Low-Level Search on Time Intervals in Branch-and-Cut-and-Price for Multi-Agent Path Finding

Edward Lam, Peter J. Stuckey

Monash University, Australia
edward.lam@monash.edu, peter.stuckey@monash.edu

Abstract

Multi-agent path finding is the problem of navigating a set of agents from their starting locations to their target locations while avoiding collisions. A leading method for optimal multi-agent path finding is branch-and-cut-and-price, a framework based on mathematical optimization. The reference implementation, named BCP-MAPF, shows highly competitive results against AI-based search. This paper presents BCP2-MAPF, a new implementation of branch-and-cut-and-price paired with a novel low-level path finder based on time intervals. Experimental results demonstrate that BCP2-MAPF significantly outperforms the other state-of-the-art optimal algorithms BCP-MAPF, Lazy CBS and CBSH2-RTC.

Introduction

Multi-agent path finding (MAPF) is an abstract computational problem in artificial intelligence and robotics. Given a map and a set of co-operative agents, each with a fixed start and target location, MAPF asks to find collision-free paths of minimum total cost that navigate every agent from its starting location to its target. The most common variants of MAPF prevent two types of collisions known as vertex conflicts and edge conflicts (Stern et al. 2019). A vertex conflict occurs when two agents occupy the same location at the same time. An edge conflict occurs when two agents swap locations at any given time. The basic MAPF problem asks to find conflict-free paths that minimize the sum of arrival times. Another common objective aims to minimize the makespan, defined as the time when the last agent arrives at its target.

Despite MAPF being NP-hard (Yu and LaValle 2013), recent advances have produced algorithms that can solve large instances that were intractable just several years ago. Conflict-based search (CBS), branch-and-cut-and-price (BCP) and lazy clause generation (LCG) are the leading approaches for optimal MAPF. CBS, BCP and LCG are all general tree search frameworks that can be customized to different problems, resulting in algorithms and implementations tailored to specific problem variants.

CBSH2-RTC (Li et al. 2021), currently the fastest version of CBS, proposed reasoning techniques for new types of conflicts. BCP-MAPF (Lam et al. 2022) is the reference implementation of BCP for MAPF and introduced many unfamiliar concepts from mathematical optimization to the MAPF community. Lazy CBS (Gange, Harabor, and Stuckey 2019), the reference implementation of LCG for MAPF, attempts to prevent every agent from increasing its path length by determining which vertex and edge conflicts cause the path length to change and then avoiding these vertices and edges.

This paper presents a new implementation of BCP, named BCP2-MAPF. The main novelties of BCP2-MAPF over BCP-MAPF are:

- A hybrid best-first depth-first search of the high-level branch-and-bound tree that balances tightening of the lower bound and upper bound.
- Constraints for reasoning about corridor conflicts borrowed from CBSH2-RTC and a mechanism for generating paths that respect these constraints.
- A low-level algorithm that breaks time symmetry by searching on a graph whose edges represent combined wait-then-move actions that jump over a time interval.
- A high-level tree search code that is specialized to MAPF and engineered from scratch for high performance, and hence removes the dependency on a generic mixed integer programming solver.

Computational experiments comparing CBSH2-RTC, Lazy CBS, BCP-MAPF and BCP2-MAPF demonstrate that BCP2-MAPF substantially outperforms the other solvers. It solves 1462 out of 2640 instances (55%) from the Moving AI benchmarks (Sturtevant 2012). In comparison, BCP-MAPF, Lazy CBS and CBSH2-RTC respectively solve 977 (37%), 957 (36%) and 608 (23%) instances. On the subset of instances solved by all methods, BCP2-MAPF averages 3.3, 10.6 and 11.7 times faster than BCP-MAPF, Lazy CBS and CBSH2-RTC respectively.

Problem Definition

Define a finite grid map with locations $\mathcal{L} = \{0, \dots, W-1\} \times \{0, \dots, H-1\}$ where $W \in \mathbb{Z}_+$ is the width and $H \in \mathbb{Z}_+$ is the height of the map. Some locations are classified as *obstacles*. Agents cannot enter locations marked as obstacles. For every location $l = (x_1, y_1) \in \mathcal{L}$, define its *neighbors* $\mathcal{N}(l) = \{(x_2, y_2) \in \mathcal{L} : |x_2 - x_1| + |y_2 - y_1| \leq 1\}$ as itself and the four locations in the north-south and east-west orthogonal directions. This paper considers the 4-connected version of the standard MAPF problem but all contributions are applicable to other graph layouts.

Let $\mathcal{T} = \mathbb{Z}_+ = \{0, 1, \ldots, \infty\}$ be the infinite set of time steps. The problem is defined on a time-expanded graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{L} \times \mathcal{T}$ is the set of vertices and $\mathcal{E} = \{((l_1, t), (l_2, t+1)) \in \mathcal{V} \times \mathcal{V} : l_2 \in \mathcal{N}(l_1)\}$ is the set of edges. A vertex $v \in \mathcal{V}$ is a location-time step pair. An edge $e \in \mathcal{E}$ is a pair of vertices representing moving from one location to a different location (a *move* action) or a staying at same location (a *wait* action). Define the *reverse* $e' = ((l_2, t), (l_1, t+1))$ of an edge $e = ((l_1, t), (l_2, t+1))$ as the edge going in the opposite direction.

Let $\mathcal{A}$ be the set of agents. Each agent $a \in \mathcal{A}$ has a fixed *start location* $s_a \in \mathcal{L}$ and *goal location* $g_a \in \mathcal{L}$. The goal location of each agent can overlap with its start location but all start locations are unique and all goal locations are unique.

