Highlights

Branch-and-Cut-and-Price for Multi-Agent Path Finding

Edward Lam, Pierre Le Bodic, Daniel Harabor, Peter J. Stuckey

- The MAPF problem finds minimal-cost collision-free paths for cooperating agents.
- This paper presents BCP, an exact algorithm for MAPF.
- BCP uses branch-and-cut-and-price to decompose MAPF into easier subproblems.
- It finds paths independently for each agent using a novel shortest path problem.
- It then resolves thirteen classes of conflicts between agents using constraints.
- BCP outperforms the two state-of-the-art solvers CBSH2-RTC and Lazy CBS.

Branch-and-Cut-and-Price for Multi-Agent Path Finding

Edward Lam, Pierre Le Bodic, Daniel Harabor, Peter J. Stuckey

Monash University, Melbourne, Victoria, Australia

Abstract

The Multi-Agent Path Finding problem aims to find a set of collision-free paths that minimizes the total cost of all paths. The problem is extensively studied in artificial intelligence due to its relevance to robotics, video games and logistics applications, but is seldom considered in the mathematical optimization community. This paper tackles the problem using a branch-and-cut-and-price algorithm that incorporates a shortest path pricing problem for finding paths for every agent independently and thirteen classes of constraints for resolving different types of conflicts. Experimental results show that this mathematical approach solves 2402 of 4430 instances compared to 2039 and 1939 by the state-of-the-art solvers Lazy CBS and CBSH2-RTC published in artificial intelligence venues.

Keywords:
multi-agent path finding, multi-agent planning, column generation, cutting plane, valid inequality

1. Introduction

Multi-Agent Path Finding (MAPF) is a family of combinatorial optimization problems that originated from artificial intelligence and automated planning. The family of problems typically features simple combinatorial structures yet it is extremely relevant to many application domains including robotics, video games and logistics (e.g., Sharon et al. (2015); Ma et al. (2017)).

Email addresses: edward.lam@monash.edu (Edward Lam), pierre.lebodic@monash.edu (Pierre Le Bodic), daniel.harabor@monash.edu (Daniel Harabor), peter.stuckey@monash.edu (Peter J. Stuckey)

URL: https://ed-lam.com/ (Edward Lam), https://harabor.net/daniel/ (Daniel Harabor), https://people.eng.unimelb.edu.au/pstuckey/ (Peter J. Stuckey)

This paper considers the simplest and most studied variant within the problem family. Space and time are discretized. Given a grid world with numerous obstacles and a group of cooperating agents, each with a unique start cell and goal cell, the MAPF problem consists in routing every agent from its start cell to its goal cell along a path of orthogonally-connected grid cells such that the agents do not collide into each other at any time and that the total number of actions is minimized (note that some variants minimize makespan). Two instances are shown in Figure 1. A solution is drawn in colored lines. Each line represents the path taken by an agent from its start cell to its goal cell. Even though the paths appear to cross, the agents traverse the same cells in different timesteps and hence do not collide.

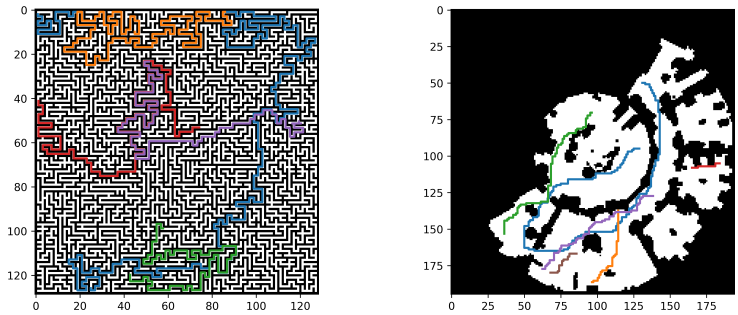

Figure 1: Agents exploring a randomly generated maze and a map from a video game.

This paper presents BCP, a branch-and-cut-and-price algorithm for MAPF. BCP decomposes the MAPF problem into (1) a master problem that selects a set of low-cost paths from a large pool of paths, (2) a pricing problem that adds lower-cost paths to the pool in the master problem or proves that none exist, (3) separation problems that add constraints to the master problem to resolve conflicts, and (4) branching rules that split the nodes in the branch-and-bound search tree. BCP includes a non-trivial adaptation of the A* shortest path algorithm to solve the pricing problem, thirteen classes of constraints for resolving different types of conflicts and two branching rules, one of which directly bounds the objective function. Experiments on 4430 instances across 16 maps from two sets of standard benchmarks indicate that BCP outperforms the two leading solvers Lazy CBS (Gange et al., 2019) and CBSH2-RTC (Li et al., 2021) by solving 2402 instances in total, compared to 2039 by Lazy CBS and 1939 by CBSH2-RTC.

This journal article presents the definitive version of the BCP algorithm first introduced in two conference papers (Lam et al., 2019; Lam and Le Bodic, 2020). This paper extends the earlier conference papers by presenting (1) technical details and examples previously omitted, (2) four more classes of constraints for resolving

new types of conflicts, (3) a technique for caching solutions from the pricing problem and (4) new experimental results.

The remainder of this paper is organized as follows. Section 2 formalizes the problem. Section 3 reviews existing methods for exact MAPF. Section 4 reviews the BCP algorithm and presents several recent improvements. Section 5 analyzes the experimental results. Section 6 concludes this paper.

2. Problem Definition

Consider a grid world with width $W \in \mathbb{Z}_+$ and height $H \in \mathbb{Z}_+$. We define $\mathcal{L} = \{0, \dots, W-1\} \times \{0, \dots, H-1\}$ as the set of locations. A *location* $l \in \mathcal{L}$ is a pair of a horizontal coordinate and a vertical coordinate on the grid. Some locations are designated as *obstacles*, i.e., agents cannot move through them. A location $l_2 = (x_2, y_2) \in \mathcal{L}$ is a *neighbor* of location $l_1 = (x_1, y_1) \in \mathcal{L}$ in the

- *north* direction if $x_2 = x_1$ and $y_2 = y_1 - 1$,
- *south* direction if $x_2 = x_1$ and $y_2 = y_1 + 1$,
- *west* direction if $x_2 = x_1 - 1$ and $y_2 = y_1$, and
- *east* direction if $x_2 = x_1 + 1$ and $y_2 = y_1$.

