Pairwise symmetry reasoning for multi-agent path finding search

Jiaoyang Li\textsuperscript{a,*}, Daniel Harabor\textsuperscript{b}, Peter J. Stuckey\textsuperscript{b}, Hang Ma\textsuperscript{c}, Graeme Gange\textsuperscript{b}, Sven Koenig\textsuperscript{a}

\textsuperscript{a} Computer Science Department, University of Southern California, USA
\textsuperscript{b} Faculty of Information Technology, Monash University, Australia
\textsuperscript{c} Computer Science Department, Simon Fraser University, Canada

\section*{ARTICLE INFO}

\textbf{Article history:}
Received 21 February 2021
Received in revised form 1 August 2021
Accepted 2 August 2021
Available online 9 August 2021

\textbf{Keywords:}
Multi-agent path finding
Symmetry breaking
Multi-robot system

\section*{ABSTRACT}

Multi-Agent Path Finding (MAPF) is a challenging combinatorial problem that asks us to plan collision-free paths for a team of cooperative agents. In this work, we show that one of the reasons why MAPF is so hard to solve is due to a phenomenon called pairwise symmetry, which occurs when two agents have many different paths to their target locations, all of which appear promising, but every combination of them results in a collision. We identify several classes of pairwise symmetries and show that each one arises commonly in practice and can produce an exponential explosion in the space of possible collision resolutions, leading to unacceptable runtimes for current state-of-the-art (bounded-sub)optimal MAPF algorithms. We propose a variety of reasoning techniques that detect the symmetries efficiently as they arise and resolve them by using specialized constraints to eliminate all permutations of pairwise colliding paths in a single branching step. We implement these ideas in the context of a leading optimal MAPF algorithm CBS and show that the addition of the symmetry reasoning techniques can have a dramatic positive effect on its performance — we report a reduction in the number of node expansions by up to four orders of magnitude and an increase in scalability by up to thirty times. These gains allow us to solve to optimality a variety of challenging MAPF instances previously considered out of reach for CBS.

\section*{1. Introduction}

Multi-Agent Path Finding (MAPF)\textsuperscript{[1]} is a combinatorial problem that asks us to plan collision-free paths for a team of moving agents while minimizing the sum of their travel times. It is a core problem in a variety of real-world applications, including (but not limited to) automated warehousing\textsuperscript{[2,3]}, autonomous intersection management\textsuperscript{[4]}, drone swarm coordination\textsuperscript{[5]}, and video game character control\textsuperscript{[6]}. High-quality MAPF solutions are important for many of these applications, and thus numerous optimal and bounded-suboptimal algorithms have been suggested in recent years, despite the fact that MAPF is NP-hard to solve optimally on general graphs\textsuperscript{[7,8]}, directed graphs\textsuperscript{[9]}, planar graphs\textsuperscript{[10]}, and grids\textsuperscript{[11]}. The current leading (bounded-sub)optimal algorithms (e.g.,\textsuperscript{[12–15]} either based on Conflict-Based Search (CBS)\textsuperscript{[16]} or employ a similar strategy to CBS, whose central idea is to plan paths for each agent independently first by ignoring other agents and

* Corresponding author.

E-mail addresses: jiaoyangli@usc.edu (J. Li), daniel.harabor@monash.edu (D. Harabor), peter.stuckey@monash.edu (P.J. Stuckey), hangma@sfu.ca (H. Ma), graeme.gange@monash.edu (G. Gange), skoenig@usc.edu (S. Koenig).

https://doi.org/10.1016/j.artint.2021.103574
0004-3702/© 2021 Elsevier B.V. All rights reserved.
resolve collisions afterwards. Though each such algorithm proceeds in a different way, they face the same essential difficulty due to a phenomenon called pairwise symmetry, which occurs when two agents have many different paths to their target locations, but every combination of them results in a collision. In order to prove that the solutions these algorithms return are (bounded-sub)optimal, they have to enumerate a dramatically large number of the combinations of these colliding paths.

**Example 1.** Fig. 1 shows an example of a rectangle symmetry. There exists for each agent multiple shortest paths. Each path is grid symmetric: it can be derived from any other path by simply changing the order of the individual RIGHT and DOWN moves. All shortest paths for the two agents collide somewhere inside the yellow rectangular area. The optimal strategy here is for one agent to wait for the other. To find such a solution, however, optimal MAPF algorithms must first prove that every combination of wait-free paths leads to collisions. Yet the number of possible combinations of wait-free paths grows exponentially with the size of the yellow rectangular area, i.e., the larger the area, the harder the optimality proof. □

In this work, we consider three challenging situations, each commonly found in popular MAPF domains and involving pairs of colliding agents:

1. **rectangle symmetry**, which arises when two agents repeatedly collide along many different shortest paths.
2. **target symmetry**, which arises when a moving agent repeatedly collides with another stopped agent along many different paths of increasing lengths.
3. **corridor symmetry**, which arises when two agents moving in opposite directions repeatedly collide inside a narrow passage along many different paths of increasing lengths.

For each type of symmetry, we propose new algorithmic reasoning techniques that can identify the situation at hand and resolve it in a single branching step by the addition of symmetry-breaking constraints. We explore these ideas in the context of a leading and popular optimal MAPF algorithm CBS [16] (or, more precisely, its advanced variant CBSH [17]). On the one hand, we give a rigorous theoretical analysis which shows that our symmetry reasoning techniques preserve the completeness and optimality of CBS. On the other hand, we evaluate the impact of these symmetry reasoning techniques in a wide range of empirical comparisons, showing that the symmetry reasoning techniques can lead to an exponential reduction in CBS node expansions. In one headline result, we show that our resulting algorithm CBSH-RTC resolves the majority of two-agent collisions in just a single branching step. In another headline result, we report substantial improvement of the symmetry reasoning techniques on CBSH and its improved variants CBSH2 [18] and Mutex Propagation [19] in terms of both runtime and percentage of instances solved within the runtime limit.

Preliminary versions of this work appeared in AAAI 2019 [20] and ICAPS 2020 [21]. Compared to those versions, this paper provides a more comprehensive description and discussion of pairwise symmetries, new generalized versions of rectangle and corridor reasoning techniques (see Sections 6 and 9), and an extended empirical evaluation, including comparison with CBSH2 and mutex propagation (see Section 11). Although we demonstrate our symmetry reasoning techniques only in the context of solving classic MAPF problems with the optimal MAPF algorithm CBS in this paper, they can be applied and, indeed, based on our preliminary work, have already been applied to other variants of MAPF problems [22] and other optimal and bounded-suboptimal MAPF algorithms [14,15,23].

2. **Problem definition**

MAPF has many variants. In this paper, we focus on the classic variant defined in [1] that (1) considers vertex and swapping conflicts, (2) uses the “stay at target” assumption, and (3) optimizes the sum of costs.

Formally, we define MAPF by a graph $G = (V, E)$ and a set of $m$ agents $\{a_1, \ldots, a_m\}$. Each agent $a_i$ has a start vertex $s_i \in V$ and a target vertex $g_i \in V$. Time is discretized into timesteps. At each timestep, every agent can either move to an adjacent vertex or wait at its current vertex. A path $p_i$ for agent $a_i$ is a sequence of vertices which are adjacent or identical (indicating a wait action), starting at vertex $s_i$ and ending at vertex $g_i$. That is, $p_i = [v_0, v_1, \ldots, v_l]$, where $v_0 = s_i$, $v_l = g_i$, and for all $0 \leq t < v_i$, $(v_t, v_{t+1}) \in E$ or $v_t = v_{t+1} \in V$. We refer to $l$ as the path length of $p_i$. Agents remain at their target vertices after they complete their paths.
Definition 1 (Conflict). A conflict is either a vertex conflict \( \langle a_i, a_j, v, t \rangle \), which arises when agents \( a_i \) and \( a_j \) are at the same vertex \( v \in V \) at the same timestep \( t \), or an edge conflict \( \langle a_i, a_j, u, v, t \rangle \), which arises when agents \( a_i \) and \( a_j \) traverse the same edge \( (u, v) \in E \) in opposite directions at the same timestep \( t \) (or, more precisely, from timestep \( t - 1 \) to timestep \( t \)).

To reason about symmetries, we further classify and group some vertex and edge conflicts into symmetric conflicts. For example, the three vertex conflicts shown in Fig. 1 correspond to the same rectangle conflict. More details about symmetric conflicts are introduced in later sections. A solution is a set of conflict-free paths, one for each agent. Our task is to find a solution with the minimum sum of costs (i.e., sum of the path lengths).

In the examples and experiments of this paper, graph \( G \) is always a 4-neighbor grid whose vertices are unblocked cells and whose edges connect vertices corresponding to adjacent unblocked cells in the four main compass directions. We use this assumption because 4-neighbor grids are arguably the most common way of representing the environment for MAPF, and MAPF on 4-neighbor grids has many real-world applications, such as video games [24] and warehouse robots [3]. Nevertheless, most of our symmetry reasoning techniques can be directly applied to general graphs, and we will provide more details when we introduce these techniques.

3. Background: CBS and its variants

In this section, we introduce CBS and many improvements to it.

3.1. Vanilla CBS

Conflict-Based Search (CBS) [16] is a two-level search algorithm for solving MAPF optimally. At the low level, CBS invokes space-time \( A^* \) [6] (i.e., \( A^* \) that searches in the space whose states are vertex-timestep pairs) to find a shortest path for a single agent that satisfies the constraints added by the high level, breaking ties in favor of the path that has the fewest conflicts with the (already planned) paths of other agents. A constraint is a spatio-temporal restriction introduced by the high level to resolve conflicts. Specifically, a vertex constraint \( \langle a_i, v, t \rangle \) prohibits agent \( a_i \) from being at vertex \( v \in V \) at timestep \( t \). Similarly, an edge constraint \( \langle a_i, u, v, t \rangle \) prohibits agent \( a_i \) from traversing edge \( (u, v) \in E \) at timestep \( t \) (or more precisely, from timestep \( t - 1 \) to timestep \( t \)). We say that a constraint blocks a path if the path does not satisfy the constraint.

At the high level, CBS performs a best-first search on a binary constraint tree (CT). Each CT node contains a set of constraints and a plan, i.e., a set of shortest paths, one for each agent, that satisfy the constraints but are not necessarily conflict-free. The cost of a CT node is the sum of costs of its plan. The root CT node contains an empty set of constraints (and thus a set of shortest paths for all agents). CBS always expands the CT node with the smallest cost, breaking ties in favor of the CT node that has the fewest conflicts in its plan, and terminates when the plan of the CT node for expansion is conflict-free, which corresponds to an optimal solution. When expanding a CT node, CBS checks for conflicts in its plan. It chooses one of the conflicts (by default, arbitrarily) and resolves it by branching, i.e., by splitting the CT node into two child CT nodes. In each child CT node, one agent from the conflict is prohibited from using the conflicting vertex or edge at the conflicting timestep by way of an additional constraint. The path of this agent does not satisfy the new constraint and is re-planned by the low-level search. All other paths remain unchanged. If the low-level search cannot find any path, this child CT node does not have any solution and therefore is pruned.

3.1.1. Theoretical analysis

CBS guarantees its completeness by exploring both ways of resolving each conflict. In other words, when expanding a CT node, any conflict-free paths that satisfy the constraints of the CT node must satisfy the constraints of at least one of its child CT nodes. So branching only excludes conflicting paths but does not lose any solutions. CBS guarantees optimality by performing best-first searches at both its high and low levels. Please refer to [16] for detailed proof.

Since we will introduce new types of constraints to resolve symmetric conflicts in this paper, we here provide the principle of designing constraints for CBS without losing its completeness or optimality guarantees.

Definition 2 (Mutually disjunctive). Two constraints for two agents \( a_i \) and \( a_j \) are mutually disjunctive iff any pair of conflict-free paths of \( a_i \) and \( a_j \) satisfies at least one of the two constraints, i.e., there does not exist a pair of conflict-free paths that violates both constraints. Moreover, two sets of constraints are mutually disjunctive iff each constraint in one set is mutually disjunctive with each constraint in the other set.

Li et al. [25] prove that using two sets of mutually disjunctive constraints to split a CT node preserves the completeness and optimality of CBS. The key idea of their proof is to show that any solution that satisfies the constraints of a CT node also satisfies the constraints of at least one of its child CT nodes, as stated in Lemma 1. See their paper for detailed proof.

---

1 Another popular optimization criterion for MAPF is makespan (i.e., the maximum length of the paths). The symmetry reasoning techniques introduced in this article can be applied to optimizing makespan as well, although the speedup may vary.
Lemma 1. For a given CT node N with constraint set C, if two constraint sets C_1 and C_2 are mutually disjunctive, any set of conflict-free paths that satisfies C also satisfies at least one of the constraint sets C \cup C_1 and C \cup C_2.

Proof. This is true because, otherwise, there would exist a pair of conflict-free paths that does not satisfy all the constraints in C_1 and does not satisfy all the constraints in C_2. That is, one of the paths violates a constraint c_1 \in C_1 and one of the paths violates a constraint c_2 \in C_2. Then, c_1 and c_2 are not mutually disjunctive, contradicting the assumption. \qed

Theorem 2. Using two sets of mutually disjunctive constraints to split a CT node preserves the completeness and optimality of CBS. \qed

Hence, the principle of designing constraints for CBS without losing its completeness or optimality guarantees is to ensure that the two constraints (or constraint sets) we use to split a CT node are mutually disjunctive.

3.2. Advanced variants of CBS

We introduce CBSH, an improved variant of CBS, that is used as the baseline algorithm in our experiments, and CBSH2, a further improved variant of CBS, that we also compare against experimentally in Section 11.2.

3.2.1. CBSH

CBSH [17] improves CBS from two aspects. It first uses the technique of prioritizing conflicts from [26] to determine which conflict to resolve first. It classifies conflicts into three types, and, here, we provide the generalized definitions that are applicable also to the symmetric conflicts.

Definition 3 (Cardinal, semi-cardinal, and non-cardinal conflicts). A conflict is cardinal iff replanning for any agent involved in the conflict (with the corresponding constraint) increases the sum of costs. A conflict is semi-cardinal iff replanning for one agent involved in the conflict increases the sum of costs while replanning for the other agent does not. Finally, a conflict is non-cardinal iff replanning for any agent involved in the conflict does not increases the sum of costs.

Boyarski et al. [26] show that CBS can significantly improve its efficiency by resolving cardinal conflicts first, then semi-cardinal conflicts, and last non-cardinal conflicts, because generating child CT nodes with larger costs first can improve the lower bound of the CT (i.e., the minimum cost of the leaf CT nodes) faster and thus produce smaller CTs.

Following [26], CBSH builds MDDs to classify conflicts. A Multi-Valued Decision Diagram (MDD) [27] MDD_i for agent a_i at a CT node is a directed acyclic graph that consists of all shortest paths of agent a_i that satisfy the constraints of the CT node. The MDD nodes at depth t in MDD_i correspond to all locations at timestep t in these paths. If MDD_i has only one MDD node (v, t) at depth t, we call this node a singleton, and all shortest paths of agent a_i are at vertex v at timestep t. So a vertex conflict (a_i, a_j, v, t) is cardinal iff the MDDs of both agents have singletons at depth t, and an edge conflict (a_i, a_j, u, v, t) is cardinal iff the MDDs of both agents have singletons at both depth t − 1 and depth t. Semi-/non-cardinal vertex/edge conflicts can be identified analogously.

The high level of CBS consists of a best-first search that prioritizes the CT node with the smallest cost for expansion. The second improvement of CBSH over CBS is to add admissible heuristics to the high-level search. It builds a cardinal conflict graph for every CT node, whose vertices represent agents and edges represent cardinal conflicts in the plan of the CT node, and uses the size of the minimum vertex cover of the cardinal conflict graph as an admissible and consistent heuristic. Felner et al. [17] show that the addition of heuristics to the high-level search often produces smaller CTs and decreases the runtime of CBS by a large factor.

3.2.2. CBSH2

Recently, Li et al. [18] introduce a more informed heuristic for the high level of CBS by using CBSH to solve a two-agent MAPF instance for every pair of conflicting agents in every CT node. The suggested algorithm, CBSH2, proceeds by building a weighted pairwise dependency graph, whose vertices represent agents and edge weights represent the sum of costs of the optimal conflict-free paths for the two agents (with respect to the constraints of the CT node) minus the sum of costs of their paths in the plan of the CT node. It then solves an edge-weighted minimum vertex cover, which is an assignment of non-negative integers, one for each vertex, that minimizes the sum of the integers subject to the constraints that, for every edge, the sum of the two corresponding integers is no smaller than the edge weight. They show that the sum of the integers is an admissible h-value and no smaller than the h-value used in CBSH. With the help of some runtime reduction techniques, CBSH2 runs faster than CBSH on all maps tested in their experiments.

4. Related work

We review existing algorithms for solving MAPF optimally, existing methods that can eliminate (some) symmetries in MAPF, and existing symmetry reasoning work for other problems.
4.1. Optimal MAPF algorithms

Optimal MAPF algorithms include search-based algorithms that either search in the joint-state space or are variants of CBS and compilation-based algorithms that reduce MAPF to other well-studied problems like ILP, SAT, and CP. Algorithms that directly search in the joint-state space are usually not scalable, so the leading variants of various optimal MAPF algorithms (e.g., CBSH2, BCP, SMT-CBS, and lazy-CBS) all use the idea of planning paths independently first by ignoring other agents and resolving conflicts afterwards. Thus, they all suffer from the pairwise symmetries. In this work, we demonstrate and develop symmetry reasoning techniques on CBS variants, but similar ideas can be, or have already been, applied to others.