A *path* p for agent $a \in \mathcal{A}$ of length $k \in \mathcal{T}$ is a sequence of k locations $(l_0, l_1, l_2, \ldots, l_{k-1})$ such that $l_0 = s_a$, $l_{k-1} = g_a$ and $((l_t, t), (l_{t+1}, t+1)) \in \mathcal{E}$ for all $t \in \{0, \ldots, k-2\}$. Path p *visits* the vertices (l_t, t) where $t \in \{0, \ldots, k-1\}$ and (g_a, t) for all $t \in \{k, k+1, \ldots, \infty\}$ because the agent remains at its goal location indefinitely after the path ends. Path p *traverses* the edges $((l_t, t), (l_{t+1}, t+1))$ where $t \in \{0, \ldots, k-2\}$ and the edges $((g_a, t), (g_a, t+1))$ where $t \in \{k-1, k, \ldots, \infty\}$. The *cost* $c_p = k - 1$ of path p is equal to the number of edges, i.e., the total number of move and wait actions taken to reach the goal (and wait there indefinitely).

A *feasible* solution is a set of paths, one for each agent $a \in \mathcal{A}$, such that each vertex is visited at most once (i.e., absence of *vertex conflicts*), and each edge and its reverse are traversed at most once (absence of *edge conflicts*). An *optimal* solution is a feasible solution that minimizes the sum of all path costs.

Related Work

Attention to MAPF has exploded in recent years as researchers and industry realize its applications in artificial intelligence and robotics. MAPF has historically been tackled through various traditional techniques, such as joint space search (Standley 2010), which have proven difficult to scale. At present, CBS, BCP and LCG are the leading approaches for optimal MAPF.

CBS made large breakthroughs in scalability by decomposing the problem into a two-level tree search (Sharon et al. 2015). On the low level, CBS solves a single-agent problem by planning the path of each agent individually. On the high level, CBS assembles the paths together for an overall plan and resolves conflicts by adding constraints to children nodes that prevent vertex and edge conflicts. Since then, CBS has received many extensions, including advanced heuristics that guide the search and prune redundant search nodes (e.g., Felner et al. 2018; Li et al. 2019). CBSH2-RTC is the most recent and fastest instantiation (Li et al. 2021). It adds sophisticated reasoning techniques for rectangle, corridor and target conflicts.

BCP is also a two-level tree search framework but its main premise is to compute a strong lower bound at each node by progressively tightening a continuous relaxation (e.g., Lübbecke and Desrosiers 2005; Desaulniers, Desrosiers, and Spoorendonk 2011; Desrosiers et al. 2024). Its instantiation for MAPF, named BCP-MAPF (Lam et al. 2022), finds paths for agents independently, like CBS, but adds these paths into a database for recall at any node in the search tree. It also features constraints for resolving numerous kinds of conflicts.

LCG is a search technique for solving constraint programs coupled with a conflict analysis procedure borrowed from propositional satisfiability (Ohrimenko, Stuckey, and Codish 2009). Whenever LCG detects a conflict between different substructures of a problem, it calculates a cause of this conflict and adds a constraint that refutes the cause, preventing the same conflict from occurring again in the future. Lazy CBS is the reference implementation of LCG for MAPF (Gange, Harabor, and Stuckey 2019). It introduces a method for determining which vertex and edge conflicts cause to an agent to increase its path length. It then prevents other agents from accessing these vertices and edges, thereby allowing the original agent to use these vertices and edges, and hence avoiding an increase to its path length.

CBS, Lazy CBS and BCP-MAPF all rely on an underlying path finder for their low-level problem. Presently, these path finders are based on a variant of time-expanded A* that searches on a graph similar to $\mathcal{G}$. A major drawback of time-expanded graphs is that they exhibit time symmetry.

Safe interval path planning (SIPP) is one approach to breaking this time symmetry (Phillips and Likhachev 2011). Provided that dynamic obstacles occupy locations during known time intervals, SIPP searches on a graph whose edges represent combined wait-then-move actions that jump through multiple time steps to a time when an obstacle has passed. These actions impose an ordering, thereby breaking symmetries arising from combinations of waiting and moving to avoid a dynamic obstacle in time-expanded A*. Empirical results demonstrate that SIPP significantly reduces computational time compared to time-expanded A*. BCP2-MAPF introduces a low-level path finder based on similar ideas but generalizes the wait-then-move actions from uniform cost to arbitrary non-negative costs required by the BCP framework.

Background

BCP is a technique developed for mathematical optimization of large-scale problems (e.g., Lübbecke and Desrosiers 2005; Desaulniers, Desrosiers, and Spoorendonk 2011; Desrosiers et al. 2024). It consists of a high-level branch-and-bound tree search in which every node iteratively builds a continuous relaxation, called the master problem, to obtain an increasingly tighter lower bound. The master problem usually assembles individual plans or patterns that make up a solution to the original problem and contains constraints that prevent certain combinations of patterns from appearing in the same solution. Every pattern is associated with a variable whose value represents the proportion of selecting the pattern.

The BCP technique was first instantiated for MAPF in an implementation named BCP-MAPF (Lam et al. 2019, 2022). It includes constraints that prohibit vertex conflicts, edge conflicts and many other types of conflicts unique to the continuous relaxation. BCP-MAPF demonstrated results competitive against and often surpassing other state-of-the-art solvers to this day.

