, C. Tovey, G. Sharon, T.K.S. Kumar, S. Koenig, Multi-agent path finding with payload transfers and the package-exchange robot-routing problem, in: AAAI, 2016, pp. 3166–3173.
[5] P.R. Wurman, R. D'Andrea, M. Mountz, Coordinating hundreds of cooperative, autonomous vehicles in warehouses, AI Mag. 29 (1) (2008) 9–20.
[6] R. Morris, C.S. Pasareanu, K. Luckow, W. Malik, H. Ma, T.K.S. Kumar, S. Koenig, Planning, scheduling and monitoring for airport surface operations, in: AAAI-16 Workshop on Planning for Hybrid Systems, 2016.
[7] G. Sharon, R. Stern, A. Felner, N.R. Sturtevant, Conflict-based search for optimal multi-agent pathfinding, Artif. Intell. 219 (2015) 40–66.
[8] E. Boyarski, A. Felner, R. Stern, G. Sharon, O. Betzalel, D. Tolpin, E. Shimony, ICBS: improved conflict-based search algorithm for multi-agent pathfinding, in: IJCAI, 2015, pp. 740–746.
[9] J. Li, A. Felner, E. Boyarski, H. Ma, S. Koenig, Improved heuristics for multi-agent path finding with conflict-based search, in: IJCAI, 2019, pp. 442–449.
[10] P. Surynek, Towards optimal cooperative path planning in hard setups through satisfiability solving, in: PRICAI, 2012, pp. 564–576.
[11] P. Surynek, A. Felner, R. Stern, E. Boyarski, Efficient SAT approach to multi-agent path finding under the sum of costs objective, in: ECAI, 2016, pp. 810–818.
[12] M. Ryan, Constraint-based multi-agent path planning, in: Australasian Joint Conference on Artificial Intelligence, 2008, pp. 116–127.
[13] E. Erdem, D.G. Kisa, U. Öztok, P. Schüller, A general formal framework for pathfinding problems with multiple agents, in: AAAI, 2013, pp. 290–296.
[14] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: DAC, 2001, pp. 530–535.
[15] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: IJCAI, 2009, pp. 399–404.
[16] G. Audemard, J. Lagniez, L. Simon, Improving glucose for incremental SAT solving with assumptions: application to MUS extraction, in: The International Conference on Theory and Applications of Satisfiability Testing, 2013, pp. 309–317.
[17] D.S. Weld, Recent advances in AI planning, AI Mag. 20 (2) (1999) 93–123.
[18] A.L. Blum, M.L. Furst, Fast planning through planning graph analysis, Artif. Intell. 90 (1997) 281–300.
[19] X. Nguyen, S. Kambhampati, Extracting effective and admissible state space heuristics from the planning graph, in: AAAI, 2000, pp. 798–805.
[20] X. Nguyen, S. Kambhampati, Reviving partial order planning, in: IJCAI, 2001, pp. 459–464.
[21] H. Kautz, B. Selman, Pushing the envelope: planning, propositional logic, and stochastic search, in: AAAI, 1996, pp. 1194–1201.
[22] J. Li, D. Harabor, P.J. Stuckey, H. Ma, S. Koenig, Symmetry-breaking constraints for grid-based multi-agent path finding, in: AAAI, 2019, pp. 6087–6095.
[23] J. Li, G. Gange, D. Harabor, P.J. Stuckey, H. Ma, S. Koenig, New techniques for pairwise symmetry breaking in multi-agent path finding, in: ICAPS, 2020, pp. 193–201.
[24] H. Zhang, J. Li, P. Surynek, S. Koenig, T.K.S. Kumar, Multi-agent path finding with mutex propagation, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2020, pp. 323–332.
[25] P. Surynek, J. Li, H. Zhang, T.K.S. Kumar, S. Koenig, Mutex propagation for SAT-based multi-agent path finding, in: Proceedings of the International Conference on Principles and Practice of Multi-Agent Systems (PRIMA), 2020, pp. 248–258.
[26] R. Stern, N.R. Sturtevant, D. Atzmon, T. Walker, J. Li, L. Cohen, H. Ma, T.K.S. Kumar, A. Felner, S. Koenig, Multi-agent pathfinding: definitions, variants, and benchmarks, in: SoCS, 2019, pp. 151–158.
[27] J. Li, D. Harabor, P.J. Stuckey, A. Felner, H. Ma, S. Koenig, Disjoint splitting for multi-agent path finding with conflict-based search, in: ICAPS, 2019, pp. 279–283.
[28] E. Lam, P.L. Bodic, D. Harabor, P.J. Stuckey, Branch-and-cut-and-price for multi-agent pathfinding, in: IJCAI, 2019, pp. 1289–1296.
[29] G. Sharon, R. Stern, M. Goldenberg, A. Felner, The increasing cost tree search for optimal multi-agent pathfinding, Artif. Intell. 195 (2013) 470–495.
[30] P. Surynek, Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs, in: ICTAI, 2014, pp. 875–882.
[31] C. Sinz, Towards an optimal CNF encoding of boolean cardinality constraints, in: CP, 2005, pp. 827–831.
[32] J.P.M. Silva, I. Lynce, Towards robust CNF encodings of cardinality constraints, in: CP, 2007, pp. 483–497.
[33] O. Bailleux, Y. Boufkhad, Efficient CNF encoding of boolean cardinality constraints, in: CP, 2003, pp. 108–122.
[34] A.K. Mackworth, Consistency in networks of relations, Artif. Intell. 8 (1) (1977) 99–118.
[35] W.F. Dowling, J.H. Gallier, Linear-time algorithms for testing the satisfiability of propositional horn formulae, J. Log. Program. 1 (3) (1984) 267–284.
[36] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T.K.S. Kumar, S. Koenig, Adding heuristics to conflict-based search for multi-agent path finding, in: ICAPS, 2018, pp. 83–87.
[37] P. Surynek, Mutex reasoning in cooperative path finding modeled as propositional satisfiability, in: IROS, 2013, pp. 4326–4331.
[38] P. Surynek, Solving difficult SAT instances using greedy clique decomposition, in: SARA, 2007, pp. 359–374.
[39] A. Biere, D.L. Berre, E. Lonca, N. Manthey, Detecting cardinality constraints in CNF, in: International Conference on Theory and Applications of Satisfiability Testing, 2014, pp. 285–301.
[40] P. Surynek, At-most-one constraints in efficient representations of mutex networks, in: ICTAI, 2020, pp. 170–177.
[41] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Barták, N.F. Zhou, Robust multi-agent path finding, in: SoCS, 2018, pp. 2–9.
[42] W. Hönig, J.A. Preiss, T.K.S. Kumar, G.S. Sukhatme, N. Ayanian, Trajectory planning for quadrotor swarms, IEEE Trans. Robot. 34 (4) (2018) 856–869.
[43] P. Surynek, Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories, in: IJCAI, 2019, pp. 1177–1183.