4.1.1. Search-based algorithms that search in the joint-state space

A* A straightforward way of solving MAPF is to use A* in the joint-state space, where the joint-states are different ways to place all the agents into $k$ out of $|V|$ vertices, one agent per vertex, and the operators between joint-states are non-conflicting combinations of actions that the agents can take. Since the size of the joint-state space grows exponentially with the number of agents, numerous techniques have been developed to improve the efficiency of A*, such as independence detection [28], operator decomposition [28], partial expansion [29], and subdimensional expansion [30].

**ICTS** Increasing Cost Tree Search (ICTS) [27] is a two-level algorithm that is conceptually different from A* but still searches in the joint-state space. Its high level searches the increasing cost tree where each node corresponds to a set of costs, one for each agent, and a child node differs from its parent node by increasing the cost of one of the agents by one. When expanding a node, its low level searches in the joint-state space to determine whether there exists a solution such that the cost of each agent is equal to the corresponding cost in the high-level node.

**Summary** Empirically, although many of the A* and ICTS variants are competitive with vanilla CBS [31], they are shown to be worse than CBSH with the symmetry reasoning technique for rectangle conflicts in our ICAPS 2019 paper [20]. This is not surprising because, as the number of agents increases, the effectiveness of the speedup techniques mentioned in the previous two paragraphs is limited, and thus they all suffer from the explosion of the joint-state space.

4.1.2. Compilation-based algorithms

**ILP** MAPF can be reduced to other well-studied NP-hard problems, relying on off-the-shelf solvers to find optimal solutions.

**SAT** MAPF can also be encoded as a Boolean satisfiability problem (SAT) [33]. Like the basic ILP encoding, the basic SAT encoding requires a Boolean variable for each agent being at each vertex at each timestep, and thus its efficiency drops as the size of the problem grows. SMT-CBS [13] is a more efficient SAT-based algorithm based on satisfiability modulo theories. Like CBS, it ignores all conflicts in the beginning and adds conflict-resolving constraints only when necessary. SMT-CBS outperforms the basic SAT-based algorithms, and there is already evidence that its efficiency can be further improved by adding the symmetry reasoning techniques [34].

**CP** Like the basic ILP and SAT encodings, MAPF can also be directly encoded as a constraint satisfaction problem and solved by an off-the-shelf Constraint-Programming (CP) solver. But, again, there is a more efficient CP-based algorithm, called lazy-CBS [12], that deploys the CBS framework. It uses the same constraint tree as CBS but traverses it using lazy clause generation instead of best-first search. Lazy-CBS is one of the leading algorithms for solving MAPF optimally, and there is already evidence that its efficiency can be further improved by adding the symmetry reasoning techniques [14].

4.2. Existing approaches to eliminating symmetries in MAPF

In pathfinding problems, symmetries have so far been studied only for single agents, e.g., by exploiting grid symmetries [35]. There is some prior work that is able to eliminate some symmetries in MAPF (but loses optimality and/or
completeness guarantees) by preprocessing the input graphs. We introduce two of them below. We then introduce a recent technique that uses mutex propagation to detect and resolve some symmetries in MAPF. We further give an empirical comparison with this technique in Section 11.1.

Graph decomposition Ryan [36,37] proposes several graph decomposition approaches for solving MAPF. Like our work, he detects special graph structures, including stacks, cliques and halls. Unlike our work, he builds an abstract graph by replacing such sub-graphs with meta-vertices during preprocessing in order to reduce the search space. His work preserves completeness but not optimality. Our work, by comparison, focuses on exploiting the sub-graphs to break symmetries without preprocessing or sacrificing optimality.

Highways Cohen et al. [38] propose highways to reduce the number of corridor conflicts (defined in Section 8). They assign directions to some corridor vertices (resulting in one or more highways) and make moving against highways more expensive than other movements. They show that highways can speed up ECBS [39], a bounded-suboptimal variant of CBS. However, the utility of highways for optimal CBS is limited because they can only be used to break ties among multiple shortest paths and are not guaranteed to resolve all corridor conflicts. Similar ideas of introducing directions to the graph edges are also explored in flow annotation re-planning [40], direction maps [41], and optimized directed roadmap graphs [42], all of which do not guarantee optimality.

Mutex propagation There is also recent work that identifies and resolves pairwise symmetries using mutex propagation [19, 34]. MDDs essentially capture the reachability information for single agents, which resemble planning graphs in classical planning. Therefore, this work adds mutex propagation on top of MDDs to capture the reachability information for pairwise agents. Two MDD nodes for two agents are mutex iff any pair of their paths that uses the two MDD nodes are in conflict. So two agents have a cardinal (symmetric) conflict iff the goal nodes of their MDDs are mutex. Given two agents with a cardinal (symmetric) conflict, it finds two MDD node sets, each consisting of the MDD nodes of one agent that are mutex with the goal MDD node of the other agent, and uses them to generate two constraint sets for branching in CBS. Therefore, the mutex propagation technique is able to automatically identify all cardinal symmetric conflicts and resolve them. However, as we show in Section 11.1, our handcrafted symmetry reasoning techniques substantially outperform the mutex propagation technique in practice because (1) we can also identify semi- and non-cardinal symmetric conflicts, (2) we induce smaller runtime overhead, and (3) our symmetry-breaking constraints are more effective in some cases.

4.3. Symmetry reasoning in other areas of AI

Symmetry is a widely-used concept that has been studied in many AI communities.

Symmetry reasoning has been shown to be a successful technique in planning [43–51]. Here, symmetries usually refer to state symmetries [45], which are defined as the automorphisms of the state transition graph, i.e., a directed multigraph, where the set of vertices contains a vertex for every state, and the set of edges contains a directed edge from state s to state s’ for every operator that is enabled at s and leads to s’. However, the state transition graph is usually too large to be given explicitly, existing work usually infers (subsets of) state symmetries from a compact description such as a semantic description of the planning task [45] or a factored representation of the planning task [48]. State symmetries take the form of symmetry groups across states. If several states from a group are encountered, only one of them is explored. In addition, information obtained during the search at different symmetric states can also improve heuristics [47].

Symmetry reasoning has been shown to a successful technique also in constraint programming [52–62]. Cohen et al. [60] propose a microstructure for a constraint satisfaction problem, i.e., a hypergraph, where the set of vertices contains a vertex for every literal (i.e., variable-value pair), and the set of edges contains a (hyper-)edge among a set of literals that corresponds to either an assignment allowed by a specific constraint or an assignment allowed because there is no constraint between the associated variables. Then, they define a constraint symmetry as an automorphism of the microstructure, which is conceptually similar to the automorphism of the state transition graph in planning. Like detecting symmetries in planning, for the sake of computational efficiency, existing work in constraint programming usually infers (subsets of) constraint symmetries from, for example, variable symmetries [59], i.e., the variables are interchangeable, or value symmetries [59,61], i.e., the values are interchangeable. The detected symmetries can then be eliminated by adding symmetry-breaking constraints [55] or performing symmetry breaking during search [54].

Similar symmetries have also been studied in propositional satisfiability problems [63,64], model checking [65,66], and path and motion planning for single agents [67,35], etc. Although symmetries have been widely studied in the literature, existing work always focuses on problem/state/solution symmetries in the sense that “renaming” (permuting somehow) some variables, values, propositions, or operators ends up with an identical problem/state/solution. However, we focus on conflict symmetries in this paper. It is unclear how to directly translate knowledge in symmetry reasoning in these domains to the improvements in CBS or other similar MAPF algorithms because they search in the conflict-resolution space as opposed to the problem/state/solution space.

---

2 An automorphism of a graph is a bijection on the vertices that preserves the edges (and hence also preserves the non-edges).
5. Rectangle symmetry

We start with some examples to show the motivations behind rectangle reasoning. Formal definitions of rectangle conflicts are introduced in the subsections of this section. According to Definition 3, the rectangle conflict in Example 1 is cardinal. Fig. 2(left) re-plots the MAPF instance from Fig. 1, Fig. 2(middle) draws the corresponding CT tree, and Fig. 2(right) shows the number of CT nodes expanded by CBS when the yellow rectangular area in the MAPF instance is larger, indicating that the size of the CT tree grows exponentially with the size of the rectangular area. So even for a 2-agent MAPF instance, CBS can time out if the cardinal rectangle conflict is undetected.

Moreover, reasoning about cardinal rectangle conflicts does not eliminate all rectangle symmetries for CBS.

Example 2. Consider the MAPF instances shown in Fig. 3 and ignore Rs, Rg, R1, and R2 for now. The conflict in Fig. 3(b) is not a cardinal rectangle conflict because agent a2 has optimal bypasses, i.e., shortest paths that do not traverse the rectangular area (e.g., path [(2, 1), (3, 1), (4, 1), (5, 1), (5, 2), (5, 3), (5, 4)]). However, if cell (5, 2) at timestep 4 and cell (5, 3) at timestep 5 are occupied by other agents, the low-level search of CBS always finds a path for agent a2 that conflicts with the path of agent a1, because the low-level search uses the number of conflicts with other agents as the tie-breaking rule. Therefore, CBS again generates many CT nodes before finally finding conflict-free paths.

We refer to the conflict in Fig. 3(b) as a semi-cardinal rectangle conflict. Similarly, we refer to the conflict in Fig. 3(c), where both agents have optimal bypasses, as a non-cardinal rectangle conflict. Together with cardinal rectangle conflicts, we refer to these three types of conflicts as rectangle conflicts. In Section 5.1, we introduce a rectangle reasoning technique that can efficiently identify and resolve rectangle conflicts between entire paths. Then, in Section 5.2, we generalize the reasoning technique for rectangle conflicts between path segments. Both techniques are applicable only on 4-neighbor grids. In Section 6, we generalize rectangle conflicts to cases where the conflicting area (i.e., the yellow area in Fig. 2(left)) is not necessarily rectangular and propose a more general reasoning technique that can work on planar graphs. We evaluate the empirical performance of all three rectangle reasoning techniques in Section 6.4.

Let us now define some notations that are used in this and the next sections. A space-time node (or node for short) (v, t) is a pair of a vertex v ∈ V and a timestep t ∈ ℕ. An MDD node is a space-time node. We say a path (or agent) visits node (v, t) iff it visits vertex v at timestep t. We say two paths (or agents) conflict at node (v, t) iff they conflict at vertex v at timestep t, and the node is referred to as the conflicting node. We focus on 4-neighbor grids in this section, as required by the two symmetry reasoning techniques. In particular, for a space-time node 5, we use (5.x, 5.y) to denote its cell and 5.t to denote its timestep.
5.1. Rectangle reasoning technique I: for entire paths

We now introduce the rectangle reasoning technique I. Consider two agents \( a_1 \) and \( a_2 \). Let nodes \( S_1, S_2, G_1 \) and \( G_2 \) be their start and target nodes. For now, we assume that the start node is at the start vertex at timestep 0 and the target node is at the target vertex at the timestep when the agent completes its path. Below are the formal definitions of the yellow rectangular area and the rectangle conflicts with some examples shown in Fig. 3.

**Definition 4 (Conflicting area).** Given start and target nodes \( S_1, S_2, G_1, \) and \( G_2 \) for agents \( a_1 \) and \( a_2 \), we define the conflicting area as the intersection cells of the rectangular area with diagonal corners \((S_1.x, S_1.y)\) and \((G_1.x, G_1.y)\) and the rectangular area with diagonal corners \((S_2.x, S_2.y)\) and \((G_2.x, G_2.y)\).

**Definition 5 (Rectangle conflict).** Two agents are involved in a rectangle conflict iff

1. they have at least one vertex conflict along their paths (or path segments, which will be discussed in Section 5.2),
2. both paths (or path segments) are Manhattan-optimal, i.e., for each agent, the length of its path (or path segment) is equal to the Manhattan distance between its start and target nodes, and
3. both paths (or path segments) move in the same direction in both \( x \) and \( y \) axes.

Conditions 1 and 2 ensure that the distances from the conflicting cell of the vertex conflict to the cells of the start nodes of the agents are equal. Together with Condition 3, we know that the distances from the cells of the start nodes of the agents to any cell inside the conflicting area are equal.

**Property 1.** Given a rectangle conflict between two agents, the distances from the cells of the start nodes of the agents to any cell inside the conflicting area are equal.

From Property 1, we know that, if two agents have a rectangle conflict, all their paths (or path segments) from their start to target nodes reach the same cell inside the conflicting area at the same timestep. We therefore define the rectangle, which is a set of nodes located inside the conflicting area, for a rectangle conflict as follows.

**Definition 6 (Rectangle).** Given start and target nodes \( S_1, S_2, G_1, \) and \( G_2 \) for agents \( a_1 \) and \( a_2 \) with a rectangle conflict, we define the rectangle as a set of nodes whose cells are the cells in the conflicting area and whose timesteps are the timesteps when a shortest path of agent \( a_1 \) or agent \( a_2 \) reaches the cell of the node. The four corner nodes of the rectangle are referred to as \( R_i, R_g, R_1, \) and \( R_2 \) where \( R_i \) and \( R_g \) are the corner nodes whose cells are closest to the cells of the start and target nodes, respectively, and \( R_1 \) and \( R_2 \) are the other corner nodes whose cells are on the opposite borders of the cells of \( S_1 \) and \( S_2 \), respectively. The border from \( R_1 \) to \( R_g \) and the border from \( R_2 \) to \( R_g \) (or, more precisely, the nodes in the rectangle whose cells are on the straight line segment from the cell of \( R_1 \) to the cell of \( R_g \) and from the cell of \( R_2 \) to the cell of \( R_g \)), are called the exit borders of agents \( a_1 \) and \( a_2 \) and denoted by \( R_1 R_g \) and \( R_2 R_g \), respectively.

In the following three subsections, we present in detail how to efficiently identify, classify and resolve the rectangle conflicts.

5.1.1. Identifying rectangle conflicts

Rectangle conflicts occur only when two agents have one or more vertex conflicts. Assume that agents \( a_1 \) and \( a_2 \) have a semi-/non-cardinal vertex conflict. Here, we do not consider cardinal vertex conflicts because a cardinal vertex conflict can be resolved in a single branching step by vertex constraints. They have a rectangle conflict iff

\[
|S_1.x - G_1.x| + |S_1.y - G_1.y| = G_1.t - S_1.t > 0 \quad (1)
\]

\[
|S_2.x - G_2.x| + |S_2.y - G_2.y| = G_2.t - S_2.t > 0 \quad (2)
\]

\[
(S_1.x - G_1.x)(S_2.x - G_2.x) \geq 0 \quad (3)
\]

\[
(S_1.y - G_1.y)(S_2.y - G_2.y) \geq 0. \quad (4)
\]

Equations (1) and (2) guarantee Condition 2 in Definition 5. Equations (3) and (4) guarantee Condition 3 in Definition 5.

5.1.2. Resolving rectangle conflicts

Let us look at Fig. 3. For cardinal rectangle conflicts, all combinations of the shortest paths are in conflict. For semi- and non-cardinal rectangle conflicts, although agents have shortest paths that are conflict-free, all combinations of the shortest paths that visit the corresponding exit borders of the agents are in conflict. We therefore propose to resolve a rectangle conflict by forcing one of the agents to leave its exit border later or take a detour. Formally, we introduce the barrier constraint \( B(a_i, R_i, R_g) = \{(a_i, (x, y), t) \mid ((x, y), t) \in R_i R_g\} \) \((i = 1, 2)\), which is a set of vertex constraints that prohibits agent
from occupying any node along its exit border \( R_i R_g \). When resolving a rectangle conflict, we generate two child CT nodes and add \( B(a_1, R_1, R_g) \) to one of them and \( B(a_2, R_2, R_g) \) to the other one. For instance, for the example in Fig. 3(a), the two barrier constraints are \( B(a_1, R_1, R_g) = \{(a_1, (4, 2 + n), 3 + n) \mid n = 0, 1 \} \) and \( B(a_2, R_2, R_g) = \{(a_2, (2 + n, 3), 2 + n) \mid n = 0, 1, 2 \} \). Adding barrier constraint \( B(a_i, R_i, R_g) \) \((i = 1, 2)\) blocks all shortest paths for agent \( a_i \) that reach its target node \( G_i \) via the rectangle. Thus, agent \( a_i \) is replanned with a longer path that does not conflict with the other agent. The rectangle conflict is thus resolved in a single branching step.

5.1.3. Classifying rectangle conflicts

To classify a rectangle conflict, we need to know whether the path length of agent \( a_i \) \((i = 1, 2)\) would increase after adding barrier constraint \( B(a_i, R_i, R_g) \). Because of the Condition 2 in Definition 5, all shortest paths between the start and target nodes for agent \( a_i \) are within the \( S_1-G_i \) rectangle. We thus only need to compare the length and width of the rectangle with those of the \( S_1-G_1 \) and \( S_2-G_2 \) rectangles. Consider the two equations

\[
R_i.x - R_g.x = S_1.x - G_i.x \\
R_i.y - R_g.y = S_1.y - G_i.y.
\]

Equation (5) holds when the length of the rectangle is equal to the length of the \( S_1-G_i \) rectangle \((i = 1, 2)\), and Equation (6) holds when the width of the rectangle is equal to the width of the \( S_1-G_i \) rectangle \((i = 1, 2)\). Also, since the rectangle is the intersection of the \( S_1-G_1 \) and \( S_2-G_2 \) rectangles, its length and width cannot be larger than the lengths and widths of the \( S_1-G_1 \) and \( S_2-G_2 \) rectangles. Therefore, if one of Equations (5) and (6) holds for \( i = 1 \) and the other one holds for \( i = 2 \), the rectangle conflict is cardinal; if only one of them holds for \( i = 1 \) or \( i = 2 \), it is semi-cardinal; otherwise, it is non-cardinal. For example, in Fig. 3(a), \( R_2.x - R_g.x = S_2.x - G_2.x = -2 \) and \( R_1.y - R_g.y = S_1.y - G_1.y = -1 \), so the conflict is cardinal.