The Master Problem

At every node of the high-level tree, BCP2-MAPF solves a *master problem* that selects paths from a large database and resolves conflicts. BCP2-MAPF inherits the master problem from BCP-MAPF, shown in Problem (1). Let $\mathcal{P}_a$ be the set of paths for agent $a \in \mathcal{A}$. Every path $p \in \mathcal{P}_a$ is associated with a variable $\lambda_p \in [0, 1]$ that represents its usage proportion.

$$\min \sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{P}_a} c_p \lambda_p \tag{1a}$$

subject to

$$\sum_{p \in \mathcal{P}_a} \lambda_p = 1 \qquad \forall a \in \mathcal{A}, \tag{1b}$$

$$\lambda_p \geq 0 \qquad \forall a \in \mathcal{A}, p \in \mathcal{P}_a. \tag{1c}$$

Objective Function (1a) minimizes the total cost of the selected paths. Constraint (1b) ensures that the total proportion of all paths used by each agent sums to 1. Constraint (1c) are the non-negativity constraints, which disallow negative proportions of a path. Constraints (1b) and (1c) together ensure that $\lambda_p \in [0, 1]$. As currently stated, the master problem allows vertex and edge conflicts. These and other types of conflicts are dynamically resolved by adding constraints as required.

Resolving Conflicts

An edge conflict occurs at $e \in \mathcal{E}$ if e and its reverse e' are used simultaneously by the paths selected by the master problem, i.e.,

$$\sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{P}_a} \left(x_e^p + x_{e'}^p\right) \lambda_p > 1,$$

where $x_e^p \in \{0, 1\}$ counts the number of times that path p uses edge $e \in \mathcal{E}$. If this condition is met, an edge conflict constraint

$$\sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{P}_a} \left(x_e^p + x_{e'}^p\right) \lambda_p \leq 1 \tag{2}$$

is added to the master problem, which is then solved again to resolve the conflict.

Many types of conflicts, including edge (Constraint (2)) and vertex conflicts, can be resolved using a constraint r expressed in the general form

$$\sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{P}_a} \left(\sum_{e \in \mathcal{E}} \alpha_{r,e}^a x_e^p \right) \lambda_p \leq b_r, \tag{3}$$

where $b_r \geq 0$ and $\alpha_{r,e}^a \in \{0, 1\}$ determines if edge e is included for agent a according to the definition of the conflict. Constraint (3) can be interpreted as a bound on the total usage of the agent-edge pairs $\{(a, e) \in \mathcal{A} \times \mathcal{E} : \alpha_{r,e}^a = 1\}$. Constraints in the form of Constraint (3) are called *robust* (de Aragao and Uchoa 2003; Fukasawa et al. 2006). Let $\mathcal{R}_1$ be the set of all robust constraints.

A few classes of conflict constraints, such as target and corridor, cannot be decomposed into a conflict over a set of agent-edge pairs. These conflicts can be resolved using a constraint r in the form

$$\sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{P}_a} x_r^p \lambda_p \leq b_r. \tag{4}$$

Constraint (4) can be interpreted as a bound on the usage of a resource to at most $b_r \geq 0$, where path p consumes $x_r^p \geq 0$ of the resource. Constraints that can be expressed in the form of Constraint (4) but not Constraint (3) are called *non-robust* (de Aragao and Uchoa 2003; Fukasawa et al. 2006). Let $\mathcal{R}_2$ be the set of all non-robust constraints.

Generating Paths

The set $\mathcal{P}_a$ is dynamically enlarged by calling a low-level path finder. Instead of finding a path with minimum cost, the low-level solver needs to find a path with minimum *reduced cost*, which can be interpreted as a combination of its cost and how much it participates in the conflicts.

Every constraint $r \in \mathcal{R}_1 \cup \mathcal{R}_2$ is associated with a dual variable $\pi_r \leq 0$ whose value can be retrieved from the solver of the master problem, similar to the values of the λ_p variables. Let ρ_a be the dual variable of Constraint (1b). The reduced cost of a path p for agent a is then defined as

$$\bar{c}_p = c_p - \rho_a - \sum_{r \in \mathcal{R}_1} \pi_r \sum_{e \in \mathcal{E}} \alpha_{r,e}^a x_e^p - \sum_{r \in \mathcal{R}_2} \pi_r x_r^p. \tag{5}$$

The reduced cost is the largest possible improvement to the value of Objective Function (1a) should the path be selected (i.e., $\lambda_p = 1$). If a path with negative reduced cost is found, it can potentially improve the objective value, so it is added to $\mathcal{P}_a$ and the master problem is solved again. Finding paths with the most negative (i.e., minimum) reduced cost intuitively decreases the objective value in the fewest iterations, although this does not always hold in practice.

According to Equation (5), every robust constraint $r \in \mathcal{R}_1$ adds a penalty $-\pi_r \alpha_{r,e}^a \geq 0$ to the reduced cost of every edge e in order to discourage competition for the edge. For example, if the constraint r for an edge conflict at edge e_0 has a dual variable whose value is $\pi_r = -4.5$, then every new path p that contests the edge (i.e., $x_{e_0}^p = 1$) will incur a penalty of $-\pi_r \alpha_{r,e_0}^a x_{e_0}^p = -(-4.5) \times 1 \times 1 = 4.5$ in its reduced cost. As seen here, penalties due to robust constraints can be trivially accumulated when extending a partial path along an edge during the search.

In contrast, a non-robust constraint $r \in \mathcal{R}_2$ requires a new mechanism in the low-level solver to correctly penalize the path by $-\pi_r \geq 0$ whenever x_r^p is incremented. These mechanisms highly depend on the definition of each class of non-robust constraints.

By adjusting the values of all ρ_a and π_r and then asking the low-level solver to find paths with negative reduced cost, the master problem will converge to a fractionally-optimal solution. If any vertex or edge is used with fractional proportion at convergence, then two children nodes are created to fix these fractional values to 0 and 1, eliminating the fractional solution. Eventually, the tree search will yield an optimal conflict-free solution.

A Hybrid Best-First Depth-First Search

BCP2-MAPF runs a hybrid search strategy that balances the focus on improving both the lower bound and upper bound.

