Conflict-Based Increasing Cost Search

Thayne T. Walker[1,2], Nathan R. Sturtevant[3], Ariel Felner[4], Han Zhang[5]
Jiaoyang Li[5] and T. K. Satish Kumar[5]

[1]University of Denver, Denver, USA
[2]Lockheed Martin Corporation, USA
[3]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
[4]Ben-Gurion University, Be'er-Sheva, Israel
[5]University of Southern California, Los Angeles, USA
thayne.walker@du.edu, nathanst@ualberta.ca, felner@bgu.ac.il, {zhan645, jiaoyanl}@usc.edu, tkskwork@gmail.com

Abstract

Two popular optimal search-based solvers for the multi-agent pathfinding (MAPF) problem, Conflict-Based Search (CBS) and Increasing Cost Tree Search (ICTS), have been extended separately for continuous time domains and symmetry breaking. However, an approach to symmetry breaking in continuous time domains remained elusive. In this work, we introduce a new algorithm, Conflict-Based Increasing Cost Search (CBICS), which is capable of symmetry breaking in continuous time domains by combining the strengths of CBS and ICTS. Our experiments show that CBICS often finds solutions faster than CBS and ICTS in both unit time and continuous time domains.

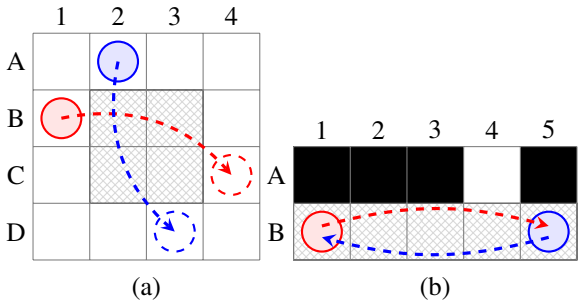

Figure 1: Illustration of MAPF instances with conflict symmetries: (a) a rectangle conflict and (b) a corridor conflict.

Introduction

The objective of multi-agent pathfinding (MAPF) (Stern et al. 2019) is to find paths for multiple agents to their respective goal states such that they do not conflict at any time. A *conflict* occurs when agents' shapes overlap at the same time. We seek optimal, conflict-free solutions to the MAPF problem in both "classic" unit time domains, where all actions have the same duration, and continuous time domains, where action durations are arbitrary real-valued quantities.

Conflict symmetries pose a particular challenge for current MAPF algorithms and can incur an exponential amount of work without special enhancements (Li et al. 2020). Conflict-Based Search (CBS) (Sharon et al. 2015) is sensitive to two types of conflict symmetries: spatial conflicts and time-extended conflicts. A *spatial conflict* occurs when all lowest-cost paths for two agents conflict when either of them move through a region called a *region of conflict*. Figure 1(a) illustrates a *rectangle conflict* (Li et al. 2019b) in a 4-neighbor grid map where the red and blue agent (shown as filled circles) try to move to their respective goals (shown with dashed lines). The shaded region in the center is the region of conflict. No matter which optimal path the agents take through the region, they will always collide. Similar spatial conflicts can also occur in continuous time domains. A *time-extended conflict* occurs when two agents incur the same conflict or set of conflicts over and over at increasing time steps until one of the agents increases its cost sufficiently (usually by waiting) and allowing the other agent to pass. Figure 1(b) illustrates a *corridor conflict* in a 4-neighbor grid. The shaded region shows the region of conflict. Similar time-extended symmetries can occur in continuous time domains.

Increasing Cost Tree Search (ICTS) (Sharon et al. 2013) is robust to spatial conflict symmetries, but will incur an exponential amount of work in the case of time-extended conflict symmetries. Reduction-based approaches re-formulate the MAPF problem for general solvers such as SAT (Surynek et al. 2016) and MIP (Lam et al. 2019). They also have to deal with conflict symmetries and are generally not formulated for continuous time. Variants of CBS and ICTS have been re-formulated for continuous time (Walker, Sturtevant, and Felner 2018; Andreychuk et al. 2019; Cohen et al. 2019; Walker, Sturtevant, and Felner 2020). Enhancements for symmetry breaking in CBS have been formulated for unit time (Li et al. 2019b, 2020; Zhang et al. 2020). One symmetry-breaking technique for continuous time exists (Walker, Sturtevant, and Felner 2020) but is limited to spatial symmetries. However, general symmetry breaking in continuous time domains has remained elusive so far.

In this work, we introduce Conflict-Based Increasing Cost Search (CBICS), a hybrid of CBS and ICTS which combines their strengths. CBICS allows symmetry breaking in both unit time and continuous time domains. Search nodes in

CBICS contain information about both conflicts (like CBS) and costs (like ICTS). This novel formulation allows us to find solutions faster and with a higher success rate in most MAPF instances versus previous state-of-the-art.

Problem Definition

MAPF was originally defined for a "classic" setting where the movements of agents are coordinated on a graph $G = (V, E)$, where edges have a unit cost/unit time duration and agents occupy a point in space. Thus, two agents can only have conflicts when on the same vertex at the same time, or traversing the same edge in opposite directions. This paper uses the definition of MAPF_R (Walker, Sturtevant, and Felner 2018), a variant of MAPF for continuous time motion where every vertex $v{\in}V$ is associated with coordinates in a metric space and every edge $e{\in}E$ is associated with a positive real-valued edge weight $w(e){\in}\mathbb{R}_+$. Weights represent the times it takes to traverse edges. Additionally, there is a set of k agents, $A{=}\{1, .., k\}$. Each agent has a start and a goal vertex $V_s{=}\{start_1, .., start_k\}{\subseteq}V$ and $V_g{=}\{goal_1, .., goal_k\}{\subseteq}V$ such that $start_i{\neq}start_j$ and $goal_i{\neq}goal_j$ for all $i{\neq}j$.

A *solution* to a MAPF_R instance is $\Pi{=}\{\pi_1, .., \pi_k\}$, a set of single-agent *paths* composed of *states*. A state $s{=}(v, t)$ is a pair composed of a vertex $v{\in}V$ and a time $t{\in}\mathbb{R}_+$. A path for agent i is a sequence of $d{+}1$ states $\pi_i{=}[s_i^0, .., s_i^d]$, where $s_i^0{=}(start_i, 0)$ and $s_i^d{=}(goal_i, t)$ where t is the time the agent arrives at its goal and all vertices in the path follow edges in E.

Agents have a shape, such as a circle or polygon, which is situated relative to an agent-specific *reference point* (Li et al. 2019c). Agents move along edges such that their motion uses constant velocity along a straight vector in the metric space. Traversing an edge is called an *action* $a{=}(s^n, s^{n+1})$. MAPF_R also allows weighted, self-directed edges for *wait* actions. The duration of actions can be fixed or dynamically computed. This paper assumes fixed-duration wait actions.

A *conflict* happens when two agents perform actions $\langle a_i, a_j\rangle$ so that their shapes overlap at the same time. A *feasible solution* has no conflicts between any pairs of its constituent paths. The objective is to minimize the sum-of-costs $c(\Pi){=}\sum_{\pi\in\Pi} c(\pi)$, where $c(\pi)$ is the sum of edge weights of all edges traversed in π. We seek Π^*, a solution with minimal cost among all feasible solutions. Optimization of the classic MAPF problem is NP-hard (Yu and LaValle 2013). Hence, optimization of the MAPF_R problem is also NP-hard.

Background

In this section, we cover previous work which we combine to formulate CBICS.

Conflict-Based Search

Conflict-Based Search (CBS) (Sharon et al. 2015) performs search on two levels. The *high level* searches a constraint tree (CT), shown in Figure 2(c). Each node N in the CT contains a solution $N.\Pi$. Each path $\pi{\in}N.\Pi$ of an agent in the root node is constructed using a *low-level* search without taking the other agents into account. Next, CBS checks for conflicts in $N.\Pi$. If $N.\Pi$ contains no conflict, then N is a goal node and CBS terminates. If $N.\Pi$ contains a conflict between agents i and j, then CBS performs a *split*, meaning that it generates two child nodes N_i and N_j of N and adds motion constraints m_i and m_j to N_i and N_j respectively.