Under this definition, the north-west corner of the grid is the origin $(0, 0)$.

Given a time horizon $T \in \mathbb{Z}_+$, let $\mathcal{T} = \{0, \dots, T-1\}$ denote the set of timesteps. The problem is defined on a time-expanded directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{L} \times \mathcal{T}$ is the set of vertices and $\mathcal{E} = \{(((x_1, y_1), t_1), ((x_2, y_2), t_2)) \in \mathcal{V} \times \mathcal{V} : |x_2 - x_1| + |y_2 - y_1| \leq 1 \wedge t_2 = t_1 + 1\}$ is the set of edges. A vertex $v \in \mathcal{V}$ is a location-timestep pair. An edge $e \in \mathcal{E}$ is a pair of vertices indicating a movement from a location at some timestep to a neighbor location (a *move* action) or a movement to the same location in the next timestep (a *wait* action). The *reverse* of an edge $e = ((l_1, t_1), (l_2, t_1 + 1))$ is denoted $e' = ((l_2, t_1), (l_1, t_1 + 1))$.

We define $\mathcal{A}$ as the set of agents. Each agent $a \in \mathcal{A}$ has a *start location* $s_a \in \mathcal{L}$ and a *goal location* $g_a \in \mathcal{L}$, which may coincide. Every start location is unique and every goal location is unique. A *path* p of length $k \in \{1, \dots, T\}$ for agent a is a sequence of k locations $(l_0, l_1, l_2, \dots, l_{k-1})$ 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 $t \in \{0, \dots, k-2\}$. Path p *visits* the vertices (l_t, t) where $t \in \{0, \dots, k-1\}$ and (g_a, t) for all $t \in \{k, \dots, T-1\}$ as the agent remains at its goal location after the path concludes. Path p *traverses* the edges $((l_t, t), (l_{t+1}, t+1))$ where $t \in \{0, \dots, k-2\}$ and the edges $((g_a, t), (g_a, t+1))$ where $t \in \{k-1, \dots, T-2\}$. Path p has a *cost*

$c_p = k - 1$ equal to the number of edges, i.e., the number of move or wait actions before the agent reaches its goal (and waits there indefinitely).

A feasible solution to MAPF is a set of paths, one for each agent $a \in \mathcal{A}$, such that (1) each vertex is visited at most once and (2) each edge or its reverse are traversed at most once (in this statement and throughout, vertices and edges are time-expanded). These two conditions are respectively called *vertex conflicts* and *edge conflicts*. An optimal solution to MAPF is a feasible solution that minimizes the sum of costs of all paths.

3. Related Work

There are three broad classes of algorithms for optimal MAPF: (1) *search-based* methods solve MAPF directly and typically implement ad hoc techniques specific to the problem, (2) compilation-based methods reduce MAPF to instances of well-known combinatorial optimization problems and thus can benefit from advances in solver techniques, and more recently, (3) hybrids.

3.1. Search-based Methods

Multi-Agent Planning. MAPF can be considered as a special case of multi-agent planning, a well-known class of problems that is often solved by modeling the set of agents as one large agent with many degrees of freedom (Torreño et al., 2017). This approach, sometimes called *joint planning*, completely fails even for small instances (Standley, 2010).

Operator Decomposition. Algorithms of this type plan in the joint space of all agents but in a way that tries to avoid an explosion in branching factor. Operator decomposition (Standley, 2010) is a well-known example that interleaves the planning of single agents to dramatically reduce the branching factor of the search. Another approach, EPEA*, is a partial expansion solver, which defers generating all but the most promising successors of a node (Goldenberg et al., 2014). Operator decomposition and EPEA* are both optimal and are often effective on MAPF problems with up to dozens of agents and with low congestion. In more challenging settings, these methods often exhaust available memory long before finding a solution (Standley, 2010).

Conflict-based Search. Conflict-based search (CBS) (Sharon et al., 2015) is a two-level tree search that resembles a simple form of branch-and-bound. At the low level, it computes a path for each agent independently. At the high level, CBS detects conflicts between pairs of agents and resolves them by splitting the current solution into two

related subproblems, each of which involves replanning a single agent. Recursively resolving conflicts by splitting a subproblem into two children implicitly defines a search tree. The high-level search explores this tree using best-first search and finds an optimal solution upon expanding the first collision-free node. The success of CBS has led to a large family of optimal and bounded suboptimal variants (Felner et al., 2017). Such leading MAPF algorithms can scale to large maps with many agents, often with the help of reasoning techniques developed specifically for MAPF, such as lower-bounding heuristics (Felner et al., 2018), branching strategies (Boyarski et al., 2015) and specialized constraints (Li et al., 2019). CBSH2-RTC (Li et al., 2021) is currently the top-performing variant within the CBS family.

3.2. Compilation-based Methods

Compilation-based methods reduce MAPF to an instance of another problem, such as constraint programming (Ryan, 2010), answer set programming (Erdem et al., 2013), propositional satisfiability (Surynek et al., 2016b,a) or integer programming (Yu and LaValle, 2013). In general, these methods create a model of MAPF that contains variables to store values representing the actions of the agents, and constraints to communicate restrictions on the possible values of the variables. Then, solving MAPF is as simple as solving the model, which can be done easily (but not necessarily quickly) using black-box solvers. Furthermore, by formulating MAPF as an instance of another problem, the latest advances in solvers are immediately available. All that is needed is a "good" model, leaving the solving process to specialized software packages.

The integer programming model of MAPF (Yu and LaValle, 2013) is particularly relevant. This reduction involves a time-expanded graph whose vertices are indexed by space and time, and specialized gadgets to account for conflicts. Despite this inefficient representation, the model remains reasonably effective on small instances since integer programming solvers are mature industry-grade software supported by decades of academic research.