Solutions to combinatorial problems often appear deep in the search tree. Therefore, depth-first search is commonly applied to find solutions quickly regardless of solution quality. The main drawback of depth-first search is that early poor decisions can cause the search to get stuck in a suboptimal subtree, requiring an exponential amount of effort to close and proceed to a more-promising subtree near the root.

In contrast, best-first search attempts to find high-quality solutions. The global lower bound is the lowest lower bound of all open nodes on the search frontier. Best-first search focuses on the nodes with the smallest lower bound, which are often near the root, despite solutions rarely appearing here. Therefore, best-first search works well in conjunction with tight lower bounds but could otherwise fail for small time limits.

Mathematical optimisation solvers, compared to other techniques, typically spend more time on tightening the lower bound and therefore best-first search will generally find low-cost solutions quickly due to the stronger lower bounds. Of course, this observation does not apply to all problems, especially those that display many symmetries. For these problems, the symmetries explode the number of equivalent nodes, requiring significant effort to close.

BCP2-MAPF implements a hybrid search strategy that periodically performs a best-first node selection and then proceeds to a limited depth-first exploration of the subtree rooted at this node. This search strategy is implemented using a stack for depth-first search and a priority queue for best-first search. It takes a node off the stack if available otherwise it pops a node off the priority queue and moves it to the stack. When a node is generated, it is inserted into the priority queue if either of the following conditions hold:

- The number of nodes in this subtree exceeds a fixed parameter, indicating that the subtree is large and another subtree could be more promising.
- The lower bound of the new node is significantly higher than the lower bound of the subtree's root, indicating that another subtree could have smaller lower bounds. The following formula proved effective in determining if a node should be inserted into the priority queue:

$$\text{node lower bound} > \lceil \text{subtree root lower bound} \rceil + 2.$$

Otherwise, the node is inserted into the stack and depth-first search continues.

Corridor Conflicts

Corridor conflicts arise when two agents traverse a single-lane section in opposite directions (Li et al. 2021). BCP-MAPF introduced pseudo-corridor reasoning, which simply treats each edge as a short corridor (Lam et al. 2019). Pseudo-corridor conflicts were later generalized to full corridor reasoning and implemented in CBSH2-RTC (Li et al. 2021). Handling corridor conflicts using dedicated reasoning techniques proved effective in maps where they often appear, such as warehouses and mazes. This section describes how the same corridor conflicts can be implemented in BCP2-MAPF.

Define a *corridor* as a sequence of connected locations such that all internal locations have degree 2 and define its *length* m as the number of locations. The first and last locations are its *endpoints*.

Figure 1 shows two agents crossing in opposite directions a corridor of length 5 with endpoints l_1 and l_2. Without loss of generality, assume that agent a_1 has priority and a_2 concedes the corridor to a_1. Then, a_1 can reach l_1 at time 5 and move to l_3 at time 6. Agent a_2 can now enter l_1 at time 6 and proceed through the corridor to reach l_2 at time 10. Therefore, a corridor conflict occurs if both a_2 reaches l_2 before time 10 and a_1 reaches l_1 before time 10 by symmetry (Li et al. 2021). Additionally, if there is a detour that allows either agent to bypass the corridor and avoid the conflict altogether, the earliest arrival time at the farthest endpoint via this detour must also be considered (Li et al. 2021).

Define $h_a(l)$ as the earliest time that agent a can reach location l from its start location and $h'_a(l)$ as the earliest time that agent a can reach location l without using the corridor. Let the earliest possible arrival time of a_1 at l_1 for avoiding a corridor conflict be $\tau_{a_1,l_1,a_2,l_2} = \min(h_{a_2}(l_2)+m, h'_{a_1}(l_1))$, defined as the earlier of either conceding the corridor to a_2 or detouring around the corridor. A corridor conflict occurs in a corridor of length m if agent a_1 visits its farthest endpoint l_1 before time τ_{a_1,l_1,a_2,l_2} and agent a_2 visits its farthest endpoint l_2 before time τ_{a_2,l_2,a_1,l_1}. These two conditions are mutually exclusive and can be captured in the constraint

$$\sum_{p \in \mathcal{P}_{a_1}} y^p_{a_1,l_1,a_2,l_2} \lambda_p + \sum_{p \in \mathcal{P}_{a_2}} y^p_{a_2,l_2,a_1,l_1} \lambda_p \leq 1,$$

where $y^p_{a_1,l_1,a_2,l_2} = 1$ if path p of a_1 visits l_1 before time τ_{a_1,l_1,a_2,l_2}.

Note that $y^p_{a_1,l_1,a_2,l_2}$ does not count the number of times that a_1 visits l_1 before τ_{a_1,l_1,a_2,l_2}, but rather, it indicates if l_1 is visited at least once before τ_{a_1,l_1,a_2,l_2}. To correctly handle the penalty induced by the dual variable of a corridor conflict constraint, the low-level solver requires a mechanism to penalize a new path on the first visit to l_1 before time τ_{a_1,l_1,a_2,l_2}. This mechanism, called a *once-off penalty*, is described in the next section.

A Low-Level Search on Time Intervals

The low-level solver generates paths by exploring a graph. Bear in mind that this graph is only an abstract concept for

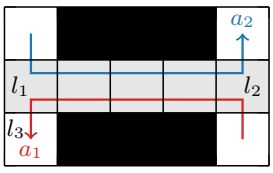

Figure 1: Two agents traversing a corridor of length 5 in opposite directions.