A *motion constraint* blocks an agent from performing the action(s) that caused the conflict. Two types of motion constraints are used with classic MAPF: *vertex* constraints and *edge* constraints. A vertex constraint blocks an agent from occupying a vertex at a specific time. An edge constraint blocks an agent from traversing an edge at a specific time. The accumulation of constraints in the CT is shown in Figure 2(c). The notation "B2@1" for a constraint means the agent is blocked from occupying vertex B2 at time step 1. Next, CBS re-plans $\pi_i{\in}N_i.\Pi$ and $\pi_j{\in}N_j.\Pi$ with motion constraints m_i and m_j and other motion constraints from ancestor nodes so that the current conflict and previously detected conflicts are avoided. CBS searches the tree in a best-first fashion prioritized by the sum-of-costs. CBS terminates when its OPEN list is empty or a feasible solution is found.

CBS is guaranteed to find a feasible solution if one exists, otherwise it may run forever. For the classic problem, a polynomial-time (non-optimal) algorithm (Botea, Bonusi, and Surynek 2018) can be run in parallel to determine if a solution exists. At this time, the existence of polynomial-time solvers for MAPF_R is an open question.

A significant amount of research has studied how to improve the performance of CBS by dealing with conflict symmetries (Boyarski et al. 2015; Gange, Harabor, and Stuckey 2019; Li et al. 2019b,a, 2020; Zhang et al. 2020; Walker, Sturtevant, and Felner 2020). Some of this work has leveraged ideas from ICTS, in particular, the analysis of conflict symmetries using multi-value decision diagrams (MDDs) (Srinivasan et al. 1990). An MDD is a directed acyclic graph where nodes are connected via edges to form paths from start to goal for an agent such that the agent arrives at its goal within a cost limit. Hence it contains one root node ($start_i$ for agent i) and one sink node ($goal_i$). Some examples of MDDs are shown in Figure 3.

Extended Increasing Cost Tree Search

Extended Increasing Cost Tree Search (E-ICTS) (Walker, Sturtevant, and Felner 2018) is an extension of ICTS (Sharon et al. 2013) for MAPF_R. E-ICTS is a two-level search algorithm. On its high level, it searches the increasing cost tree (ICT), as shown in Figure 2(b), where every node consists of a k-ary vector of cost ranges $\langle(c_1, c_1 + \delta], .., (c_k, c_k + \delta]\rangle$, where δ is an *increment* value which represents the path cost ranges of a solution. For the example in Figure 2(a), with $\delta{=}1$, an ICT node may contain the vector of ranges: $\langle[2, 3), [2, 3), [2, 3)\rangle$, which represents a solution with path costs between 2 and 3 for all agents. When δ is fixed, the upper bound of each range is implied, and ICT nodes contain the lower bounds only (e.g., $\langle 2, 2, 2\rangle$), as shown in Figure 2(b).The ICT root node contains the smallest path cost for each agent, ignoring the other agents.

For each ICT node, the low level is invoked. Its task is

to determine a lowest-cost feasible solution that falls in the ranges $\langle [c_1, c_1+\delta), .., [c_k, c_k+\delta) \rangle$. For each agent i, ICTS stores all single-agent paths in the range $[c_i, c_i+\delta)$ as an MDD. Figure 3(a) shows an example of two MDDs for agents x and z for the MAPF instance in Figure 2(a).

If no feasible solution is found by the low level, the ICT node is split. A node is split by generating k child nodes, each with the cost range of one agent increased by δ. For example, for the ICT node $\langle 2, 2, 2 \rangle$ and δ=1, a split would generate three child nodes: $\langle 3, 2, 2 \rangle$, $\langle 2, 3, 2 \rangle$ and $\langle 2, 2, 3 \rangle$, respectively.

The low level searches the Cartesian product of all time-overlapping actions of the k MDDs to find a set of k non-conflicting paths. Multiple solutions of different sum-of-costs may exist. If any feasible solution exists, the low-level returns the minimum sum-of-costs. Otherwise, it returns infinity. In case the low level finds a feasible solution, it becomes the new incumbent solution at the high level. The high level orders the OPEN list by sum-of-costs. The search halts either when no node in the OPEN list has a better cost than the incumbent or the OPEN list is empty.

Mutex Propagation

Mutex propagation (MP), a technique for finding unreachable states in planning graphs (Weld 1999), has been integrated into CBS for symmetry breaking (Zhang et al. 2020). MP helps to determine sets of mutually exclusive states for conflicting agents, often allowing for an immediate resolution of conflict symmetries. This technique is general, and applies to all types of conflict symmetries. MP for unit time is carried out in four steps:

1. Build an MDD for each agent.
2. Discover initial mutexes between MDDs.
3. Propagate the mutexes.
4. Extract motion constraints for CBS.

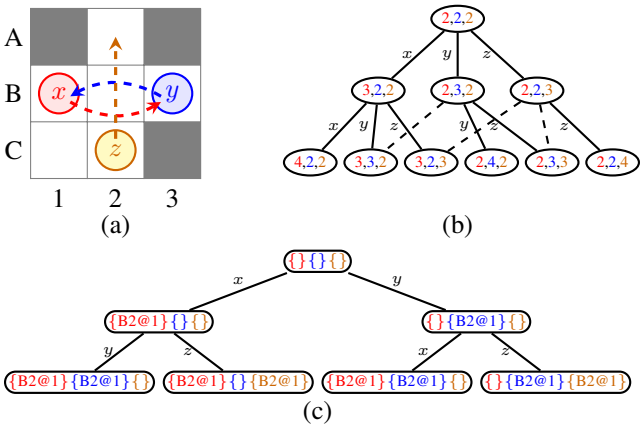

Figure 2: (a) An example unit time MAPF instance, (b) a *partial* ICT with an implied δ of 1 for the MAPF instance and (c) a *partial* CT for the MAPF instance.

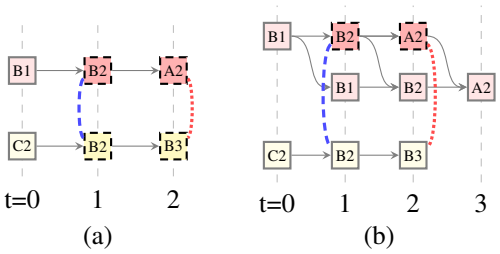

Figure 3: (a) Mutex propagation for agents x and z for the MAPF instance in Figure 2(a) with cost limits $\langle 2, -, 2 \rangle$ and (b) the same analysis as (a) but with cost limits $\langle 3, -, 2 \rangle$.

In Step 1, MDDs are built for two conflicting agents as shown in Figure 3(a).

In Step 2, the MDDs are traversed in parallel from start to goal. At each time step, the Cartesian product of the two MDDs are merged and checked for conflicts. For example, in Figure 3(a), the combination {B1}×{C2}={(B1,C2)} is checked at time step t=0, and so on, for each time step. A mutex $\langle s_i^t, s_j^t \rangle$ is created for any pair $s_i^t \in MDD_i$ and $s_j^t \in MDD_j$ of MDD nodes (or edges) which conflict (e.g., (B2,B2) at time step 1). These initial mutexes are depicted as blue dashed lines in Figure 3(a).

In Step 3, the initial mutexes are propagated, meaning that whenever all parent MDD nodes of s_i^t are mutex with all parent MDD nodes of s_j^t, a new mutex $\langle s_i^t, s_j^t \rangle$ is created. These propagated mutexes are depicted as red dotted lines in Figure 3(a). For example, at time step 2, B3 and A2 get a propagated mutex because their only parents (B2 and B2) respectively, at time step 1 are mutex.