3.3. Hybrids

Lazy CBS. Conflict analysis is an essential technique in modern propositional satisfiability (Marques Silva and Sakallah, 1996) and constraint programming (Ohrimenko et al., 2009) solvers. Conflict analysis prunes an exponential number of nodes in the search tree by dynamically building constraints to prevent the same infeasibility from occurring elsewhere in the search tree. Lazy CBS (Gange et al., 2019) is an improvement of CBS that implements conflict analysis. Lazy CBS records the change in costs from forcing an agent to avoid a vertex or edge upon branching and then uses

this information to deduce that certain combinations of branchings are suboptimal and consequently prune all nodes, even those from disparate parts of the search tree, that contain these branchings.

4. The BCP Algorithm

This section presents the main contributions of this paper. Before proceeding further, readers unfamiliar with integer programming or branch-and-cut-and-price are recommended to first consult the intuition and background information provided in the supplementary material. For a formal treatment, we advise them to consult introductory material for integer programming (e.g., Rader (2010)) and then the tutorials on branch-and-cut-and-price (Desrosiers and Lübbecke, 2010; Lübbecke and Desrosiers, 2005; Desaulniers et al., 2005; Barnhart et al., 1998).

4.1. Overview

Branch-and-cut-and-price is a general framework for solving a combinatorial optimization problem via a sequence of easier subproblems (Desrosiers and Lübbecke, 2010; Lübbecke and Desrosiers, 2005; Desaulniers et al., 2005; Barnhart et al., 1998). In particular, it can solve large-scale graph optimization problems for which other approaches fail. Its power comes from three main sources: (1) it can call dedicated algorithms to solve the subproblems, (2) it can solve exponential-size integer programming models whose linear relaxation is theoretically tighter than all other known models (e.g., Letchford and Salazar-González (2006)), and (3) it presents alternative views into a problem and hence enables cutting planes in different spaces. BCP is a branch-and-cut-and-price algorithm that exploits (1) and (3). BCP consists of four main components:

- a *master problem* that assembles a set of low-cost paths, each represented by a variable/column,
- a *pricer* that finds lower-cost paths,
- eleven *separators* that resolve conflicts in candidate solutions proposed by the master problem, and
- two *branching rules* that resolve fractionalities in the master problem.

```
1  open ← NewPriorityQueue()                          // create priority queue of nodes
2  open.Add(CreateRootNode())                                  // create root node
3  while ¬open.IsEmpty() do                                    // solve every node
4  |  node ← open.Pop()                                 // get node from priority queue
5  |  separated ← true                                  // initialize control variable
6  |  while separated do                                // loop until no cuts are found
7  |  |  priced ← true                                 // initialize control variable
8  |  |  while priced do                          // loop until no new paths are found
9  |  |  |  SolveMasterProblem(node.master)            // solve master problem
10 |  |  |  priced ← Pricer(node.master)                // generate new paths
11 |  |  separated ← Separators(node.master)             // generate new cuts
12 |  if node.master.IsFractional() then          // split the node if fractional
13 |  |  left_child, right_child ← CreateChildren(node)  // create children nodes
14 |  |  open.Add(left_child)                  // add left child to priority queue
15 |  |  open.Add(right_child)                 // add right child to priority queue
```

Algorithm 1: A basic branch-and-cut-and-price algorithm loosely followed by BCP.

The pseudocode for BCP is given in Algorithm 1. BCP is implemented in the SCIP integer programming solver, which mostly follows but does not strictly adhere to the algorithm shown. Firstly, a priority queue for storing the open nodes of the branch-and-bound tree is created (Line 1). Next, the root node is created and added to the priority queue (Line 2).

BCP then loops through every open node (Lines 3 and 4), as ordered by the best-first scoring function. The outer *separation* loop begins in Lines 5 and 6. The separation loop contains the inner *pricing* loop, which begins in Lines 7 and 8.

The pricing loop alternates between solving the master problem (Line 9) and the pricing problem (Line 10). Given a large set of paths for each agent, the master problem solves a linear program to determine the fraction that each path is selected in a candidate solution. The pricing problem is solved after the master problem. The pricing problem attempts to find lower-cost paths for inclusion in the sets from which the master problem operates, in the hope that these paths will be chosen in the next iteration of the master problem. Even if lower-cost paths are found, the master problem is not guaranteed to use them because they may be incompatible with the paths of other agents or for other reasons (e.g., degeneracy).

Eventually, the pricer will declare that it cannot find a potentially-improving path. At this point, BCP proceeds to the separators (Line 11). BCP currently contains thirteen separators, each handling a different class of conflicts: two are the vertex and edge conflicts, which are *problem constraints* and are necessary to correctly define the problem, and the remaining eleven are *valid inequalities*, which are redundant constraints that improve performance in practice and give theoretical insights into the

geometry of the search space. The candidate solution to the master problem is passed to each separator in turn, which checks whether the candidate solution violates a conflict in its class. If so, the constraint is added to the master problem, requiring the master problem (together with the pricing problem) to propose a different candidate solution that does not contain this conflict.

The node is solved when the outer loop is completed. Branching occurs if the master problem solution has at least one variable taking a fractional value (Line 12). BCP splits the node into two children nodes according to a branching rule (Line 13) and adds them to the priority queue (Lines 14 and 15).

To correctly define BCP as an instantiation of the branch-and-cut-and-price algorithm for the MAPF domain, one must define a master problem that assembles the paths and constraints, define at least one pricer for finding paths and show that it will find a lower-cost path (i.e., a path with negative reduced cost) if and only if one exists, define separators of problem constraints and show that if a solution violates a constraint, the separators will find this constraint and also show that all constraints added by the separators will retain at least one feasible solution, define separators of valid inequalities and show that all constraints added by the separators will retain at least one feasible solution, and define at least one branching rule for resolving fractionalities and show that it cannot remove valid integer solutions. The master problem, pricer, separators and branching rules of BCP are formalized below.

4.2. The Master Problem

Given a set of possible paths for every agent, the master problem uses linear programming to minimize the sum-of-costs by selecting, ideally, a path, but often, a set of fractional paths for every agent such that all selected paths are fractionally free of conflicts.