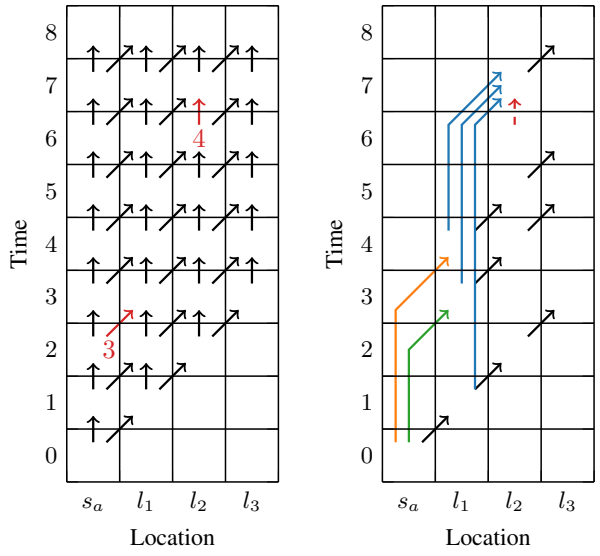

Figure 2: The time-expanded graph (left) includes all move and wait edges, regardless of whether they have zero (black) or non-zero (red) penalty. The time-interval graph (right) only contains edges with non-zero penalty (green), edges that bypass a penalty (orange and blue) and edges that move from the start vertex or from the destination of edges of the previous two types to the next time step (black).

defining the search problem and is never explicitly instantiated. BCP-MAPF defines the search problem on a time-expanded graph very similar to $\mathcal{G}$. This section describes how the time intervals from SIPP can be generalized from uniform-cost actions to arbitrary non-negative costs to define the low-level search problem in BCP2-MAPF.

The Time-Interval Graph

A *time interval* of length $k \in \mathcal{T}$ starting at $t \in \mathcal{T}$ is a consecutive sequence of time steps $(t, t+1, \ldots, t+k-1)$ that is associated with an origin location $i \in \mathcal{L}$, a destination location $j \in \mathcal{L}$, which may coincide with the origin, and a penalty c incurred when departing the origin at any time during the sequence of time steps to the destination.

Figure 2 compares the time-expanded graph and equivalent time-interval graph. To avoid a messy illustration, assume that the agent can only move from left to right. The vertices of the time-interval graph are the same but the vast majority are unreachable and hence can be discarded. In the time-expanded graph (left), the edges indiscriminately exist at every time step until infinity (or an arbitrarily-imposed finite time horizon). It is oblivious to the two penalties shown in red: the move edge $((s_a, 2), (l_1, 3))$ with penalty 3 and the wait edge $((l_2, 6), (l_2, 7))$ with penalty 4. The time-interval graph (right) recognizes the times of these penalties and its edges mostly correspond to jumping to or over the penalties. Specifically, the edge set consists of:

- Move edges outgoing from the agent's start vertex or from the destination vertex of the three types of edges mentioned below. Shown in black, these edges simply propel the agent forward just like the time-expanded case.
- Wait-then-move edges that wait and then move with a penalty. The green edge $((s_a, 0), (l_1, 3))$ corresponds to waiting at s_a from time 0 until time 2 and then moving, which incurs the penalty at $(s_a, 2)$. This edge has cost $2 + (1 + 3) = 6$ due to waiting 2 time steps at s_a and then moving with cost 1 plus the penalty 3.
- Wait-then-move edges that bypass a move penalty. For every edge of the previous type, there is another edge that arrives one time step later to avoid the move penalty. The orange edge $((s_a, 0), (l_1, 4))$ bypasses the penalty incurred by the green edge and has cost 4.
- Wait-then-move edges that bypass a wait penalty. At every neighbor location of a wait penalty, there is an edge that first waits from the arrival time of every incoming edge and then moves to arrive after the wait penalty. The cost of these edges include the penalties for waiting at the origin location, if any. The blue edges avoid the red wait penalty at $(l_2, 6)$.
- Wait-then-move edges that bypass a once-off penalty incurred for visiting a location at or before a given time. These edges are discussed later.

The time-interval graph breaks time symmetry by imposing that agents must wait first before moving to reach a given vertex. Specifically, one wait-then-move edge prevents all except one combination of moving and waiting in the time-expanded graph.

Once-Off Penalties

Robust constraints induce penalties on the edges, which are handled by adding wait-then-move edges described previously. The corridor and target non-robust constraints induce penalties on the first time that a location is visited before or after the associated time. Penalties on the first visit to a location cannot be implemented via multiple edge costs at consecutive time steps because the cost will be incurred multiple times should a path visit the location subsequently. This type of penalty, called a *once-off penalty*, necessitates a distinct mechanism to handle.

Assuming that the only non-robust constraints are the target and corridor conflict constraints, a once-off penalty induced by a target or corridor conflict constraint $r \in \mathcal{R}_2$ is defined by a value for the associated dual variable $\pi_r \leq 0$, a location $l_r \in \mathcal{L}$, a time $t_r \in \mathcal{T}$ and a time direction $d_r \in \{\text{“}\leq\text{”}, \text{“}\geq\text{”}\}$ that indicates if the penalty is to be paid upon visiting l_i at a time equal to or earlier than t_r if $d_r = \text{“}\leq\text{”}$, or equal to or later than t_r if $d_r = \text{“}\geq\text{”}$. The time direction d_r is “$\leq$” for all corridor conflict constraints r and d_r is “$\geq$” for all target conflict constraint r.

Every partial path p is now associated with an additional bit vector $(q^p_1, \ldots, q^p_{|\mathcal{R}_2|}) \in \{0,1\}^{|\mathcal{R}_2|}$ whose elements indicate whether each penalty has been paid.

