New Techniques for Pairwise Symmetry Breaking
in Multi-Agent Path Finding

Jiaoyang Li,[1] Graeme Gange,[2] Daniel Harabor,[2] Peter J. Stuckey,[2] Hang Ma[3] and **Sven Koenig[1]**

[1]University of Southern California, [2]Monash University, [3]Simon Fraser University

jiaoyanl@usc.edu, {graeme.gange, daniel.harabor, peter.stuckey}@monash.edu, hangma@sfu.ca, skoenig@usc.edu

Abstract

We consider two new classes of pairwise path symmetries which appear in the context of Multi-Agent Path Finding (MAPF). The first of them, *corridor symmetry*, arises when two agents attempt to pass through the same narrow passage in opposite directions. The second, *target symmetry*, arises when the shortest path of one agent passes through the target location of a second agent after the second agent has already arrived at it. These symmetries can produce an exponential explosion in the space of possible collision resolutions, leading to unacceptable runtimes even for state-of-the-art MAPF algorithms such as Conflict-Based Search (CBS). We propose to *break these symmetries* using new reasoning techniques that: (1) detect each class of symmetry and, (2) resolve them by introducing specialized constraints. We experimentally show that our techniques can, in some cases, more than double the success rate of CBS and improve its runtime by one order of magnitude.

1 Introduction

Multi-Agent Path Finding (MAPF) is a problem that requires one to compute a set of collision-free paths on a given graph for a team of moving agents while minimizing the makespan or the sum of path lengths. The problem appears in variety of applications including warehouse logistics (Wurman, D'Andrea, and Mountz 2008), traffic management (Dresner and Stone 2008), aircraft towing (Morris et al. 2015) and computer games (Silver 2005). MAPF is known to be NP-hard (Yu and LaValle 2013). It remains hard even under a variety of simplifying assumptions. One such setting, also NP-hard (Banfi, Basilico, and Amigoni 2017) but popular in practice, models the operating environment as a 4-neighbor grid. Agents can either move from one unblocked cell to an adjacent unblocked cell or wait in place. We use benchmarks of this form for all our experiments, although our techniques also work for general graphs.

One of the leading methods for solving MAPF optimally (Gange, Harabor, and Stuckey 2019; Li et al. 2019a) employs a strategy known as Conflict-Based Search (CBS) (Sharon et al. 2015). The central idea is to plan paths for each agent independently and resolve collisions between two agents by branching. Each branch is a new candidate
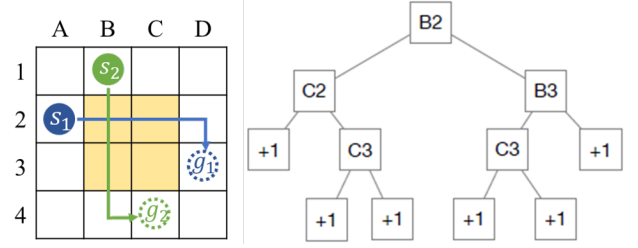
Figure 1: An example of rectangle symmetry. The left figure shows two shortest paths for two agents a_1 and a_2 that move them from cells A2 and B1 to cells D3 and C4, respectively, and collide at cell B2 at timestep 1. The right figure shows the constraint tree (CT) generated by CBS. Each left branch constrains agent a_2, while each right branch constrains agent a_1. Each non-leaf CT node is marked with the cell of the chosen collision. Each leaf CT marked "+1" contains an optimal solution, whose sum of path lengths is one larger than the sum of path lengths of the plan in the root CT node.

plan wherein one agent or the other is forced to find a new path that avoids the chosen collision.

Recent work (Li et al. 2019c) shows that this strategy, i.e., branching and then replanning, suffers from unacceptable runtimes when the collision at hand is *symmetric*. Figure 1 shows an example of a *rectangle symmetry*. There exist, for each agent, multiple shortest paths, each of which can be derived, one from the other, by changing the order of the individual RIGHT and DOWN moves. Any shortest path for one agent is in collision with any shortest path for the other agent. The only feasible resolution is for one of the agents to wait or take a detour. However, to generate such a path, CBS has to branch multiple times, and each branch potentially doubles the amount of search remaining. This rectangle symmetry arises because of the use of 4-neighbor grids.

In this work, we explore two new classes of such pairwise path equivalency, namely *corridor symmetry* and *target symmetry*. Like their rectangle counterpart, each one describes a specific situation that arises in MAPF. But both of these symmetries are applicable to MAPF on arbitrary graphs rather than just 4-neighbor grids. Moreover, each one is commonly found in current benchmark domains and, therefore, in practice. The behavior of CBS in such symmetric situations is the same, resulting in unacceptable runtimes

due to an explosion of branches. To handle these new symmetries, we introduce new constraint-based reasoning techniques designed to detect corridor and target symmetries and to resolve them in a single branching step. Our experimental results show that these new constraints can, in some cases, more than double the success rate of CBS and improve its runtime by one order of magnitude.