5.1.4. Theoretical analysis

Now, we present a sequence of properties of the rectangle reasoning technique I and prove its completeness and optimality.

Property 2. For agents \( a_1 \) and \( a_2 \) with a rectangle conflict found by the rectangle reasoning technique I, all paths for agent \( a_1 \) that visit a node on its exit border \( R_i R_g \) must visit a node on its entry border \( R_i R_3 \), and all paths for agent \( a_2 \) that visit a node on its exit border \( R_2 R_g \) must visit a node on its entry border \( R_i R_1 \).

Property 2 is straightforward to prove but lengthy. We thus include the formal proof only in Appendix A.

Property 3. For all combinations of paths of agents \( a_1 \) and \( a_2 \) with a rectangle conflict found by the rectangle reasoning technique I, if one path violates barrier constraint \( B(a_1, R_1, R_g) \) and the other path violates barrier constraint \( B(a_2, R_2, R_g) \), then the two paths have one or more vertex conflicts within the rectangle.

Proof. According to Property 2, any path that violates \( B(a_1, R_1, R_g) \) must visit a node on border \( R_i R_2 \) and a node on border \( R_1 R_g \), and any path that violates \( B(a_1, R_1, R_g) \) must visit a node on border \( R_g R_1 \) and a node on border \( R_2 R_g \). Since \( R_1 R_2 \) and \( R_3 R_1 \) are the opposite sides of \( R_1 R_2 \) and \( R_3 R_1 \) of the conflicting area, respectively, such two paths must cross each other, i.e., they visit a common cell within the conflicting area. According to Property 1, they must visit this cell at the same timestep. Therefore, the property holds.

Property 3 tells us that barrier constraints \( B(a_1, R_1, R_g) \) and \( B(a_2, R_2, R_g) \) are mutually disjunctive (recall Definition 2). According to Theorem 2, using them to split a CT node preserves the completeness and optimality of CBS.

Theorem 3. Using the rectangle reasoning technique I preserves the completeness and optimality of CBS.

Note that we add barrier constraints on the exit borders of the agents instead of their entry borders because there might be an optimal solution that violates both “entry-border” barrier constraints. For instance, given the rectangle conflict shown in Fig. 3(a), if we use “entry-border” barrier constraints \( B(a_1, R_1, R_g) = \{(a_1, (2 + n, 2), 1 + n) \mid n = 0, 1 \} \) and \( B(a_2, R_3, R_1) = \{(a_2, (2 + n, 2), 1 + n) \mid n = 0, 1, 2 \} \), then the pair of paths, \([\{(1, 2), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3)\} \mid \text{for agent } a_1 \text{ and } \{(2, 1), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)\} \mid \text{for agent } a_2 \} \), is an optimal solution that violates both “entry-border” barrier constraints.

5.2. Rectangle reasoning technique II: for path segments

The rectangle reasoning technique II does not reason about obstacles and constraints, so it can only apply to rectangle conflicts for entire paths. In some cases, however, rectangle conflicts exist for path segments but not for entire paths, such as the cardinal rectangle conflict in Fig. 3(a). Since the paths are not Manhattan-optimal, the rectangle reasoning technique I fails to identify this rectangle conflict. Therefore, we extend the rectangle reasoning technique to reasoning about rectangle
Algorithm 1: Rectangle reasoning for path segments.

```
1 Input: A semi/non-cardinal vertex conflict \( (a_1, a_2, v, t) \).
2 \( N^c_{\ell} \leftarrow \) singletons in MDD, no later than timestep \( \ell \);
3 \( N^c_{\ell} \leftarrow \) singletons in MDD, no earlier than timestep \( \ell \);
4 \( N^c_{\ell} \leftarrow \) singletons in MDD, no earlier than timestep \( \ell \);
5 \( N^c_{\ell} \leftarrow \) Not-Rectangle; \( \text{area} \leftarrow 0 \);
6 foreach \( S_1 \in N^c_1, S_2 \in N^c_2, G_1 \in N^c_1, G_2 \in N^c_2 \) do
7 if isRectangle\( (S_1, S_2, G_1, G_2) \) then
8 \( \{R_1, R_2, R_3, R_4\} \leftarrow \text{getIntersection}(S_1, S_2, G_1, G_2); \)
9 \( \text{type} \leftarrow \text{classifyRectangle}(R_1, R_2, R_3, S_1, S_2, G_1, G_2); \)
10 \( \text{area} \leftarrow |R_1 - R_2| \times |R_3 - R_4| \)
11 if \( \text{type} = \text{Not-Rectangle} \) or type is better than type’ or \( \text{type} = \text{type’} \) and \( \text{area} > \text{area’} \) then
12 \( \text{type} \leftarrow \text{type’}; \text{area} \leftarrow \text{area}; \)
13 \( \{R_1’, R_2’, R_3’, R_4’\} \leftarrow \{R_1, R_2, R_3, R_4\}; \)
14 if type’ \neq \text{Not-Rectangle} then
15 \( B_1, B_2 \leftarrow \text{GENERATEBarriers}(\text{MDD}_1, \text{MDD}_2, R_1’, R_2’, R_3’, R_4’); \)
16 if \( a_1 \)'s path violates \( B_1 \) and \( a_2 \)'s path violate \( B_2 \) then
17 return \( B_1 \) and \( B_2 \);
18 return Not-Rectangle;
```

Algorithm 1 eventually discards this rectangle conflict.

Example 3. When running Algorithm 1 for the vertex conflict \( (a_1, a_2, (3, 2), 2) \) shown in Fig. 5, Line 4 assigns two singletons \( G_2 = (5, 5, 7) \) and \( G_2’ = ((3, 3), 3) \) to \( N^c_2 \). Therefore, Lines 7-10 in Algorithm 1 find two rectangle conflicts, namely a cardinal rectangle conflict with the conflicting area highlighted in yellow (i.e., the rectangular area with corner cells (2, 2) and (5, 4)), and a semi-cardinal rectangle conflict with the conflicting area highlighted in yellow with shadows (i.e., the rectangular area with corner cells (2, 2) and (3, 3)). Line 11 in Algorithm 1 prefers the cardinal rectangle conflict.

Example 4. Consider the MAPF instance shown in Fig. 3(b) and assume that the paths for agent \( a_1 \) and \( a_2 \) are \{ (1, 2), (2, 2), (3, 2), (4, 2), (4, 3) \} and \{ (2, 1), (3, 1), (4, 1), (4, 2), (5, 2), (5, 3), (5, 4) \}, respectively. Algorithm 1 (before Line 16) identifies the vertex conflict \( (a_1, a_2, (4, 2), 3) \) as a rectangle conflict with the conflicting area highlighted in yellow. The cells of the four corner nodes are shown in the figure. However, the resulting barrier constraint \( B_2 = B(a_2, R_2, R_8) \) does not block the path of agent \( a_2 \). So Algorithm 1 eventually discards this rectangle conflict.

Fig. 4. Examples of rectangle conflicts for path segments. The cells of the start and target nodes are shown in the figures. In (a), the cells of \( S_1 \) and \( G_2 \) are indicated by \( s_1 \) and \( g_2 \). In (b) and (c), \( G_1.t = S_1.t + |G_1.x - S_1.x| + |G_1.y - S_1.y|, i = 1, 2 \). In (b), \( S_1.t = S_2.t - 1 \). In (c), \( S_1.t = S_2.t - 2 \).
5.2.1. Identifying rectangle conflicts

The start and target nodes of a rectangle conflict have to satisfy not only Equations (1) to (4) but also

\[(S_1.x - S_2.x)(S_1.y - S_2.y)(S_1.x - G_1.x)(S_1.y - G_1.y) \leq 0.\] (7)

This guarantees that the start nodes are on different sides of the rectangle since, otherwise, adding barrier constraints might disallow a pair of paths that move both agents to the constrained border without waiting, such as in the example of Fig. 4(c).\(^3\) We also require that \(S_1 \neq S_2\) because, otherwise, the two agents have a cardinal vertex conflict at node \(S_1\) (recall that \(S_1\) and \(S_2\) are singletons) that can be resolved by vertex constraints in a single branching step.

5.2.2. Resolving rectangle conflicts

When reasoning about entire paths, all paths of agent \(a_1\) visit its start node \(S_1\) as node \(S_1\) is at its start vertex at timestep 0. However, when reasoning about path segments, only the shortest paths of agent \(a_1\) are guaranteed to visit node \(S_1\) as node \(S_1\) is a singleton. Its non-shortest paths do not necessarily visit node \(S_1\). In this case, barrier constraints may disallow pairs of conflict-free paths and thus lose the completeness guarantees.

Example 5. Fig. 6 provides a counterexample where a CT node \(N\) has the set of constraints listed in Fig. 6(b). The constraints force agent \(a_2\) to wait for at least one timestep before reaching its target vertex. It can either wait before entering the conflicting area, which leads to a conflict with agent \(a_1\), or enter the conflicting area without waiting and wait later, which might avoid conflicts with agent \(a_1\). However, all shortest paths (of length 6) of agent \(a_2\) that satisfy the constraints in \(N\) have to wait for one timestep before entering the conflicting area, see \(MDD_2\) shown in Fig. 6(c). Therefore, node \(S_2 = (4,2,2)\) is a singleton, and agents \(a_1\) and \(a_2\) have a cardinal rectangle conflict. If this conflict is resolved using barrier constraints, the CT sub-tree of \(N\) disallows the pair of conflict-free paths where agent \(a_1\) directly follows the blue arrow (which visits node \((5,3,4)\) constrained by \(B(a_1, R_1, R_2)\)) and agent \(a_2\) follows the green dashed arrow but waits at cell \((4,4)\) for two timesteps (which visits node \((4,4,4)\) constrained by \(B(a_2, R_2, R_3, R_4)\)). Barrier constraints fail here because the constrained node \((4,4,4)\) is not in \(MDD_2\) and thus agent \(a_2\) could have a path with a larger cost that does not visit node \(S_2\) but visits node \((4,4,4)\). □

\(^3\) We do not check Equation (7) in the rectangle reasoning technique I because, when the start nodes are at the same timestep, situations like Fig. 4(c) would never occur.
Therefore, we redefine barrier constraints by considering only the border nodes that are in the MDD of the agent. That is, \( B(a_i, R_1, R_g) = \{ (a_i, (x,y), t) \mid ((x,y), t) \in R_R \cap MDD_i \} (i = 1, 2) \).4 When resolving a rectangle conflict for path segments, we generate two child CT nodes and add \( B(a_1, R_1, R_g) \) to one of them and \( B(a_2, R_2, R_g) \) to the other one.

5.2.3. Classifying rectangle conflicts

We reuse the method in Section 5.1.3 to classify rectangle conflicts.

5.2.4. Theoretical analysis

We first present a property of MDDs.

**Property 4.** Given an MDD \( MDD_i \) for agent \( a_i \) and an MDD node \((v,t) \in MDD_i\) for any path \( p \) for agent \( a_i \) that visits node \((v,t)\), all the nodes that path \( p \) visits before timestep \( t \) are also in \( MDD_i \).

**Proof.** We prove the property by contradiction. Assume that there is a path \( p \) for agent \( a_i \) that visits node \((v,t) \in MDD_i\) and node \((u, \tau) \notin MDD_i \) with \( \tau < t \). Since \((v,t) \in MDD_i\), there exists a sub-path \( p' \) that moves agent \( a_i \) from node \((v,t)\) to node \((g_i, l)\), where \( l \) is the length of the shortest path for agent \( a_i \). So a path that first follows path \( p \) from node \((s_1, 0)\) to node \((v,t)\) via node \((u, \tau)\) and then follows sub-path \( p' \) to node \((g_i, l)\) is a shortest path for agent \( a_i \). So all nodes on path \( p' \) are in \( MDD_i \), which is contradicted to the assumption that \((u, \tau) \notin MDD_i \). Therefore, the property holds. \( \square \)

We then present three properties of barrier constraints.

**Property 5.** If agents \( a_1 \) and \( a_2 \) have a rectangle conflict found by the rectangle reasoning technique II, any path of agent \( a_i \) (\( i = 1, 2 \)) that visits a node constrained by \( B(a_i, R_1, R_g) \) also visits its start node \( S_i \).

**Proof.** Let \((v,t)\) be a node constrained by \( B(a_i, R_1, R_g) \). Thus node \((v,t)\) is in \( MDD_i \). Since node \( S_i \) is a singleton of \( MDD_i \) and the timestep of \( S_i \) is no larger than \( t \), from Property 4, any path for agent \( a_i \) that visits node \((v,t)\) also visits its start node \( S_i \). \( \square \)

**Property 6.** For agents \( a_1 \) and \( a_2 \) with a rectangle conflict found by the rectangle reasoning technique II, all paths for agent \( a_1 \) that visit a node constrained by \( B(a_1, R_1, R_g) \) must visit a node on the entry border \( R_2R_2 \), and all paths for agent \( a_2 \) that visit a node constrained by \( B(a_2, R_2, R_g) \) must visit a node on the entry border \( R_2R_1 \). \( \square \)

**Property 7.** For all combinations of paths of agents \( a_1 \) and \( a_2 \) with a rectangle conflict found by the rectangle reasoning technique II, if one path violates \( B(a_1, R_1, R_g) \) and the other path violates \( B(a_2, R_2, R_g) \), then the two paths have one or more vertex conflicts within the rectangle.

**Proof.** The proof for Property 3 can be applied here by replacing Property 2 with Property 6. \( \square \)

**Example 6.** Fig. 7(a) shows a MAPF sub-instance on a \( 32 \times 32 \) empty map. The distance between cells \( s_1 \) and \( g_1 \) is 32 while the distance between cells \( s_2 \) and \( g_2 \) is 34. Agent \( a_1 \) has no constraints, and thus all of its shortest paths are Manhattan-optimal and of length 32. Agent \( a_2 \) has a barrier constraint \( B_2 \) that forces the agent to first take a wait action at one of the cells in the top yellow row and then follow its Manhattan-optimal path to its target cell. Its shortest paths are thus of length 35. Due to this wait action, both agents reach every purple cell at the same timestep and thus have a conflict there if they both visit the same purple cell following their shortest paths. Since the two agents need to cross each other to reach

---

4 In our implementation, a barrier constraint is encoded as a set of vertex constraints.
their target cells, there is no way for them to reach their target cells without visiting some common purple cell via their shortest paths. Therefore, the optimal resolution is either for agent $a_1$ to wait for one timestep (resulting in a path of length 33) or for agent $a_2$ to wait for two timesteps or take a detour (resulting in a path of length 36).

This looks like a cardinal rectangle conflict as defined in Section 5. However, the only two singletons in MDD$_2$ are $(s_2, 0)$ and $(g_2, 35)$, which do not satisfy Equations (1) and (2). Therefore, the rectangle reasoning techniques in Section 5 fail to identify it as a rectangle conflict, and, as a result, CBS needs to spend exponential time to solve it.

**Example 7.** Fig. 7(b) shows a 2-agent MAPF instance on a 32 × 32 map with random obstacles. Both agents reach every purple cell at the same timestep if they follow their shortest paths, and they need to topologically cross each other to reach their target cells (or, formally, the line segments between their start and target cells need to cross each other). Therefore, the optimal resolution is for one of the agents to wait for one timestep.

However, the rectangle reasoning techniques in Section 5 fail to identify this as a rectangle conflict because they cannot find a pair of singletons around the purple area of agent $a_2$ that are Manhattan-optimal. In fact, the conflicting area here is not of a rectangular shape.

Example 6 behaves like a rectangle conflict, but there do not exist any appropriate singletons. Example 7 behaves like a rectangle conflict, but the conflicting area is not rectangular. They motivate us to define a more general rectangle conflict between two agents. These generalized rectangle cardinal conflicts have the following properties: (1) There is a purple area that both agents reach at the same timestep if they follow their shortest paths, and (2) the two agents have to topologically cross each other inside the purple area.

In Section 6.1, we present the high-level idea of our generalized rectangle reasoning technique. Then, in Section 6.2, we present the algorithm in detail. We provide a proof sketch of the soundness of the proposed technique in Section 6.3 and a formal proof in Appendix B. We empirically evaluate our generalized rectangle reasoning technique together with the rectangle reasoning techniques in Section 6.4.

The generalized rectangle reasoning technique can be applied to not only 4-neighbor grids but also other planar graphs, which covers most ways of representing 2D (or even 2.5D) environments for MAPF.

### 6.1. High-level idea

Let us consider the conflict in Fig. 7(b). Fig. 8(a) shows an abstract illustration of it. Agent $a_1$ enters the purple area from (one of) the blue solid lines and leaves it from (one of) the blue dotted lines. Similarly, agent $a_2$ enters the purple area from one of the yellow solid lines and leaves it from one of the yellow dotted lines. If we scan the border of the purple area anticlockwise, we find the pattern of “blue solid lines → yellow dotted lines → blue dotted lines → yellow solid lines”. So, from geometry, any line that connects a point on one of the blue solid lines with a point on one of the blue dotted lines without going outside the purple area must intersect with any line that connects a point on one of the yellow solid lines with a point on one of the yellow dotted lines without going outside the purple area. If the two agents follow such two lines, then they must have a vertex conflict at the intersection point. Therefore, any path for agent $a_1$ that visits the blue dotted lines must conflict with any path for agent $a_1$ that visits the yellow dotted lines. Following the idea in Section 5, we generate two barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$, where the cells of $R_1$, $R_2$ and $R_g$ are marked in Fig. 8(a).
and $B(a_i, R_1, R_g)$ ($i = 1, 2$) is a set of vertex constraints that prohibits agent $a_1$ from occupying all vertices along the border from $R_1$ to $R_g$ at the timestep when $a_1$ would optimally reach the vertex. This pair of barrier constraints gives one of the agents priority within the purple area and forces the other agent to leave it later or take a detour.