Let $\mathcal{P}_a$ be a large pool of candidate paths for every agent $a \in \mathcal{A}$. For all $a \in \mathcal{A}, p \in \mathcal{P}_a$, define $\lambda_p \in [0, 1]$ as a variable representing the proportion of selecting path p. Because the master problem is solved using linear programming, λ_p can take fractional values.

The master problem begins as the linear programming problem:

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

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 every agent uses exactly one path. Constraint (1c) are the non-negativity constraints, which disallow negative proportions of a path. Constraints (1b) and (1c) together ensure that $\lambda_p \in [0, 1]$. Constraints enforcing vertex conflicts and edge conflicts are initially omitted and added dynamically as necessary.

4.3. Resolving Conflicts

BCP resolves vertex conflicts and edge conflicts by calling separators to add constraints to the master problem. Even though BCP can solve MAPF with only the vertex conflicts and edge conflicts, its performance can be substantially improved by reasoning about different classes of conflicts. This section describes how vertex conflicts and edge conflicts are implemented in BCP, and then introduces nine classes of conflicts that either reason about the MAPF problem structure or combine various combinations of vertex conflicts and edge conflicts into one stronger constraint.

4.3.1. Reasoning About Vertices and Edges in the Master Problem

It is not yet clear how to enforce constraints on vertices and edges since the variables in the master problem concern paths. The vertices and edges used by the paths selected by the master problem can be obtained as follows.

Recall from Section 2 that a path $p = (l_0, l_1, l_2, \ldots, l_{k-1})$ for an agent $a \in \mathcal{A}$ visits the vertices (l_t, t) for all $t \in \{0, \ldots, k-1\}$ and (g_a, t) for all $t \in \{k, \ldots, T-1\}$, and traverses the edges $((l_t, t), (l_{t+1}, t+1))$ for all $t \in \{0, \ldots, k-2\}$ and $((g_a, t), (g_a, t+1))$ for all $t \in \{k-1, \ldots, T-2\}$. Let $x_v^p \in \{0, 1\}$ be a constant taking value 1 if path p visits vertex $v \in \mathcal{V}$ and take value 0 otherwise. Let $x_e^p \in \{0, 1\}$ similarly indicate whether path p traverses edge $e \in \mathcal{E}$. Under this definition, x_v^p and x_e^p include the vertices $(g_a, t), t \in \{k, \ldots, T-1\}$, and edges $((g_a, t), (g_a, t+1)), t \in \{k-1, \ldots, T-2\}$, used to indicate the agent remaining at its goal location after the path is completed.

The proportion $X_e^a \in [0, 1]$ that agent $a \in \mathcal{A}$ traverses an edge $e \in \mathcal{E}$ can then be computed as

$$X_e^a = \sum_{p \in \mathcal{P}_a} x_e^p \lambda_p. \tag{2}$$

The proportion $X_v^a \in [0, 1]$ that agent $a \in \mathcal{A}$ visits a vertex $v = ((x, y), t) \in \mathcal{V}$ can be calculated by summing the five edges incoming to v:

$$\begin{aligned} X_v^a = {} & X^a_{(((x-1,y),t-1),v)} + X^a_{(((x+1,y),t-1),v)} + X^a_{(((x,y-1),t-1),v)} + \\ & X^a_{(((x,y+1),t-1),v)} + X^a_{(((x,y),t-1),v)}. \end{aligned} \tag{3}$$

4.3.2. Vertex Conflicts

A vertex conflict occurs at $v \in \mathcal{V}$ whenever v is visited by more than one agent. That is, whenever

$$\sum_{a \in \mathcal{A}} X_v^a > 1. \tag{4}$$

Given a solution to the master problem, the separator for vertex conflicts first computes X_v^a for all $a \in \mathcal{A}$ and $v \in \mathcal{V}$ using Equations (2) and (3). Next, it builds the constraint

$$\sum_{a \in \mathcal{A}} X_v^a \leq 1, \tag{5}$$

for every $v \in \mathcal{V}$ that satisfy Condition (4). It then substitutes for X_v^a in Constraint (5) using Equations (2) and (3), forming a constraint over the λ_p variables in the master problem. Finally, the separator adds this constraint to the master problem.

4.3.3. Edge Conflicts

Consider a move edge $e = ((l_1, t), (l_2, t+1)) \in \mathcal{E}$ where $l_1 \neq l_2$. An edge conflict occurs at e whenever e or its reverse e' are traversed by more than one agent. That is,

$$\sum_{a \in \mathcal{A}} (X_e^a + X_{e'}^a) > 1.$$

The separator for edge conflicts is similar to the separator for vertex conflicts. The edge conflict at e is removed by adding the constraint

$$\sum_{a \in \mathcal{A}} (X_e^a + X_{e'}^a) \leq 1 \tag{6}$$

to the master problem after substitution using Equation (2).

4.3.4. Wait-Edge Conflicts

Constraint (6) permits an edge or its reverse to be used by at most one agent. Wait-edge conflicts lift the edge conflicts by also prohibiting a wait within the same constraint.

Figure 2 shows a move edge $e_1 = ((l_1, t), (l_2, t+1)) \in \mathcal{E}$ where $l_1 \neq l_2$, its reverse $e_1' = ((l_2, t), (l_1, t+1))$ and a wait edge $e_2 = ((l_1, t), (l_1, t+1))$ that is incompatible with both e_1 because of the vertex conflict at (l_1, t) and with e_1' because of the vertex conflict at $(l_1, t+1)$. Since the three edges are pair-wise incompatible, at most one of e_1, e_1' or e_2 can be traversed. Then, this conflict can be removed using the constraint

$$\sum_{a \in \mathcal{A}} (X_{e_1}^a + X_{e_1'}^a + X_{e_2}^a) \leq 1. \tag{7}$$

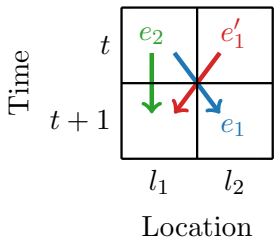

Figure 2: A wait-edge conflict.