2 Problem Definition

MAPF has many variants (Stern et al. 2019) and, in this paper, we focus on the variant defined in (Stern et al. 2019) 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 an undirected 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 (goal) 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 the start vertex s_i and ending at the target vertex g_i. Agents remain at their target vertices after they complete their paths. A *conflict* (or, synonymously, *collision*) is either a *vertex conflict* $\langle a_i, a_j, v, t \rangle$, where agents a_i and a_j are at the same vertex $v \in V$ at the same timestep t, or an *edge conflict* $\langle a_i, a_j, u, v, t \rangle$, where 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). 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).

3 Background: Conflict-Based Search

Conflict-Based Search (CBS) (Sharon et al. 2015) is a two-level state-of-the-art search algorithm for solving MAPF optimally. At the low level, CBS invokes state-time A* (Silver 2005) to find a shortest path for each agent that satisfies constraints added by the high level. It breaks ties by preferring the path that has the fewest conflicts with the paths of other agents. At the high level, CBS performs a best-first search on a binary *constraint tree* (CT). Each CT node contains a *plan*, i.e., a set of paths, one for each agent, and a set of constraints that are used to coordinate agents and avoid conflicts. 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 a set of shortest paths, one for each agent. CBS proceeds from one CT node to the next one, checking for conflicts and calling its low-level search to replan paths one at a time. CBS succeeds when the plan of the current CT node is conflict-free, which corresponds to an optimal solution.

Constraints   A constraint is a spatio-temporal restriction introduced by CBS to resolve situations where the paths of two agents conflict. Specifically, a *vertex constraint* $\langle a_i, v, t \rangle$ means that agent a_i is prohibited from being at vertex $v \in V$ at timestep t. Similarly, an *edge constraint* $\langle a_i, u, v, t \rangle$ means that agent a_i is prohibited from traversing edge $(u, v) \in E$ at timestep t (or more precisely, from timestep $t-1$ to timestep t).

Branching   When expanding a CT node, CBS checks for conflicts in the plan of the CT node. If there are none, the CT node is a goal CT node, and CBS terminates. Otherwise, CBS 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 replanned by the low-level search. All other paths remain unchanged. CBS guarantees completeness by exploring both ways of resolving each conflict. CBS guarantees optimality by performing best-first searches on both its high and low levels.

Cardinal, semi-cardinal and non-cardinal conflicts   Boyarski et al. (2015) classify conflicts into three types. A conflict is *cardinal* iff, when CBS uses the conflict to split CT node N, the costs of both resulting child CT nodes are larger than the cost of CT node N. A conflict is *semi-cardinal* iff the cost of one child CT node is larger than the cost of CT node N, and the cost of the other child CT node is equal to the cost of CT node N. Finally, a conflict is *non-cardinal* iff the costs of both child CT nodes are equal to the cost of CT node N. They show that CBS can significantly improve its efficiency by choosing to resolve 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.

Rectangle symmetry   Li et al. (2019c) analyze rectangle symmetry for grid-based MAPF and introduce barrier constraints to resolve it efficiently. For the example in Figure 1, the rectangle symmetry is resolved by splitting the root CT node into two child CT nodes, one with a barrier constraint that prohibits agent a_2 from being at cell B3 at timestep 2 or cell C3 at timestep 3, and one with a barrier constraint that prohibits agent a_1 from being at cell C2 at timestep 2 or cell C3 at timestep 3. In each child CT node, one of the agents cannot take a path of length 4. Hence, the barrier constraints immediately increase the lower bound of the CT by 1, thus avoiding an exponential explosion of the runtime, and each child CT node contains a pair of conflict-free paths. See (Li et al. 2019c) for more details.

4 Corridor Symmetry

A *corridor* $C = C_0 \cup \{b, e\}$ of graph $G = (V, E)$ is a chain of connected vertices $C_0 \subseteq V$, each of degree 2, together with two endpoints $\{b, e\} \in V$ connected to C_0. The *length* of the corridor is the distance between its two endpoints, i.e., the number of vertices in C_0 plus 1. Figure 2 shows a corridor of length 3 made up of $C_0 = \{B3, C3\}$, $b = $ A3 and $e = $ 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*. Figure 2 shows an example. CBS detects the edge conflict $\langle a_1, a_2, B3, C3, 3 \rangle$ and branches, thereby generating
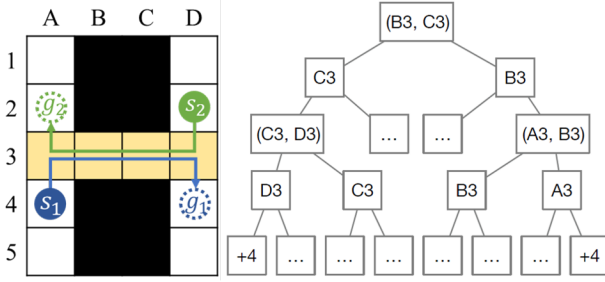
Figure 2: An example of corridor symmetry. The left figure shows the shortest paths of two agents a_1 and a_2 that have an edge conflict inside the corridor at edge (B3, C3) at timestep 3. The right figure shows the CT. Each left branch constrains agent a_2, while each right branch constrains agent a_1. Each non-leaf CT node is marked with the vertex/edge of the chosen conflict. Each leaf CT node marked "+4" contains an optimal solution, whose sum of costs is the cost of the root CT node plus 4. Each leaf CT node marked "..." contains a plan with conflicts and eventually produces suboptimal solutions in its descendant CT nodes.