In Step 4, the pair of MDDs is analyzed against the mutexes. For any s_i^t which is mutex with all s_j^t at the same time step, vertex constraints are created for s_i^t. These are shown in Figure 3(a) as dashed nodes. For example, B2 at time step 1 in the upper MDD is used for a constraint for agent x because it is mutex with all nodes at the same time step in the lower MDD. In Figure 3(b), the cost limit for agent x has been increased to 3, resulting in additional MDD nodes being added to the upper MDD. Because of these additional MDD nodes, B2 in the lower MDD is no longer mutex with all MDD nodes in the upper MDD at t=1. Hence it can no longer be used for a constraint in CBS.

Conflict-Based Increasing Cost Search

In this section, we describe Conflict-Based Increasing Cost Search (CBICS), a new hybrid of CBS and ICTS. The main idea of CBICS is that useful information is learned by analyzing the path costs, which can be exploited to avoid unnecessary work. In Figure 2(a), the path costs of the agents are x=2, y=2, z=2 when conflicts between agents are ignored. To resolve the conflict between agents y and z, at location B2 at the first time step, agent y must wait for agent z to enter B2, or vice versa. This results in the path cost combinations $\langle y=2, z=3 \rangle$ and $\langle y=3, z=2 \rangle$. Hence, $y+z$=5. We refer to this sum $lb_{i,j}$ (for arbitrary agents i and j) as the *pairwise lower bound*, since there is no feasible solution whose sum

is less than this bound.

Without the information that $lb_{y,z}$=5, $lb_{x,z}$ also appears to be 5 (agent z must wait for agent x or vice versa). However, given $lb_{y,z}$=5, it must be that $lb_{x,z}$=6: If we fix y=2 and z=3, the path cost for agent x must be $x\geq 4$. For example, if agent y follows the path *left, left* and agent z follows the path *wait, up, up*, then the only lowest-cost path for agent x is *down, right, up, right*. In general, by fixing the path costs of some agents, we can learn about the path costs of other agents.

Consider the *partial* search trees for CBS (Figure 2(c)) and ICTS (Figure 2(b)). CBS adds only constraints at each depth of the CT, eventually resulting in enough cumulative constraints to eliminate infeasible path combinations. But, little insight is gained at each depth in the CT. ICTS systematically increases the path cost for each agent, but does not learn that some subsets of path costs can *never* lead to a feasible solution.

CBICS gains insight about feasible path cost combinations and uses both motion constraints *and* cost constraints in order to reduce the size of the search tree and find solutions more quickly.

CBICS High Level

CBICS searches the cost-range constraint tree (CRCT) as shown in Figure 4. A CRCT node is a tuple: $\langle R, M, LB, \Pi, SoC\rangle$ where R=$\{r_1, .., r_k\}$ is a set of path cost ranges for each agent where r_i=$[lb, ub)$, for example r_i=$[2, \infty)$. We use the shorthand 2+ for $[2, \infty)$ and 2 for $[2, 2]$. M=$\{M_1, .., M_k\}$ is a set of motion constraint sets for each agent. LB=$\{lb_{1,2}, lb_{1,3}, .., lb_{(k-1),k}\}$ is the set of lower bounds for the sum of path costs for all pairs of agents (pairwise path costs). SoC is the sum-of-costs of all agents. Π is a solution. For a CRCT node N we use the shorthand $N.r_i$ to refer to the cost range in $N.R$ for the ith agent, $N.M_i$ to mean the set of motion constraints in $N.M$ for the ith agent, $N.\pi_i$ to mean the path in $N.\Pi$ for the ith agent and $N.lb_{i,j}$ to mean the sum of costs for the ith and jth agents.

Each node in Figure 4 shows the path cost ranges R (in colors) and SoC (in black parentheses) in the top row, pairwise cost information LB in the second row (in black) and motion constraint sets M in the remaining rows (in colors).

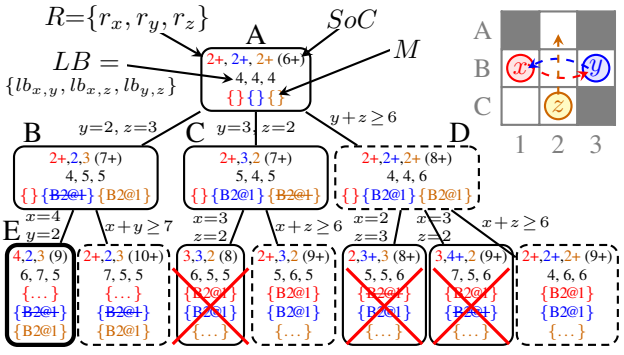

Figure 4: The *entire* CRCT for the MAPF instance in Figure 2(a).

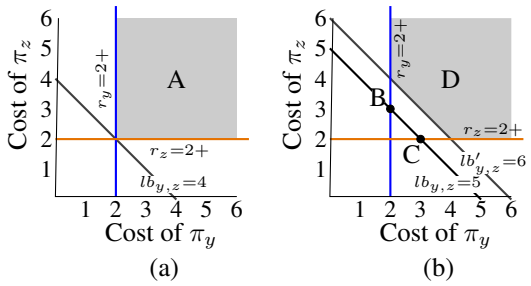

Figure 5: (a) The combined cost range of agents y and z for (a) node A and (b) for nodes B, C and D of Figure 4.

When motion constraint sets contain more than one member, "..." is used to indicate this. Recall from CBS that motion constraints $m\in N.M_i$ restrict agent i from performing an action at a specific time. A motion constraint is a tuple m=$\langle n, a\rangle$ where n is the agent number and a=$\langle s^t, s^{t+1}\rangle$ is the action for the agent to avoid. In this paper, the notation for motion constraints are in the format B1@0→B2@1 meaning the agent is prohibited from moving from location B1 at time step 0 and arriving at location B2 at time step 1. However, in Figure 4 motion constraints are shown with abbreviated notation like B2@1, which is shorthand for an action that ends at location B2 at time step 1. The start location can be inferred from the MAPF instance. The color indicates the agent number. When a motion constraint is struck through (e.g., ~~B2@1~~), it means that the constraint was conditionally removed. Conditional constraints are discussed later. The solution Π of each node is not shown.

Cost constraints restrict agents to paths with costs inside certain cost ranges. For $N.r_i$=$[lb, ub)$, $N.\pi_i$ is restricted to a path such that $c(N.\pi_i)\in[lb, ub)$. The pairwise path cost $N.lb_{i,j}$ similarly restricts paths based on the sum of two path costs. That is, for $N.lb_{i,j}$=ℓ, $N.\pi_i$ and $N.\pi_j$ are restricted such that $c(\pi_i)$+$c(\pi_j)\geq\ell$. The range of possible path costs for agents y and z for node A is shown in Figure 5(a). The cost region (shown in gray) represents all values bounded by r_y=2+ represented by the blue line, r_z=2+ represented by the orange line, and all pairwise path costs $lb_{y,z}$=4 represented by the black line.

Pseudocode for CBICS is shown in Algorithm 1. On line 3, the root CRCT node N (e.g., node A in Figure 4) is constructed. $N.\Pi$ in the root node contains paths for all agents, planned for shortest paths without taking the other agents into account. Each $r_i\in N.R$ is set to $[c(\pi_i), \infty)$ respectively, SoC is $c(N.\Pi)$, $N.LB$ gets the sum of costs for each pair in $N.\Pi$ and all elements of $N.M$ are set to empty. CBICS checks for conflicts between the paths in Π (line 10). If no conflict is found, $N.\Pi$ is set as the new incumbent (line 12). This incumbent is needed due to the lazy evaluation of some nodes. Some nodes may have a cost increase after evaluation. The OPEN list is ordered by SoC. If no better solution exists in the OPEN list, the incumbent is returned as the solution (line 14). If a conflict is found, the *low-level* subroutine PAIRWISECONSTRAINTSEARCH is called for the two conflicting agents (generically, agent i and j) (line 17).