Consider the extension of a partial path p along a wait-then-move edge that waits from (l_1, t_1) until $(l_1, t_2 - 1)$ then moves to (l_2, t_2). For every corridor constraint r (i.e., $d_r = \text{“}\leq\text{”}$) whose penalty is not yet paid (i.e., $q^p_r = 0$), the penalty $-\pi_r \geq 0$ is added to the reduced cost $\bar{c}_{p'}$ of the

extended partial path p' if $l_2 = l_r$ and $t_2 \leq t_r$. For every target constraint r (i.e., $d_r =$ "$\geq$") whose penalty is not yet paid, the penalty $-\pi_r \geq 0$ is accumulated if either $l_1 = l_r$ and $t_2 - 1 \geq t_r$, or $l_2 = l_r$ and $t_2 \geq t_r$. Next, the penalty is marked as paid by setting $q_r^{p'}$ to 1. Then, future visits to l_r will not incur the penalty again.

Every corridor constraint r induces a penalty $-\pi_r$ for visiting l_r at or before time t_r. This penalty can be bypassed by arriving at l_r after t_r. To facilitate this bypass, the time-interval graph also requires the addition of wait-then-move edges at every neighbor location l of l_r that wait from every arrival time at l until t_r, then move to arrive after the penalty (i.e., arrive at $(l_r, t_r + 1)$).

Every target constraint r induces a penalty for visiting l_r at or after time t_r. Since this penalty can be incurred indefinitely into the future, it cannot be bypassed and hence no wait-then-move edge needs to be added to the time-interval graph.

The Dominance Rule

Dominance rules are commonly used to discard partial paths found during one invocation of the low-level search. In addition to breaking time symmetry, the time-interval graph allows for stronger dominance rules that can compare two partial paths ending at the same location but at different times, instead of only comparing paths ending at the same location at the same time.

In the time-expanded graph, if two partial paths end at the same location at the same time, the path with higher cost is dominated and can be discarded. In the time-interval graph, if two partial paths end at the same location at different times and extending the earlier path to the time of the later path (e.g., by waiting) yields a path with the same or lower cost, then the later path can be discarded.

Consider two partial paths in Figure 2. A partial path p_1 that solely consists of the edge $((s_a, 0), (l_1, 1))$ ends at $(l_1, 1)$ with cost 1. Another partial path p_2 that includes only the green edge ends at $(l_1, 3)$ with cost 6. Extending p_1 from $(l_1, 1)$ to $(l_1, 3)$ by waiting costs 2, giving a new partial path p_1' with cost $1 + 2 = 3$. Since p_1' has lower cost than p_2, path p_1 dominates p_2 and p_2 is pruned.

This dominance rule can be strengthened by calculating the minimum reduced cost of a path from $(l_1, 1)$ to $(l_1, 3)$, instead of merely waiting. However, this stronger bound requires an expensive search and is unlikely to be beneficial given that there are relatively few wait penalties in comparison to the total number of edges. Therefore, this improvement is not considered any further.

Formally, consider two partial paths p_1 and p_2 ending at the same location l but possibly at different times t_{p_1} and t_{p_2}. Path p_1 dominates p_2 if

$$t_{p_1} \leq t_{p_2}$$

and

$$\bar{c}_{p_1} - \sum_{t=t_1}^{t_2-1} \sum_{r \in \mathcal{R}_1} \pi_r \alpha^a_{r,((l,t),(l,t+1))} x^p_{((l,t),(l,t+1))} - \sum_{r \in \mathcal{R}_2 : q_r^{p_1} = 0 \wedge q_r^{p_2} = 1} \pi_r \leq \bar{c}_{p_2}.$$

The first summation is the penalties of waiting until the time t_2 of the second path. The second summation is the once-off penalties paid by the second path but not yet paid by the first path. This dominance rule says that, should the first path wait until the time of the second path, pay the extra penalties paid by the second path and still cost the same or less than the second path, then the second path is dominated. By dominating across time, more states can be removed from the search.

This dominance rule can also be used in the time-expanded search but checking the condition is much more complex. Furthermore, it needs to ensure that a descendant is not dominated by an ancestor. Iterating through the ancestors of every extension to a partial path to check the dominance condition is surely a very expensive computation.

Note that checking ancestry is unnecessary in the time-interval graph. Consider an extension p' of a partial path p that moves away from its current location l, returns to this location l and then is immediately dominated by the ancestor p. This dominance is valid in the time-interval graph because there are wait-then-move edges outgoing from every arrival time. For every wait-then-move edge outgoing from p', there is a corresponding wait-then-move edge outgoing from p that arrives at the same vertex. In conjunction with the fact that p' is dominated, every extension of p' has an equivalent extension of p that arrives at the same destination vertex with the same or lower cost. Therefore, a descendant can be dominated by an ancestor but doing so does not prune any feasible path.

A Data Structure for the Time Intervals

A high-performance implementation of these algorithms proved annoyingly elusive. Several simpler implementations ran much slower than the reference code BCP-MAPF. The difficulty originates in the interaction between components. For example, loop bodies that were once tight and branchless in a standard A* code now contains nested loops for looking up various data structures for the penalties. This section describes a data structure for efficient look-up of the wait-then-move edges and their costs.

The expansion step in the time-expanded search of BCP-MAPF first takes a vertex and then looks up the cost of each outgoing edge from this vertex. Because each look-up is performed independently, the edge penalties can be stored in a hash table, allowing constant-time look-up.

In contrast, the expansion step in the time-interval search of BCP2-MAPF takes a vertex and then iterates through all later wait-then-move edges. Therefore, the penalties at each location and each outgoing direction need to be stored in an ordered list. This order requirement prevents the use of a hash table, requiring a sorted array, linked list, binary tree, skip list or similar. The best performing data structure is a linked list whose nodes are allocated in a contiguous section of memory (i.e., a memory pool). Storing nodes close together mitigates some of the poor cache locality issues of linked lists. The sorted array performed surprisingly poorly, presumably due to the insertions at arbitrary positions during its construction.