Table 1: Number of expanded CT nodes to resolve a corridor conflict for different corridor lengths k.

k	3	5	7	9	11	13
Nodes	16	64	256	1,024	4,096	16,384

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) – 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 at least four times to find conflict-free paths in such a situation and has to branch even more times to prove their optimality. Figure 2 (right) shows the corresponding CT. Only two of the sixteen leaf CT nodes contain optimal solutions. This example highlights an especially pernicious characteristic of the 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.

Table 1 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}. We therefore propose a new reasoning technique which can identify and resolve corridor conflicts efficiently.

4.1 Identifying Corridor Conflicts

The detection of corridor conflicts is straightforward. We check every vertex and edge conflict. A vertex/edge conflict is a corridor conflict iff the conflict occurs inside a corridor, and the two agents involved in the conflict are coming from opposite directions. 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.

4.2 Resolving Corridor Conflicts

Consider a corridor C of length k with endpoints b and e. Assume that a shortest path of agent a_1 traverses the corridor from b to e and a shortest path of agent a_2 traverses the corridor from e to b. They conflict with each other inside the corridor. Let t_1 be the earliest timestep when agent a_1 can reach e and t_2 be the earliest timestep when agent a_2 can reach b.

We first assume that there are no *bypasses* (i.e., paths that move the agent from its start vertex to its target vertex without traversing corridor C) for either agent. 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 is $t_1 + 1$. Therefore, the earliest timestep when agent a_2 can reach b is $t_1 + 1 + k$. Similarly, if we prioritize agent a_2 and let agent a_1 wait, then the earliest timestep when agent a_1 can reach e is $t_2 + 1 + k$. Therefore, any paths of agent a_1 that reach e before or at timestep $t_2 + k$ must conflict with any paths of agent a_2 that reach b before or at timestep $t_1 + k$.

Now we consider bypasses. Assume that agent a_1 has bypasses to reach e without traversing corridor C and the earliest timestep when it can reach e using a bypass is t'_1. Similarly, assume that agent a_2 also has bypasses to reach b without traversing corridor C and the earliest timestep when it can reach b using a bypass is t'_2. If we prioritize agent a_1, then agent a_2 can either wait or use a bypass. So the earliest timestep when agent a_2 can reach b is $\min(t'_2, t_1 + 1 + k)$. Similarly, if we prioritize agent a_2, then the earliest timestep when agent a_1 can reach e is $\min(t'_1, t_2 + 1 + k)$. Therefore, any paths of agent a_1 that reach e before or at timestep $\min(t'_1 - 1, t_2 + k)$ must conflict with any paths of agent a_2 that reach b before or at timestep $\min(t'_2 - 1, t_1 + k)$. In other words, for every pair of conflict-free paths for the two agents, at least one of the two following constraints hold:

- $\langle a_1, e, [0, \min(t'_1 - 1, t_2 + k)] \rangle$ or
- $\langle a_2, b, [0, \min(t'_2 - 1, t_1 + k)] \rangle$,

where $\langle a_i, v, [t_{min}, t_{max}] \rangle$ is a *range constraint* that prohibits agent a_i from being at vertex v at any timestep between timesteps t_{min} and t_{max} (Atzmon et al. 2018). Therefore, to resolve this corridor conflict, we split the CT node and generate two child CT nodes, each with one of the two range constraints as an additional constraint. We use state-time A* to compute t_1, t'_1, t_2 and t'_2.

For example, for the corridor conflict in Figure 2, we calculate $t_1 = t_2 = 4$, $t'_1 = t'_2 = +\infty$, 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 [A4, A4, A4, A4, A4, A3, B3, C3, D3, D4] ([D2, D2, D2, D2, D2, D3, C3, 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 other vertices inside the corridor because CBS breaks ties by preferring the path that has the fewest conflicts with the paths of other agents. Hence, the paths in both child CT nodes are conflict-free, and the corridor symmetry here is resolved in a single branching step.

However, this branching method cannot be applied to all corridor conflicts. We use this branching method only when the path of agent a_1 in the current CT node violates the range constraint $\langle a_1, e, [0, \min(t'_1 - 1, t_2 + k)] \rangle$ and the path of agent a_2 in the current CT node violates the range constraint $\langle a_2, b, [0, \min(t'_2 - 1, t_1 + k)] \rangle$. This guarantees that the paths in both child CT nodes are different from the paths in the current CT node. Otherwise, we use the standard branching method (discussed in Section 3) to resolve the conflict.

Theorem 1. *Resolving corridor conflicts with range constraints preserves the completeness and optimality of CBS.*

The proof is given in the appendix. We add range constraints at the exit endpoints of the corridor for each agent instead of the entry endpoints because there might be an optimal solution where one of the conflicting agents has to move into the corridor, move out from the same side of the corridor before colliding with the other agent, move into it again after the other agent has traversed the corridor and finally traverse it.

