Efficient Multi Agent Path Finding with Turn Actions

Yue Zhang, Daniel Harabor, Pierre Le Bodic, Peter J. Stuckey

Monash University, Australia
{Yue.Zhang, Daniel.Harabor, Pierre.LeBodic, Peter.Stuckey}@monash.edu

Abstract

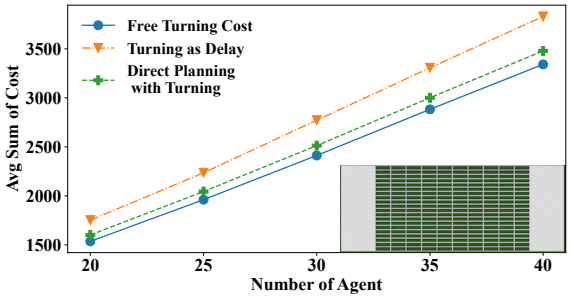
Current approaches for real-world Multi-Agent Path Finding (MAPF) usually start with a simplified MAPF model and modify the resulting plans so they are kinematically feasible. We investigate one such problem, called MAPF with turn actions (MAPF$_T$), and show that ignoring the kinematic constraints significantly increases solution cost. A first modification of the popular Conflict-Based Search algorithm to MAPF$_T$ yields significantly better plans but comes at the cost of substantial decreases in scalability. We then introduce several techniques that can improve the performance of CBS for MAPF$_T$, including stronger and generalised versions of existing symmetry-breaking constraints and a novel pruning technique that eliminates redundant branches in the CBS constraint tree. Experimental results on six popular MAPF domains show convincing improvements for CBS success rate and substantial reductions in node expansions and runtime.

Introduction

Multi-Agent Path Finding (MAPF) (Stern et al. 2019) is the well-studied problem of planning paths for multiple agents so that they can travel from their predefined start locations to goal locations without any collisions. MAPF has a wide range of applications, including online video games (Silver 2005), traffic-controlling system (Dresner and Stone 2008) and automatic warehouse (Hönig et al. 2019).

Existing algorithms, like Conflict-Based Search (CBS) (Sharon et al. 2015), can efficiently solve MAPF problems with up to hundreds of agents in simulation environments. However, deploying MAPF solvers in practical applications, such as automated warehouses, can result in large gaps between planned and achieved solution costs. This is because MAPF solvers employ a simplified model in which each agent moves without considering their orientation and without considering turning costs. Figure 1 shows the gap; the blue line represents the optimal plan assuming turning is free. These plans can be converted to executable plans with turning costs by treating the turn actions as delays (see (Ma, Kumar, and Koenig 2017)) which gives the orange line. Directly planning with turn actions, the green line, is a clear improvement, deserving of further investigation.

(a) Example of an Amazon Robotics drive unit.



(b) Achieved costs for different planning models.

Figure 1: 1(a) shows typical operations in an automatic warehouse. Robots have an orientation (facing direction), and need to perform turn actions. 1(b) compares objective values for three different planning methods, on a synthetic warehouse domain with different numbers of agents.

Approaches for planning with turn actions have appeared in the literature. One idea involves the use of state lattices; e.g., (McNaughton et al. 2011; Cohen et al. 2019; Pivtoraiko, Knepper, and Kelly 2009), to model agents that can rotate while moving (e.g., automated forklifts). Unfortunately, this model is not appropriate for warehouse settings, where rotations take the form of separate in-place actions. In some recent works, researchers have attempted to incorporating kinematic constraints, including turning cost, directly into MAPF planning for Multi-Agent Pickup and Delivery (MAPD) problem (Ma et al. 2019) and solver MAPF. These approaches more closely resemble real warehouse applications. The problem they aim to solve involves adding kinematic constraints and considering task allocations, which has made it challenging to compute plans. To address this, their solver uses a prioritized strategy to plan paths, but this approach may not provide solution quality guarantees. Other approaches, such as (Varambally, Li, and

Koenig 2022; Hönig et al. 2019) focus on direct planning with additional uniform-cost actions, such as turning and attaching or detaching to/from shelves. They design algorithms based on modifications to an existing bounded suboptimal MAPF algorithm, called ECBS (Barer et al. 2014). Unfortunately, because no optimal method exists, the optimality gap of these plans is unknown. Also unknown is the relative difficulty of directly computing optimal plans for these extended models vs. conventional MAPF.

In this paper, we study optimal *MAPF with Turn actions* (MAPF$_T$), where we model turning as a separate unit-cost action. Although the problem has previously appeared in the literature, to the best of our knowledge this is the first approach that solves it optimally. First, we show how to adapt and generalise CBSH2-RTC (Li et al. 2021), a state-of-the-art optimal MAPF algorithm, to CBS with turn actions (CBS$_T$) to solve MAPF$_T$. Next, we show that MAPF$_T$ is substantially more challenging than MAPF. We then describe how to adapt a range of recent enhancements, originally developed for MAPF, to improve its performance. Finally, we show that CBS$_T$ can generate new types of *redundant high level nodes*; ones that define sub-problems appearing elsewhere in the search tree. We introduce a novel pruning approach, based on logical subsumption, which detects and eliminates these nodes. Experimental results indicate this approach can avoid substantial amounts of duplicated work and we report convincing runtime improvements for CBS$_T$. The same kind of redundant nodes also arise in standard MAPF, though less frequently, and we show how the same ideas can also improve performance there.