Algorithm 1 CBICS Algorithm

```
1: Input: a MAPF instance
2: OPEN ← ∅
3: Initialize the root node and add it to OPEN
4: incumbent ← dummy with SoC=∞
5: while OPEN ≠ ∅ do
6:    N ← OPEN.pop()
7:    if N has any empty π_i ∈ N.Π then
8:       Re-plan each empty π_i with cost and motion constraints
9:    end if
10:   A ← FINDCONFLICT(N.Π) ▷ Find conflicting actions ⟨a_i, a_j⟩
11:   if A = ∅ then
12:      incumbent ← N if N.SoC < incumbent.SoC
13:      if incumbent.SoC ≤ OPEN.top.SoC then
14:         return incumbent
15:      end if
16:   else
17:      P, M'_i, M'_j, lb'_{i,j} ← PAIRWISECONSTRAINTSEARCH(i, j, N)
18:      if M'_i ≠ ∅ ∨ M'_j ≠ ∅ then ▷ Conjunctive split
19:         for ⟨π'_i, π'_j⟩ ∈ P do ▷ Nodes with pairwise costs = lb_{i,j}
20:            N' ← N
21:            N'.r_i ← [c(π'_i), c(π'_i)]
22:            N'.r_j ← [c(π'_j), c(π'_j)]
23:            N'.lb_{i,j} ← c(π'_i)+c(π'_j)
24:            N'.SoC ← N.SoC − c(N.lb_{i,j}) + N'.lb_{i,j}
25:            N'.M_i ← N'.M_i ∪ M'_i ▷ Cost-cond. constraints
26:            N'.M_j ← N'.M_j ∪ M'_j
27:            N'.π_i ← π'_i
28:            N'.π_j ← π'_j
29:            OPEN ← OPEN ∪ N'
30:         end for
31:         N' ← N ▷ Node for path costs at and above the next frontier
32:         N'.lb_{i,j} ← lb'_{i,j}
33:         N'.SoC ← N.SoC − c(N.lb_{i,j}) + N'.lb_{i,j}
34:         N'.M_i ← N'.M_i ∪ M'_i ▷ Cost-cond. constraints
35:         N'.M_j ← N'.M_j ∪ M'_j
36:         N'.π_i ← ∅ ▷ Will be replanned lazily on line 8
37:         N'.π_j ← ∅
38:         OPEN ← OPEN ∪ N'
39:      else ▷ Disjunctive, CBS-style split for conflicting agents
40:         N' ← N ▷ Create node for agent i
41:         N'.r_i ← [MIN_{π'_i∈P} c(π'_i), ∞)
42:         N'.lb_{i,j} ← c(P_0.π'_i)+c(P_0.π'_j) ▷ P_0 is the first pair in P
43:         N'.SoC ← N.SoC − c(N.π_i) + c(π'_i)
44:         N'.M_i ← N'.M_i ∪ {⟨i, a_i⟩} ▷ Regular motion constraint
45:         N'.π_i ← ∅ ▷ Will be replanned lazily on line 8
46:         N'.SoC ← N.SoC − c(N.π_i) + c(π'_i)
47:         OPEN ← OPEN ∪ N'
48:         N' ← N ▷ Create node for agent j
49:         N'.r_j ← [MIN_{π'_j∈P} c(π'_j), ∞)
50:         N'.lb_{i,j} ← c(P_0.π'_j)+c(P_0.π'_j) ▷ P_0 is the first pair in P
51:         N'.M_j ← N'.M_j ∪ {⟨j, a_j⟩} ▷ Regular motion constraint
52:         N'.π_j ← ∅ ▷ Will be replanned lazily on line 8
53:         N'.SoC ← N.SoC − c(N.π_i) + c(π'_i)
54:         OPEN ← OPEN ∪ N')
55:      end if
56:   end if
57: end while
58: return "No solution"
```

The PAIRWISECONSTRAINTSEARCH (PCS) plans two conflicting agents (agents i and j) jointly to find a feasible solution and discover motion constraints and cost constraints at the same time. PCS takes as input two path cost constraints r_i and r_j, a pairwise cost constraint $lb_{i,j}$ (also known as the *current pairwise cost frontier*) and motion constraints M_i and M_j. Line 17 shows N as the input because N contains all of the information needed. The output is: (1) a pair of new motion constraint sets M'_i and M'_j, (2) a *set* of lowest-cost path pairs $P=\{\langle\pi_i, \pi_j\rangle_1, .., \langle\pi_i, \pi_j\rangle_n\}$ such that $\forall(\pi_i, \pi_j)\in P$:

- π_i and π_j have no conflicts.
- $c(\pi_i)+c(\pi_j)\geq lb_{i,j}$ (The sum of each pair of path costs conforms to the pairwise cost constraint.)
- $c(\pi_i)\in r_i$ and $c(\pi_j)\in r_j$ (Each path cost conforms to the path cost constraints.)

and (3) $lb'_{i,j}$, the *next lowest pairwise cost frontier*. Further details of PCS are covered after the high level.

For example, in Node A of Figure 4, based on the conflict between agents y and z, r_y, r_z, $lb_{y,z}$, M_y and M_z in Figure 5(a) are passed to PCS in order to plan agents y and z jointly. If PCS was called for the example instance with the cost constraints r_y=2+, r_z=2+ and $lb_{y,z}$=4 with no motion constraints, PCS would return two path pairs in P with path costs $\langle 2, 3\rangle$ and $\langle 3, 2\rangle$. These points are shown as B and C in Figure 5(b). The resulting pairwise cost frontiers, $lb_{y,z}$=5 and $lb'_{y,z}$=6 are also shown.

Constraint Sets for Splitting CBICS generates child nodes based on outputs from PCS. Recall from the discussion of CBS that, during a split, agents i and j receive a set of new motion constraints M_i and M_j, respectively. This helps the agents to avoid a conflict. Completeness is ensured only when the actions blocked by M_i are mutually conflicting with all actions blocked by M_j (Walker, Sturtevant, and Felner 2020). This is known as the mutually disjunctive property (Li et al. 2019c) for constraint sets. We say that motion constraint sets are *valid* iff no solution exists when both agents i and j violate any constraints $m_i\in M_i$ and $m_j\in M_j$, respectively, simultaneously.

There are two possibilities for PCS outputs M'_i, M'_j: (1) motion constraints are found for at least one of the two agents or (2) no motion constraints are found. In case (1) CBICS performs a *conjunctive split*. In case (2), CBICS performs a (CBS-style) *disjunctive split*.

Conjunctive Splitting In a conjunctive split, the same motion constraint sets are applied to all child nodes. See lines 19-38 of Algorithm 1. A child node N' is created for each path pair π'_i, π'_j in P, where $N'.r_i$ and $N'r_j$ are assigned the costs $c(\pi'_i)$ and $c(\pi'_j)$, respectively, the pairwise lower bound, $N'.lb_{i,j}$ gets $c(\pi'_i)+c(\pi'_j)$, $N'.\Pi$ is updated with π'_i and π'_j and $N'.M$ is updated with M'_i and M'_j (lines 19-30).

For example, node A of Figure 4 is split based on the conflict between agents y and z, producing two child nodes because PCS returned two paths in P with costs $\langle 2, 3\rangle$ and $\langle 3, 2\rangle$. Child nodes B and C take on *fixed* values for r_y and r_z ($\langle 2, 3\rangle$ and $\langle 3, 2\rangle$, respectively). Child B and C also get $lb_{y,z}$=5, and the same sets of constraints are added to both. CBICS also replaces $B.\pi_y$, $B.\pi_z$, $C.\pi_y$ and $C.\pi_z$ with the respective paths from P.

CBICS generates an additional node whose cost constraints represent the cost range based on the next pairwise cost frontier, $lb'_{i,j}$ (lines 31-38). Note that paths for this node

are not filled in, but are lazily planned on line 8. The creation of the additional node is illustrated by node D, which is generated based on $lb'_{y,z}$=6, and also receives the same motion constraint sets as nodes B and C. The cost range for node D in Figure 4 is shown as the shaded region in Figure 5(b).

Disjunctive Splitting Lines 39-54 show the steps for a disjunctive split. In a disjunctive split, only two nodes are created, one for each agent in conflict. Each node then gets motion constraints for one agent respectively. The cost constraints are based on the minimum costs of path pairs in P.