Constraint (7) contains all the terms in Constraint (6). In the presence of wait-edge conflict constraints, the ordinary edge conflict constraints are redundant and can be omitted. Unlike Constraint (6), Constraint (7) is asymmetric since swapping l_1 and l_2 results in a different constraint. This cannot be reconciled by, e.g., including the edge $((l_2, t), (l_2, t+1))$ in Constraint (7) since this edge is compatible with e_2. In the implementation, the left-hand side of both the wait edge at l_1 and at l_2 is computed and the constraint with the larger violation of the two is added. Wait-edge conflict constraints were first presented in the earlier conference paper (Lam and Le Bodic, 2020).

4.3.5. Two-Agent Wait-Edge Conflicts

Wait-edge conflicts span all agents. Two-agent wait-edge conflicts exploit additional reasoning available for a pair of agents. Figure 3 shows an example. Consider an agent $a_1 \in \mathcal{A}$ fractionally using the edges $e_1 = ((l_1, t), (l_2, t+1))$ and $e_2 = ((l_1, t), (l_1, t+1))$, and another agent $a_2 \in \mathcal{A}$ using the edge $e'_1 = ((l_2, t), (l_1, t+1))$ and an edge $e_3 \in \mathcal{E}_3$ from the set $\mathcal{E}_3$ of edges incoming to $(l_1, t) = ((x_1, y_1), t)$, i.e.,

$$\begin{aligned}\mathcal{E}_3 = \{&(((x_1 - 1, y_1), t-1), (l_1, t)),\\ &(((x_1 + 1, y_1), t-1), (l_1, t)),\\ &(((x_1, y_1 - 1), t-1), (l_1, t)),\\ &(((x_1, y_1 + 1), t-1), (l_1, t)),\\ &((l_1, t-1), (l_1, t))\} \cap \mathcal{E}.\end{aligned}$$

By construction, any edge in $\mathcal{E}_3$ is incompatible with e'_1 because agent a_2 cannot be in two locations at the same time and is incompatible with e_1 and e_2 because of a vertex conflict at (l_1, t). This reasoning results in the constraint

$$X^{a_1}_{e_1} + X^{a_1}_{e_2} + X^{a_2}_{e'_1} + \sum_{e \in \mathcal{E}_3} X^{a_2}_{e} \leq 1. \tag{8}$$

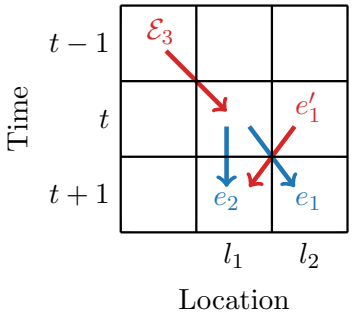

Figure 3: A two-agent wait-edge conflict.

If, instead, e_2 is defined as $e_2 = ((l_2, t), (l_2, t+1))$, the same constraint is valid if the set $\mathcal{E}_3$ is redefined as the edges outgoing from (l_2, t), the edges incoming to $(l_2, t+1)$ or the edges outgoing from $(l_2, t+1)$. Two-agent wait-edge conflicts are presented for the first time here.

4.3.6. Corridor Conflicts

Corridor conflicts can appear when two agents fractionally cross a space of unit height and some length in opposite directions, such as when two agents enter a corridor or maneuver around a U-shaped bend. Figure 4 shows an agent a_1 fractionally using $e_1 = ((l_1, t), (l_2, t+1))$ and $e_2 = ((l_1, t+1), (l_2, t+2))$ one timestep later, and another agent a_2 using $e_1' = ((l_2, t), (l_1, t+1))$ and $e_2' = ((l_2, t+1), (l_1, t+2))$. These four edges for the two agents are pair-wise incompatible. In an integer solution, if a_1 uses e_1, it arrives at l_2 at time $t+1$ and hence cannot use e_2, which would require a_1 to be at l_1 at time $t+1$. Agent a_2 cannot use e_1' because it would incur an edge conflict with a_1 using e_1 nor use e_2' because it would incur a vertex conflict at $(l_2, t+1)$. By same arguments, at most one of these four agent-edge pairs can be used. This leads to the constraint

$$X_{e_1}^{a_1} + X_{e_2}^{a_1} + X_{e_1'}^{a_2} + X_{e_2'}^{a_2} \leq 1. \tag{9}$$

Corridor conflict constraints were first presented in the previous conference paper (Lam et al., 2019).

4.3.7. Wait-Corridor Conflicts

Wait-corridor conflicts lift the corridor conflicts in the same manner that the wait-edge conflicts lift the edge conflicts. Figure 5 shows a wait-corridor conflict. The two agents are fractionally traversing the same four edges in a corridor conflict. Agent a_1 is using $e_1 = ((l_1, t), (l_2, t+1))$ and $e_2 = ((l_1, t+1), (l_2, t+2))$. Agent a_2 is using $e_1' = ((l_2, t), (l_1, t+1))$ and $e_2' = ((l_2, t+1), (l_1, t+2))$. In addition, agent a_1 is now

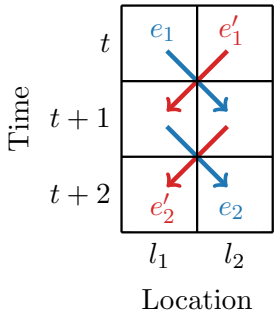

Figure 4: A corridor conflict.

also using the edges $e_3 = ((l_2, t), (l_2, t+1))$ and $e_4 = ((l_1, t+1), (l_1, t+2))$. Edge e_3 is incompatible with e_1, e_2 and e_4 because the agent can only be at one location at any given time. It is also incompatible with e'_1 because of the vertex conflict at (l_2, t) and incompatible with e'_2 because of the vertex conflict at $(l_2, t+1)$. Edge e_4 is also incompatible with the other five edges for the same reasons. Hence, these six edges are pair-wise incompatible and induce the constraint

$$X_{e_1}^{a_1} + X_{e_2}^{a_1} + X_{e_3}^{a_1} + X_{e_4}^{a_1} + X_{e'_1}^{a_2} + X_{e'_2}^{a_2} \leq 1. \tag{10}$$

If wait-corridor constraints are implemented, the ordinary corridor constraints are superseded and can be ignored. Wait-corridor conflict constraints are first presented here.