Another complication is the vertex and edge constraints, which place the same penalty on the same vertex/edge of all

agents. BCP-MAPF retrieves the dual solution and naively creates the same penalty on the same edges in the low-level graph of every agent. Because the edge penalties are backed by a hash table in BCP-MAPF, this operation is cheap, despite the large number of copies. In BCP2-MAPF, this copying is a major bottleneck for large numbers of agents (e.g., 200) because every insertion into a linked list requires iterating from the start to find the insertion position.

To overcome this issue, a copy-on-write data structure is developed. This data structure consists of a hash table that maps every location and outgoing direction to a linked list of intervals sorted by time. This hash table, described as *global*, is initially shared across all agents. The penalties from the vertex and edge conflict constraints, which span all agents, are first created. Next, the code loops through all agent-specific robust constraints (e.g., pseudo-corridor) to add the associated penalties. Whenever this procedure encounters an agent that still shares the global hash table, it makes a copy of the global hash table and duplicates the penalties at the location and direction of the penalty. Then, the penalty is inserted into this agent-specific copy of the linked list. The penalties at the other locations or directions remain shared. This shared data structure means that the penalties originating from the vertex and edge conflict constraints are created once in the global hash table, which can be used by all agents not involved in agent-specific constraints.

Computational Results

The experiments are carried out on 16 maps from the Moving AI benchmarks (Sturtevant 2012). These maps are selected to span a wide range of structures, including cities, video games, empty squares, mazes and warehouses. The experiments are conducted on 11 different numbers of agents on 15 instances of the *random* category, which indicates that the start and goal locations of the agents are randomly distributed. The experiments are run on a total of 2640 instances. All instances for each solver are run in parallel with a time limit of 5 minutes on an Intel Xeon Gold 6338 CPU with 64 cores.

Comparison Against Other Solvers

Overall, BCP2-MAPF, BCP-MAPF, Lazy CBS and CBSH2-RTC respectively take a total of 6777, 7219, 9244 and 10495 minutes to solve 1462 (55%), 977 (37%), 957 (36%) and 608 (23%) instances. Instances that time out are counted as 5 minutes. For the 461 instances solved by all four methods, the total time taken is 13, 41, 134 and 148 minutes respectively, indicating that BCP2-MAPF is 3.3 times faster than its nearest competitor BCP-MAPF, 10.6 times faster than Lazy CBS and 11.7 times faster than CBSH2-RTC.

Figure 3 plots the percentage of instances solved categorized by map and number of agents. The results show that BCP2-MAPF performs consistently well across all maps. It dominates CBSH2-RTC on all maps and dominates BCP-MAPF except for orz900d. BCP2-MAPF has the greatest lead against the competing CBSH2-RTC and Lazy CBS algorithms on the five game maps. It also closes all tested instances on the small empty square empty-8-8, which have at most 32 agents. However, it is inferior to Lazy CBS when the square is enlarged and the number of agents increase in the empty-32-32 map and when 10% of the map is replaced with obstacles in random-32-32-10. BCP2-MAPF is also no match for Lazy CBS on warehouse-10-20-10-2-2, the warehouse with corridors of width 2, where corridor reasoning is rendered inapplicable.

Comparison of the Contributions

Figure 3 also shows an ablation study of the three main contributions of BCP2-MAPF. The complete BCP2-MAPF solves 1462 instances, compared to 1401 without the hybrid search, 1343 without corridor reasoning and 1299 without time intervals.

The time-interval graph makes the biggest contribution because it breaks time symmetries, of which many intrinsically exist due to the problem definition. This difference is most prominent on the maps with higher congestion, such as brc202d, orz900d, random-32-32-20, maze-128-128-1 and maze-128-128-2. The time expansion performs similarly well on maps with more open space, including Berlin_1_256, lt_gallowstemplar_n, empty-32-32, random-32-32-10 and maze-128-128-10.

The corridor conflicts make negligible difference except on maze-128-128-1, maze-128-128-2 and warehouse-10-20-10-2-1, where many long corridors appear. Without explicit handling of corridors, BCP2-MAPF reverts to pseudo-corridor conflicts, which require many iterations to clear.

The hybrid best-first depth-first search enables BCP2-MAPF to close the final instance of empty-8-8. It also assists in empty-32-32 but has more muted impact across the other maps. Nevertheless, it still contributes to solving an additional 61 instances, so overall it is clearly beneficial.

Conclusion

This paper presents a new implementation of branch-and-cut-and-price for multi-agent path finding named BCP2-MAPF. Its main novelties are a low-level search that breaks time symmetries using a graph based on time intervals, corridor reasoning techniques borrowed from CBSH2-RTC and a hybrid best-first depth-first search that balances improving both the lower and upper bounds. The implementation, consisting of more than 22000 lines of code, is engineered from scratch and underwent substantial tuning for high performance.

Experimental results on 2640 instances across 16 maps indicate that BCP2-MAPF far exceeds the capabilities of other state-of-the-art solvers, including the reference implementation BCP-MAPF and the competing CBSH2-RTC and Lazy CBS techniques. BCP2-MAPF is 3.3, 10.6 and 11.7 times faster than BCP-MAPF, Lazy CBS and CBSH2-RTC respectively on the subset of instances solved by all methods. It dominates CBSH-RTC and BCP-MAPF except for a few instances. It also outperforms Lazy CBS on all maps except the larger (but still small) empty square and the warehouse with corridors of width 2.