Cost-Conditional Motion Constraints In order to ensure completeness, motion constraints used in a conjunctive split must be *cost-conditional* motion constraints. Conditional motion constraints (Walker, Sturtevant, and Felner 2020) are similar to regular constraints, except that they can be turned off, meaning that they are omitted from the low-level search based on some criteria. The need for cost-conditional motion constraints in conjunctive splitting scenarios can be understood using the example in Figures 3(a) and (b). In Figure 3(a), CBS constructs motion constraints for the mutexed MDD nodes shown with dashed outlines. Figure 3(b) shows the same analysis if the cost limit is increased to 3 for agent x. The MDD node B2 at time step 1 for agent z is no longer mutex with all MDD nodes for agent x, hence the motion constraints for agent x remain valid, but the motion constraints for agent z are no longer valid. This can be seen by comparing the nodes with dashed outlines between (a) and (b).

We therefore create motion constraints that are conditional on the cost of the other agents. The analysis in Figure 3(b) leads us to define a *cost-conditional* motion constraint $\langle n, a, \textit{ref}, c\rangle$, where n is the agent being constrained, a is the action to be avoided, *ref* is the *reference agent* for the constraint, and c is the upper cost bound of the reference agent. Hence, the motion constraint is only valid when the cost of the reference agent is no greater than c. Referring back to Figure 3(b), a cost-conditional motion constraint $\langle n{=}x, a{=}B1@0{\rightarrow}B2@1, \textit{ref}{=}z, c{=}2\rangle$ would be returned by PCS.

The CRCT is searched in a best-first fashion using the sum-of-costs of nodes as priorities. Some nodes in the CRCT which are generated before node E are pruned quickly because the combination of costs is infeasible. These pruned nodes are shown with red 'X's in Figure 4. For example, the left child of node C is pruned because setting r_x=3 and r_y=3 makes conflict-free paths for agents x and y impossible.

The search continues in this fashion, finding the current pairwise cost frontier, creating child nodes for fixed costs on the frontier and one child node representing pairwise costs at and above the next lowest pairwise cost frontier.

Prior to calling PCS, CBICS checks each $m_i{\in}M_i$ and $m_j{\in}M_j$ against the upper bound of each $r{\in}R$. All m_i with $m_i.ref{=}j$ and $m_i.c{<}r_j.ub$ are removed from the constraint set (i.e., turned off). Then, the search is started. For example, the analysis in Figure 3(a) creates four conditional motion constraints based on the actions terminating at dashed nodes.

1. $\langle n{=}x, a{=}B1{\rightarrow}B2@0, ref{=}z, c{=}2\rangle$
2. $\langle n{=}x, a{=}B2{\rightarrow}B3@1, ref{=}z, c{=}2\rangle$
3. $\langle n{=}z, a{=}C2{\rightarrow}B2@0, ref{=}x, c{=}2\rangle$
4. $\langle n{=}z, a{=}B2{\rightarrow}B3@1, ref{=}x, c{=}2\rangle$.

If PCS were called with parameters r_x=2 and r_z=3, constraints (1) and (2) would be omitted because r_z=3 is greater than c=2 for both of them.

The upper cost bound of a conditional motion constraint $m_i.c$ is valid iff it is mutually disjunctive with all of the opposing agent (agent j)'s actions at the same time when agent j's path cost is less than or equal to $m_i.c$. Hence, we say cost-conditional motion constraint sets are *valid* iff no solution exists when both agents i and j violate any (non-turned off) motion constraint from M_i and M_j simultaneously. By *non-turned off*, we mean motion constraints m_i for which $m_i.c{\leq}r_j.ub$ where $r_j.ub$ is the upper bound of the cost range for agent j (and analogously for m_j).

In node E in Figure 4, because of the motion constraints and the cost constraints for agent y from node B, only one feasible solution for agents x and y exists with costs $\langle 4, 2\rangle$. This results in pushing the individual cost limit for agent x from 2+ (in node B) to 4 (in node E) and $lb_{x,y}$ from 4+ (in node B) to 6 (in node E). Because of the cost increase for agent x, we can infer that $lb_{x,z}$=7 (because r_x=4 and r_z=3).

In this instance, the goal node E is found sooner than would have been the case with either CBS or ICTS for two reasons: (1) CBICS can apply motion constraints to multiple agents at the same time. Hence, both agents y and z get motion constraints in nodes B, C and D where CBS would have applied them only to one agent. (2) CBICS can increase the path costs of multiple agents at the same time, by the maximum amount possible. ICTS would have increased the cost of only one agent and only by the fixed amount δ.

Pairwise Constraint Search

PCS plans two conflicting agents jointly to find feasible solutions and discover motion constraints and cost constraints at the same time. The pseudocode for PCS shown in Algorithm 2 is based on A*, where the state space is the joint state space for the two agents. A state in the joint state space is $S{=}\{s_i, s_j\}$, where s_i and s_j are single-agent states. It respects motion constraints M_i and M_j for the agents during successor generation (line 13). It respects cost constraints by pruning successors (line 60). PCS terminates after all lowest-cost solutions have been found and the first next-lowest cost solution is found (line 7) or when OPEN is empty.

During the successor generation phase (lines 12-18), successors are generated to maintain time overlap between the actions of the agents. *Time overlap* means that for $a_i{=}\langle s_i, s'_i\rangle$ and $a_j{=}\langle s_j, s'_j\rangle$, $s_i.t{\in}[s_j.t, s'_j.t]$ or $s'_i.t{\in}[s_j.t, s'_j.t]$. This is only needed for continuous time (Walker, Sturtevant, and Felner 2018). Successor nodes are marked as infeasible if their parent is infeasible. This is similar to how MP checks the mutexes of predecessor