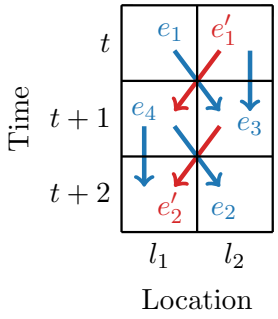

Figure 5: A wait-corridor conflict.

4.3.8. Wait-Delay Conflicts

Figure 6 shows a wait-delay conflict between two agents $a_1, a_2 \in \mathcal{A}$ where $a_2 \neq a_1$. Agent a_2 is attempting to visit $l_1 = (x, y)$ at time t or time $t+1$ but is impeded by

another agent a_1 waiting at l_1, i.e., traversing $e_1 = ((l_1, t), (l_1, t+1))$. Let

$$\begin{aligned}\mathcal{E}_2 = \{&((l_1, t-1), (l_1, t)),\\ &(((x-1, y), t-1), (l_1, t)),\\ &(((x+1, y), t-1), (l_1, t)),\\ &(((x, y-1), t-1), (l_1, t)),\\ &(((x, y+1), t-1), (l_1, t)),\\ &(((x-1, y), t), (l_1, t+1)),\\ &(((x+1, y), t), (l_1, t+1)),\\ &(((x, y-1), t), (l_1, t+1)),\\ &(((x, y+1), t), (l_1, t+1))\} \cap \mathcal{E}.\end{aligned}$$

The first five edges of $\mathcal{E}_2$ lead into (l_1, t). The last four edges lead into $(l_1, t+1)$ from a neighbor location. Agent a_2 can use at most one edge from $\mathcal{E}_2$ because it can only be at one location at a time and the first five edges do not lead into the last four edges. Note that $\mathcal{E}_2$ excludes $((l_1, t), (l_1, t+1))$ because this edge is compatible with the first five edges. Since a_1 traversing e_1 and a_2 traversing any edge $e \in \mathcal{E}_2$ is incompatible due to a vertex conflict at (l_1, t) or $(l_1, t+1)$, the wait-delay conflict constraint is

$$X_{e_1}^{a_1} + \sum_{e \in \mathcal{E}_2} X_e^{a_2} \leq 1. \tag{11}$$

Wait-delay conflict constraints were first presented in the conference paper (Lam and Le Bodic, 2020).

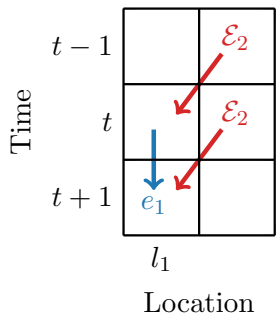

Figure 6: A wait-delay conflict.

4.3.9. Exit-Entry Conflicts

Exit-entry conflicts are similar to wait-delay conflicts. Figure 7 shows an agent $a_1 \in \mathcal{A}$ moving from $l_1 = (x_1, y_1)$ to $l_2 = (x_2, y_2)$ at time t, i.e., it takes the edge

$e_1 = ((l_1, t), (l_2, t+1))$. Consider another agent $a_2 \in \mathcal{A}$, $a_2 \neq a_1$, with a set of edges

$$\begin{aligned}\mathcal{E}_2 = \{&((l_1, t), (l_1, t+1)),\\ &((l_1, t), ((x_1 - 1, y_1), t+1)),\\ &((l_1, t), ((x_1 + 1, y_1), t+1)),\\ &((l_1, t), ((x_1, y_1 - 1), t+1)),\\ &((l_1, t), ((x_1, y_1 + 1), t+1)),\\ &((l_2, t), (l_2, t+1)),\\ &(((x_2 - 1, y_2), t), (l_2, t+1)),\\ &(((x_2 + 1, y_2), t), (l_2, t+1)),\\ &(((x_2, y_2 - 1), t), (l_2, t+1)),\\ &(((x_2, y_2 + 1), t), (l_2, t+1)),\\ &((l_2, t), (l_1, t+1))\} \cap \mathcal{E}.\end{aligned}$$

The first five edges in $\mathcal{E}_2$ exit (l_1, t). The next five edges enter $(l_2, t+1)$. The last edge is the reverse of e_1. Note that some of the edges in $\mathcal{E}_2$ can be duplicates, which are removed because $\mathcal{E}_2$ is a set. All edges in $\mathcal{E}_2$ use the same timestep, and hence, are pairwise incompatible. The first five edges have a vertex conflict with e_1 at (l_1, t). The next five edges have a vertex conflict with e_1 at $(l_2, t+1)$. The final edge is incompatible with e_1 by the definition of an edge conflict. Using this reasoning, the exit-entry conflict constraint is

$$X_{e_1}^{a_1} + \sum_{e \in \mathcal{E}_2} X_e^{a_2} \leq 1. \tag{12}$$

Exit-entry conflict constraints were first presented in the conference paper (Lam and Le Bodic, 2020).

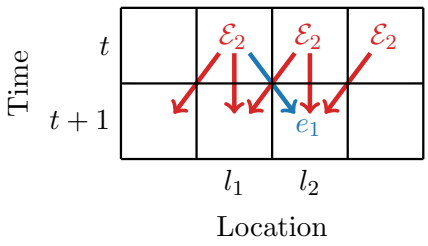

Figure 7: An exit-entry conflict.