4.3 Classifying Corridor Conflicts

We classify corridor conflicts based on the type of the vertex/edge conflict inside the corridor. A corridor conflict is cardinal iff the corresponding vertex/edge conflict is cardinal; it is semi-cardinal iff the corresponding vertex/edge conflict is semi-cardinal; and it is non-cardinal iff the corresponding vertex/edge conflict is non-cardinal. This is an approximate way of classifying corridor conflicts. We use Figure 2 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 has a path 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, 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 nodes are larger than the cost of N.

We follow the conflict prioritization in (Boyarski et al. 2015; Li et al. 2019c) and resolve cardinal conflicts first, then semi-cardinal conflicts, and finally non-cardinal conflicts. For conflicts of the same type, we resolve corridor conflicts first, then rectangle conflicts, and finally vertex and edge conflicts. Corridor conflicts have higher priority than rectangle conflicts 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, while, when resolving rectangle conflicts, the costs of the child CT nodes
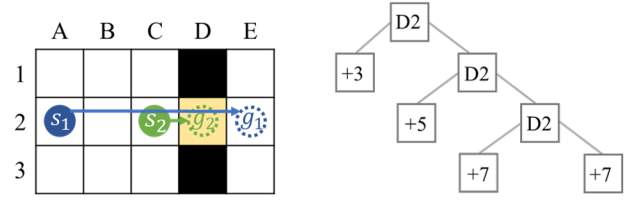

Figure 3: An example of target symmetry. In the left figure, agent a_2 arrives at cell D2 at timestep 1. Two timesteps later, agent a_1 traverses the same cell, leading to a vertex conflict $\langle a_1, a_2, D2, 3 \rangle$. The right figure shows the CT. Each left branch constrains agent a_2, while each right branch constrains agent a_1. Each non-leaf CT node is marked with the vertex of the chosen conflict. The leaf CT node marked "+3" contains an optimal solution, whose sum of costs is the cost of the root CT node plus 3. Each leaf CT node marked "+5" or "+7" contains a suboptimal solution, whose sum of costs is the cost of the root CT node plus 5 or 7.

Table 2: Number of expanded CT nodes to resolve a target conflict of the type shown in Figure 3 for different distances k between vertices s_1 and g_2.

k	10	20	30	40	50
Nodes for 2-agent instances	10	20	30	40	50
Nodes for 4-agent instances	50	150	300	500	750

are typically at most one larger (Li et al. 2019d). Vertex and edge conflicts have the lowest priority because we prefer to resolve all symmetry conflicts first.

5 Target Symmetry

A target symmetry occurs when one agent traverses 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*. Figure 3 shows an example. 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, 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 to agent a_1 that prohibits it from being at vertex D2 at timestep 3. Thus, agent a_1 can arrive at D2 only 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 can wait 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 D2 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 of the root CT node is optimal.

Target symmetry has the same pernicious characteristics as corridor 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 Figure 3 for different distances k between vertices s_1 and g_2. While the increase in CT nodes is linear in k, which may not seem too problematic, only one of the leaf CT nodes actually resolves the conflict. Later, when other conflicts occur elsewhere on the map, each of the leaf CT nodes will 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 in the number of CT nodes become exponential in m.

5.1 Identifying Target Conflicts

The detection of the target conflicts is straightforward. We check every vertex conflict. A vertex conflict is a *target conflict* iff the conflict happens after one agent has arrived at its target vertex.

5.2 Resolving Target Conflicts

The key to resolving target conflicts is to reason about the path length of an agent directly. Suppose agent a_2 arrives at its target vertex g_2 at timestep t' and stays there forever. The path of agent a_1 traverses vertex g_2 at timestep t ($t \geq t'$). We resolve this 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 traverse 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 possible 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 traverse vertex g_2 at or after timestep t.

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. Given bounds $e \leq l_i \leq u$ on the path length l_i of agent a_i: If the low-level search reaches target vertex g_i 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 is 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 possible solution, and the CT node is thus pruned. We require the agent to not be at the target vertex in 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 Figure 3, 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 conflict with agent a_1. In the right child CT node, agent a_1 cannot occupy vertex D2 at or after timestep 3 and we thus fail to find a path and prune the right child CT node. Therefore, the target symmetry is resolved in a single branching step.

The completeness and optimality of the length constraints for target conflicts are straightforward. Therefore, we omit the proof of the following theorem.

Theorem 2. *Resolving target conflicts with length constraints preserves the completeness and optimality of CBS.*

5.3 Classifying Target Conflicts

Similar to corridor 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 guaranteed to be larger than the cost of the current 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 in a CT node N, both child CT nodes have larger costs than CT node N.

Similar to corridor conflicts, we resolve cardinal conflicts first, then semi-cardinal conflicts, and finally non-cardinal conflicts. For conflicts of the same type, 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.

6 Experiments

We implement CBSH (Felner et al. 2018) in C++, an advanced variant of CBS that uses admissible heuristics for its high-level search. We add rectangle reasoning (i.e., CBSH-RM in (Li et al. 2019c)), corridor reasoning and target reasoning on top of CBSH. We refer to these three reasoning techniques as R, C and T, respectively. The experiments are conducted on a 2.80 GHz Intel Core i7-7700 laptop with 8 GB RAM and a runtime limit of 1 minute.