Algorithm 2 Pairwise Constraint Search Algorithm

1: Input: $start_i, start_j, goal_i, goal_j, M_i, M_j, r_i, r_j, lb_{i,j}$
2: $OPEN \leftarrow \langle start_i, start_j \rangle$
3: $M'_i \leftarrow \emptyset, M'_j \leftarrow \emptyset, P \leftarrow \emptyset, B_i \leftarrow \emptyset, B_j \leftarrow \emptyset$
4: **while** $OPEN \neq \emptyset$ **do**
5: $S = \langle s_i, s_j \rangle \leftarrow OPEN.pop()$
6: **if** $S'.feasible \wedge S'$ is goal $\wedge$
7: $P \neq \emptyset \ \wedge \ f(s'_i)+f(s'_j) >$ all path costs in P **then**
8: $M'_i \leftarrow M'_i \setminus \{\forall m'_i \in M'_i : m'_i.c < r_j.lb \vee m'_i.a \notin B_i\}$
9: $M'_j \leftarrow M'_j \setminus \{\forall m'_j \in M'_j : m'_j.c < r_i.lb \vee m'_j.a \notin B_j\}$
10: **return** $P, M'_i, M'_j, lb'_{i,j}=f(s'_i)+f(s'_j)$
11: **end if**
12: **if** $s_i.time < s_j.time$ **then** ▷ Get successors for continuous time
13: $S'_i \leftarrow successors(s_i, M_i)$
14: $S'_j \leftarrow \{s_j\}$
15: **else**
16: $S'_j \leftarrow successors(s_j, M_j)$
17: $S'_i \leftarrow \{s_i\}$
18: **end if**
19: Compute f-cost for each $s'_i \in S'_i, s'_j \in S'_j$
20: $\mathbf{S'} \leftarrow S'_i \times S'_j$ ▷ Cartesian product
21: **for** $S' \in \mathbf{S'}$ **do**
22: $S'.feasible \leftarrow S.feasible$
23: **if** $s'_i \in S'$ conflicts with $s'_j \in S'$ **then**
24: $S'.feasible \leftarrow false$
25: **end if**
26: **if** $\exists m'_i \in M'_i$ blocks $\langle s_i, s'_i \rangle$ **then**
27: $m'_i \leftarrow m'_i \in M'_i$
28: $M'_i \leftarrow M'_i \setminus m'_i$
29: **else**
30: $m'_i \leftarrow \langle n{=}i, a{=}(s_i, s'_i), ref{=}j, c{=}f(s'_j) \rangle$
31: **end if**
32: **if** $S'.feasible = false$ **then**
33: **if** not $m'_i.costIsCapped$ **then**
34: $m'_i.c \leftarrow \text{MAX}(f(s'_j), m'_i.c)$ ▷ Increase range
35: **end if**
36: **else**
37: **if** $f(s'_j) \leq m'_i.c$ **then**
38: $m'_i.c \leftarrow f(s'_j) - \epsilon$ ▷ Decrease range permanently
39: $m'_i.costIsCapped \leftarrow true$
40: **end if**
41: **end if**
42: $M'_i \leftarrow M'_i \cup m'_i$
43: Analogously for s'_j
44: **if** S' is goal $\wedge f(s'_i) \leq r_i.ub \wedge f(s'_j) \leq r_j.ub$ **then**
45: **if** $S'.feasible \wedge \langle f(s'_i), f(s'_j) \rangle$ is unique in $P \wedge$
46: $f(s'_i) \geq r_i.lb \wedge f(s'_j) \geq r_j.lb \wedge f(s'_i)+f(s'_j) \geq lb_{i,j}$ **then**
47: $P \leftarrow P \cup$ path to S' ▷ Save unique-cost solutions
48: **else if** $\neg S'.feasible$ **then**
49: **while** $\neg S.feasible$ **do** ▷ Add to infeasible action sets
50: $B_i \leftarrow \langle S.s_i, s'_i \rangle$
51: $B_j \leftarrow \langle S.s_j, s'_j \rangle$
52: $s'_i \leftarrow S.s_i$
53: $s'_j \leftarrow S.s_j$
54: $S \leftarrow S.parent$
55: **end while**
56: **end if**
57: **if** $f(s'_i) < r_i.ub \wedge f(s'_j) < r_j.ub$ **then**
58: $OPEN \leftarrow OPEN \cup S'$
59: **end if**
60: **else if** $f(s'_i) \leq r_i.ub \wedge f(s'_j) \leq r_j.ub$ **then**
61: $OPEN \leftarrow OPEN \cup S'$ ▷ Add to open if in cost bounds
62: **end if**
63: **end for**
64: **end while**
65: **return** $\emptyset, \emptyset, \emptyset, \infty$ ▷ No solution

MDD nodes for computing *propagated mutexes*. PCS performs mutex propagation, but it does this without the use of MDDs. Instead, it keeps a list of motion constraints, updating the upper cost bound $m'.c$ as necessary.

Consider the example instance in Figure 2(a). If PCS were planning for agents x and y, the root state would be $S{=}\langle s_x{=}\text{B1@0}, s_y{=}\text{B3@0}, f{=}2, 2, feasible{=}true \rangle$, where s_x and s_y are the states of agents x and y and f is $f(s_x)$ and $f(s_y)$ respectively. $f(s_i)$ is the *f-cost* of s which is a lower-bounded estimate of the cost of a path from $start_i$ to $goal_i$ that passes through s_i (line 19). The successors of this joint state as produced by lines 12-18 would be:

1. $\langle s_x{=}\text{C1@1}, s_y{=}\text{B3@1}, f{=}4, 3, feasible{=}true \rangle$
2. $\langle s_x{=}\text{C1@1}, s_y{=}\text{B2@1}, f{=}4, 2, feasible{=}true \rangle$
3. $\langle s_x{=}\text{B1@1}, s_y{=}\text{B3@1}, f{=}3, 3, feasible{=}true \rangle$
4. $\langle s_x{=}\text{B1@1}, s_y{=}\text{B2@1}, f{=}3, 2, feasible{=}true \rangle$
5. $\langle s_x{=}\text{B2@1}, s_y{=}\text{B3@1}, f{=}2, 3, feasible{=}true \rangle$
6. $\langle s_x{=}\text{B2@1}, s_y{=}\text{B2@1}, f{=}2, 2, feasible{=}false \rangle$

Lines 26-31 will get an existing motion constraint (for updating) or add a new motion constraint (as an *initial mutex*) with an initial upper cost bound set to the f-cost of the opposing agent (line 30).

Continuing the example, 5 new cost-conditional motion constraints will be created from the successor nodes, whose upper cost bound $m'.c$ will be updated.

1. $\langle n{=}x, a{=}\text{B1@0}{\rightarrow}\text{B1@1}, ref{=}y, c{=}3 \rangle$
2. $\langle n{=}x, a{=}\text{B1@0}{\rightarrow}\text{B2@1}, ref{=}y, c{=}3 \rangle$
3. $\langle n{=}x, a{=}\text{B1@0}{\rightarrow}\text{C1@1}, ref{=}y, c{=}3 \rangle$
4. $\langle n{=}y, a{=}\text{B3@0}{\rightarrow}\text{B3@1}, ref{=}x, c{=}4 \rangle$
5. $\langle n{=}y, a{=}\text{B3@0}{\rightarrow}\text{B2@1}, ref{=}x, c{=}4 \rangle$

The logic for updating $m'.c$ occurs at lines 32-41. Consider what happens for motion constraint 1, listed above. Because agent x's action B1@0→B1@1 does not conflict with either of agent y's actions B3@0→B3@1 and B3@0→B2@1, the cost is capped so that it cannot grow (see line 39) and its upper cost limit, $m'.c$ is reduced, (after comparing to both states) to $2{-}\epsilon$ where 2 is the f-cost of agent y's B3@0→B2@1 action, and ϵ is a small constant. Now consider what happens for motion constraint 2 which is for agent x's action B1@0→B2@1. It does not conflict with agent y's action B3@0→B3@1, but it does conflict with B3@0→B2@1. Thus, the cost gets capped at $3{-}\epsilon$, (for the f-cost of B3@0→B3@1), but does not get decreased when checked versus B3@0→B2@1. If the actions had been checked in the reverse order, the result would be the same. By inspection, it is apparent that as long as agent y's cost does not go above $3{-}\epsilon$, agent x can never perform the action B1@0→B2@1 without conflict.

Subsequent expansions will either increase or decrease $m'.c$ values appropriately so that each $m'.c$ is the top of the continuous conflicting path cost range for the opposing agent. Note that the resulting $m'.c$ for some cost-conditional motion constraints will be less than the lowest possible path

cost for the opposing agent, hence are unusable and must be omitted entirely (line 8. For example, $m'.c$ for motion constraint 1 is $2-\epsilon$ which is below the cost of a shortest path for agent y, and can be omitted from M_x.

In cases where heuristics that inform f-costs are not exact (without the additional logic described here), PCS could include motion constraints in M_i' and M_j' which block actions for paths with costs that are outside the cost bounds of r_i and r_j respectively. This would lead to incompleteness. In order to remove motion constraints which fall outside the cost bounds, PCS computes the sets of infeasible actions (literally $feasible{=}false$) B_i and B_j for agents i and j at lines 3 and 48-55. These sets include only infeasible actions included in infeasible paths to the goal that fall inside the cost bounds. Finally, in addition to removing all m' where $m'.c$ is less than the lower cost bounds, we also remove any m' where $m'.a$ is not in the set of infeasible actions B_i and B_j respectively (lines 8 and 9).

Completeness and Optimality

We believe CBICS to be optimal and complete, which is supported by our experimental results, where it has always returned optimal solutions. We are finalizing a complete and formal proof of correctness that will be presented in future work.

Additional Enhancements

Some additional enhancements make CBICS more efficient. Instead of letting it dynamically find the pairwise costs for conflicting agents, runtime is reduced if all pairwise costs are updated in the root node (line 3 of Algorithm 1). This can be done by performing a conflict check between all initial paths. Any pairs of paths that have a conflict are then planned together with PCS. The resulting pairwise costs are then updated in the root node. This is called the *preprocessing* enhancement.