Problem Definition

In this paper, we consider each agent to have an orientation and need to perform turn actions. We call this problem MAPF$_T$. It is defined as follows. **Input**: The input is an undirected gridmap $G = (V, E)$ and a set of k agents $\{a_1...a_k\}$ each with start state s_i and goal state g_i. Time is assumed to be discretized into timesteps. At each timestep, each agent has a state and takes a single action to transit to another state. **State**: The state for an agent is a triple (v, t, θ) containing the current timestep t, the vertex $v \in V$ occupied by the agent at this time and the current orientation $\theta \in \{North, South, East, West\}$. **Action**: Each action has a unit cost. Each agent has four possible actions: moving forward to the next cell (assuming it is not blocked), turning left by 90°, turning right by 90°, and waiting at the current location. **Conflict**: A conflict occurs when two agents a_i and a_j occupy the same vertex $v \in V$ at the same timestep t, called *vertex conflict* denoted $\langle a_i, a_j, v, t\rangle$, or two agents a_i and a_j pass through the same edge $e \in E$ in opposite directions at the same timestep called *edge conflict* denoted $\langle a_i, a_j, e, t\rangle$. **Cost**: The cost of a path l_i is the sum of timesteps that transit agent a_i from the start state to the goal state. **Solution**: A solution in MAPF is a set of feasible action sequences that can transit the given set of agents from start vertex, start orientation to goal vertex, goal orientation without any conflict. **Objective**: The objective is to find a feasible solution with the minimal sum of individual costs of agents.

Conflict-Based Search

Among the most popular and effective methods for optimal MAPF is CBS (Sharon et al. 2015). This is a two-level search algorithm. **Low level search**: At the low level, CBS invokes a single-agent search algorithm with both time and space dimensions considered to find the optimal path for each agent that satisfies all its constraints given by the high level search while ignoring other agents. **High level search**: At the high level, CBS performs best-first search on a binary search tree called the *constraint tree* (CT). Each node N in the CT contains a set of CBS constraints ($N.const$), a MAPF solution ($N.solution$) that satisfies $N.const$ and the sum of the solution cost $N.cost$. Basically, it maintains a priority queue of the unexplored nodes and selects the nodes from CT with the lowest $N.cost$ to expand first. CBS first generates the root node of the CT from the initial low level search with no constraints. Then, it finds all existing conflicts between pair of agents in the current solution and chooses a conflict to resolve. When a conflict between a_i and a_j is chosen to resolve, the high level search generates two successors $N.left$ and $N.right$. In each successor, CBS adds a constraint that prohibits one agent from being at a vertex or using an edge at the conflict timestep to prevent the chosen conflict. Then, each child node C runs a low-lever solver with the new constraint $C.new$ to find a new plan for the affected agent $C.agent$ which is one of a_i or a_j. After replanning, the successors are inserted into the priority queue. CBS iteratively selects candidates from the priority queue, splits a node to create two children to resolve conflicts, and inserts them into the priority queue until it selects a candidate from the priority queue with no conflicts.

From CBS to CBS$_T$

Adapting CBS to the MAPF$_T$ problem is in principle a simple process, involving modifications to the state and graph expansions based on the MAPF$_T$ action model in low level search. However, the turning cost in MAPF$_T$ means optimal solutions are usually found deeper in CBS's CT. Since the size of the CT is exponential in the depth of the optimal plan, the resulting algorithm is unlikely to perform well. Thus, we must also adapt a range of recent enhancements developed in recent years to improve the performance of CBS.

Prioritisation (PC)

CBS selects which conflict to split and resolve. This process is crucial, as different choices significantly influence the number of high level node expansions in CBS. Boyarski et al. (2015) classify and prioritise conflicts based on three types. A conflict C is: (1) *cardinal* if replanning for both agent to resolve C increases both their cost; or (2) *semi-cardinal* if replanning for both agents increases the cost for exactly one of them; or (3) *non-cardinal* if replanning for both agents increases the cost for neither.

The state-of-the-art approach relies on Multi-Valued Decision Diagrams (MDD) (Sharon et al. 2013) to determine the conflict type. An MDD for an agent a in a CT node N is a rooted directed graph that represents all optimal paths subject to the set of constraints $N.const$. Each location v

reached at timestep t on an optimal path is represented by an MDD node (v, t) that is at depth t. If there is a single MDD node (v, t) at depth t, and a constraint is added that prevents it from being at v at timestep t, then all current optimal paths become infeasible, which results in a cost increase. Given this, an MDD can efficiently identify the conflict type.

MDDs in CBS$_T$ work as in CBS, except that (1) paths come from the modified low level search, (2) MDD nodes record the state (v, t, θ), and (3) to identify a cardinal conflict, we count the number of locations at a given depth. This is because multiple nodes with the same location and different orientations can exists at a same depth. Counting the number of MDD nodes, as in CBS, still works, but fail to identify some cardinal conflicts.

Heuristic Estimators

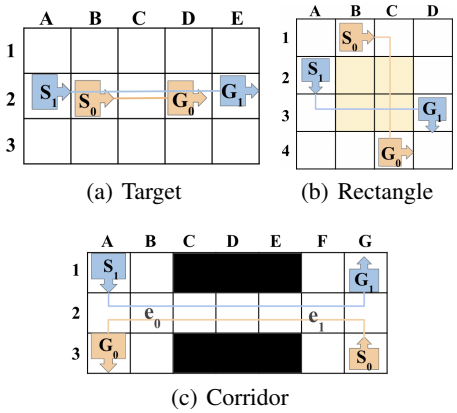
An important ingredient for the performance of CBS is high level heuristics that estimate cost increases along the current branch before all conflicts are resolved. The WDG heuristic (Li et al. 2019a) is one such popular and leading estimator. For every conflicting pair of agents, WDG solves a two-agent MAPF problem, and then uses their combined cost to build an eponymous Weighted Dependency Graph. The graph is then used to compute an admissible estimate. As the WDG heuristic acts only on the high level search, it can be directly adapted to MAPF$_T$.

Symmetry Breaking

For an efficient search, symmetries need special handling when a pair of agents have many equivalent optimal individual paths, but they all conflict with each other. Symmetry breaking plays a crucial role in scaling MAPF. Three common symmetries: target symmetries, rectangle symmetries and corridor symmetries are identified and handled directly in (Li et al. 2021) leading to significant performance improvements. As shown in Figure 2, pairwise symmetries also exist in MAPF$_T$. The existing target reasoning and rectangle reasoning can be straightforwardly enabled in CBS$_T$, but for corridor reasoning in MAPF$_T$, the cost of turnings matters when calculating the constraints, so we show how to modify corridor reasoning to generate more accurate constraints.