We evaluate our algorithms on eight maps of different sizes and structures, including: (1) A small warehouse map from (Li et al. 2019b): It is a 10×30 grid with 9 rectangular obstacles of size 2×6 in the center area. Half the agents move from left to right, and half the agents move
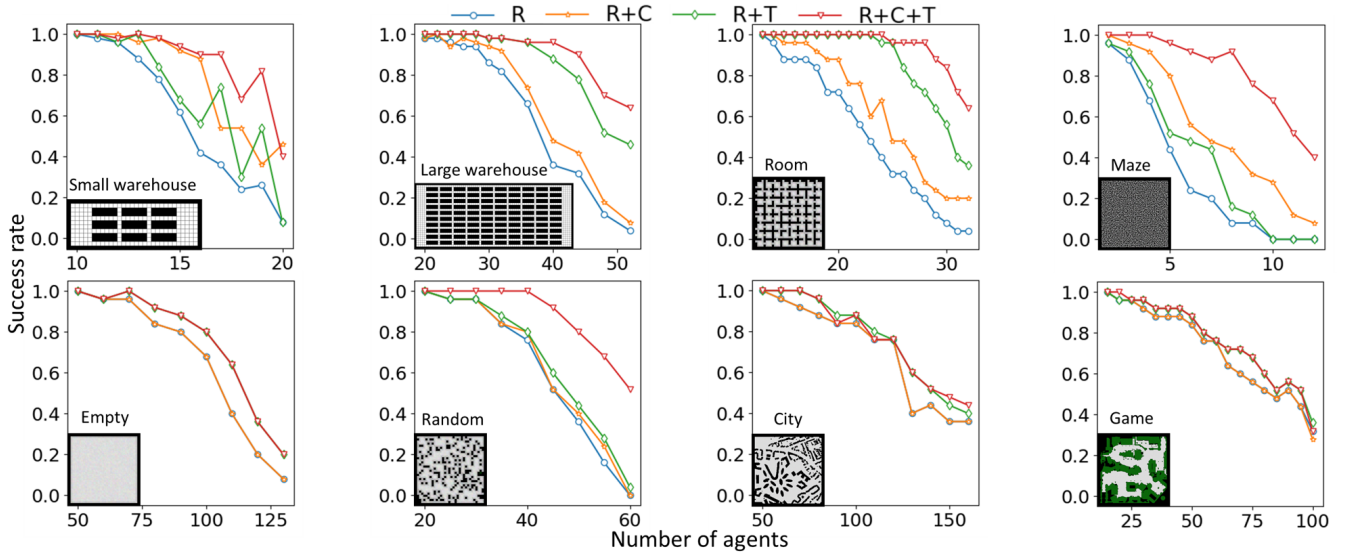
Figure 4: Success rates within the time limit. Many parts of the blue and green lines in the figures for the empty, city and game maps are hidden by the orange and red lines, respectively.

Table 3: Average runtimes in seconds. The runtime limit of 60 seconds is included in the average for unsolved instances.

	Small warehouse					Large warehouse					Room					Maze			
m	R	R+C	R+T	R+C+T	m	R	R+C	R+T	R+C+T	m	R	R+C	R+T	R+C+T	m	R	R+C	R+T	R+C+T
12	4.62	**1.30**	3.42	1.95	20	1.39	1.33	1.15	**0.12**	16	7.69	3.93	0.20	**0.13**	2	2.72	**0.01**	2.70	**0.01**
14	21.91	4.74	17.56	**4.23**	28	5.42	3.74	0.11	**0.09**	20	21.51	14.70	1.19	**0.34**	4	20.14	7.52	15.7	**1.10**
16	41.92	12.04	34.25	**11.40**	36	24.20	18.08	4.20	**2.81**	24	39.98	31.36	5.59	**3.33**	6	46.00	28.83	33.79	**11.19**
18	48.65	32.94	45.01	**26.07**	44	46.02	39.48	18.10	**11.26**	28	51.53	45.73	22.45	**10.91**	8	55.70	35.29	52.28	**15.19**
20	55.42	**40.14**	55.45	41.41	52	58.38	55.99	37.80	**30.18**	32	59.95	54.02	39.81	**27.29**	10	60.00	46.63	60.00	**30.13**
	Empty					Random					City					Game			
m	R	R+C	R+T	R+C+T	m	R	R+C	R+T	R+C+T	m	R	R+C	R+T	R+C+T	m	R	R+C	R+T	R+C+T
50	0.06	0.06	**0.01**	0.03	20	0.08	0.07	0.02	**0.01**	60	3.64	3.71	**3.05**	**3.05**	20	**2.85**	**2.85**	2.87	**2.85**
70	2.76	2.76	**0.09**	0.11	30	4.99	3.88	3.20	**0.12**	80	11.18	11.13	10.58	**10.49**	40	10.41	10.38	9.16	**8.94**
90	13.68	13.67	**7.95**	10.65	40	18.45	16.30	13.87	**0.72**	100	12.51	12.30	12.36	**12.27**	60	21.57	21.82	**20.55**	20.97
110	41.32	41.31	27.18	**25.83**	50	42.41	40.50	36.39	**18.29**	120	22.08	20.49	20.49	**19.50**	80	34.34	34.35	32.67	**32.39**
130	55.81	55.81	**51.04**	52.11	60	60.00	60.00	58.95	**36.88**	140	38.92	37.67	32.52	**31.70**	100	49.80	49.95	**49.33**	49.51