4.3.10. Two-Edge Conflicts

Figure 8 illustrates three distinct locations l_1, l_2, l_3 such that l_1 and l_3 are neighbors of l_2. Consider an agent $a_1 \in \mathcal{A}$ fractionally using two edges $e_1 = ((l_1, t), (l_2, t+1))$ and $e_2 = ((l_2, t), (l_3, t+1))$. In an integer solution, a_1 can use at most one of these two edges since they occur at the same time. Denote their reverse edges as $e'_1 = ((l_2, t), (l_1, t+1))$ and $e'_2 = ((l_3, t), (l_2, t+1))$. If a_1 uses either e_1 or e_2, then another agent $a_2 \in \mathcal{A}$, $a_2 \neq a_1$, cannot simultaneously use e'_1 and e'_2. The edge e'_1 is incompatible with e_1 because of an edge conflict and incompatible with e_2 because of a vertex conflict at (l_2, t). For the same reasons, e'_2 is incompatible with e_1 and e_2. These ideas lead to the two-edge conflict constraint

$$X_{e_1}^{a_1} + X_{e_2}^{a_1} + X_{e'_1}^{a_2} + X_{e'_2}^{a_2} \leq 1. \tag{13}$$

Two-edge conflict constraints were first presented in the conference paper (Lam and Le Bodic, 2020).

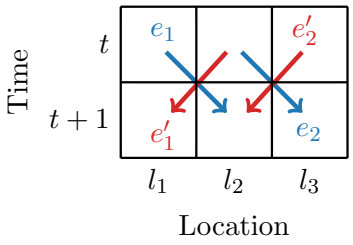

Figure 8: A two-edge conflict.

4.3.11. Wait-Two-Edge Conflicts

Wait-two-edge conflicts lift the two-edge conflicts with an additional wait edge, as shown in Figure 9. Consider an agent a_1 using the edges $e_1 = ((l_1, t), (l_2, t+1))$, $e_2 = ((l_2, t), (l_3, t+1))$ and $e_3 = ((l_2, t), (l_2, t+1))$. Also consider another agent a_2 using $e'_1 = ((l_2, t), (l_1, t+1))$, $e'_2 = ((l_3, t), (l_2, t+1))$ and e_3.

Agent a_1 taking edge e_3 is incompatible with agent a_2 taking edges e'_1 because of a vertex conflict at (l_2, t) and e'_2 and e_3 because of a vertex conflict at $(l_2, t+1)$. Using similar reasoning, all three edges for agent a_1 are incompatible with the three edges of agent a_2. This leads to the constraint

$$X_{e_1}^{a_1} + X_{e_2}^{a_1} + X_{e_3}^{a_1} + X_{e'_1}^{a_2} + X_{e'_2}^{a_2} + X_{e_3}^{a_2} \leq 1. \tag{14}$$

If wait-two-edge conflict constraints are included, the regular two-edge conflict constraints can be omitted. Wait-two-edge conflict constraints are presented here for the first time.

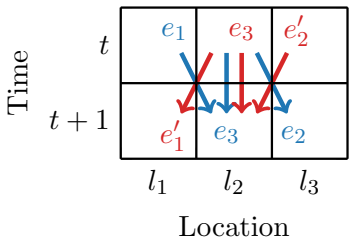

Figure 9: A wait-two-edge conflict.

4.3.12. Rectangle Conflicts

The previous eight classes of conflicts attempt to tighten the master problem by summing combinations of Constraints (5) and (6) within one constraint. In contrast, rectangle conflicts reason directly about the MAPF structure, and therefore, are applicable in other algorithms such as CBS.

In fact, rectangle conflicts are originally developed for CBS (Li et al., 2019). The idea is that two agents entering and exiting a rectangle at precisely the right time must sustain a vertex conflict somewhere within the rectangle. Consider two agents a_1 and a_2 entering the gray rectangle in Figure 10. If the two agents use a single-agent-optimal path (i.e., a path without waiting), the agents will always conflict somewhere within the rectangle. Let the (time-indexed) edges of the two sides used by a_1 (the left and right sides in Figure 10) be $\mathcal{E}_1 \subset \mathcal{E}$. Let the (time-indexed) edges of the two sides used by a_2 (the top and bottom sides in Figure 10) be $\mathcal{E}_2 \subset \mathcal{E}$. The edges $\mathcal{E}_1$ and $\mathcal{E}_2$ are timed precisely so that the two agents will conflict somewhere inside the rectangle. Then, rectangle conflicts are implemented using the constraint

$$\sum_{e \in \mathcal{E}_1} X_e^{a_1} + \sum_{e \in \mathcal{E}_2} X_e^{a_2} \leq 3. \tag{15}$$

Constraint (15) stipulates that at most three of the four sides can be used at those exact timesteps. Note that the constraint is valid even when any of the locations in the rectangle are obstacles. The correctness of this constraint relies on the proof given by Li et al. (2019). Rectangle conflict constraints were adapted to BCP in the original conference paper (Lam et al., 2019).

4.3.13. Step-aside Conflicts

Figure 11 shows a corridor of unit width and length $h = 4$ drawn in gray. The locations of the two ends are denoted l_1 and l_4. If agent a_1 needs to get from l_1 to l_4 and agent a_2 needs to get from l_4 to l_1, then one of the two agents must step out of the corridor to let the other pass then step back in, incurring a cost of at least two

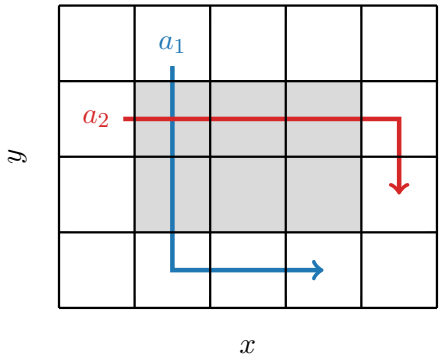

Figure 10: A rectangle conflict.

extra movements on top of its direct path. In other words, the agent cannot take the direct path nor take a path with one extra movement.

More formally, a_1 crosses from l_1 to l_2 at time t or $t+1$, and then exits the corridor from l_3 to l_4 at time $t+h$ or $t+h+1$. (If a_1 enters at t and exits at $t+h+1$, then it waited for one timestep inside the corridor.) Let the edges of these crossings be $e_1^1 = ((l_1, t), (l_2, t+1))$, $e_2^1 = ((l_1, t+1), (l_2, t+2))$, $e_3^1 = ((l_3, t+h), (l_4, t+h+1))$ and $e_4^1 = ((l_3, t+h+1), (l_4, t+h+2))$.