Corridor Reasoning A *corridor* is a chain of connected vertices, each of degree 2, and the two endpoints of this chain. The endpoint e_i is the first vertex that agent a_i exits the corridor. The corridor length, denoted as k, is the map distance between two endpoints. Corridor conflicts happen when two agents travel the corridor in opposite directions at the same time. For each agent, it must either wait for the other agent to fully traverse the corridor and then enter the corridor, or take an alternate path not using the corridor to reach the endpoint. Corridor reasoning technique resolves corridor conflicts by generating range constraints on endpoints ($\langle a_i, e_i, [t_0, t_1]\rangle$), which disallows each agent to stay at the endpoint from timestep t_0 to timestep t_1.

We denote the shortest path cost that a agent a_i travel from the start to its corridor endpoint e_i as $t_i(e_i)$, and the shortest path cost a_i travel from start to e_i that not crossing the corridor as $t_i^{'}(e_i)$. In Figure 2(c), if a_1 traverses the corridor after



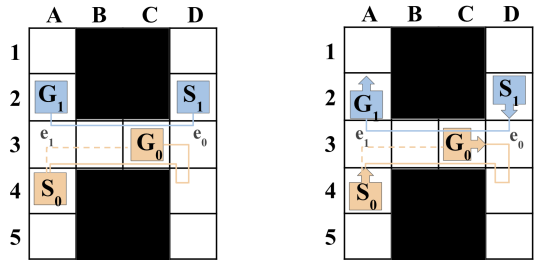
(a) Target (b) Rectangle (c) Corridor

Figure 2: Pairwise symmetries in MAPF$_T$. 2(a) incurs a target conflict that a_1 traverses g_0 after a_0 arrived. 2(c) is a corridor conflict that two agents travel the corridor from B2 to F2 at the same time. 2(b) is a rectangle conflict that all shortest path of these two agents conflict in the yellow area.

a_0 fully passing it, the earliest timestep that a_1 reach e_1 is $t_0(e_0)+k+1$. If a_1 has a route to e_1 not crossing the corridor and arriving at $t_1^{'}(e_1)$, the earliest timestep for a_1 to reach e_1 is $min(t_1^{'}(e_1), t_0(e_0)+k+1)$, and hence, a_1 cannot be at e_1 from timestep 0 to $min(t_1^{'}(e_1)-1, t_0(e_0)+k)$. Similar for a_0, if it allows a_1 traverse the corridor first, then it cannot be at e_0 from timestep 0 to $min(t_0^{'}(e_0)-1, t_1(e_1)+k)$. The constraints are: $\langle a_0, e_0, [0, min(t_0^{'}(e_0)-1, t_1(e_1)+k)]\rangle$ on one branch and $\langle a_1, e_1, [0, min(t_1^{'}(e_1)-1, t_0(e_0)+k)]\rangle$ on the other branch.

In MAPF$_T$, we further consider two kinds of turning costs. First, the turning cost are paid when the corridor is not a straight line. Simply using the corridor length k could underestimate the travel time to cross the corridor. Therefore, instead of adding k in the range constraints, we add the travel time between two endpoints, denoted as $dist(e_0, e_1)$ in which turning cost is considered. Second, the cost of one agent leaving the endpoint may differ depending on the next vertex the agent intends to enter. In Figure 2(c), if a_1 waits for a_0 to travel the corridor first, then a_0 can leave e_0 forward to A2, or take one extra timestep to turn left at e_0 and forward to B3. If a_0 chooses forward to A2, then a_1 can only wait at B1 or B3 to enter the corridor from e_0, and needs an additional cost of turning at e_0 to traverse the corridor. As a result, one agent needs to have one additional turning cost at the intersection B2.

The existing corridor reasoning generates inefficient range constraints as it ignores the turning cost, so the search will select nodes that are still infeasible because of the inefficient constraint before exploring the more promising nodes. This constraint can be easily improved to be more efficient by calculating the turning cost. Therefore, we modified the constraints by adding the additional cost due to the additional turn actions, which are $\langle a_0, e_0, [0, min(t_0^{'}(e_0) - 1, t_1(e_1) + dist(e_0, e_1) + 1)]\rangle$ on one branch and $\langle a_1, e_1, [0, min(t_1^{'}(e_1) - 1, t_0(e_0) +$



(a) MAPF instance (b) MAPF$_T$ instance

Figure 3: Target corridor conflict in MAPF and MAPF$_T$.

$dist(e_0, e_1) + 1)]\rangle$ on the other branch.

Corridor-target Reasoning Corridor-target conflicts occur when two agents have corridor conflicts and the target vertex of one agent is inside the corridor. We find that the existing corridor-target reasoning technique (Li et al. 2021) may not always solve this problem in one split, even in MAPF instances. As an example of the limitations of the current strategy and our modifications to close the gap, we present Figure 3. Since the goal g_0 of a_0 is inside the corridor, agent a_0 can either keep its current path while a_1 reach its goal using a bypass that does not use the corridor, or reaches its goal after a_1 traverses the corridor. Let the path cost for a_0 that reaches its goal after waiting for a_1 to traverse the corridor be denoted as l, and let the cost for a_1 that reaches its endpoint e_1 using a bypass that does not use the corridor be $t_1^{'}(e_1)$. Then the resulting constraints on the two branches are $l_0 \leq l, \langle a_1, e_1, [0, t_1^{'}(e_1) - 1]\rangle$ and $l_0 > l$.