from right to left. Their start and target vertices are randomly located in the left/right open areas of size 10×5. (2) A large warehouse map: It is a 31×79 grid with 100 rectangular obstacles of size 2×6 in the center area. The start and target vertices are randomly located over the entire map. (3) A room map "room-32-32-4" from the MAPF benchmark (Stern et al. 2019): It is a 32×32 grid with 64 rooms of size 3×3 connected by single-cell doors. (4) A maze map "maze-128-128-1" from the MAPF benchmark: It is a 128×128 grid with corridors that are one-cell wide. (5) An empty map "empty-32-32" from the MAPF benchmark: It is a 32×32 grid without any obstacles. (6) A random map "random-32-32-20" from the MAPF benchmark: It is a 32×32 grid with 20% randomly blocked cells. (7) A city map "Paris_1_256" from the MAPF benchmark: It is a 256×256 grid encoding a map of Paris. (8) A game map "den520d" from the MAPF benchmark: It is a 257×256 grid from the video game *Dragon Age: Origins*. We show the maps in Figure 4. For both warehouse maps, we generate 50 instances with random start and target vertices for each map and each number of agents m. For all other maps from the MAPF benchmark, we use the "even" scenarios in

the benchmark, yielding 25 instances for each map and each number of agents m.

6.1 Success Rate and Runtime

Figure 4 plots the *success rate*, i.e., the percentage of solved instances within the time limit, on all maps. Overall, corridor reasoning improves the success rates when the maps contain many corridors, which is the case for the small and large warehouse maps, the room map and the maze map. When the maps contain no or only a few corridors, which is the case for the empty map, the random map, the city map and the game map, corridor reasoning does not improve the success rates but does not deteriorate the success rates either. Target reasoning, on the other hand, improves the success rates on most maps substantially. Again, when target reasoning does not improve the success rates, it does not deteriorate them either. Corridor and target reasoning together improves the success rates the most. For example, the success rate of R is 0 on the maze map with 10 agents but R+C+T improves it to 0.68.

Table 3 reports the average runtimes. Again, corridor and target reasoning seldom increase the runtimes, and often re-

Table 4: Conflict distributions in R+C+T. "Nodes" represents the number of expanded CT nodes within the time limit. "Rectangle", "Corridor" and "Target" represent the percentage of CT nodes expanded by rectangle, corridor and target reasoning, respectively.

Map	m	Nodes	Rectangle	Corridor	Target
Small warehouse	16	6,564	9.99%	16.05%	2.58%
Large warehouse	40	5,417	2.24%	9.48%	13.74%
Room	24	1,687	1.93%	23.29%	8.68%
Maze	6	11	0.00%	32.06%	8.71%
Empty	100	23,573	12.58%	0.00%	12.95%
Random	50	17,803	4.14%	2.89%	9.27%
City	100	87	5.98%	0.18%	10.30%
Game	70	414	0.87%	0.00%	1.89%

Table 5: Average runtimes per expanded CT node in milliseconds.

Map	m	R	R+C	R+T	R+C+T
Small warehouse	16	0.72	1.59	0.83	1.74
Large warehouse	40	1.00	1.26	1.53	1.39
Room	24	0.74	1.01	0.89	1.31
Maze	6	30.99	541.81	30.28	983.12
Empty	100	0.40	0.40	0.55	0.55
Random	50	0.69	0.77	0.84	0.94
City	100	91.48	90.40	141.70	144.54
Game	70	45.27	45.45	45.06	45.22

Table 6: Average lower bound improvements.

Map	m	R	R+C	R+T	R+C+T
Small warehouse	16	12.74	14.36	13.02	14.42
Large warehouse	40	17.18	19.76	23.74	24.04
Room	24	19.40	22.92	27.16	27.20
Maze	6	18.44	33.96	93.08	100.88
Empty	100	9.52	9.52	9.88	9.88
Random	50	20.68	21.16	21.00	27.92
City	100	6.32	6.36	8.72	8.72
Game	70	5.96	5.96	6.60	6.60

duce the runtimes substantially. For example, R+C+T improves the runtime of R by a factor of 40 on the random map with 30 agents.

We notice an interesting behavior on the maze and random maps: Corridor and target reasoning applied separately do not result in substantial improvements, but their combination does for the following reason: Maze and random maps have many corridor and target conflicts. Solving either class of conflicts by the standard branching method of CBS could result in unacceptable runtimes. Thus, CBS with only one of the reasoning techniques does not solve many instances within the time limit, but CBS with both techniques overcomes both symmetries and solves many instances within the time limit.