In some scenarios, $|P|$, the number of unique-cost path combinations found by PCS can be large, which for conjunctive splits, results in a large branching factor at the high level. In order to keep the branching factor small, some of the cost combinations can be combined into a single node. Of all approaches we tried, the best approach is to create four nodes from P: (1) a node with the minimum cost for r_i and maximum cost for r_j. (2) a node with the maximum cost for r_i and minimum cost for r_j. (3) a node that combines all costs in between (but not equal to) the minimum and maximum costs. (4) the node for where the sum of costs is at least $lb'_{i,j}$. For example, with the following cost combinations: $\langle 2,5\rangle$, $\langle 3,4\rangle$, $\langle 4,3\rangle$, $\langle 5,2\rangle$, four nodes with the following path cost ranges would be generated: $\langle r_i{=}2, r_j{=}5\rangle$, $\langle r_i{=}[3,4], r_j{=}[3,4]\rangle$, $\langle r_i{=}5, r_j{=}2\rangle$, $\langle r_i{=}2{+}, r_j{=}2{+}\rangle$. This is called the *combination* enhancement.

Analysis

Figure 6 shows an example of a typical CBICS tree. Orange nodes represent conjunctive nodes with an unlimited upper cost bound (unlimited nodes), red nodes represent conjunctive nodes with finite cost bounds (limited nodes), and blue nodes represent disjunctive nodes. The search usually explores the sub-trees of limited nodes to completion - eliminating the sub-trees from consideration before moving on to expanding unlimited nodes. In practice, conjunctive splits occur most often at unlimited nodes. This is because lower bounds on individual costs are rarely increased for unlimited nodes and, when the costs are low, PCS (and mutex propagation in general) is more likely to find motion constraints. Conversely, as costs increase, PCS is less likely to find motion constraints and disjunctive splitting is used. Often, the pattern results in the sub-trees of limited nodes resembling a regular CBS tree. However, occasionally a sufficient number of disjunctive motion constraints are added in a sub-tree to allow PCS to find motion constraints, triggering a conjunctive split.

When compared to CBS, which has a branching factor of 2, CBICS has a larger branching factor on average. However, unlike CBS, nodes in CBICS have cost constraints which tend to eliminate a significant proportion of the sub-trees quickly. Thus, while CBS and ICTS tend to build fuller binary and k-ary-like trees, respectively, CBICS tends to build a more unbalanced search tree.

The branching factor of the joint state space for PCS is $b{=}(b_{base})^2$, where b_{base} is the single-agent branching factor. The depth of the solution Δ, can be no larger than $\text{MAX}(d_i, d_j)$ where d_i and d_j are the lengths (number of states) in π_i and π_j in the unit time case. Δ can be no larger than $d_i{+}d_j$ in the continuous time case because of a phenomenon related to operator decomposition (Standley 2010). Hence, the computational complexity of PCS is no worse than for regular mutex propagation, namely $O(b^d)$ in unit time domains, and $O(b^{2d})$ for continuous time domains.

Empirical Results

All experiments were run using an Intel i9 processor at 2.4GHz with 64GB of memory. The implementation is publicly available[1]. The experiments come from the MAPF benchmarks set (Stern et al. 2019) which includes various grid maps and MAPF instances with randomly selected start

[1]https://github.com/thaynewalker/hog2/tree/id/apps/CBICS

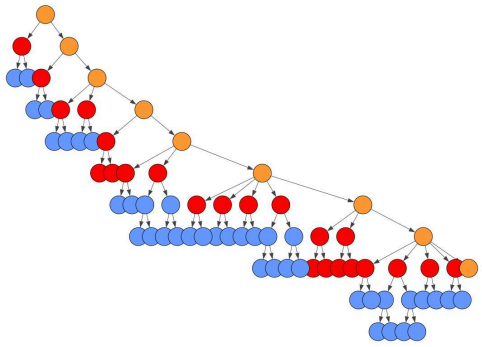

Figure 6: Structure of a CBICS tree.

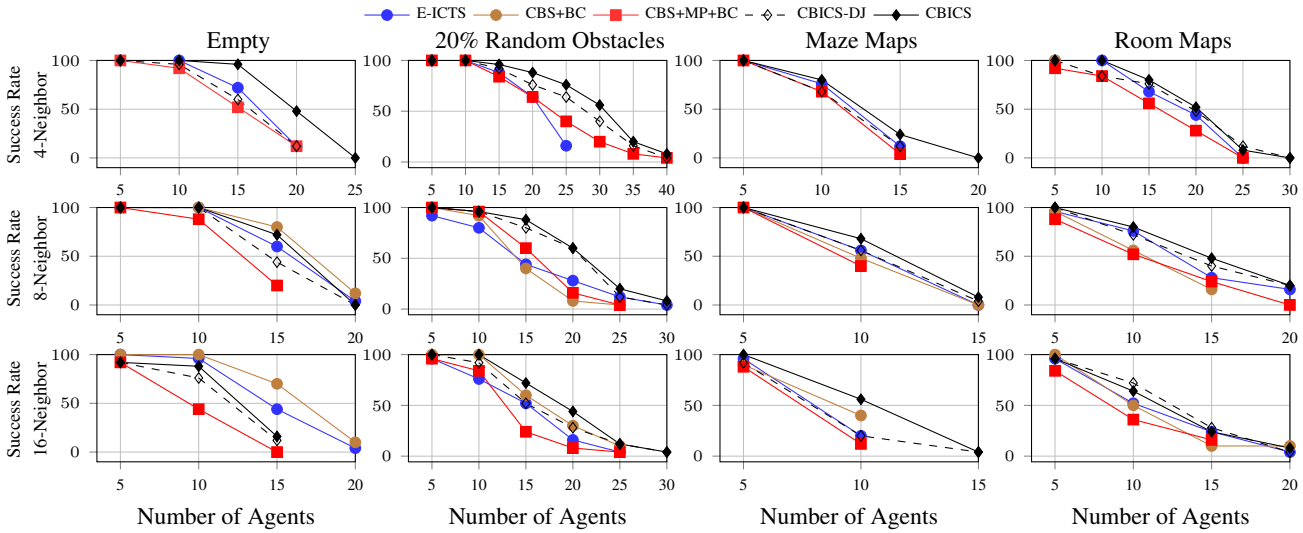

Figure 7: Success rates of E-ICTS, CBS+BC, CBS+MP+BC and two variants of CBICS on benchmark problems.

Map	E-ICTS		CBS+BC		CBS+MP+BC		CBICS	
	4	16	4	16	4	16	4	16
Empty	0.9	1.8	-	**1.3**	1.0	6.0	**0.1**	2.5
Random Obs.	0.02	7.2	-	5.4	0.06	5.5	**0.01**	**0.8**
Maze	9.2	25.3	-	25.5	10.4	26.5	**8.2**	**18.3**
Room	1.4	15.6	-	20.0	4.9	21.7	**0.5**	**12.0**

Table 1: Overall runtime for benchmarks with 10 agents.

and goal vertices for agents. There are 25 instances for each map. The benchmarks were run by starting with five agents, incrementally increasing the number of agents and recording the runtime and number of instances that were solvable within 30 seconds. Any MAPF instance that was not solved within this runtime limit was marked as a failure. Figure 7 shows the success rates for four different benchmark instances, namely empty 8x8 grid maps, 32x32 grid maps with 20% random blocked cells, 32x32 maze maps with corridors that are two cells wide, and 32x32 grid maps with rooms with doorways that are one cell wide.

All MAPF instances were solved for circular agents with a radius of $1/(2\sqrt{2})$ cells, a fixed wait time of 1, and the sum-of-costs time-not-on-target objective, meaning an agent never incurs cost when it is on its goal. Even if it moves away and back to its goal location, prior actions staying on its goal location have no cost. We have run experiments on 4-, 8- and 16-neighbor grid maps which are 2^k neighborhoods (Rivera, Hernández, and Baier 2017). 8- and 16-neighborhoods have continuous time. All experiments were run on an implementation which has collision detection, the conflict avoidance table and other aspects optimized for continuous time. This causes differences in our results versus MAPF solvers which are optimized for unit time.