In MAPF, as shown in Figure 3(a), when calculating l in the existing method, there are two cases that a_0 enters the corridor: (1) from e_1 (the dotted orange line), and (2) from e_0 (the solid orange line). For (1), a_0 should enter the corridor from e_1 after a_1 travels the corridor and leaves e_1. Then, the minimal time for a_0 to reach g_0 is $max\{t_0(e_1), t_1(e_1) + 1\} + dist(e_0, g_0)$. For (2), similarly, the minimal time that a_0 reaches g_0 is $max\{t_0(e_1), t_1(e_0) + 1\} + dist(e_0, g_0)$. Combining case (1) and (2), $l = min_{i=0,1}\{max(t_0(e_i) - 1, t_1(e_i)\} + dist(e_i, g_0)$. In case (2), if a_0 enters the corridor and reaches the goal from e_0, it needs to first travel the corridor from e_1 to e_0 and leave e_0. Then, it needs to reenter e_0 after a_1 enters e_0 to travel the corridor. The current limitation is that the cost for a_0 to reach e_0 ($t_0(e_0)$) is the first time that a_0 arrives at e_0 in the existing method, which does not consider the cost of leaving and reentering the corridor. As a result, is the additional cost of leaving and reentering is not considered, the resulting constraints may not solve this example in one split. Therefore, we further refine case (2) to two sub-cases: (2.a) agent a_0 enters from e_0 using a bypass that does not traverse the corridor, which does not require reentering the corridor, the cost to reach e_0 is $t_0^{'}(e_0)$; (2.b) a_0 enters the corridor from e_1 to e_0, and uses at least 2 timesteps to leave and reenter, the cost to reach e_0 is $t_1(e_0) + 2$. Combining (2.a) and (2.b), the cost for a_0 to reach e_0 is $min(t_0^{'}(e_0), t_0(e_0) + 2)$, and minimal path cost for a_0 to reach its goal in case (2) is $max\{min(t_0^{'}(e_0), t_0(e_0) + 2), t_1(e_0) + 2\} + dist(e_0, g_0)$, and l is the minimal between the path cost for a_0 in case (1) and the modified path cost in case (2).

We apply the same reasoning for MAPF$_T$, with more cost considered in l: (1) turning cost in $dist()$, including the turnings when corridor is not a straight line and the turnings to the goal orientation when agent reach the goal vertex; (2) same as in corridor reasoning, the additional turning cost when one agent leave the corridor and another agent attempt to enter; (3) for case (2), we also need to consider that in MAPF$_T$, the cost of leaving a vertex and reentering it is at least 4 due to turnings. Clearly, in case (1), the minimal time for a_0 to reach its goal is $max\{t_0(e_1), t_1(e_1) + 2\} + dist(e_1, g_0)$. In case (2.b), the minimal time a_0 to reach goal from e_0 is $min(t_0^{'}(e_0), t_0(e_0) + 4)$, so the minimal path cost of a_0 in case (2) is $max\{min(t_0^{'}(e_0), t_0(e_0) + 4), t_1(e_0) + 2\} + dist(e_0, g_0)$.

Note that Varambally, Li, and Koenig (2022) consider turn actions with a slightly different setting where agents are allowed to turn 180° in one timestep. Our improvements can also be easily adapted to this action model.

k	5	10	15	20	25	30
CT node	183	1,288	5,073	14,772	34,869	71,054
time(s)	0.01	0.16	0.92	4.37	15.59	40.97

Table 1: CT node expansions and time on solving the follow-up conflicts with CBS$_T$+Target reasoning of the type shown in Figure 2(a) for different distance k between s_0 and g_0.

Follow-up Conflicts in Target Reasoning

We use the term **follow-up** to describe that new conflicts are found between the same pair of agents after applying constraints to resolve a conflict between two agents. We observe that unlike in MAPF, target conflict resolution may not prevent *follow-up* conflicts at the target.

Just as for MAPF (Li et al. 2021), we generate two disjoint length constraints to resolve the target conflict. Suppose agent a_j reaches its goal g_j at timestep t_j and stays there forever, while agent a_i visits g_j at timestep t_i ($t_i \geq t_j$), the usual constraints applied to resolve this conflict are: (1) $l_j > t_i$: a_j can only finish after a_i visits its goal. and (2) $l_j \leq t_i$: a_j must finish its path by timestep t_i, and other agents cannot visit g_j at any time equal or after t_i.

Figure 2(a) is an example of the follow-up conflict in target reasoning. The target conflict occurs at vertex $D2$ at timestep $t = 3$. On one branch, CBS constrain a_0 to finish no later than 3; we generate a solution of cost 11, as a_1 is forced to divert around D2. On the other branch, CBS constrains a_0 to finish later than 3; it will try to finish at 4, which will still cause a conflict with agent 0, either at C2 at time 2, or D2 at time 3. Unlike MAPF, it has no length 4 path to D2 that gets out of the way of agent 1. Hence, resolving the target conflict requires more conflicts to be resolved, and if we are not careful, we may try to simply resolve the conflict at D2 at time 3 in the same way we already did.

This kind of follow-up frequently happens when the shortest paths of two agents are in the same row. Table 1 demon-

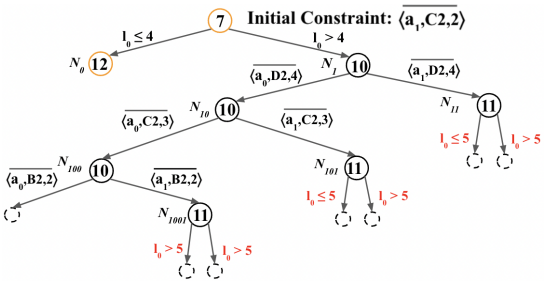


Figure 4: CT tree for the problem instance in Figure 2(a). To simplify the illustration, we give an initial constraint on root and run CBS$_T$ with target reasoning. The branch with orange nodes leads to the optimal solution. Solid nodes are expanded nodes labelled with cost, and edges are labelled with constraints added. The constraints shown in red illustrate that different nodes encounter the same sub-problem and generate the same constraints.

strates the impact that the follow-up problems can have on solving time and CT node expansions. In the CT tree, we observe that some constraints added in CT nodes may be redundant, because the constraints added when splitting is not disjoint. This redundancy also exists in the unexpected expansions in target follow-up conflicts, so we further develop a pruning method to reduce the redundant expansions.