Fig. 8(b) shows a slightly different example where agent $a_1$ can leave the purple area also from the blue dotted line on the right. Therefore, the two agents can traverse the purple area without collisions, for instance, by following the dotted arrows. But, just like Example 2, CBS is not guaranteed to find such a pair of conflict-free paths efficiently. And, in fact, this example is a semi-cardinal generalized rectangle conflict as we can use barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$ to resolve it. This is so because, for the child CT node with constraint $B(a_1, R_1, R_g)$, agent $a_1$ will find a path that does not increase the length, such as the path indicated by the dotted blue line, while, for the other child CT node, all shortest paths are blocked by $B(a_2, R_2, R_g)$, and thus agent $a_2$ will find a path that does increase the length.

Fig. 8(c) draws another example where agent $a_1$ can enter the purple area also from the blue dotted line on the right. This time, however, we cannot use barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$ because there is a pair of conflict-free paths that violates both barrier constraints, indicated by the two arrows in the figure. Therefore, we do not recognize this example as a generalized rectangle conflict.

To sum up, how the solid lines of different colors distribute determines whether the conflict is a generalized rectangle conflict, and how the dotted lines of different colors distribute only affects the type of the conflict. Therefore, when we identify generalized rectangle conflicts, we only focus on the solid lines, see Fig. 8(d). We denote the nodes on the border with the smallest and largest timesteps as $R_s$ and $R_g$, respectively. $R_s$ and $R_g$ divide the border into two segments. If all blue solid lines are on only one of the segments and all yellow solid lines are on only the other segment, then the conflict is a generalized rectangle conflict. We denote the node on the blue and yellow solid lines that are furthest from $R_s$, i.e., closest to $R_g$ as $R_2$ and $R_1$, respectively. Then, we can prove that using barrier constraints $B(a_1, R_1, R_g)$ and $B(a_2, R_2, R_g)$ to resolve this conflict preserves the completeness and optimality of CBS.

Now, let us consider the case where the purple area has holes. The holes can be caused by either obstacles or constraints. The key point is to exclude the cases where the lines can cross each other within the hole because, otherwise, the agents might cross the intersection point in the hole at different timesteps and thus have conflict-free paths. Therefore, we also draw blue and yellow solid lines on the border of each hole to indicate where the agents can enter the purple area from the hole. If every hole inside the purple area has solid lines of at most one color, such as Fig. 8(e), then this is still a generalized rectangle conflict. Otherwise, as in the example of Fig. 8(f), such a conflict is not a generalized rectangle conflict.
As for classifying conflicts, we simply check whether barrier constraint \( B(a_i, R_i, R_g) \) \((i = 1, 2)\) blocks all shortest paths of agent \(a_i\) by looking at \(MDD_i\). The conflict is cardinal iff both barrier constraints block all shortest paths; it is semi-cardinal iff only one of them blocks all shortest paths; it is non-cardinal iff neither of them blocks all shortest paths.

6.2. Algorithm

Now, we provide the detailed methodology for identifying, classifying, and resolving generalized rectangle conflicts. There are five key steps:

1. finding the generalized rectangle (i.e., the purple area in Fig. 8);
2. scanning the border;
3. checking the holes;
4. generating the constraints; and
5. classifying the conflict,

which correspond to the following five subsections, respectively. Given a semi- or non-cardinal vertex conflict between two agents, the generalized rectangle reasoning algorithm returns either a pair of barrier constraints or “Not-Rectangle”.

6.2.1. Step 1: finding the generalized rectangle

**Definition 7 (Generalized rectangle).** Given two agents \(a_1\) and \(a_2\) with a vertex conflict at node \((v, t)\), the **generalized rectangle** is a connected directed acyclic graph \(G = (V, E)\) such that (1) \(G \subseteq MDD_1 \cap MDD_2\), (2) \((v, t) \in V\), and (3) for every node \((u, t_u) \in V\), any shortest path of either agent that visits vertex \(u\) visits it only at timestep \(t_u\). We use the term **conflicting area** to denote the vertices (e.g., cells in 4-neighbor grids) of the nodes in \(V\), which represent a connected area on the plane to which graph \(G\) is mapped.

Condition (3) is important because it guarantees that, if the shortest paths of agents \(a_1\) and \(a_2\) visit a common vertex in the conflicting area, they must have a vertex conflict. From conditions (1) and (3), we know that \((u, t) \in V\) only if, for both \(i = 1\) and \(i = 2\), \((u, t_u) \in MDD_i\) and \((u, t'_u) \notin MDD_i, \forall t'_u \neq t_u\). Formally, to find a generalized rectangle, we first project the MDD nodes of the MDDs of both agents to the vertices in \(V\). Let \(M_i \ (i = 1, 2)\) be such a mapping, where \(M_i[u], u \in V\) is a list of MDD nodes in \(MDD_i\) whose vertices are \(u\). Then, we run a search starting from the conflicting vertex \(v\) to generate \(G\) whose nodes \((u, t)\) satisfy the constraint that both \(M_1[u]\) and \(M_2[u]\) contain only one MDD node \((u, t)\). If \(V\) is empty or contains only one node, we terminate and report “Not-Rectangle”.

During the search, we also collect the entry edges \(E_1\) and \(E_2\) for the conflicting area (corresponding to the blue and yellow solid lines in Fig. 8).

**Definition 8 (Entry edge).** The set of **entry edges** \(E_i \ (i = 1, 2)\) is a set of directed MDD edges of \(MDD_i\) whose “from” node is not in \(V\) and whose “to” node is in \(V\).

Since the start nodes \((s_1, 0)\) and \((s_2, 0)\) of agents \(a_1\) and \(a_2\) are different, they must be located outside of the conflicting area, and, thus, both \(E_1\) and \(E_2\) contain at least one entry edge.

6.2.2. Step 2: scanning the border

Let \(R_s\) and \(R_g\) denote the nodes with the smallest and largest timesteps on the border, respectively. Scan the border from \(R_s\) to \(R_g\) on both sides and check whether the entry edges of one agent are all on one side of \(R_s\) and \(R_g\) and the entry edges of the other agent are all on the other side of \(R_s\) and \(R_g\). If not, we terminate and report “Not-Rectangle”.

Recall that the underlying graph is a planar graph. So we embed the graph into a plane and then scan the border clockwise and counterclockwise from \(R_s\) to \(R_g\). During the scanning, we mark the “to” nodes of the last-seen entry edges of \(E_1\) and \(E_2\) as \(R_2\) and \(R_1\), respectively. We also remove every visited entry edge from \(E_1\) or \(E_2\) so that all remaining edges in \(E_1\) and \(E_2\) are entry edges on the borders of the holes, which will be used in the next step. For clarification, we use \(E_i^b\) to denote the removed edges from \(E_i\) and \(E_1^b = E_1 \setminus E_1^b\) to denote the remaining edges in \(E_i \ (i = 1, 2)\).

6.2.3. Step 3: checking the holes

For each entry edge in \(E_1^b\), we scan the border of its corresponding hole and check whether the “to” node of any edge in \(E_1^b\) is on the border. If so, then this hole contains entry edges of both agents, so we terminate and report “Not-Rectangle”. If we succeed in examining every edge in \(E_1^b\) without terminating, then there are no holes in the conflicting area that contain an entry edge of both agents. We thus move to the next step.
Table 1
Benchmark details. We have 8 maps, each with 6 different numbers of agents. We have 25 instances for each setting, yielding \(8 \times 6 = 120\) instances in total.

| Map       | Map name       | Map size | #Empty cells | #Agents |
|-----------|----------------|----------|--------------|---------|
| Random    | random-32-32-20| 32 \times 32 | 819          | 20, 30, ..., 70 |
| Empty     | empty-32-32    | 32 \times 32 | 1,024        | 30, 50, ..., 130 |
| Warehouse | warehouse-10-20-10-2-1 | 161 \times 63 | 5,699 | 30, 50, ..., 130 |
| Game1     | den520d        | 256 \times 257 | 28,178 | 40, 60, ..., 140 |
| Room      | room-64-64-8   | 64 \times 64 | 3,232 | 15, 20, ..., 35 |
| Maze      | maze-128-128-1 | 128 \times 128 | 8,191 | 3, 6, ..., 18 |
| City      | Paris_1_256    | 256 \times 256 | 47,240 | 30, 60, ..., 180 |
| Game2     | br202d         | 530 \times 481 | 43,151 | 20, 30, ..., 70 |

6.2.4. Step 4: generating the constraints
We generate barrier constraints \(B(a_1, R_1, R_g)\) and \(B(a_2, R_2, R_g)\), where \(B(a_i, R_i, R_g)\) \((i = 1, 2)\) is a set of vertex constraints that prohibits agent \(a_i\) from occupying all nodes along the border from \(R_i\) to \(R_g\). All prohibited nodes are on the MDDs of the agents, so we do not need to worry about situations like Example 5 where the two agents might have conflict-free paths that traverse the prohibited nodes. Like Line 16 in Algorithm 1, we check whether the generated barrier constraints block the current paths of both agents. If not, we terminate and report “Not-Rectangle”.

6.2.5. Step 5: classifying the conflict
From Figs. 8(a) and 8(b), it seems that we can classify conflicts by checking whether the border segment \(R_iR_g\) covers all dotted lines of the color corresponding to agent \(a_i\). However, this is not correct because the agent might have a shortest path that does not visit the purple area at all. Therefore, we run a search on the MDD of each agent and check whether the barrier constraint \(B(a_i, R_i, R_g)\) \((i = 1, 2)\) blocks all paths on \(MDD\) from its start node to its target node, i.e., the nodes constrained by the barrier constraint form a cut of the MDD. The generalized rectangle conflict is cardinal iff both barrier constraints block all paths on the corresponding MDD; it is semi-cardinal iff only one of the barrier constraint blocks all paths on the corresponding MDD; it is non-cardinal iff neither barrier constraint blocks all paths on the corresponding MDD.

6.3. Theoretical analysis

Property 8. For all combinations of paths of agents \(a_1\) and \(a_2\) with a generalized rectangle conflict, if one path violates \(B(a_1, R_1, R_g)\) and the other path violates \(B(a_2, R_2, R_g)\), then the two paths have one or more vertex conflicts within the generalized rectangle \(G\).

Proof sketch. We show a proof sketch here and a formal proof in Appendix B:

1. All paths for agent \(a_i\) \((i = 1, 2)\) that visit a node constrained by \(B(a_i, R_i, R_g)\) must traverse an entry edge in \(E_i^b\).
2. Any sub-path from an entry edge in \(E_i^b\) to a node constrained by \(B(a_1, R_1, R_g)\) must visit at least one common vertex with any sub-path from an entry edge in \(E_k^b\) to a node constrained by \(B(a_2, R_2, R_g)\).
3. The common vertex must be inside the conflicting area, i.e., not inside one of the holes.
4. Following the two sub-paths, agents \(a_1\) and \(a_2\) must conflict at the common vertex in the conflicting area. QED

Property 8 tells us that barrier constraints \(B(a_1, R_1, R_g)\) and \(B(a_2, R_2, R_g)\) are mutually disjunctive, and thus, based on Theorem 2, using them to split a CT node preserves the completeness and optimality of CBS.

Theorem 5. Using the generalized rectangle reasoning technique preserves the completeness and optimality of CBS. QED

6.4. Empirical evaluation on rectangle reasoning

In this and future sections, we evaluate the algorithms on eight maps of different sizes and structures from the MAPF benchmark suite [1]. We test six different numbers of agents per map. We use the “random” scenarios from the benchmark suite in which the start and target vertices were generated randomly, yielding 25 instances for each map and each number of agents. The details of the benchmark instances are shown in Table 1, and a visualization of the maps is shown in Fig. 9. The algorithms are implemented in C++, and the experiments are conducted on Ubuntu 20.04 LTS on an Intel Xeon 8260 CPU with a memory limit of 16 GB and a time limit of 1 minute.

In this subsection, we compare CBSH (denoted None), CBSH with rectangle reasoning for entire paths (denoted R), CBSH with rectangle reasoning for path segments (denoted RM), and CBSH with generalized rectangle reasoning (denoted GR).\(^5\)

\(^5\) We demonstrate the symmetry reasoning techniques on top of CBSH instead of CBSH2 here (and in the following several sections) because, for CBSH2, the symmetry reasoning techniques can be applied to both the main CBSH2 and the two-agent sub-MAPF solver CBSH, which makes it sophisticated to
The results are reported in Fig. 9. A point \((x, y)\) in the figure indicates that there are \(x\) instances solved within \(y\) seconds. That is, lower is better. As expected, the improvements of our rectangle reasoning techniques depend on the structure of the maps. On maps that have little open space, such as Random, Room, Maze, and Game2, rectangle reasoning techniques do not improve the performance. But fortunately, due to their small runtime overhead, they do not deteriorate the performance either. On other maps that have open space, some or even all of the rectangle reasoning techniques speed up CBSH, and GR is always the best. Specifically, on map Empty, the shortest path of an agent (ignoring other agents) is always Manhattan-optimal, so R significantly speeds up CBSH, while RM and GR further speed it up, but only by a little bit. Similar is the performance on map Warehouse, as the obstacles on this map are all of rectangular shapes. Maps Game1 and City, however, contains obstacles of various shapes, so the shortest path of an agent is not necessarily Manhattan-optimal, and the conflicting area is not necessarily of rectangular shapes. Thus, R performs similarly with None, but GR significantly outperforms RM, which in turn significantly outperforms R.

7. Target symmetry

A target symmetry occurs when one agent visits the target vertex of a second agent after the second agent has already arrived at it and stays there forever. We refer to the corresponding conflict as a target conflict.

**Definition 9** (Target conflict). Two agents are involved in a target conflict iff they have a vertex conflict that happens after one agent has arrived at its target vertex and stays there forever.

**Example 8.** In Fig. 10, agent \(a_2\) arrives at its target vertex D2 at timestep 1, but an unavoidable vertex conflict occurs with agent \(a_1\) at the target vertex D2 at timestep 3. When CBS branches to resolve this vertex conflict, it generates two child CT nodes. In the left child CT node, CBS adds a vertex constraint for agent \(a_2\) that prohibits it from being at vertex D2 at timestep 3. The low-level search finds a new path \([C2, C3, C2, D2]\) for agent \(a_2\), which does not conflict with agent \(a_1\). The cost of this CT node is three larger than the cost of the root CT node. In the right child CT node, CBS adds a vertex
constraint for agent $a_1$ that prohibits it from being at vertex $D_2$ at timestep 3. Thus, agent $a_1$ can arrive at vertex $D_2$ at timestep 4, and the cost of this CT node is one larger than the cost of the root CT node. There are several alternative paths for agent $a_1$ where it waits at different vertices for the requisite timestep, e.g., path [A2, A2, B2, C2, D2, E2]. However, each of these paths produces a further conflict with agent $a_2$ at vertex $D_2$ at timestep 4. Although the left child CT node contains conflict-free paths, CBS has to split the right child CT nodes repeatedly to constrain agent $a_1$ (because it performs a best-first search) before eventually proving that the solution of the left child CT node is optimal.

Target symmetry has the same pernicious characteristics as rectangle symmetry since, if undetected, it can explode the size of the CT and lead to unacceptable runtimes. Table 2 shows how many CT nodes CBS expands to resolve a target conflict of the type shown in Fig. 10 for different distances between vertices $s_1$ and $g_2$. While the increase in CT nodes is linear in the distance, which may not seem too problematic, only one of the leaf CT nodes actually resolves the conflict (the paths in the other CT nodes still contain conflicts between the two agents). Later, when other conflicts occur, each of the leaf CT nodes might be further fruitlessly expanded. With two copies of the problem (resulting in 4-agent instances), Table 2 shows a quadratic increase in the number of CT nodes. For $m$-agent instances, the increases become exponential in $m$. Hence, we propose a target reasoning technique that can efficiently detect and resolve all target symmetries on general graphs. We introduce this technique in detail in the following four subsections and present its empirical performance in Section 7.5.

7.1. Identifying target conflicts

The detection of target conflicts is straightforward. For every vertex conflict, we compare the conflicting timestep with the agents’ path lengths.

7.2. Resolving target conflicts

The key to resolving target conflicts is to reason about the path length of an agent. Suppose that agent $a_2$ arrives at its target vertex $g_2$ at timestep $t'$ and stays there forever. Agent $a_1$ then visits vertex $g_2$ at timestep $t \geq t'$. We resolve this target conflict by branching on the path length $l_2$ of agent $a_2$ using the following two length constraints, one for each child CT node:

- $l_2 > t$, i.e., agent $a_2$ can complete its path only after timestep $t$, or
- $l_2 \leq t$, i.e., agent $a_2$ must arrive at vertex $g_2$ and stay there forever before or at timestep $t$, which also requires that any other agent cannot visit vertex $g_2$ at or after timestep $t$.

The first constraint $l_2 > t$ affects only the path of agent $a_2$, while the second constraint $l_2 \leq t$ could affect the paths of all agents.

The advantage of this branching method is immediate. In the first case, agent $a_2$ cannot finish until timestep $t + 1$, so its path length increases from its current value $t'$ to at least $t + 1$. In the second case, agent $a_1$ is prohibited from being at vertex $g_2$ at or after timestep $t$. If agent $a_1$ has no alternate path to its target vertex, the CT node with this constraint has no solution and is thus pruned. If agent $a_1$ has alternate paths that do not use vertex $g_2$ at or after timestep $t$ and the shortest one among them is longer than its current path, then its path length increases. We do not need to replan for agent $a_2$ since its current path is no longer than $t$. Nevertheless, we have to replan the paths for all other agents that visit vertex $g_2$ at or after timestep $t$. This is a very strong constraint as vertex $g_2$ can be viewed as an obstacle after timestep $t$ for all agents except agent $a_2$.