Figure 7 shows the average success rates for various MAPF algorithms over 25 instances with increasing numbers of agents. *CBS+BC* means CBS with biclique motion constraints (Walker, Sturtevant, and Felner 2020) (for non-4-neighbor MAPF instances only since biclique motion constraints are not valid for unit time), *CBS+MP+BC* means CBS with (continuous time) mutex propagation and biclique motion constraints (for continuous time instances only), *CBICS-DJ* means CBICS with only disjunctive splitting, and *CBICS* is the full algorithm with both the preprocessing and combination enhancements. Both implementations of CBICS use biclique motion constraints in disjunctive splits when applicable. The latter three algorithms need to use PCS for continuous-time mutex propagation.

With the exception of empty maps in continuous time, the results show that CBICS is able to find solutions faster, and with a higher success rate than all other variants. CBICS solves for up to 5 or more agents in the same amount of time compared to previous state of the art. This is significant because each additional agent increases the difficulty of solving the problem instance exponentially. In 8- and 16-neighbor empty 8x8 grid maps, CBS+BC has a larger success rate than CBICS because the work for extended-time symmetry breaking of mutex propagation in PCS is not beneficial. Biclique motion constraints are formulated for spatial symmetries and so perform better on empty maps. In most of the other maps, CBICS and CBICS-DJ outperform the other variants, with CBICS consistently having the best performance. The results in Table 1 which shows the average runtime for the same MAPF instances with 10 agents, show the same trend, with CBICS having the smallest runtimes.

Table 2 shows the average number of high-level node expansions for the benchmarks with 10 agents. When an algorithm was not able to solve an instance, the number of node expansions before the timeout was used instead. All algorithms expanded more nodes for the empty-8x8 map than for the other maps because agent density is high and thus many conflicts need to be resolved and many cost increases are needed to find a solution. Also, a smaller map means shorter

Map	E-ICTS		CBS+BC		CBS+MP+BC		CBICS	
	4	16	4	16	4	16	4	16
Empty	**219**	28369	-	**2048**	4221	41994	592	6647
Random	16	12722	-	938	5	114	**4**	**48**
Maze	103	**167**	-	774	291	589	**80**	428
Room	169	10132	-	798	84	465	**15**	**212**

Table 2: Nodes expanded for benchmarks with 10 agents.

path lengths and the low level of each algorithm thus does less work, enabling it to expand more nodes in less time.

Mutex propagation is an effective yet computationally expensive algorithm. Although CBS+MP+BC is using the same implementation of the low-level as CBICS, its average number of expansions is consistently larger than for CBICS because of its ability to propagate conjunctive motion constraints to multiple agents and pruning of infeasible cost combinations. Additionally, when running PCS with cost limits, a lot of work is reduced, allowing CBICS to evaluate nodes faster.

Table 3 shows the average runtime per high-level node expanded for the low-level of each algorithm in milliseconds. For instances that were unsolvable, the runtime of all expanded high-level nodes were used. E-ICTS generally has a very small runtime per node evaluated because it uses small, cost-exact pairwise checks and eliminates infeasible combinations before doing a final k-agent search. The majority of its low-level searches never perform the full k-agent search because the pairwise tests fail quickly. PCS evaluates the entire pairwise space, including the infeasible combinations. This causes longer low-level runtimes, but more information is gleaned in the process. The runtime per node of CBS+MP+BC is consistent across all 32x32 maps, but the low-level runtime per node of CBICS has more variance. This is because *limited* nodes and the nodes in their sub-trees have tighter cost constraints, and so can be evaluated faster. The *unlimited* nodes in CBICS, having unlimited upper cost bounds, are similar to CBS+MP+BC nodes and tend to take longer to evaluate.

Conclusion

In this work, we have solved the open issue of generalized symmetry breaking in continuous time domains by formulating Pairwise Constraint Search (PCS). PCS computes motion constraint sets for conflicting agents by performing mutex propagation and simultaneously computes cost validity ranges for the motion constraints.

We have also formulated Conflict-Based Increasing Cost Search (CBICS), which leverages strengths from the CBS and ICTS algorithms. CBICS dynamically employs both conjunctive and disjunctive splitting in the high-level search to simultaneously increase costs and resolve conflicts. CBICS works well for both unit time and continuous time domains and outperforms current state of the art algorithms for continuous time domains.

Map	E-ICTS		CBS+BC		CBS+MP+BC		CBICS	
	4	16	4	16	4	16	4	16
Empty	**0.6**	5.4	-	**3.0**	3.4	4.5	1.8	3.8
Random	**0.7**	29.7	-	28.3	9.9	48.2	4.0	**17.7**
Maze	89.4	151.5	-	**20.1**	**35.8**	48.8	38.7	42.7
Room	**0.6**	**1.5**	-	27.1	27.5	41.8	25.8	36.5

Table 3: Low-level runtime for benchmarks with 10 agents.

Acknowledgements

We are grateful for the significant contributions of Sven Koenig. The research at the University of Denver was supported by the National Science Foundation (NSF) grant number 1815660 and Lockheed Martin Corp. Research at the university of Alberta was funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Research at Ben Gurion University was supported by BSF grant number 2017692.

References

Andreychuk, A.; Yakovlev, K.; Atzmon, D.; and Stern, R. 2019. Multi-Agent Pathfinding with Continuous Time. In *International Joint Conferences on Artifical Intelligence*, 39–45.

Botea, A.; Bonusi, D.; and Surynek, P. 2018. Solving multi-agent path finding on strongly biconnected digraphs. *Journal of Artificial Intelligence Research* 273–314.

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 *International Joint Conferences on Artifical Intelligence*, 223–225.

Cohen, L.; Uras, T.; Kumar, T. K. S.; and Koenig, S. 2019. Optimal and Bounded Sub-Optimal Multi-Agent Motion Planning. In *Symposium on Combinatorial Search*, 44–51.

Gange, G.; Harabor, D.; and Stuckey, P. J. 2019. Lazy CBS: Implict conflict-based search using lazy clause generation. In *International Conference on Planning and Scheduling*, 155–162.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2019. Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19), International Joint Conferences on Artificial Intelligence Organization*, 1289–1296.

Li, J.; Gange, G.; Harabor, D.; Stuckey, P. J.; Ma, H.; Koenig, S.; Li, J.; Sun, K.; Ma, H.; Felner, A.; et al. 2020. New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding. In *International Conference on Automated Planning and Scheduling*, 6087–6095.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2019a. Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search. In *International Conference on Planning and Scheduling*, 279–283.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Sven, K. 2019b. Symmetry-Breaking Constraints for Grid-Based

Multi-Agent Pathfinding. In *AAAI Conference on Artificial Intelligence*, 6087–6095.

Li, J.; Surynek, P.; Felner, A.; Ma, H.; and Satish, K. T. 2019c. Multi-Agent Pathfinding for Large Agents. In *AAAI Conference on Artificial Intelligence*, 7627–7634.

Rivera, N.; Hernández, C.; and Baier, J. A. 2017. Grid Pathfinding on the 2k Neighborhoods. In *AAAI Conference on Artificial Intelligence*, 891–897.

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. *AIJ* 470–495.

Srinivasan, A.; Ham, T.; Malik, S.; and Brayton, R. K. 1990. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, 92–95. IEEE.

Standley, T. S. 2010. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *AAAI Conference on Artificial Intelligence*, 28–29.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Symposium on Combinatorial Search*, 151–159.

Surynek, P.; Felner, A.; Stern, R.; and Boyarski, E. 2016. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *ECAI/PAIS*, 810–818.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In *International Joint Conferences on Artifical Intelligence*, 534–540.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2020. Generalized and Sub-Optimal Bipartite Constraints for Conflict-Based Search. In *AAAI Conference on Artificial Intelligence*.

Weld, D. S. 1999. Recent advances in AI planning. *AI magazine* 20(2): 93–93.

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

Zhang, H.; Li, J.; Surynek, P.; Koenig, S.; and Kumar, T. K. S. 2020. Multi-Agent Pathfinding with Mutex Propagation. In *International Conference on Planning and Scheduling*.