High level Search Node Pruning

CBS and CBS$_T$ split nodes by adding constraints to resolve conflicts. However, the splitting is not *disjoint*. There may be solutions that appear in both child nodes of a parent. Disjoint splitting (Li et al. 2019b) can be used for simple conflicts but is too difficult to implement for rectangle and corridor conflicts. While the set of constraints we generate in each node in the CT tree is guaranteed to be different, it does not mean we cannot generate one node that is redundant (can only have a subset of solutions of) another node.

Redundant Nodes in High level Search

Consider Figure 4. Node N_{11}, N_{101} and N_{1001} generate the same splitting constraints. The constraints of node N_{1001} are: $l_0 > 4 \wedge \overline{\langle a_0, D2, 4\rangle} \wedge \overline{\langle a_0, C2, 3\rangle} \wedge \overline{\langle a_1, B2, 2\rangle}$. For node N_{101}, they are: $l_0 > 4 \wedge \overline{\langle a_0, D2, 4\rangle} \wedge \overline{\langle a_1, C2, 3\rangle}$.

Now the constraint $\overline{\langle a_1, B2, 2\rangle}$ forces a_1 not to be at B2 at time 2, and as a consequence, it cannot be at C2 at time 3, so $\overline{\langle a_1, B2, 2\rangle}$ implies $\overline{\langle a_1, C2, 3\rangle}$, written $\overline{\langle a_1, B2, 2\rangle} \rightarrow \overline{\langle a_1, C2, 3\rangle}$. That means that any solution of the node N_{1001} must also be a solution of N_{101}. Similarly, because $\overline{\langle a_1, C2, 3\rangle} \rightarrow \overline{\langle a_1, D2, 4\rangle}$, N_{101} defines a sub-problem of N_{11}, and hence, N_{1001} is also a sub-problem of N_{11}. This shows that the exploration of N_{1001} and N_{101} would be redundant efforts, as the same solutions exist under N_{11}.

We say that a node N_1 *subsumes* another node N_2 if $N_2.const \rightarrow N_1.const$. In this case, we can safely remove N_2 from the nodes to be explored, since any solution will still be available under N_1. As shown in Table 2, eliminating

k	5	10	15	20	25	30
CT node	30	74	144	239	360	505
time(s)	<0.01	<0.01	0.01	0.03	0.07	0.11

Table 2: CT node expansions and time on solving the follow-up conflicts with CBS$_T$+Target reasoning + subsumption pruning of the same experiments in Table 1.

subsumed nodes significantly reduces the number of expansions in target follow-up problems. In this section, we show a novel method that can be used in CBS and CBS$_T$ to reduce searching these redundant sub-problems This method works by pruning high level search nodes and can be directly enabled with the existing advances.

Subsumption Identification

When expanding a CT node to resolve a conflict, for example, a conflict $\langle a_0, a_1, v, t\rangle$, CBS generates two children, the left child with constraint $\overline{\langle a_0, v, t\rangle}$, and the right child with constraint $\overline{\langle a_1, v, t\rangle}$, which causes the solution that satisfies $\overline{\langle a_0, v, t\rangle} \wedge \overline{\langle a_1, v, t\rangle}$ is kept in both children. Therefore, we want to prune the duplicate part in one child while keeping this in another children by subsumption checking. However, since the number of nodes in the CT tree can grow very large, simply checking a new node against all existing nodes is too expensive. Therefore, to reduce the time overhead on finding and checking subsumption, our method checks against only some existing nodes.

We need to be careful when we eliminate a subsumed node that we have not relied (implicitly) on the subsumed node to allow the earlier subsumption of another node. To avoid this problem we only allow subsumption of a node N by a node that appears to the right of it in the tree (i.e. after it in a depth-first left to right traversal of the tree). In this way we can never subsume a node we have relied on by using information of nodes to its left.

We introduce two properties that help us to check subsumption of nodes.

Property 1. *If a node N_1 cannot subsume another node N_2, then any ancestor node of N_1 also cannot subsume N_2.*

Proof. As node N_1 cannot subsume node N_2, by definition, is $N_2.const \not\rightarrow N_1.const$. Since any ancestor node n of N_1 adds constraints to $N_1.const$, hence $N_2.const \not\rightarrow n.const$ because $N_1.const \subset n.const$. □

Property 2. *If a node N_1 is a child of an ancestor A of node N_2 where $N_1.agent \neq N_2.agent$ then N_1 cannot subsume node N_2 unless it also subsumes N_2's parent.*

Proof. Suppose N_1 subsumes N_2 then $N_2.const \rightarrow N_1.const$. Let N_3 be the parent of N_2. Now all of N_1, N_2 and N_3 share all constraints of the common ancestor $A.const$ and $N_1.const = A.const \wedge N_1.new$ since A is its parent, while $N_2.const = N_3.const \wedge N_2.new$. Now $N_2.const \rightarrow N_1.const$ hence $N_2.const \equiv N_3.const \wedge N_2.new \rightarrow N_1.new$. Suppose $N_2.new$ constrains a different agent then $N_1.new$, then since all constraints only

Algorithm 1: CheckSubsumed(N)

```
1: INPUT: Node N
2: OUTPUT: return true if N can be subsumed by one of the right
   children of its ancestors.
3: A ← Root
4: while A ≠ N do
5:    C ← A.right
6:    if C is not an ancestor of N then
7:       if C.agent = N.agent ∧ Subsume(C, N, N.agent)
         then
8:          return true
9:       else
10:         A ← A.left
11:      end if
12:   else
13:      A ← A.right
14:   end if
15: end while
16: return false
```

involve one agent, it must be the case that $N_3.const \rightarrow N_1.new$ and hence N_3 is subsumed by N_1 since the remaining constraints in N_1 ($A.const$) are shared by N_3. □

Property 1 means that for a node N_2 we only need to check the children of its ancestor nodes for a possibly subsuming node N_1, since N_1's descendants will only subsume node N_2 if N_1 subsumes N_2. We will not check all children of ancestors, just the right children, in order to avoid using a node to subsume another and then removing part of its search space by subsumption possibly losing solutions. Property 2 means that we only have to check children of ancestors on N_1 which added a constraint on the same agent as N_1.