6.2 Conflict Distribution

Table 4 reports how often CBS uses each reasoning technique to expand CT nodes, which also indicates how often different conflicts occur on different maps. Clearly, corridor conflicts are frequent and more common than rectangle conflicts on maps with corridors. Target conflicts are frequent on all maps and even occur on the small warehouse map, despite target vertices not being located in corridors. The high frequency of both kinds of conflicts results in the gains that we see in Figure 4 and Table 3.

6.3 Runtime Overhead

Table 5 reports the average runtime per expanded CT node. As expected, CBS spends more time to expand a CT node on a large map than on a small map. Corridor and target reasoning incur only a reasonably small runtime overhead on all maps except for the maze and city maps.

On the maze map, the overhead of corridor reasoning comes from computing t'_1 and t'_2. The empty cells in the

maze map form a tree, and thus there are no bypasses for agents to avoid any corridors, i.e., t'_1 and t'_2 are always infinite. However, since we use state-time A* to compute them, state-time A* can only determine that there is no bypass when it has expanded all reachable states, which is time-consuming. Still, corridor and target reasoning are most beneficial on the maze map because the time-consuming alternate path search avoids a much more time-intensive CBS search, which essentially repeats the reasoning by creating many CT nodes. Moreover, if we know the map a priori, then we can preprocess the map, mark those corridors that do not have bypasses, and save the runtime of computing t'_1 and t'_2.

On the city map, the overhead of target reasoning comes from the low-level space-time A* search for replanning an extremely long path. The length constraint $l_2 > u$ can substantially increase the path length, but finding a long path is also time-consuming for space-time A*. We might be able to address this issue by replacing space-time A* with Safe Interval Path Planning (Phillips and Likhachev 2011), but leave this for future work.

6.4 Lower Bound Improvement

Table 6 reports the *lower bound improvement*, i.e., the minimum f-value of the CT nodes in the open list when CBS terminates minus the cost of the root CT node. If an algorithm finds an optimal solution within the time limit, the lower bound improvement is equal to the optimal cost minus the cost of the root CT node. Thus, if all algorithms find an optimal solution within the time limit, the lower bound improvement is the same for all of them. For those hard instances which none of the algorithms solve within the time limit, R+C+T always provides a higher lower bound than the other algorithms. On the maze map with 6 agents, for instance, the lower bound improvement of R+C+T is 5 times higher than that of R alone.

7 Related Work

Ryan (2006; 2007) proposed several graph decomposition approaches for solving MAPF. Like our work, he reasoned about special graph structures, including stacks, cliques, and halls. Unlike our work, he built 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 analyzing sub-graphs to break symmetries without preprocessing and without sacrificing optimality.

Cohen et al. (2016) proposed highways to reduce the number of corridor conflicts. They assigned a direction to some corridor vertices (resulting in a highway), and agents moving against highways need to pay larger costs. They showed that highways can speed up ECBS, a bounded-suboptimal version of CBS. However, the efficiency of highways for optimal CBS is limited because highways can then only be used to break ties among multiple shortest paths and are not guaranteed to resolve all corridor conflicts.

Lam et al. (2019) proposed a novel algorithm BCP based on Integer Linear Programming (ILP) for solving MAPF optimally. They also found rectangle and corridor symmetry in their fractional solutions and designed dedicated constraints to break them. However, since the frameworks of CBS and BCP are different, the approaches to reasoning about symmetries and designing constraints are different as well. For example, the form of corridor symmetry addressed in BCP is quite different from ours, arising when two agents swap locations, and independent of corridors in our sense. The constraints used in BCP remove fractional solutions that do not arise in CBS.

Recently, Li et al. (2019a) made a significant improvement to MAPF by using CBS 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. It remains future work to implement corridor reasoning and target reasoning in this framework, but we expect to be able to speed up the calculation of the informed heuristics significantly since both reasoning techniques apply directly to solving the two-agent sub-MAPF instances.

8   Conclusion

In this paper, we introduced corridor and target reasoning to reason directly about symmetry conflicts that occur between two agents when using CBS to solve MAPF. As Table 4 shows, these kinds of conflicts occur quite frequently on many classes of maps, particularly warehouse maps, which reflect one of the key applications for MAPF. We showed experimentally that directly reasoning about these conflicts leads to substantial improvements in both success rate and runtime.

Appendix

In this section, we prove the correctness of Theorem 1. We first explain *mutually disjunctive* constraint sets, a pair of constraint sets that we can use to split a CT node with completeness and optimality guarantees. We then show that the pair of range constraints extracted from a corridor conflict is mutually disjunctive.

Li et al. (2019d) define that two vertex constraints for agents a_i and a_j, respectively, 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 do not exist two conflict-free paths such that both constraints are violated. Moreover, they define that two sets of vertex constraints are

mutually disjunctive iff each constraint in one set is mutually disjunctive with each constraint in the other set. They 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 given in Lemma 3. Please refer to their paper for detailed proof.