In order to handle the length constraints, we need the low-level search to take into account bounds on the path length. This is fairly straightforward for given bounds $e \leq l_2 \leq u$ on the path length $l_2$ of agent $a_2$. If the low-level search reaches target vertex $g_2$ before timestep $e$, then it cannot terminate but must continue searching: if it reaches the target vertex between timesteps $e$ and $u$ (and the agent was not at the target vertex at the previous timestep), then it terminates and returns the corresponding path; if it reaches the target vertex after timestep $u$, then it terminates, the corresponding CT node has no solution, and the CT node is thus pruned. We require the agent to not be at the target vertex at the previous timestep because, otherwise, the agent could simply take its current path to the target vertex and wait there until timestep $e$ is reached, which does not help to resolve the conflict.

For example, to resolve the target conflict in Fig. 10, we split the root CT node and add the length constraints $l_2 > 3$ and $l_2 \leq 3$. In the left child CT node, we replan the path of agent $a_2$ and find a new path [C2, C3, C3, C2, D2], which does not

---

Table 2

| Distances between vertices $s_1$ and $g_2$ | 10 | 20 | 30 | 40 | 50 |
|--------------------------------------------|----|----|----|----|----|
| Expanded CT nodes for 2-agent instances    | 10 | 20 | 30 | 40 | 50 |
| Expanded CT nodes for 4-agent instances    | 50 | 150| 300| 500| 750|
conflict with agent $a_1$. In the right child CT node, agent $a_1$ cannot occupy vertex D2 at or after timestep 3. We thus fail to find a path for it and prune the right child CT node. Therefore, the target symmetry is resolved in a single branching step.

7.3. Classifying target conflicts

Target conflicts are classified based on the vertex conflict at the target vertex: A target conflict is cardinal iff the corresponding vertex conflict is cardinal; and it is semi-cardinal iff the corresponding vertex conflict is semi-cardinal. It can never be non-cardinal because the cost of the child CT node with the additional length constraint $l_2 > t$ is always larger than the cost of the parent CT node. This is an approximate way of classifying target conflicts since it is possible that, when we branch on a semi-cardinal target conflict, the costs of both child CT nodes increase.

7.4. Theoretical analysis

Showing the completeness and optimality of CBS when using length constraints for target conflicts is straightforward. Therefore, we omit the proof of the following theorem.

**Theorem 6.** Resolving target conflicts with length constraints preserves the completeness and optimality of CBS. \(\square\)

7.5. Empirical evaluation on target reasoning

In this subsection, we compare CBSH (denoted **None**) with CBSH with target reasoning (denoted **T**). As shown in Fig. 11, on all maps except for **Maze**, target reasoning speeds up CBSH, and the improvement is usually larger on denser maps. The performance on **Maze** is an exception due to the low-level space-time A* search for replanning an extremely long or non-existing path. On the one hand, the length constraint $l_1 > t$ can substantially increase the path length of agent $a_1$, but finding a long path is time-consuming for space-time A*. On the other hand, the length constraint $l_1 \leq t$ prohibits all agents other than agent $a_1$ from being at vertex $g_i$ for all timesteps at and after timestep $t$, which might make it impossible for an agent to reach its target vertex. However, to realize that such a path does not exist, space-time A* has to enumerate all reachable pairs of vertex and timesteps, which is terribly time-consuming. We might be able to address both issues by replacing space-time A* with Safe Interval Path Planning [68], but leave this for future work.

8. Corridor symmetry

**Definition 10** (Corridor). A corridor $C = C_0 \cup \{e_1, e_2\}$ of graph $G = (V, E)$ is a chain of connected vertices $C_0 \subseteq V$, each of degree 2, together with two endpoints $\{e_1, e_2\} \in V$ connected to $C_0$. Its length is the distance between its two endpoints, i.e., the number of vertices in $C_0$ plus 1.

Fig. 12 shows a corridor of length 3 made up of $C_0 = \{B3, C3\}$, $e_1 = A3$ and $e_2 = D3$. A corridor symmetry occurs when two agents attempt to traverse a corridor in opposite directions at the same time. We refer to the corresponding conflict as a corridor conflict.
Definition 11 (Corridor conflict). Two agents are involved in a corridor conflict iff they come from opposite directions and have one or more vertex or edge conflicts inside a corridor.

Example 9. In Fig. 12, CBS detects the edge conflict \((a_1, a_2, B3, C3, 3)\) and branches, thereby generating two child CT nodes. There are many shortest paths for each agent that avoid edge \((B3, C3)\) at timestep 3 (e.g., path [A4, A3, B3, B3, C3, D3, D4] for agent \(a_1\) and path [D2, D2, D3, C3, B3, A3, A2] for agent \(a_2\)), all of which involve one wait action and differ only in where the wait action is taken. However, each of these single-wait paths remains in conflict with the path of the other agent. CBS has to branch at least four times to find conflict-free paths in such a situation and has to branch even more times to prove the optimality. Fig. 12(middle) shows the corresponding CT. Only two of the sixteen leaf CT nodes contain optimal solutions. \(\square\)

This example highlights an especially pernicious characteristic of corridor symmetry: CBS may be forced to continue branching and exploring irrelevant and suboptimal resolutions of the same corridor conflict in order to eventually compute an optimal solution. Fig. 12(right) shows how large a problem corridor symmetry can be for CBS more generally. As the corridor length \(k\) increases, the number of expanded CT nodes grows exponentially as \(2^{k+1}\) (This is because, when resolving a corridor conflict, the cost of a CT node at depth \(d\) is \(d\) plus the cost of the root CT node, and the cost of the optimal solution is \(k\) plus the cost of the root CT node). We therefore propose a new reasoning technique that can identify and resolve corridor conflicts efficiently. We present this technique in the following four subsections. We then extend it to handle several different special corridor symmetries more efficiently and evaluate the empirical performance in the next section.

8.1. Identifying corridor conflicts

The detection of corridor conflicts is straightforward by checking every vertex and edge conflict. We find the corridor on-the-fly by checking whether the conflicting vertex (or an endpoint of the conflicting edge) is of degree 2. To find the endpoints of the corridor, we check the degree of each of the two adjacent vertices and repeat the procedure until we find either a vertex whose degree is not 2 or the start or target vertex of one of the two agents.

8.2. Resolving corridor conflicts

Consider a corridor \(C\) of length \(k\) with endpoints \(e_1\) and \(e_2\) (see Fig. 13). Assume that the path of agent \(a_1\) traverses the corridor from \(e_2\) to \(e_1\) and the path of agent \(a_2\) traverses the corridor from \(e_1\) to \(e_2\). They conflict with each other inside the corridor. Let \(t_1(e_1)\) be the earliest timestep when agent \(a_1\) can reach \(e_1\) and \(t_2(e_2)\) be the earliest timestep when agent \(a_2\) can reach \(e_2\).

We first assume that there are no bypasses (i.e., paths that move the agent from its start vertex to \(e_1\) without traversing corridor \(C\)) for either agent (see Fig. 13(a)). Therefore, one of the agents must wait until the other one has fully traversed the corridor. If we prioritize agent \(a_1\) and let agent \(a_2\) wait, then the earliest timestep when agent \(a_2\) can start to traverse the corridor from \(e_1\) is \(t_1(e_1) + 1\). Therefore, the earliest timestep when agent \(a_2\) can reach \(e_2\) is \(t_1(e_1) + 1 + k\). Similarly, if we prioritize agent \(a_2\) and let agent \(a_1\) wait, then the earliest timestep when agent \(a_1\) can reach \(e_1\) is \(t_2(e_2) + 1 + k\). Therefore, any paths of agent \(a_1\) that reach \(e_1\) before or at timestep \(t_2(e_2) + k\) must conflict with any paths of agent \(a_2\) that reach \(e_2\) before or at timestep \(t_1(e_1) + k\).

Now we consider bypasses (see Fig. 13(b)). Assume that agent \(a_1\) has bypasses to reach \(e_1\) without traversing corridor \(C\) and the earliest timestep when it can reach \(e_1\) using a bypass is \(t_1'(e_1)\). Similarly, assume that agent \(a_2\) also has bypasses to reach \(e_2\) without traversing corridor \(C\) and the earliest timestep when it can reach \(e_2\) using a bypass is \(t_2'(e_2)\). If we prioritize agent \(a_1\), then agent \(a_2\) can either wait or use a bypass, then the earliest timestep when agent \(a_2\) can reach \(e_2\) is \(\min(t_2'(e_2), t_1(e_1) + 1 + k)\). Similarly, if we prioritize agent \(a_2\), then the earliest timestep when agent \(a_1\) can reach \(e_1\) is...
min(t′_1(e_1), t_2(e_2)+1+k). Therefore, any paths of agent a_1 that reach e_1 before or at timestep min(t′_1(e_1)−1, t_2(e_2)+k) must conflict with any paths of agent a_2 that reach e_2 before or at timestep min(t′_2(e_2)−1, t_1(e_1)+k). In other words, the following two constraints are mutually disjunctive:

- \langle a_1, e_1, [0, \min(t′_1(e_1)−1, t_2(e_2)+k)] \rangle
- \langle a_2, e_2, [0, \min(t′_2(e_2)−1, t_1(e_1)+k)] \rangle,

where \langle a, v, [t_{\min}, t_{\max}] \rangle is a range constraint \cite{69} that prohibits agent a from being at vertex v at any timestep between t_{\min} and t_{\max}. Thus, to resolve a corridor conflict, we split the CT node with two range constraints. We use state-time A* to compute t_1(e_1), t′_1(e_1), t_2(e_2), and t′_2(e_2). We cannot simply use the timesteps when the current paths of the agents traverse e_1 and e_2 as t_1(e_1) and t_2(e_2) because these paths minimize only the timesteps to reach the target vertices and thus do not necessarily minimize the timesteps to reach e_1 and e_2.

For example, for the corridor conflict in Fig. 12, we calculate t_1(D3) = t_2(A3) = 4, t′_1(D3) = t′_2(A3) = +∞ and k = 3. Hence, to resolve this conflict, we split the root CT node and add the range constraints \langle a_1, D3, [0, 7] \rangle and \langle a_2, A3, [0, 7] \rangle. In the right (left) child CT node, we replan the path of agent a_1 (a_2) and find a new path \langle A4, A4, A4, A4, A4, A3, B3, C3, D3, D4 \rangle ([D2, D2, D2, D2, D2, D2, D2, D3, B3, B3, A3, A2]), that waits at its start vertex for 4 timesteps before moving to its target vertex. It waits at its start vertex rather than any vertex inside the corridor because CBS breaks ties by preferring the path that has the fewest conflicts with the paths of the other agents. Hence, the paths in both child CT nodes are conflict-free, and the corridor symmetry is resolved in a single branching step.

Like the rectangle reasoning techniques, we use this branching method only when the paths of both agents in the current CT node violate their corresponding range constraints because this guarantees that the paths in both child CT nodes are different from the paths in the current CT node.

8.3. Classifying corridor conflicts

Similarly to target conflicts, we classify corridor conflicts based on the type of the vertex/edge conflict inside the corridor. A corridor conflict is cardinal if the corresponding vertex/edge conflict is cardinal; it is semi-cardinal if the corresponding vertex/edge conflict is semi-cardinal; and it is non-cardinal if the corresponding vertex/edge conflict is non-cardinal. This is an approximate way of classifying corridor conflicts. We use Fig. 12 to show an example where, after branching on a non-cardinal corridor conflict in a CT node N, the costs of both resulting child CT nodes have costs larger than the cost of N. Assume that N has two constraints, each of which prohibits one of the agents from being at its target vertex at timestep 5, so both agents have to wait for one timestep and thus have paths of length 6. If agent a_1 waits at vertex D3 at timestep 5 and agent a_2 waits at vertex A3 at timestep 5, then they have a non-cardinal edge conflict \langle a_1, a_2, B3, C3, 3 \rangle. As a result, the corridor conflict is classified as a non-cardinal conflict. However, as we saw above, when we use the range constraints \langle a_1, D3, [0, 7] \rangle and \langle a_2, A3, [0, 7] \rangle to resolve the corridor conflict, the costs of both child CT nodes are larger than the cost of N.

8.4. Theoretical analysis

Property 9. For all combinations of paths of agents a_1 and a_2 with a corridor conflict, if one path violates \langle a_1, e_1, [0, \min(t′_1(e_1)−1, t_2(e_2)+k)] \rangle and the other path violates \langle a_2, e_2, [0, \min(t′_2(e_2)−1, t_1(e_1)+k)] \rangle, then the two paths have one or more vertex or edge conflicts inside the corridor.\hfill \Box
Since we already intuitively prove Property 9 when we introduce range constraints, we move the formal proof to Appendix C. Property 9 tells us that range constraints are mutually disjunctive, and thus, according to Theorem 2, using them to split a CT node preserves the completeness and optimality of CBS.

**Theorem 7.** Resolving corridor conflicts with range constraints preserves the completeness and optimality of CBS.

Note that we add range constraints at the exit endpoints of the agents instead of their entry endpoints because there might be an optimal solution that violates both "entry-endpoint" range constraints. For instance, given the corridor conflict shown in Fig. 12(left), if we use "entry-endpoint" range constraints \( \{a_1, A_3, [0, 4]\}\) and \( \{a_2, D_3, [0, 4]\}\), then the pair of paths, \( \{A_4, A_3, B_3, C_3, D_3, D_4\}\) for agent \( a_1 \) and \( \{D_2, D_3, C_3, D_3, D_2, D_3, C_3, B_3, A_3, A_2\}\) for agent \( a_2 \), is an optimal solution that violates both "entry-endpoint" range constraints.

9. Generalized corridor symmetries

The corridor reasoning technique in the previous section has some limitations when handling three special corridor symmetries, namely pseudo-corridor symmetries, corridor symmetries with start vertices inside the corridor, and corridor-target symmetries. In this section, we first discuss and address these special cases in detail in the following three subsections. We then present the framework of the generalized corridor reasoning technique that can handle all types of corridor symmetries in Section 9.4. We last show some empirical results in Section 9.5.

9.1. Pseudo-corridor conflicts

Pseudo-corridor symmetry is a special corridor symmetry that behaves like a corridor conflict but occurs in a non-corridor region.

**Example 10.** In Fig. 14(left), CBS detects the edge conflict \( \langle a_1, a_2, C_2, D_2, 4 \rangle \) and branches, thereby generating two child CT nodes. There are many shortest paths for each agent that avoid edge (\( C_2, D_2 \)) at timestep 4 (e.g., path \( \{A_3, A_2, B_2, C_2, D_2, E_2, F_2, F_3, F_4\}\) for agent \( a_1 \) and path \( \{F_3, F_2, E_2, D_2, C_2, B_2, A_2, A_3, A_4\}\) for agent \( a_2 \)), but they all involve one wait action and differ only in where the wait action is taken. However, each of these single-wait paths remains in conflict with the path of the other agent. CBS has to branch again to find conflict-free paths in such a situation. Fig. 14(right) shows the corresponding CT. Only the left-most and right-most leaf CT nodes contain optimal solutions.

Like corridor conflicts, a pseudo-corridor conflict occurs when (1) two agents move in opposite directions, (2) they have a vertex or edge conflict, and (3) adding one wait action to one of the agents before the timestep of the vertex or edge conflict, no matter where, must lead to another edge or vertex conflict. In fact, a pseudo-corridor conflict can be viewed as a corridor conflict whose corridor is of length 1, i.e., consist of only two endpoints. Although, compared to corridor conflicts, a pseudo-corridor conflict seems to be less problematic as the size of the CT does not grow exponentially, it could occur more frequently as it is not restricted to maps that have corridors.

We reuse the corridor reasoning technique to resolve pseudo-corridor conflicts. That is, when we find a corridor conflict of length \( k = 1 \), we generate two range constraints \( c_1 = \langle a_1, e_1, \max(\min(t'_1(e_1) - 1, t_2(e_2) + 1)) \rangle \) and \( c_2 = \langle a_2, e_2, \min(\min(t'_2(e_2) - 1, t_1(e_1) + 1)) \rangle \), where \( t_i(e_i) \) (\( i = 1, 2 \)) is the earliest timestep for agent \( a_i \) to reach endpoint \( e_i \) and \( t'_i(e_i) \) (\( i = 1, 2 \)) is the earliest timestep for agent \( a_i \) to reach endpoint \( e_i \) without using edge \( e_i \). All properties listed in Section 8.4 hold here. By reusing their proofs without changes, we can show that resolving a pseudo-corridor conflict with constraints \( c_1 \) and \( c_2 \) preserves the completeness and optimality of CBS.