Algorithm 1 demonstrates the subsumption checking algorithm. For each new node N as its created we call CheckSubsumed(N), if this returns *true* we dont insert the node in the tree. The algorithm walks the ancestors A of N from $Root$ to n. If the right child C of an ancestor A is not an ancestor of N and added a new constraint on the same agent we check to see if it subsumes N, and if so return. Otherwise we update A to be the next ancestor of N. If we reach N the subsumption check fails.

What remains is how to compute the subsumption check $Subsume(N_1, N_2, a)$ which checks that N_1 subsumes N_2 with respect to the constraints on agent a. It is implemented as follows. Since $N_1.parent.const \subset N_2.const$, so for the last added constraint $N_1.new$ constraints agent a, we run a low level search for a subject to $N_2.const$ to check if it can violate $N_1.new$. This amounts to checking if vertex v is reachable at time t for a constraint resulting from a vertex conflict $\langle a, a', v, t\rangle$, or checking a disallowed edge e at time t is reachable for an edge conflict, or any of the disallowed barrier vertex/time points are reachable in the case of a barrier constraint arising from rectangle or corridor reasoning. For constraint $N_1.new$ arising from target reasoning we can directly return *false* since the split is disjoint.

$CheckSubsumed$ is efficient since for each node n we only check a number of nodes less than its depth in the tree. We only check subsumption on the new constraint agent $N.agent = C.agent$. Because we only check the last added constraint, the check is simply running a low level search with a given timestep.

We now prove that the subsumption checking never loses solutions.

Theorem 1. *Removing nodes from the CT tree using $CheckSubsumed$ never eliminates a solution $P = \{p(a) \mid a \in \{a_1, \ldots, a_k\}\}$.*

Proof. We first show that every solution P in the pruned node n must appear in the subsuming node C. Since C only further constrains $a = C.agent = n.agent$, for each other agent a', $p(a')$ is a solution for A (since its a solution for n its ancestor) and also for C. Now $Subsume(C, n, a)$ succeeded so all possible paths for a in node n do not violate $C.new$.

Second we need to show that the solution P will not be pruned from the subtree under C, unless it appears elsewhere in the CT tree. By the correctness of CBS solution P is never removed by a CBS split (splits never remove any solution), so P must appear as a solution of at least one descendent of C say n' at every level of its subtree. So the only way P can disappear as a solution of the CT tree under C is if n' is then subsumed. Note this node n' occurs to the right of n in the CT tree. We can imagine a chain of subsumed nodes n, n', n",... each holding solution P. But this chain must be finite since each node appears further to the right of the previous one in the CT tree, and the CT tree is finite. Hence there is a node at the end of the chain which is not subsumed, and shows that the solution P remains in the pruned CT tree. □

Conflict Prioritisation Based on Pruning

Each node of the CT tree is processed in polynomial time, therefore the reason why CBS is an exponential-time algorithm is that the CT tree can itself have an exponential size. Because the size of the tree depends on the choices of conflicts resolved at each node, improving the splitting choice has a high potential for runtime improvements.

Earlier in the paper we introduce pruning, which offers a way of reducing the size of the CT tree. Current pruning works only by chance that we create a node that can subsume or be subsumed by another. The novelty we suggest now, in combination with pruning, is to not leave this entirely to chance, but rather to choose to resolve conflicts if they are more likely to lead to pruning. In order to do this, when choosing a conflict to resolve, we prioritise conflicts that have been resolved previously in the tree, to increase the probably that descendants of the current node can trigger a subsumption. We break ties using the previous prioritisation based on cardinality.

Experiments

The implementation is programmed in C++ and based on heuristics, prioritisation and symmetric reasoning of Li et al. (2021) and the modifications to existing advances and pruning method are made on top of it. [1] We denote by PCRTC only modification to the low level search in CBS$_T$ (with

[1] Code is at https://github.com/YueZhang-studyuse/MAPF_T

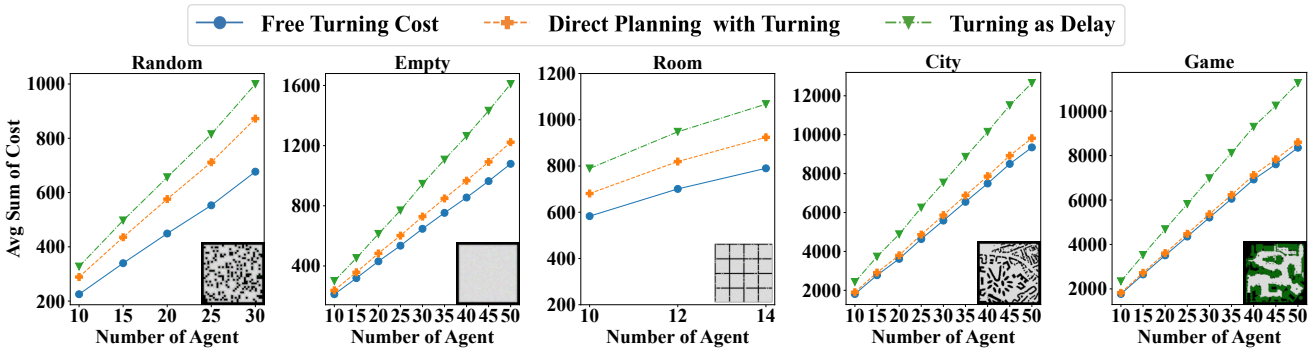


Figure 5: The same experiments as in Figure 1 in other five domains.