Of the three scientific contributions, the low-level search has the greatest impact on its performance, allowing it to solve many more instances of the larger game maps brc202d and orz900d. The other advances achieve a more modest result but still enable several more instances to be solved.

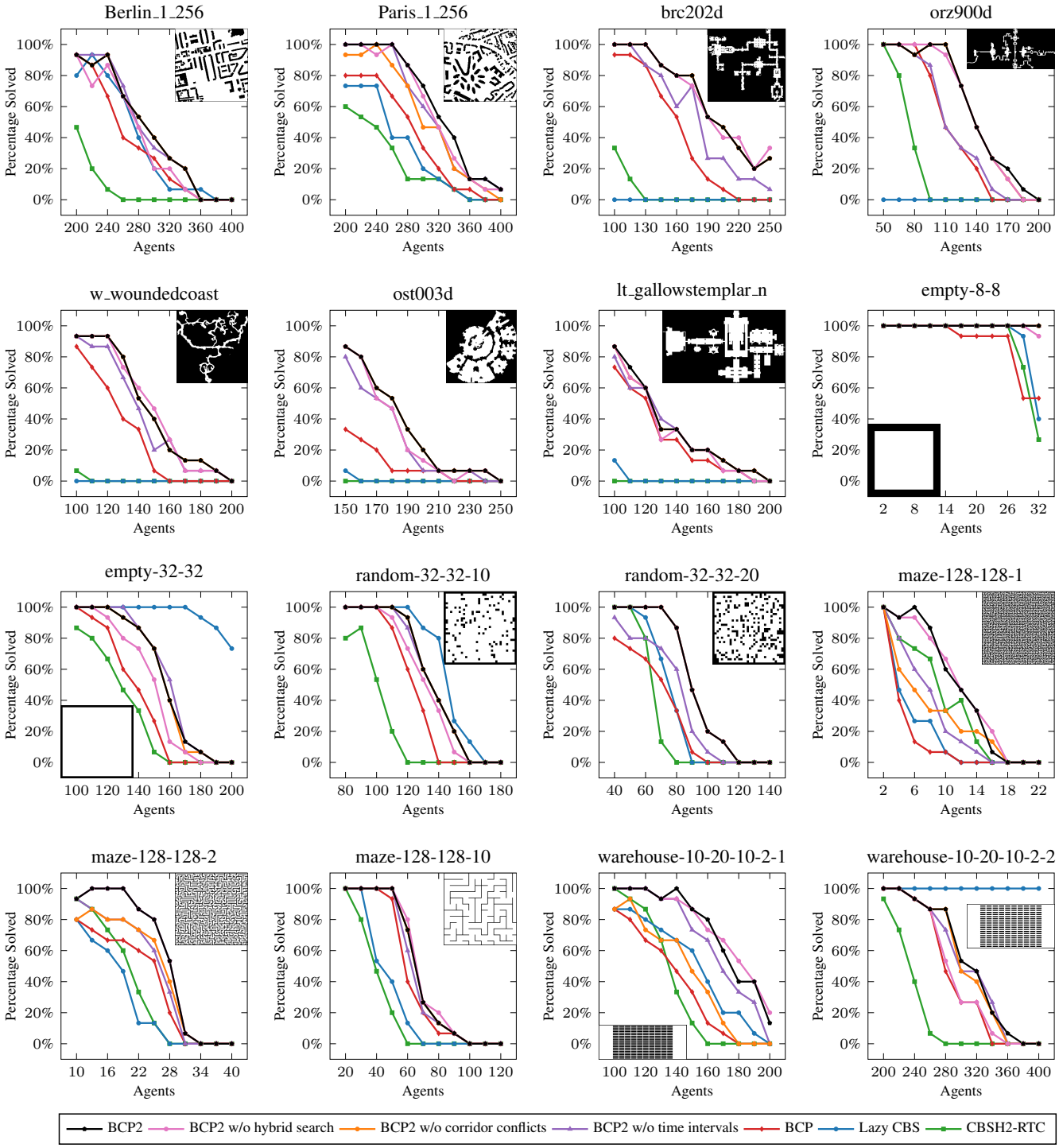

Figure 3: Success rate of the algorithms by map. Higher is better.

Acknowledgments

This research is supported by the Australian Research Council under grant DE240100042.

References

de Aragao, M. P.; and Uchoa, E. 2003. Integer program reformulation for robust branch-and-cut-and-price algorithms. In *Mathematical Program in Rio: a conference in honour of Nelson Maculan*, 56–61.

Desaulniers, G.; Desrosiers, J.; and Spoorendonk, S. 2011. Cutting planes for branch-and-price algorithms. *Networks*, 58(4): 301–310.

Desrosiers, J.; Lübbecke, M.; Desaulniers, G.; and Gauthier, J.-B. 2024. Branch-and-Price. Les Cahiers du GERAD G-2024-36, Groupe d'études et de recherche en analyse des décisions.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 28(1): 83–87.

Fukasawa, R.; Longo, H.; Lysgaard, J.; De Aragão, M. P.; Reis, M.; Uchoa, E.; and Werneck, R. F. 2006. Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem. *Mathematical Programming*, 106(3): 491 – 511.

Gange, G.; Harabor, D.; and Stuckey, P. 2019. Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS)*, volume 29, 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)*, 1289–1296. International Joint Conferences on Artificial Intelligence Organization.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2022. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research*, 144: 105809.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

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.

Lübbecke, M. E.; and Desrosiers, J. 2005. Selected topics in column generation. *Operations Research*, 53(6): 1007–1023.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints*, 14(3): 357–391.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, 5628–5635.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. 2015. Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence*, 219: 40–66.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, 173–178.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 151–158.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 144–148.

Yu, J.; and LaValle, S. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1): 1443–1449.