In practice, we only use range constraints \( c_1 \) and \( c_2 \) to resolve the conflict if the path of agent \( a_1 \) violates range constraint \( c_1 \) and the path of agent \( a_2 \) violates range constraint \( c_2 \), and we are only interested in cardinal pseudo-corridor conflicts because semi-/non-cardinal pseudo-corridor conflicts are easy to resolve. A necessary but insufficient condition to ensure
Algorithm 2: Pseudo-corridor reasoning.

```
Input: Vertex conflict $c = (a_1, a_2, v, t)$ or edge conflict $c = (a_1, a_2, v, u, t)$.

1. $e_1, e_2 \leftarrow$ NULL;
2. if $c$ is a vertex conflict and, for $i = 1, 2$, MDD$_i$ has only one MDD node at timesteps $t - 1$, $t$, and $t + 1$ and the MDD node of MDD$_i$ at timestep $t - 1$ is identical to the MDD node of MDD$_{i-1}$ at timestep $t + 1$ then
   3. $e_1 \leftarrow v$;
4. else if $c$ is an edge conflict and, for $i = 1, 2$, MDD$_i$ has only one MDD node at both timesteps $t - 1$ and $t$ then
   5. $e_1 \leftarrow u$;
   6. $e_2 \leftarrow v$;
8. if $e_1 \neq$ NULL then
   9. $c_1 \leftarrow (a_1, e_1, [0, \min(t'_1(e_1) - 1, t_2(e_1) + 1)])$;
10. $c_2 \leftarrow (a_2, e_2, [0, \min(t'_2(e_2) - 1, t_1(e_2) + 1)])$;
11. if the path of $a_1$ violates $c_1$ and the path of $a_2$ violates $c_2$ then
    12. return $c_1$ and $c_2$;
13. return “Not Corridor”;
```

![Fig. 15. Examples of corridor conflicts with start vertices inside the corridor.](image)

This is that, if the conflict between the two agents is a vertex conflict at timestep $t$, then the MDD of both agents have only one MDD node at timesteps $t - 1$, $t$ and $t + 1$, and the MDD node of one agent at timestep $t - 1$ is identical to the MDD node of the other agent at timestep $t + 1$; or if the conflict is an edge conflict at timestep $t$, then the MDD of both agents have only one MDD node at timesteps $t - 1$ and $t$. Therefore, before we generate range constraints $c_1$ and $c_2$, we check the MDDs of both agents to eliminate some non-pseudo-corridor conflicts, as checking MDDs is substantially computationally cheaper than computing $t'_1(e_1)$ and $t'_2(e_2)$ for generating range constraints. Algorithm 2 summarizes the pseudo-code for the pseudo-corridor reasoning technique. All pseudo-corridor conflicts returned by Algorithm 2 are cardinal.

9.2. Corridor conflicts with start vertices inside the corridor

The corridor reasoning technique cannot resolve corridor conflicts efficiently when the start vertices of one or both agents are inside the corridor.

Example 11. Fig. 15(a) shows the same example as in Fig. 12(left) except that the start vertex of agent $a_1$ is inside the corridor. If the two agents follow their individual shortest paths, they have an edge conflict at ($C_3, D_3$) at timestep 2. Thus, when we use the corridor reasoning technique described in Section 8.1, we find a corridor $C = (B_2, C_3, D_3)$ of length 2 and generate a pair of range constraints $a_1, D_3, [0, 5])$ and $a_2, B_3, [0, 4])$. However, when we generate the left child node with the first constraint, we cannot find a shortest path for agent $a_1$ that does not conflict with agent $a_2$. In fact, the shortest path for agent $a_1$ that does not conflict with agent $a_2$ is to first move to $A_4$, wait there until agent $a_2$ reaches $A_3$, then traverse the corridor and reach its target vertex.

This example shows that the previous corridor reasoning technique cannot resolve the corridor conflict in a single branch, because it stops detecting the corridor after it finds a start vertex. Therefore, in this subsection, we modify the corridor reasoning technique by allowing start vertices to be inside the corridor. Below are the details of the modification.

Identifying corridor conflicts For every vertex and edge conflict, we first find the corridor on-the-fly by checking whether the conflicting vertex (or an endpoint of the conflicting edge) is of degree 2. To find the endpoints of the corridor, we check all degree of each of the two adjacent vertices and repeat the procedure until we find either a vertex whose degree is not 2 or the target vertex of one of the two agents. Then, we say the two agents are involved in a corridor conflict if they (1) leave the corridor from different endpoints and (2) have to cross each other inside the corridor. The second condition is to avoid cases like Fig. 15(c). Although the paths for the two agents shown in Fig. 15(c) do not conflict, when considering
constraints in the CT node, it is possible that the shortest paths of the two agents are longer than the paths shown in the figure and conflict inside the corridor. But we should not view it as a corridor conflict.

**Resolving and classifying corridor conflicts** It is the same as the original technique shown in Sections 8.2 and 8.3.

**Theoretical analysis** All properties listed in Section 8.4 hold here. We can reuse their proofs without changes. Therefore, this modified technique preserves the completeness and optimality of CBS.

### 9.3. Corridor-target conflicts

Another interesting case occurs when the target vertex of an agent is inside the corridor.

**Example 12.** Fig. 16(a) shows the same example as in Fig. 12(left) except that the target vertex of agent $a_1$ is inside the corridor. If the two agents follow their individual shortest paths, they have an edge conflict at $(B3, C3)$ at timestep 3. Thus, when we use the corridor reasoning technique described in Section 8.1, we find a corridor $C = \{A3, B3, C3\}$ of length 2 and generate a pair of range constraints $(a_1, C3,[0,6])$ and $(a_2, B3,[0,5])$. In the left child CT node with the first constraint, agent $a_1$ waits until agent $a_2$ leaves the corridor and then starts to enter the corridor from A3 at timestep 5. In the right child CT node with the second constraint, however, we cannot find a shortest path for agent $a_2$ that does not conflict with agent $a_1$. In fact, the best resolution under this node is to first let agent $a_1$ travel through the corridor and leave D3, then agent $a_2$ enter the corridor from D3, and last agent $a_1$ reenter the corridor from D3. In other words, the paths of both agents have to be changed!  

This example shows that the previous corridor reasoning technique cannot resolve the corridor conflict in a single branching step because it stops detecting the corridor after it finds a target vertex. Therefore, in this subsection, we modify the corridor reasoning technique by allowing target vertices to be inside the corridor. Below are the details of the modified technique. In particular, we refer to a corridor conflict with one or two target vertices inside the corridor as a **corridor-target conflict**.

#### 9.3.1. Identifying corridor-target conflicts

For every vertex and edge conflict, we first find the corridor on-the-fly by checking whether the conflicting vertex (or an endpoint of the conflicting edge) is of degree 2. To find the endpoints of the corridor, we check the degree of each of the two adjacent vertices and repeat the procedure until we find a vertex whose degree is not 2. Then, we say the two agents are involved in a corridor conflict iff they have to cross each other inside the corridor. Note that we remove the condition that requires agents to move in opposite directions. This is because, when the start and target vertices are inside the corridor, it is possible that the two agents move in the same direction but still have an unavoidable conflict, e.g., the conflict in the second plot on the first row of Fig. 17.

We use a function $\text{MustCross}(a_1, a_2, C)$ to determine whether agents $a_1$ and $a_2$ have to cross each other in corridor $C$. We use vertex $b_i$ $(i = 1,2)$ to denote the start vertex of agent $a_i$ if it is inside the corridor and the endpoint from where agent $a_i$ enters the corridor otherwise. Similarly, we use vertex $d_i$ $(i = 1,2)$ to denote the target vertex of agent $a_i$ if it is inside the corridor and the endpoint from where agent $a_i$ leaves the corridor otherwise. If $b_1 \neq b_2$, $d_1 \neq d_2$, and the direction of moving from $b_1$ to $b_2$ is opposite to the direction of moving from $d_1$ to $d_2$, then agents $a_1$ and $a_2$ must cross each other. Fig. 17 shows more examples.

#### 9.3.2. Resolving corridor-target conflicts

We combine the corridor reasoning technique with the target reasoning technique to resolve corridor-target conflicts.

**Case 1: only one target vertex is inside the corridor** Without loss of generality, we assume that $g_1$ is inside the corridor and $g_2$ is not. Let us use Fig. 16(a) as a running example where the path of agent $a_2$ traverses the corridor from endpoint $e_1$ (i.e., cell D3 in Fig. 16(a)) to endpoint $e_2$ (i.e., cell A3 in Fig. 16(a)). However, agent $a_2$ might have alternative paths that can reach
Fig. 17. Examples for cases where the target vertices of agents $a_1$ and $a_2$ are inside corridor $C$. Only the cases shown in the first row are classified as corridor conflicts by Function MustCross($a_1, a_2, C$).

e_2$ without traversing the corridor (which is omitted in Fig. 16(a)). So it can choose to use the corridor or not. If it uses the corridor, then agent $a_1$ has to finally use the corridor after agent $a_2$ because it has to eventually wait at its target vertex forever. (1) If agent $a_1$ enters the corridor from endpoint $e_2$, then it has to let agent $a_2$ traverse through the corridor first. So the earliest timestep for it to enter the corridor from endpoint $e_2$ is $\max\{t_1(e_2), t_2(e_2) + 1\}$, and, therefore, the earliest timestep for it to reach its target vertex $g_1$ is $\max\{t_1(e_2), t_2(e_2) + 1\} + \text{dist}(e_2, g_1)$. (2) If agent $a_1$ enters the corridor from endpoint $e_1$, then the earliest timestep for it to reach its target vertex $g_1$ is $\max\{t_1(e_1), t_2(e_1) + 1\} + \text{dist}(e_1, g_1)$. In other words, if agent $a_1$ reaches its earliest target vertex at or before timestep $l = \min_{i=1,2}\{\max\{t_1(e_i) - 1, t_2(e_i)\} + \text{dist}(e_i, g_1)\}$, then agent $a_2$ cannot traverse through the corridor without conflicting with $a_1$, i.e., the earliest timestep for it to reach endpoint $e_2$ is $t_3(e_2)$ (i.e., using a bypass that does not traverse the corridor). Therefore, to resolve this corridor-target conflict, we generate two child CT nodes, each with one of the constraint sets $C_1 = \{l_1 > l\}$ and $C_2 = \{l_1 \leq l, (a_2, e_2, [0, t'_2(e_2) - 1])\}$.

**Case 2: both target vertices are inside the corridor** The reasoning is similar to Case 1. Let us use Fig. 16(b) as a running example. Agent $a_2$ has to enter the corridor to reach its target vertex $g_2$, and it can enter from either endpoint $e_1$ (i.e., cell A3 in Fig. 16(b)) or endpoint $e_2$ (i.e., cell A1 in Fig. 16(b)). If agent $a_2$ enters the corridor from endpoint $e_1$, then it has to traverse vertex $g_1$ before agent $a_1$ eventually reaches its target vertex $g_1$ and waits at vertex $g_1$ forever. (1) If agent $a_1$ enters the corridor from endpoint $e_2$, then it has to let agent $a_2$ traverse through the corridor first. So the earliest timestep for agent $a_1$ to enter the corridor from endpoint $e_2$ is $\max\{t_1(e_2), t_2(e_2) + 1\}$, and, therefore, the earliest timestep for agent $a_1$ to reach its target vertex $g_1$ is $\max\{t_1(e_2), t_2(e_2) + 1\} + \text{dist}(e_2, g_1)$. (2) If agent $a_1$ enters the corridor from endpoint $e_1$, then the earliest timestep for it to reach its target vertex $g_1$ is $\max\{t_1(e_1), t_2(e_1) + 1\} + \text{dist}(e_1, g_1)$. In other words, if agent $a_1$ reaches its target vertex $g_1$ at or before timestep $l = \min_{i=1,2}\{\max\{t_1(e_i) - 1, t_2(e_i)\} + \text{dist}(e_i, g_1)\}$, then agent $a_2$ cannot traverse through vertex $g_1$ without conflicting with agent $a_1$, i.e., the earliest timestep for agent $a_2$ to reach its target vertex $g_2$ is $t'_3(g_2)$, which represents the earliest timestep for agent $a_2$ to reach its target vertex $g_2$ via a bypass, i.e., a path that enters the corridor from vertex $e_2$. Therefore, to resolve this corridor-target conflict, we generate two child CT nodes, each with one of the constraint sets $C_1 = \{l_1 > l\}$ and $C_2 = \{l_1 \leq l, l_2 > t'_2(g_2) - 1\}$.

**9.3.3. Classifying corridor-target conflicts**
We reuse the method in Section 8.3 to classify corridor-target conflicts.

**9.3.4. Theoretical analysis**

**Property 10.** For all combinations of paths of agents $a_1$ and $a_2$ with a corridor-target conflict, if one path violates constraint set $C_1$ and the other path violates constraint set $C_2$, then the two paths have one or more vertex or edge conflicts inside the corridor. \(\square\)

Since we already intuitively prove Property 10 when we introduce constraint sets $C_1$ and $C_2$ in Section 9.3.2, we move the formal proof to Appendix D. Property 10 tells us that constraint sets $C_1$ and $C_2$ are mutually disjunctive, and thus, according to Theorem 2, using them to split a CT node preserves the completeness and optimality of CBS.

**9.4. Summary**

Up to now, we have discussed all types of generalized corridor conflicts, namely standard corridor conflicts (including the cases when start vertices are inside the corridor), corridor-target conflicts, and pseudo-corridor conflicts. Algorithm 3 shows the pseudo-code for generalized corridor reasoning.

Combining the theoretical analysis for each type of generalized corridor conflicts, we have the following theorem.
Algorithm 3: Generalized corridor reasoning.

Input: Vertex conflict \(c = (a_1, a_2, v, t)\) or edge conflict \(c = (a_1, a_2, v, u, t)\).

1. Construct the corridor \(C\) from vertex \(v\) or edge \((v, u)\).

2. if \(C\) is of length 1 then
   3. return \(\text{PseudoCorridorReasoning}(c)\);

4. if \(\text{MustCross}(a_1, a_2, C)\) returns \(\text{False}\) then
   5. return \(\text{"Not Corridor"}\);

6. if \(g_1\) and \(g_2\) are inside the corridor then
   7. \(l \leftarrow \min_{i=1,2}[\max(t_1(e_i) - 1, t_2(e_i)] + \text{dist}(e_i, g_i)];\)
   8. \(C_1 \leftarrow (l_1 > l);\)
   9. \(C_2 \leftarrow (l_1 \leq l, l_2 > l_2(g_2));\)

10. else if \(g_1\) or \(g_2\) is inside the corridor then
   11. WLOG, let \(a_1\) be the agent whose target vertex is inside the corridor;
   12. \(l \leftarrow \min_{i=1,2}[\max(t_1(e_i) - 1, t_2(e_i)] + \text{dist}(e_i, g_i)];\)
   13. \(C_1 \leftarrow (l_1 > l);\)
   14. \(C_2 \leftarrow (l_1 \leq l, l_2 \geq l_2(g_2));\)

15. else
   16. \(C_1 \leftarrow (a_1, e_1, [0, \min(t_1(e_1) - 1, t_2(e_1)] + \text{dist}(e_1, e_2)];\)
   17. \(C_2 \leftarrow (a_2, e_2, [0, \min(t_2(e_2) - 1, t_1(e_2)] + \text{dist}(e_2, e_1)];\)

18. if The path of \(a_1\) violates \(C_1\) and the path of \(a_2\) violates \(C_2\) then
   19. return \(C_1\) and \(C_2\);

   20. else
   21. return \(\text{"Not Corridor"}\);

Fig. 18. Runtime distribution of CBSH with different corridor reasoning techniques. In the two game-map figures, red and green lines are hidden by purple lines, while yellow lines are hidden by grey lines.

Theorem 8. Resolving generalized corridor conflicts with the constraint sets returned by Algorithm 3 preserves the completeness and optimality of CBS.

9.5. Empirical evaluation on generalized corridor reasoning

In this subsection, we empirically compare the effectiveness of all corridor reasoning techniques. The results are shown in Fig. 18. In particular, \textbf{None} represents CBSH, \textbf{C} represents CBSH with the basic corridor reasoning technique described in Section 8, \textbf{PC} represents CBSH with the basic corridor reasoning technique described in Section 8 plus the pseudo-corridor reasoning described in Section 9.1, \textbf{STC} represents CBSH with the basic corridor reasoning technique described in Section 8 plus the modification of handling start and target vertices differently, as described in Sections 9.2 and 9.3, and \textbf{GC} represents CBSH with generalized corridor reasoning, namely the reasoning technique shown in Algorithm 3.

Map \textbf{Empty} contains no obstacles and thus no corridors. So none of the corridor reasoning techniques can speed up CBSH, but luckily, they do not slow down CBSH either. Maps \textbf{Game1}, \textbf{City}, and \textbf{Game2} rarely have corridors, but they all have obstacles of various shapes, where pseudo-corridor reasoning can be useful. As a result, although \textbf{C} and \textbf{STC} do not improve the performance of CBSH, \textbf{PC} and \textbf{GC} do. Maps \textbf{Random}, \textbf{Warehouse}, \textbf{Room}, and \textbf{Maze} all have many corridors, and as a result, all corridor techniques speed up CBSH. Among all maps, the improvements on map \textbf{Maze} are the largest. Among all corridor reasoning techniques, \textbf{GC} is always the best.
Algorithm 4: Symmetry reasoning.

Input: Vertex conflict \( c = (a_1, a_2, v, t) \) or edge conflict \( c = (a_1, a_2, v, u, t) \).

1. \( (C_1, C_2) \leftarrow \text{GeneralizedCorridorReasoning}(c) \);
2. if \( (C_1, C_2) \neq \) “Not Corridor” then
   3. return “Corridor Conflict” and constraint sets \( [C_1, C_2] \);
4. if \( t \) is larger than the length of the path of agent \( a_1 \) or \( a_2 \) then
   5. \( (C_1, C_2) \leftarrow \text{TargetReasoning}(c) \);
   6. return “Target Conflict” and the constraint sets \( [C_1, C_2] \);
7. if \( t \) is a semi-/non-cardinal vertex conflict then
   8. \( (C_1, C_2) \leftarrow \text{GeneralizedRectangleReasoning}(c) \);
   9. if \( (C_1, C_2) \neq \) “Not Rectangle” then
      10. return “Rectangle Conflict” and the constraint sets \( [C_1, C_2] \);
11. \( (C_1, C_2) \leftarrow \text{StandardCBSplitting}(c) \);
12. return “Vertex/Edge Conflict” and the constraint sets \( [C_1, C_2] \);

10. Symmetry reasoning framework

Until now, we have described and empirically evaluated each symmetry reasoning technique independently. In this section, we present the complete framework of our pairwise symmetry reasoning technique, namely how to identify different classes of symmetry conflicts and, when multiple conflicts exist, which conflict to choose to resolve first. We then show some empirical results for combining all symmetry reasoning techniques together.