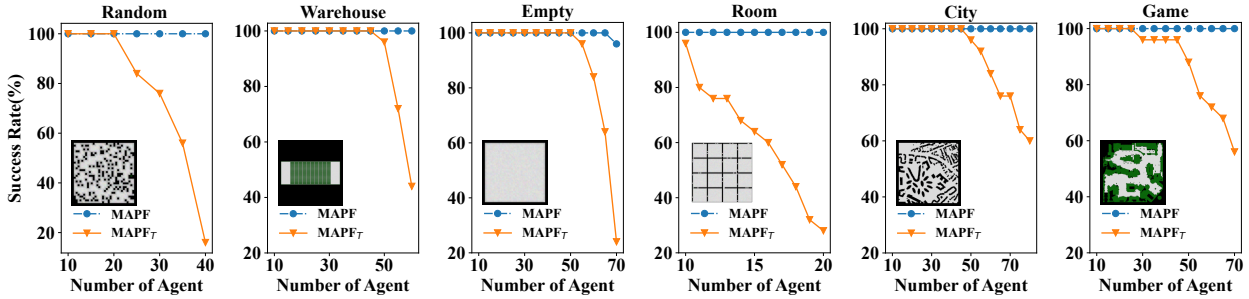


Figure 6: Success rate between CBS with PCRTC and CBS$_T$ with PCRTC.

WDG heuristic, prioritisation modification (1),(2) in Subsection Prioritisation (PC) and unmodified symmetric reasoning) which we consider as baseline. We denote by PRCTC2 our CBS$_T$ with improved prioritisation and corridor reasoning, and our high level node pruning as -P. The experiments are conducted on a server with 16 VCPUs and 32GB RAM. We do experiments on six different maps in different domains using grid-based MAPF benchmarks from Stern et al. (2019), including *random-32-32-20* (denoted Random), *warehouse-10-20-10-2-1* (denoted Warehouse), *empty-32-32* (denoted Empty), *room-64-64-16* (denoted Room), *Paris_1_256* (denoted City) and *den520d* (denoted Game). For each map, we use all 25 random scenario files from the benchmark sets of the chosen map for the experiments. As for the number of agents, for Room, since the success rate is observed to decline significantly when the number of agents is set to 20, we start the number of agents at 10, increasing by 1 for this specific map. For the other 5 maps, we start the number of agents at 10, increasing by 5. For each number of agents, we solve all 25 instances. In order to ensure replicability, for each instance, we assign the start and target direction to be North. The runtime limit is set to 60 seconds for each instance. For the failed results, the runtime is set to be 60 seconds.

Experiment 1: Solution Quality in other domains through direct turning planning. We conduct experiments as in Figure 1 in five additional domains. We note that the post-execution method incurs negligible time overhead ($\leq$ 0.01s per instance), and direct planning with turn actions may result in longer planning time than leaving it to the post-processing phase, which could also be indicated from Experiment 2. We believe that the improved solution quality, as presented in Figure 5, justifies the extra computational effort, and our improvements could also help reduce the computational challenge incurred by direct planning.

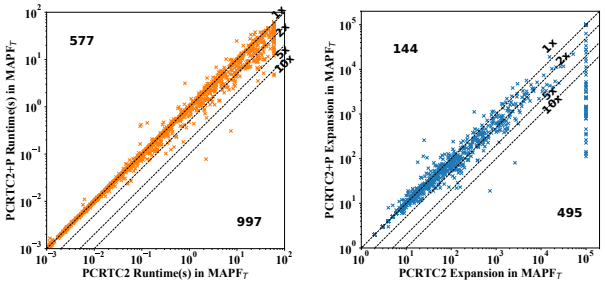
Experiment 2: Investigating the Challenges of MAPF$_T$ Even with modern enhancements, MAPF$_T$ problem remains challenging. In Figure 6, we present experimental results comparing success rates. The results indicate that, for all MAPF$_T$ instances in all domains, the success rate of PCRTC drops significantly as the number of agents increases when compared to CBS solving the simpler MAPF problem.

Experiment 3: Our improvements in MAPF$_T$. Table 3 shows the performance of PCRTC2, PCRTC+P and PCRTC2+P compared to PCRTC, including the number of instance solved, the total runtime and the pruning overhead. Our experiments demonstrate that, compared to PCRTC, pruning (PCRTC+P) results in better runtime and more instances solved in all domains. PCRTC2 performs better on solved instances in City, Random, and Empty maps, and has significant improvements on Room maps. In maps where PCRTC2 outperforms PCRTC, PCRTC2+P further significantly improves the number of instances solved. Our results also show that subsumption checking only accounts for around 1% of the total runtime for each type of map.

We also observe that: (1) PCRTC2+P does not always perform well on runtime compared to PCRTC+P; (2) PCRTC2 solves less instances than PCRTC in some domains; (3) our improvements on large maps, like Warehouse, Game and City, are not as significant as that on smaller maps. To the