Lemma 3. *For a given CT node N with constraint set C, if two vertex constraint sets C_1 and C_2 are mutually disjunctive, any set of conflict-free paths that satisfy C also satisfies at least one of the constraint sets $C \cup C_1$ and $C \cup C_2$.*

Proof. Lemma 3 is true because, otherwise, there would exist two conflict-free paths such that both of them are consistent with C but one path violates a constraint $c_1 \in C_1$ and one path violates a constraint $c_2 \in C_2$. Then, c_1 and c_2 are not mutually disjunctive, contradicting the assumption.   □

Therefore, in order to show that CBS with corridor reasoning is complete and optimal, we only need to show that the constraint sets that correspond to the two range constraints are mutually disjunctive.

Lemma 4. *For the pair of range constraints $\langle a_1, e, [0, \min(t'_1 - 1, t_2 + k)] \rangle$ and $\langle a_2, b, [0, \min(t'_2 - 1, t_1 + k)] \rangle$ extracted from a corridor conflict, their corresponding vertex constraint sets $C_1 = \{ \langle a_1, e, t \rangle \mid t \in [0, \min(t'_1 - 1, t_2 + k)] \}$ and $C_2 = \{ \langle a_2, b, t \rangle \mid t \in [0, \min(t'_2 - 1, t_1 + k)] \}$ are mutually disjunctive.*

Proof. According to the definition of mutually disjunctive, we only need to show that every vertex constraint $c_1 = \langle a_1, e, i \rangle \in C_1$ is mutually disjunctive with every vertex constraint $c_2 = \langle a_2, b, j \rangle \in C_2$. Let path p_1 be an arbitrary path of agent a_1 that traverses vertex e at timestep i and path p_2 be an arbitrary path of agent a_2 that traverses vertex b at timestep j. That is, p_1 is a path that violates constraint c_1 and p_2 is a path that violates constraint c_2. Since $i \le \min(t'_1 - 1, t_2 + k) \le t'_1 - 1 < t'_1$ (where t'_1 is the earliest timestep when agent a_1 can reach vertex e without using the corridor between vertices b and e), path p_1 must traverse the corridor. Similarly, path p_2 must traverse the corridor as well. Since $i \le \min(t'_1 - 1, t_2 + k) \le t_2 + k$ (where k is the distance between vertices e and b), the latest timestep when path p_1 traverses vertex b is no larger than timestep t_2. t_2 is the earliest timestep when path p_2 can traverse vertex b, so path p_1 traverses vertex b before path p_2. Similarly, path p_2 traverses vertex e before path p_1. Therefore, paths p_1 and p_2 must have a conflict in the corridor between vertices b and e. Since paths p_1 and p_2 were chosen arbitrarily, every pair of paths that violate both c_1 and c_2 are in conflict. So, c_1 and c_2 are mutually disjunctive, and, therefore, C_1 and C_2 are mutually disjunctive as well.   □

By Lemmata 3 and 4 and the proof in (Li et al. 2019d), we conclude that resolving corridor conflicts with range constraints preserves the completeness and optimality of CBS.

Acknowledgements

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2018. Robust multi-agent path finding. In *Proceedings of the Annual Symposium on Combinatorial Search (SoCS)*, 2–9.

Banfi, J.; Basilico, N.; and Amigoni, F. 2017. Intractability of time-optimal multirobot path planning on 2D grid graphs with holes. *IEEE Robotics and Automation Letters* 2(4):1941–1947.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, S. E. 2015. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 740–746.

Cohen, L.; Uras, T.; Kumar, T. K. S.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Improved solvers for bounded-suboptimal multi-agent path finding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 3067–3074.

Dresner, K., and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research* 31:591–656.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Keonig, S. 2018. Adding heuristics to conflict-based search for multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 83–87.

Gange, G.; Harabor, D.; and Stuckey, P. J. 2019. Lazy CBS: Implict Conflict-based Search Using Lazy Clause Generation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 155–162.

Lam, E.; Bodic, P. L.; Harabor, D. D.; and Stuckey, P. J. 2019. Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1289–1296.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019a. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 442–449.

Li, J.; Harabor, D.; Stuckey, P. J.; Felner, A.; Ma, H.; and Koenig, S. 2019b. Disjoint splitting for conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 279–283.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2019c. Symmetry-breaking constraints for grid-based multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 6087–6095.

Li, J.; Surynek, P.; Felner, A.; Ma, H.; Kumar, T. K. S.; and Koenig, S. 2019d. Multi-agent path finding for large agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 7627–7634.

Morris, R.; Chang, M. L.; Archer, R.; Cross, E.; Thompson, S.; Franke, J.; Garrett, R.; Malik, W.; McGuire, K.; and Hemann, G. 2015. Self-driving aircraft towing vehicles: A preliminary report. In *Workshop on AI for Transportation*, 35–42.

Phillips, M., and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 5628–5635.

Ryan, M. R. K. 2006. Multi-robot path planning with subgraphs. In *Proceedings of the Australasian Conference on Robotics and Automation*, 1–8.

Ryan, M. R. K. 2007. Graph decomposition for efficient multi-robot path planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003–2008.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219:40–66.

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

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

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1):9–20.

Yu, J., and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1444–1449.