10.1. Framework

During the expansion of a CT node, we run symmetry reasoning for each vertex and edge conflict. Algorithm 4 shows the pseudo-code. We first run generalized corridor reasoning by calling Algorithm 3 (Line 1). If the input conflict \( c \) turns out not to be a corridor conflict, we then check whether it is a target conflict by comparing the path length of the agents with the conflicting timestep \( t \) (Line 4). If, say, agent \( a_1 \)'s path length is smaller than or equal to \( t \), then it is a target conflict, and we generate the constraint sets \( [C_1 = \{l_1 > t\}, C_2 = \{l_2 \leq t\}] \) by function TargetReasoning\((c)\) (Line 5). If conflict \( c \) is not a target conflict but a semi- or non-cardinal vertex conflict, we then run generalized rectangle reasoning by calling the algorithm described in Section 6.2 (Line 8). If conflict \( c \) turns out not to be any class of symmetric conflicts, we use the standard CBS splitting method to generate constraints (Line 11).

When choosing conflicts for expansion, we prioritize conflicts by resolving cardinal conflicts first, then semi-cardinal conflicts, and last non-cardinal conflicts. The cardinality of symmetric conflicts are determined during the symmetry reasoning procedure, although we do not show it explicitly in Algorithm 4. When there are multiple conflicts of the same cardinality, we break ties using the same motivation described in Section 3.2.1, i.e., in favor of conflicts that can increase the costs of the child CT nodes more. To be specific, we give target conflicts the highest priority because, when resolving a target conflict, the cost of at least one child node is larger than the cost of the current CT node by at least one and often by much more. Corridor conflicts have the second highest priority because, when resolving a corridor conflict, the costs of the child CT nodes can be more than one larger than the cost of the parent CT node. Rectangle conflicts have the third highest priority because, when resolving a rectangle conflict, the costs of the child CT nodes are typically at most one larger. Vertex and edge conflicts have the lowest priority because we prefer to resolve all symmetric conflicts first, and also, when resolving a vertex or edge conflict, the costs of the child CT nodes are typically at most one larger.

10.2. Empirical evaluation

In this subsection, we compare CBSH (denoted None), CBSH with the best variant of each of the reasoning technique, namely generalized rectangle reasoning (denoted GR), target reasoning (denoted T), and generalized corridor reasoning (denoted GC), and CBSH with their combination (denoted GR+T+GC, or RTC for short).

Runtimes and success rates Fig. 19 presents the runtimes, and Fig. 20 presents the success rates, i.e., the percentage of instances solved within the time limit of one minute. As expected, all of GR, T, and GC are able to speed up CBSH, and the significance of their speedup depends on the structure of the maps. The combination of them, i.e., RTC, is always the best. In Fig. 20, we notice an interesting behavior on many maps, such as Empty, Warehouse, Game, and City: the success rate improvements of the combination RTC is substantially larger than those of GR, T, and GC separately. This is because when an instance contains more than one class of symmetric conflicts, solving any class of symmetric conflicts with the standard splitting method of CBSH could result in unacceptable runtimes. Thus, CBSH with only one of the reasoning techniques does not solve many instances within the time limit, while CBSH with all techniques does.

Scalability To show the scalability of CBSH with and without our reasoning techniques, instead of using the instances described in Table 1, we run None and RTC on the same 6 maps with the number of agents increasing by one at a time,
starting from 2. We report the largest number of agents that each algorithm can solve with a success rate of 100% in Table 3. We see that, except for map Maze, RTC dramatically improves the scalability of CBSH, especially on large maps with many open space, such as maps Game1 (with an improvement of 13 times) and City (with an improvement of 30 times).

Size of CTs Fig. 21 compares the number of expanded CT nodes of None and RTC. We can see that our reasoning techniques can reduce the size of CTs by up to four orders of magnitude. Among the 890 instances that are solved by at least one of the algorithms, RTC performs worse than None only on 24 (= 2% of) instances and beats it on 782 (= 88% of) instances.

Runtime overhead Table 4 reports the runtime overhead of rectangle and corridor reasoning in RTC. The runtime overhead of rectangle reasoning mainly comes from manipulating MDDs because it has to search on the MDDs twice, once for finding the generalized rectangle and once for classifying rectangle conflicts. However, they can both be done relatively fast, and, as a result, the overall runtime overhead of rectangle reasoning is manageable, i.e., always less than 7% in Table 4. The runtime overhead of corridor conflicts mainly comes from calculating $t_i(x)$ and $t'_{i}(x)$, as each of them, in our implementation, is a
Fig. 21. CT node expansions of None and RTC. If an instance is not solved within the time limit, we set its node expansions infinite. Among the 1,200 instances, 310 instances are solved by neither algorithm; 418 instances are solved by RTC but not by None; and only 3 instances are solved by None but not by RTC. Among the 469 instances solved by both algorithms, RTC expands fewer nodes than None on 364 instances, the same number of nodes on 84 instances, and more nodes only on 21 instances.

| Map     | Rect | Corr |
|---------|------|------|
| Random  | 3.62%| 10.86%|
| Room    | 2.42%| 30.12%|

| Map     | Rect | Corr |
|---------|------|------|
| Empty   | 5.79%| 0.30%|
| Maze    | 0.14%| 0.57%|
| Warehouse | 1.26%| 5.69%|
| City    | 1.12%| 0.98%|

Table 5 reports how often RTC uses each reasoning technique to expand CT nodes, which also indicates how often different conflicts occur on different maps. Clearly, rectangle conflicts are more frequent on maps with more open space. An extreme case is on map Maze, where RTC does not branch on any rectangle conflicts as there is no open space on this map. Target conflicts are highly frequent on all maps for two reasons: one is that we always choose to resolve target conflicts first, and the other is that the likelihood of a target conflict happening is high given the high density of the agents in our instances and regardless of the structures of the maps. The only exception is map Maze, because there most target conflicts are classified as corridor-target conflicts by generalized corridor reasoning. Corridor conflicts are detected on all maps and frequent on maps with obstacles. Thanks to pseudo-corridor reasoning, we find many corridor conflicts not only on maps with many corridors, such as Random, Warehouse, Room, and Maze, but also maps with few or even zero corridors, such as Empty, Game1, City, and Game2. Rectangle, target, and corridor conflicts together account for approximately 70% of conflicts that are used to expand CT nodes on many of the maps. Together with the efficiency of our reasoning techniques and the effectiveness of our symmetry-breaking constraints, this high frequency results in the gains that we see in Figs. 19 to 21.

Conflict distribution. Table 4 reports how often RTC spent on rectangle reasoning (denoted “Rect”) and corridor reasoning (denoted “Corr”). The runtime overhead of target reasoning is negligible, and thus is not reported here.

| Map     | Rect | Corr |
|---------|------|------|
| Random  | 3.62%| 10.86%|
| Room    | 2.42%| 30.12%|

Two-agent analysis. An interesting question to our reasoning techniques is that: do rectangle, target, and corridor reasoning find all pairwise symmetries in MAPF? To answer this question, we design a two-agent experiment. Recall that CBSH2 (introduced in Section 3.2.2) calls CBSH to solve a 2-agent sub-MAPF instance for each pair of conflicting agents at each CT
node to compute heuristics.\textsuperscript{6} Here, we record the number of CT nodes to solve such 2-agent instances by None and RTC, respectively, and report the results in Table 6. Compared to None, RTC requires substantially fewer nodes to resolve 2-agent instances. And impressively, RTC is able to solve up to 99\% of 2-agent instances by expanding only one CT node. Even in the worst case, it solves 81\%. Except for map Maze, there are less than 0.5\% of instances that RTC expands more than 10 CT nodes to resolve. As for map Maze, the percentage is less than 1.5\%. Therefore, we conclude that RTC is able to identify most of the pairwise symmetries in MAPF.

\textbf{Conflict prioritization} In order to show the effectiveness of our proposed conflict prioritization strategy (i.e., for conflicts of the same cardinality, we first choose target conflicts, then corridor conflicts, rectangle conflicts, and last vertex and edge conflicts), we create a strawman algorithm rRTC that chooses conflicts of the same cardinality randomly and compare its performance with RTC in Table 7. On all maps, RTC solves 3.9\%-31.1\% more instances than rRTC, which clearly shows that our fine-grained conflict prioritization strategy that sorts conflicts according to their cardinality and then breaks ties according to their symmetry types is better than the existing conflict prioritization strategy [70] that sorts the conflicts according to their cardinality only.

\section{11. Empirical comparison with existing algorithms}

In this section, we compare our reasoning techniques with existing related algorithms, namely mutex propagation and CBSH2.

\subsection{11.1. Comparison with mutex propagation}

As we introduced in Section 4.2, mutex propagation is a symmetry reasoning technique that can identify all cardinal symmetric conflicts and resolve them with a pair of vertex constraint sets. To provide a extensive comparison of RTC and mutex propagation, we test four versions of CBSH: (1) CBSH with mutex propagation only (denoted $\textbf{M}$); (2) CBSH with RTC only (denoted $\textbf{RTC}$); (3) CBSH with both techniques where, for each vertex/edge conflict, we always perform mutex propagation first and then perform RTC only if mutex propagation fails to identify this conflict as a symmetric conflict.

\begin{table}[h]
\centering
\caption{Numbers of solved instances by rRTC and RTC within one minute. The total number of instances for each map is $25 \times 6 = 150$.}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Map & rRTC & RTC & Map & rRTC & RTC & Map & rRTC & RTC \\
\hline
Random & 97 & 113 & Empty & 116 & 126 & Warehouse & 90 & 118 \\
Room & 90 & 111 & Maze & 46 & 49 & City & 128 & 133 \\
Game2 & 150 & & & & & Game1 & 114 & 119 \\
\hline
\end{tabular}
\end{table}

\textsuperscript{6} In practice, CBSH2 does not do so for all pairs, as it uses a memoization technique to avoid solving the same 2-agent sub-MAPF instance (at different CT nodes) twice.
(denoted M+RTC); and (4) CBSH with both techniques where, for each vertex/edge conflict, we always perform RTC first and then perform mutex propagation only if RTC fails to identify this conflict as a symmetric conflict (denoted RTC+M).

Fig. 22 reports the runtime distribution of these four algorithms. First, RTC alone always performs better than mutex propagation alone. One of the reasons is that mutex propagation only reasons about cardinal symmetric conflicts but ignores semi- and non-cardinal symmetric conflicts. Therefore, when we apply RTC after mutex propagation, M+RTC performs better than M in many cases. However, RTC still always performs better than M+RTC for two reasons, i.e., mutex propagation has larger runtime overhead than RTC and mutex propagation uses vertex constraint sets to resolve target (and corridor-target) conflicts, which are less effective than the length constraints that RTC uses. The performance of RTC and RTC+M is competitive. In some cases, RTC is slightly better than RTC+M because RTC+M has larger runtime overhead. In other cases, RTC is slightly worse than RTC+M because mutex propagation can identify some cardinal symmetric conflicts that RTC fails to identify. The negligible improvement of RTC+M over RTC also implies that, although we developed RTC by enumerating possible symmetries manually, it is already able to identify most of the cardinal symmetric conflicts. In summary, RTC is a more effective symmetry reasoning technique than mutex propagation on the instances we test. The combination of them does not outperform RTC alone.

11.2. Comparison with CBSH2

CBSH2 uses CBSH to solve a two-agent sub-MAPF instance for each pair of agents in the original MAPF instance to generate informed heuristic guidance for the high-level search of CBS. We already show in Table 6 that our reasoning technique can significantly reduce the number of CT nodes required by CBSH for solving the two-agent instances. Now we show that our reasoning technique can also reduce the number of CT nodes required by CBSH2 for solving the original MAPF instance and thus reduce its runtime. In addition, we add the bypassing strategy [70] to CBSH2 which can greedily resolve some semi- and non-cardinal conflicts without branching.

Fig. 23 shows the runtime distribution of CBSH and CBSH2 with and without RTC. In particular, CBSH2+RTC uses RTC both in the main CBSH2 and in the two-agent sub-MAPF solver CBSH. As expected, both CBSH with RTC and CBSH2 outperform CBSH in most cases. In particular, RTC always performs better than CBSH2, which indicates that, although RTC and the heuristics used in CBSH2 both reasons about pairs of agents, RTC using symmetry-breaking constraints to resolve symmetries directly is more effective than CBSH2 relying on the heuristics to eliminate symmetries. Not surprisingly, CBSH2 with RTC performs the best as it makes use of both symmetry-breaking constraints and informed heuristics.

In order to show that the gain of RTC over CBSH2 is not just because it speeds up CBSH to solve the two-agent instances, we plot the number of CT nodes expanded by CBSH2 with and without RTC in Fig. 24. We see that RTC can reduce the size of CTs of CBSH2 by up to three orders of magnitude. Among the 961 instances that are solved by at least one of the algorithms, CBSH2 with RTC performs worse than CBSH2 only on 44 (≈ 5% of) instances and beats it on 676 (≈ 70% of) instances.

12. Summary and future work

Researchers have made significant progress on scaling up MAPF algorithms in the past decade. Most previous work focuses on developing advanced techniques for particular MAPF algorithms, like partial expansion for A*, node pruning for ICTS, and conflict selection for CBS. Here, we try to improve our understanding of what makes MAPF hard. The symmetry issues we identify must be eventually resolved by every optimal MAPF algorithm although the encodings are algorithm
specific. We give instantiations for classic MAPF with optimal CBS. Other recent work has applied these ideas in other optimal MAPF algorithms like BCP [23,14], bounded-suboptimal MAPF algorithms like EECBS [15], and other MAPF variants like k-robust MAPF [22].

We showed that symmetric conflicts arise extremely frequently in MAPF. Rectangle conflicts occur when two agents must cross paths and have many equivalent ways to do so. The generalized rectangle reasoning applies to any planar graphs, which represents almost all the real-world circumstances for MAPF problems in 2D scenarios. Generalized corridor and target reasoning concentrate on spatial and temporal reasoning where we try to avoid symmetries resulting from multiple waiting actions. Both of them are applicable to any graphs and critical problems for one of the main commercial uses of MAPF, namely routing robots in automated warehouses. We showed that our reasoning techniques scaled up CBSH by up to thirty times and reduced its node expansion by up to four orders of magnitude. They significantly outperformed mutex propagation and significantly improved CBSH2.

There remain many open questions. As Table 6 indicates, although our reasoning techniques resolve most pairwise symmetries in a single branching step, there remain some undetected pairwise symmetries. Also, complex interactions between more than two agents can arise in congested settings. Our work can also be extended to more complex MAPF problems. For example, in k-robust MAPF [69], agents need to keep safety time between each other. So, agents being at the same vertex at different timesteps can conflict with each other, which introduces more types of symmetric conflicts. The rectangle, target and corridor reasoning techniques have been shown to be effective there [22]. Similarly, in large-agent MAPF [20], agents are of different sizes. So, agents at different vertices/edges can conflict with each other, which also introduces more types of symmetric conflicts. In addition, if we allow agents to have different speeds, then a chasing symmetry arises when a fast agent tries to overtake a slow agent.
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.

Acknowledgement

The research at Monash University was partially supported by Australian Research Council (ARC) under grant numbers DP200100025 and DP190100013 as well as a gift from Amazon. The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779, and 1935712 as well as a gift from Amazon. The research at Simon Fraser University was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant number RGPIN2020-06540. 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.

Appendix A. Proof for rectangle reasoning techniques

Property 2. For agents $a_1$ and $a_2$ with a rectangle conflict found by the rectangle reasoning technique I, all paths for agent $a_1$ that visit a node on the exit border $R_1R_g$ must visit a node on the entry border $R_1R_2$, and all paths for agent $a_2$ that visit a node on the exit border $R_2R_g$ must visit a node on the entry border $R_1R_1$.

Proof. We assume that the vertex conflict between agents $a_1$ and $a_2$ is at node $C$. We then assume $S_{1,x} \leq G_{1,x}$ and $S_{1,y} \leq G_{2,y}$ without loss of generality (because MAPF is invariant under rotations of axes). From Equations (3) and (4), we know $S_{2,x} \leq G_{2,x}$ and $S_{2,y} \leq G_{2,y}$. Since Equations (1) and (2) ensure that the paths for the two agents from their start to target nodes are within the conflicting area, we know that the cell of node $C$ is also within the conflicting area, i.e.,

\[
\begin{align*}
\max\{S_{1,x}, S_{2,x}\} & \leq C_x \leq \min\{G_{1,x}, G_{2,x}\} \\
\max\{S_{1,y}, S_{2,y}\} & \leq C_y \leq \min\{G_{1,y}, G_{2,y}\}
\end{align*}
\] (A.1)

Then, since the two agents reach cell $(C_x, C_y)$ at the same time via their Manhattan-optimal paths, we know

\[
(C_x - S_{1,x}) + (C_y - S_{1,y}) = (C_x - S_{2,x}) + (C_y - S_{2,y}),
\] (A.3)

which can be simplified to

\[
S_{1,x} + S_{1,y} = S_{2,x} + S_{2,y}.
\] (A.4)

We assume that $S_{1,x} \leq S_{2,x}$ without loss of generality (because MAPF is invariant under swaps of the indexes of agents), which implies $S_{1,y} \geq S_{2,y}$. According to the definition of the four corners of the rectangle in Definition 6, we have $R_{c,x} = S_{2,x}, R_{c,y} = S_{1,y}, R_g,x = \min\{G_{1,x}, G_{2,x}\} \geq S_{2,x}, R_g,y = \min\{G_{1,y}, G_{2,y}\} \geq S_{1,y}, R_1,x = R_g,x, R_1,y = S_{1,y}, R_2,x = S_{2,x}$ and $R_2,y = R_g,y$. Thus,

\[
S_{1,x} \leq S_{2,x} = R_{c,x} = R_2,x \leq R_g,x = R_1,x
\] (A.5)

\[
S_{2,y} \leq S_{1,y} = R_{c,y} = R_1,y \leq R_g,y = R_2,y.
\] (A.6)

Consequently, the relative locations of the start, target and rectangle corner nodes are exactly the same as given in Fig. 3. Since the $S_1$-$R_g$ rectangle and the $R_2$-$R_g$ rectangle are of the same length (i.e., $|S_1,x - R_g,x| = |R_2,x - R_g,x| = |R_1,x - R_g,x|$) and any sub-path $p_1$ from node $S_1$ to a node on border $R_1R_g$ must be Manhattan-optimal, sub-path $p_1$ must visit a node on border $R_1R_2$. Similarly, since the $S_2$-$R_g$ rectangle and the $R_2$-$R_g$ rectangle are of the same width (i.e., $|S_2,y - R_g,y| = |R_2,y - R_g,y| = |R_1,y - R_g,y|$) and any sub-path from node $S_2$ to a node on border $R_2R_g$ must be Manhattan-optimal, sub-path $p_2$ must visit a node on border $R_1R_1$. Therefore, the property holds.