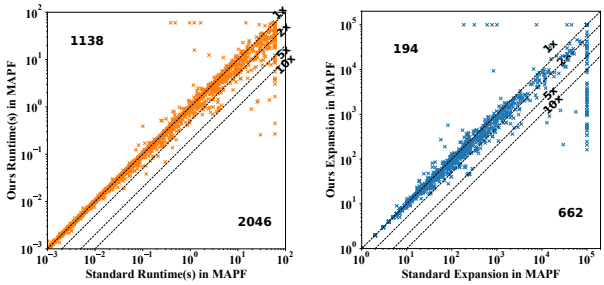
Map	Number of Instances					Total Runtime(all failed excluded)				
	Total Attempts	PCRTC Solved	ΔSolved Compared to PCRTC			PCRTC Total Runtime(s)	Speedups Compared to PCRTC			Pruning Overhead
			PCRTC2	PCRTC+P	PCRTC2+P		PCRTC2	PCRTC+P	PCRTC2+P	
Random	175	133	+8	+6	**+13**	1225.76	x1.83	x1.64	**x2.23**	1.68%
Warehouse	275	256	-9	**+4**	-4	1432.15	x0.71	**x1.15**	x1.04	0.07%
Empty	325	293	+11	+19	**+21**	1904.36	x1.5	**x1.86**	x1.6	0.5%
Room	275	167	+46	+23	**+62**	4552.31	x2.24	x1.29	**x2.15**	1.59%
City	375	338	+5	+9	**+11**	2310.04	x1.09	**x1.21**	x1.09	0.4%
Game	325	285	-4	**+8**	+3	2975.93	x0.87	**x1.16**	x1.09	0.37%

Table 3: Improvement results. Column 4-6 is the difference on number of solved instances compared to PCRTC. ΔSolved = Total Solved of the given solver - Total Solved of PCRTC. Column 7-11 shows runtime on the instances excluding those where all methods failed, i.e. including instances solved by at least one solver. We set failed instance's runtime to the runtime limit (60s). Column 8-10 shows the runtime speedups compared to the PCRTC. We divide the PCRTC total runtime by total runtime of the given solver to compute speedup. Column 11 is the pruning overhead for PCRTC2+P on all instances (including failed), which is the percentage of time spent on subsumption checking relative to the total runtime.



(a) Runtime (b) Node Expansion

Figure 7: Scatter plot for runtime in seconds and node expansion for pruning comparison in MAPF$_T$ instances. If an instance is not solved within the time limit, we set its runtime to 60s and node expansions to 10^5. We compare PCRTC2+P with PCRTC2. For each node in the plot, the x coordinate is the runtime on PCRTC2, and the y coordinate is the runtime on PCRTC2+P. The number in the left corner shows the number of instances that PCRTC2 outperforms PCRTC2+P, while the number in the right corner is the number of instances that PCRTC2+P outperforms PCRTC2



(a) Runtime (b) Node Expansion

Figure 8: Scatter plot of runtime and expansions for pruning comparison in MAPF instances. We denote CBS + WDG heuristic + PCRTC as *Standard*, and compare with enabling pruning in standard and denote it as *Ours*. For timeout instances, we have the same settings as in Figure 7.

best of our knowledge, we identify three problems for future improvements: (1) generating new range constraints incurs runtime overheads; (2) low level search runtime on large maps is sometimes the major cause of the performance and (3) identifying cardinal conflicts and resolving them first, as in PCRTC2, may not be the best strategy for MAPF$_T$.

Experiment 4: Improved search efficiency through subsumption pruning. Figure 7 shows pruning improvements in a total of 1550 instances. Our results indicate that pruning saves significant search effort and outperforms no pruning in almost all cases. In addition, pruning also helps in MAPF problems, so we also conduct experiments in MAPF instances. Since CBS can solve MAPF problems with more agents compared to MAPF$_T$, we increase the number of agents to solved in experiments on MAPF with a total number of 3025 instances including: 20-70 agents in Random, 20-140 agents in Warehouse, 20-140 agents in Empty, 15-50 agents in Room, 20-150 agents in Game and 20-200 agents in City. For each map, we increase the number of agents by 5. Figure 8 shows our pruning also improves the runtime and node expansions in MAPF. In summary, for MAPF instances, in the instances solved by at least one solver, our pruning method achieved around 1.41 times speedup.

Conclusion and Future Work

We show the benefits of direct planning with turn actions in MAPF, called MAPF$_T$, and adapt the existing state-of-the-art algorithm to solve planning with turn actions optimally. While MAPF$_T$ increases the difficulty to solve because turning is not free, this research also proposes a novel method to speed up runtime and reduce node expansions. As for future work, we address two current limitations. First, we have not yet explored non-uniform-cost turns and other kinematic constraints, such as the model proposed in (Ma et al. 2019), which is an interesting aspect for future work. Second, as shown in experiments, more speedups need to be considered such as reducing overheads and changing splitting strategy.

Acknowledgements

This work is supported by the Australian Research Council under grant DP200100025, and by a gift from Amazon.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search*, 19–27.

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

Cohen, L.; Uras, T.; Kumar, T. S.; and Koenig, S. 2019. Optimal and bounded-suboptimal multi-agent motion planning. In *Proceedings of the Twelfth Annual Symposium on Combinatorial Search*, 44–51.

Dresner, K.; and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research*, 31: 591–656.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J. W.; and Ayanian, N. 2019. Persistent and robust execution of mapf schedules in warehouses. *IEEE Robotics and Automation Letters*, 4(2): 1125–1131.

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 Twenty-Eighth International Joint Conference on Artificial Intelligence*, 442–449.

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

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; Gange, G.; and Koenig, S. 2021. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301: 103574.

Ma, H.; Hönig, W.; Kumar, T. S.; Ayanian, N.; and Koenig, S. 2019. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7651–7658.

Ma, H.; Kumar, T. S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1, 3605–3612.

McNaughton, M.; Urmson, C.; Dolan, J. M.; and Lee, J.-W. 2011. Motion planning for autonomous driving with a conformal spatiotemporal lattice. In *Proceedings of the 2011 IEEE International Conference on Robotics and Automation*, 4889–4895.

Pivtoraiko, M.; Knepper, R. A.; and Kelly, A. 2009. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(3): 308–333.

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial intelligence*, 195: 470–495.

Silver, D. 2005. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, 1, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the Twelfth Annual Symposium on Combinatorial Search*, 151–158.

Varambally, S.; Li, J.; and Koenig, S. 2022. Which MAPF model works best for automated warehousing? In *Proceedings of the International Symposium on Combinatorial Search*, 1, 190–198.