Property 6. For agents $a_1$ and $a_2$ with a rectangle conflict found by the rectangle reasoning technique II, all paths for agent $a_1$ that visit a node constrained by $B(a_1, R_1, R_g)$ must visit a node on the entry border $R_1R_2$, and all paths for agent $a_2$ that visit a node constrained by $B(a_2, R_2, R_g)$ must visit a node on the entry border $R_1R_1$.

\[7\text{ Note that, when } S_{1,x} = G_{1,x}, \text{ it is possible that } S_{2,x} > G_{2,x} \text{ by Equation (3). In this case, we flip the } x \text{ axis so that } S_{1,x} \leq G_{1,x} \text{ and } S_{2,x} \leq G_{2,x} \text{ both hold. Same for the } y \text{ axis.}\]
Proof. By Property 5, we need to prove that any path for agent $a_1$ from its start node $S_1$ to one of the nodes constrained by $B(a_1, R_1, R_g)$ must visit a node on the entry border $R_1 R_2$ and any path of agent $a_2$ from its start node $S_2$ to one of the nodes constrained by $B(a_2, R_2, R_g)$ must visit a node on the entry border $R_2 R_1$. This holds by applying the proof for Property 2 after replacing Equation (A.4) by Equation (7).

Appendix B. Proof for the generalized rectangle reasoning technique

For a given generalized rectangle $G = (\mathcal{V}, \mathcal{E})$, we use $\mathcal{V}' = \{u| (u, t) \in \mathcal{V}\}$ to denote the vertices in the conflicting area.

Lemma 9. Any path for agent $a_i (i = 1, 2)$ that visits a node in the generalized rectangle $\mathcal{V}$ must visit an entry edge in $E_i$.

Proof. Consider an arbitrary path $p$ for agent $a_i$ that visits a node in $\mathcal{V}$. Let edge $e = ((u, t), (w, t + 1))$ be the edge on path $p$ such that $(u, t) \notin \mathcal{V}$ and $(w, t + 1) \in \mathcal{V}$. Since $(w, t + 1) \in \mathcal{V}$, node $(w, t + 1)$ is in MDD$_i$. By Property 4, node $(u, t)$ is also in MDD$_i$. By the definition of the entry edges in Definition 8, edge $e \in E_i$. Therefore, any path for agent $a_i$ that visits a node in $\mathcal{V}$ must visit one of the entry edges in $E_i$.

Lemma 10. Any path for agent $a_i (i = 1, 2)$ that visits a node in $\mathcal{V}$ must visit one of the entry edges in $E^b_i$.

Proof. According to Lemma 9 and the fact that $E_i = E^b_i \cup E^g_i$, we only need to prove that any path for agent $a_i$ that visits an edge in $E^g_i$ also visits an edge in $E^b_i$. Consider an arbitrary path $p$ for agent $a_i$ that visits an edge $e = ((u, t), (w, t + 1))$ in $E^g_i$. In geometry, since vertex $s_i$ is outside the conflicting area while vertex $u$ is in a hole, path $p$ must visit at least one vertex in $\mathcal{V}'$. We use $u'$ to denote the first vertex in $\mathcal{V}'$ visited by path $p$, $u''$ to denote the vertex visited by path $p$ right before vertex $u'$, and $(u'', t_{u''})$ to denote the corresponding node in $\mathcal{V}$. By Definition 7, node $(u'', t_{u''})$ is the only MDD node in MDD$_i$ that visits vertex $u'$. By Property 4 and the fact that node $(w, t + 1)$ is in MDD$_i$, all nodes before timestep $t + 1$ on path $p$, including the node whose vertex is $u'$, are in MDD$_i$. So path $p$ visits vertex $u'$ at timestep $t_{u''}$ and vertex $u''$ at timestep $t_{u''} - 1$. So $(u'', t_{u''} - 1) \notin \mathcal{V}$, $(u', t_{u'}) \in \mathcal{V}$, and both node $(u'', t_{u''} - 1)$ and node $(u', t_{u'})$ are in MDD$_i$. Therefore, edge $e' = ((u'', t_{u''} - 1), (u', t_{u'}))$ is an entry edge in $E^b_i$. Therefore, the lemma holds.

Property 8. For all combinations of paths of agents $a_1$ and $a_2$ with a generalized rectangle conflict, if one path violates $B(a_1, R_1, R_g)$ and the other path violates $B(a_2, R_2, R_g)$, then the two paths have one or more vertex conflicts within the conflicting area $G$.

Proof. Since all nodes prohibited by $B(a_1, R_1, R_g)$ $(i = 1, 2)$ are in $\mathcal{V}$, from Lemma 10, any path for agent $a_i (i = 1, 2)$ that visits a node prohibited by $B(a_i, R_i, R_g)$ must visit one of the entry edges in $E^b_i$. The four nodes $R_s, R_g, R_1$ and $R_2$ cut the border of the generalized rectangle $G$ into four segments $R_1 R_2$, $R_2 R_g$, $R_g R_1$ and $R_1 R_2$, denoted $Seg_1$, $Seg_2$, $Seg_3$ and $Seg_4$, respectively. The “to” nodes of all entry edges in $E^b_i$ are on $Seg_1$, and the “to” nodes of all entry edges in $E^g_i$ are on segment $Seg_4$. The nodes prohibited by $B(a_1, R_1, R_g)$ are on segment $Seg_3$, and the nodes prohibited by $B(a_2, R_2, R_g)$ are on segment $Seg_2$. Therefore, we only need to prove that any path $p_1$ for agent $a_1$ that visits a node on $Seg_1$ and a node on segment $Seg_3$ must conflict with any path $p_2$ for agent $a_2$ that visits a node on $Seg_4$ and a node on segment $Seg_2$. By the geometric property, paths $p_1$ and $p_2$ must cross each other, i.e., must visit at least one common vertex $u$. According to Section 6.2.3, vertex $u$ is not in one of the holes, i.e., $u \in \mathcal{V}$. Let node $(u, t_u)$ be the corresponding node in $\mathcal{V}$. Then both path $p_1$ and path $p_2$ must visit node $(u, t_u)$, i.e., they conflict at vertex $u$ at timestep $t_u$. Therefore, the property holds.

Appendix C. Proof for the corridor reasoning technique

Property 9. For all combinations of paths of agents $a_1$ and $a_2$ with a corridor conflict, if one path violates $\{ (a_1, e_1, [0, \min(t'_1(e_1) - 1, t_2(e_2) + k)]) \}$ and the other path violates $\{ (a_2, e_2, [0, \min(t'_2(e_2) - 1, t_1(e_1) + k)]) \}$, then the two paths have one or more vertex or edge conflicts inside the corridor.

Proof. Let path $p_1$ be an arbitrary path of agent $a_1$ that visits vertex $e_1$ at timestep $t_1 \in [0, \min(t'_1(e_1) - 1, t_2(e_2) + k)]$ and path $p_2$ be an arbitrary path of agent $a_2$ that visits vertex $e_2$ at timestep $t_2 \in [0, \min(t'_2(e_2) - 1, t_1(e_1) + k)]$. We need to prove that paths $p_1$ and $p_2$ have one or more vertex or edge conflicts inside the corridor.

Since $t_1 \leq \min(t'_1(e_1) - 1, t_2(e_2) + k) \leq t'_1(e_1) - 1 < t'_1(e_1)$ (where $t'_1(e_1)$ is the earliest timestep when agent $a_1$ can reach vertex $e_1$ without using the corridor between vertices $e_1$ and $e_2$), path $p_1$ must traverse the corridor. Similarly, path $p_2$ must traverse the corridor as well.

Since $t_1 \leq \min(t'_1(e_1) - 1, t_2(e_2) + k) \leq t_2(e_2) + k$ (where $k$ is the distance between vertices $e_1$ and $e_2$), the latest timestep when path $p_1$ visits vertex $e_2$ is no larger than timestep $t_2(e_2)$. $t_2(e_2)$ is the earliest timestep when path $p_2$ can visit vertex $e_2$, so path $p_1$ visits vertex $e_2$ before path $p_2$. Similarly, path $p_2$ visits vertex $e_1$ before path $p_1$. Therefore, paths $p_1$ and $p_2$ must have a conflict in the corridor between vertices $e_1$ and $e_2$. Therefore, the property holds.
Appendix D. Proof for the corridor-target reasoning technique

**Property 10.** For all combinations of paths of agents $a_1$ and $a_2$ with a corridor-target conflict, if one path violates constraint set $C_1$ and the other path violates constraint set $C_2$, then the two paths have one or more vertex or edge conflicts inside the corridor.

**Proof.** Since a path of agent $a_1$ cannot violate the length constraints $l_1 > l$ and $l_1 ≤ l$ simultaneously, we only need to consider the case where a path of agent $a_1$ violates $l_1 > l$ and a path of agent $a_2$ violates $(a_2, e_2, [0, t^*_2(e_2) − 1])$ or $l_2 > t^*_2(g_2)$.

**Case 1** Let us first consider the case where the target vertex of agent $a_2$ is not inside the corridor. Let path $p_1$ be an arbitrary path of agent $a_1$ that is of length no larger than $l$ and path $p_2$ be an arbitrary path of agent $a_2$ that visits vertex $e_2$ at timestep $t_2 \in [0, t^*_2(e_2) − 1]$. We need to prove that paths $p_1$ and $p_2$ have one or more vertex or edge conflicts inside the corridor. Since $t_2 ≤ t^*_2(e_2) − 1 < t^*_2(e_2)$ (where $t^*_2(e_2)$ is the earliest timestep when agent $a_2$ can reach vertex $e_2$ without using the corridor between vertices $e_1$ and $e_2$), path $p_2$ must traverse the corridor. Since the target vertex of $a_1$ is inside the corridor, eventually path $p_1$ must enter the corridor via endpoints $e_1$ or $e_2$ without leaving again. Assume that path $p_1$ enters the corridor via endpoint $e_i$ (for $i = 1, 2$) at timestep $t_1$ (without leaving again), then

$$
t_1 ≤ |p_1| − \text{dist}(e_i, g_1) ≤ l − \text{dist}(e_i, g_1) = \min_{i=1,2} \left( \max\left(\text{dist}(e_i, g_1), \text{dist}(e_i, g_1) \right) − \text{dist}(e_i, g_1) \right) ≤ \max\left(\text{dist}(e_i, g_1), \text{dist}(e_i, g_1) \right) ≤ t^*_2(e_i). \tag{D.1}$$

where $|p_1|$ represents the length of path $p_1$. This equation indicates that path $p_1$ enters the corridor via endpoint $e_i$ at or before path $p_2$ without leaving again. Therefore, paths $p_1$ and $p_2$ must have one or more vertex or edge conflicts inside the corridor.

**Case 2** Now let us consider the case where the target vertices of both agents are inside the corridor. Let path $p_1$ be an arbitrary path of agent $a_1$ that is of length no larger than $l$ and path $p_2$ be an arbitrary path of agent $a_2$ that is of length no larger than $t^*_2(g_2) − 1$. We need to prove that paths $p_1$ and $p_2$ have one or more vertex or edge conflicts inside the corridor. Since $|p| ≤ t^*_2(g_2) − 1 < t^*_2(g_2)$ (where $t^*_2(g_2)$ is the earliest timestep when agent $a_2$ can reach its target vertex $g_2$ via vertex $e_2$), path $p_2$ must reach its target vertex $g_2$ via vertex $e_1$, i.e., path $p_2$ reaches its target vertex $g_2$ after leaving again. If path $p_1$ enters the corridor via endpoint $e_1$, then path $p_1$ reaches its target vertex $g_2$ via vertex $g_1$. So paths $p_1$ and $p_2$ have one or more vertex or edge conflicts inside the corridor. If path $p_1$ enters the corridor via endpoint $e_1$, say at timestep $t_1$, then according to Equation (D.1), we know $t_1 ≤ t^*_2(e_1)$, which indicates that path $p_1$ enters the corridor via endpoint $e_1$ at or before path $p_2$ without leaving again. Therefore, paths $p_1$ and $p_2$ must have one or more vertex or edge conflicts inside the corridor.

Therefore, the property holds.

**References**

[1] R. Stern, N.R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T.K.S. Kumar, E. Boyarski, R. Bartak, Multi-agent pathfinding: definitions, variants, and benchmarks, in: Proceedings of the International Symposium on Combinatorial Search, SoCS, 2019, pp. 151–159.

[2] P.R. Wurman, R. D’Andrea, M. Mounz, Coordinating hundreds of cooperative, autonomous vehicles in warehouses, AI Mag. 29 (2008) 9–20.

[3] J. Li, A. Tinka, S. Kiesel, J.W. Durham, T.K.S. Kumar, S. Koenig, Lifelong multi-agent path finding in large-scale warehouses, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI, 2021, pp. 11272–11281.

[4] K. Dresner, P. Stone, A multiagent approach to autonomous intersection management, J. Artif. Intell. Res. 31 (2008) 591–656.

[5] W. Höing, J.A. Preiss, T.S. Kumar, G.S. Sukhatme, N. Ayanian, Trajectory planning for quadrotor swarms, IEEE Trans. Robot. 34 (2018) 856–869.

[6] D. Silver, Cooperative pathfinding, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE, 2005, pp. 117–122.

[7] J. Yu, S.M. LaValle, Structure and intractability of optimal multi-robot path planning on graphs, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI, 2013, pp. 1444–1449.

[8] 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.

[9] B. Nebel, On the computational complexity of multi-agent pathfinding on directed graphs, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2020, pp. 212–216.

[10] J. Yu, Intractability of optimal multi-robot path planning on planar graphs, IEEE Robot. Autom. Lett. 1 (2016) 33–40.
[50] D. Gnad, A. Torralba, A. Shleyfman, J. Hoffmann, Symmetry breaking in star-topology decoupled search, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2017, pp. 125–134.
[51] G. Röger, S. Sievers, M. Katz, Symmetry-based task reduction for relaxed reachability analysis, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2018, pp. 208–217.
[52] J. Puget, On the satisfiability of symmetrical constrained satisfaction problems, in: Proceedings of International Symposium on the Methodologies for Intelligent Systems (ISMIS), vol. 689, 1993, pp. 350–361.
[53] B. Benhamou, Study of symmetry in constraint satisfaction problems, in: Proceedings of the Workshop on Principles and Practice of Constraint Programming, PPCP, 1994, pp. 246–254.
[54] R. Backofen, S. Will, Excluding symmetries in constraint-based search, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP, 1999, pp. 73–87.
[55] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Breaking row and column symmetries in matrix models, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP, 2002, pp. 462–476.
[56] J. Puget, Symmetry breaking using stabilizers, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP, 2003, pp. 585–599.
[57] J. Puget, Automatic detection of variable and value symmetries, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), vol. 3709, 2005, pp. 475–489.
[58] M. Sellmann, P.V. Hentenryck, Structural symmetry breaking, in: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 2005, pp. 298–303.
[59] T. Walsh, General symmetry breaking constraints, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP, 2006, pp. 650–664.
[60] D.A. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, B.M. Smith, Symmetry definitions for constraint satisfaction problems, Constraints 11 (2006) 115–137.
[61] Y.-C. Law, J.H.-M. Lee, Symmetry breaking constraints for value symmetries in constraint satisfaction, Constraints 11 (2006) 221–267.
[62] C. Mears, M.J.G. de la Banda, M. Wallace, B. Demoen, A novel approach for detecting symmetries in CSP models, in: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR, 2008, pp. 158–172.
[63] J.M. Crawford, M.L. Ginsberg, E.M. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, KR, 1996, pp. 148–159.
[64] F.A. Aloul, K.A. Sakallah, J.L. Markov, Efficient symmetry breaking for Boolean satisfiability, IEEE Trans. Comput. 55 (2006) 549–558.
[65] E.A. Emerson, A.P. Sistla, Symmetry and model checking, Form. Methods Syst. Des. 9 (1996) 105–131.
[66] D. Bosnacki, M. Scheffer, Partial order reduction and symmetry with multiple representatives, in: Proceedings of the International Symposium on NASA Formal Methods (NFM), vol. 9058, 2015, pp. 97–111.
[67] P. Cheng, E. Frazzoli, S.M. LaValle, Improving the performance of sampling-based motion planning with symmetry-based gap reduction, IEEE Trans. Robot. 24 (2008) 488–494.
[68] M. Phillips, M. Likhachev, SIPP: safe interval path planning for dynamic environments, in: Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, 2011, pp. 5628–5635.
[69] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Barták, N.-F. Zhou, Robust multi-agent path finding, in: Proceedings of the Annual Symposium on Combinatorial Search, SoCS, 2018, pp. 2–9.
[70] E. Boyarski, A. Felner, G. Sharon, R. Stern, Don’t split, try to work it out: bypassing conflicts in multi-agent pathfinding, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2015, pp. 47–51.