Agent a_2 enters the corridor from the opposite end l_4 at time t or $t+1$ and exits from l_2 to l_1 at time $t+h$ or $t+h+1$. Let the edges of a_2 be $e_1^2 = ((l_4, t), (l_3, t+1))$, $e_2^2 = ((l_4, t+1), (l_3, t+2))$, $e_3^2 = ((l_2, t+h), (l_1, t+h+1))$ and $e_4^2 = ((l_2, t+h+1), (l_1, t+h+2))$.

These four crossings cannot occur together because the two agents will collide somewhere within the corridor. Hence, at most three of these eight edges can occur. This condition is enforced by the constraint

$$X^{a_1}_{e_1^1} + X^{a_1}_{e_2^1} + X^{a_1}_{e_3^1} + X^{a_1}_{e_4^1} + X^{a_2}_{e_1^2} + X^{a_2}_{e_2^2} + X^{a_2}_{e_3^2} + X^{a_2}_{e_4^2} \leq 3. \tag{16}$$

Step-aside conflicts are new to this journal paper.

4.3.14. Goal Conflicts

All the previous classes of conflicts reason about a set of incompatible edges traversed by a set of agents. The constraints prohibiting these conflicts are expressed as a sum over agents and edges. In contrast, goal conflicts reason about whole paths and hence their constraints cannot be expressed as a sum over edges.

Figure 12 shows two agents involved in a goal conflict. Agent $a_{\text{pass}} \in \mathcal{A}$ is (fractionally) passing through the goal location $l := g_{a_{\text{goal}}} \in \mathcal{L}$ of another agent $a_{\text{goal}} \in \mathcal{A}$ at some time t after a_{goal} has already (fractionally) reached its goal. Goal

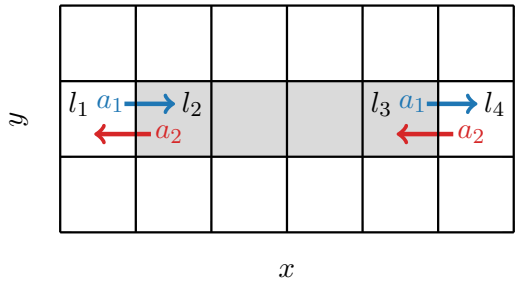

Figure 11: A step-aside conflict.

conflicts are resolved using the constraint

$$\sum_{p \in \mathcal{P}_{a_{\text{goal}}}} W_t^p \lambda_p + \sum_{p \in \mathcal{P}_{a_{\text{pass}}}} Q_{l,t}^p \lambda_p \leq 1, \tag{17}$$

where W_t^p takes value 1 if path p finishes at time t or earlier, and takes value 0 otherwise, and $Q_{l,t}^p$ takes value 1 if path p visits location l at time t or later, and takes value 0 otherwise. This constraint restricts the solution so that either a_{goal} reaches its goal location l at or before time t and remains there, or a_{pass} passes through location l at or after time t, or neither. At most one of these two events can occur since a_{goal} will wait indefinitely at l if it reaches its goal location. Goal conflict constraints were first presented in the conference paper (Lam and Le Bodic, 2020).

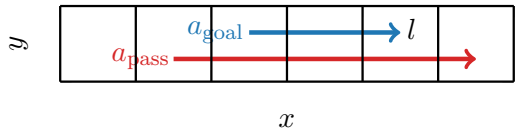

Figure 12: A goal conflict.

4.4. Finding Lower Cost Paths

The set $\mathcal{P}_a$ of candidate paths for every agent $a \in \mathcal{A}$ is dynamically filled by the pricer. The pricer solves the pricing problem of every agent $a \in \mathcal{A}$ to either find one or more paths that may appear in a solution better than the current solution to the master problem, or prove that an improving path does not exist.

4.4.1. The Pricing Problem

According to well-known results from linear programming, any variable that appears in a (future) solution with a cost lower than the current solution must have

negative *reduced cost*. In BCP, the reduced cost of a variable measures the cost-effectiveness of an agent using the resources (i.e., vertex, edge or goal location) of a path compared against all other paths using the same resource in the current solution of the master problem. The pricing problem is a resource-constrained shortest path problem on the reduced cost instead of the regular unit cost per edge. For each agent $a \in \mathcal{A}$ in turn, the pricer calls the A* algorithm to find a path with negative reduced cost, and if one exists, adds it to $\mathcal{P}_a$, so that the master problem may select it in the next iteration. If such a path cannot be found for any agent, the current solution to the master problem is optimal.

There are four main considerations for defining the reduced cost. Firstly, according to Constraint (1b), every agent must use exactly one path. This constraint can be viewed as restricting each agent to use exactly one "path resource". Let $\pi_a \in \mathbb{R}$ be the dual variable of Constraint (1b) for agent a. As every path will consume one "path resource", the reduced cost of every path is penalized by $-\pi_a$.

Secondly, consider the constraints for all classes of conflicts except the goal conflicts (i.e., Constraints (5) to (16)). Let $\mathcal{I}$ denote the set of these constraints that currently exist in the master problem. Every conflict constraint $i \in \mathcal{I}$ can be written in the form

$$\sum_{a \in \mathcal{A}} \sum_{e \in \mathcal{E}} \alpha_{i,e}^a X_e^a \leq \beta_i, \tag{18}$$

where $\alpha_{i,e}^a \geq 0$ and $\beta_i \geq 0$ are constants. Let $\pi_i \leq 0$ be the dual variable of constraint $i \in \mathcal{I}$ as defined by linear programming theory.

If a path p traverses an edge e, this action must be penalized by the value of the dual variable of all constraints that include e. Formally, every edge $e \in \mathcal{E}$ has reduced cost

$$1 + \sum_{i \in \mathcal{I}} -\pi_i \alpha_{i,e}^a. \tag{19}$$

Notice that the summation is non-negative because every $\pi_i \leq 0$ and $\alpha_{i,e}^a \geq 0$. Hence, the summation can be interpreted as a cost penalty for the agent using edge e compared against every other path (belonging to the same agent or a different agent) using the same edge in the current solution of the master problem. Adding more of these conflict constraints to the master problem simply adds more penalties to summation and does not impact the A* algorithm in any other way.

Thirdly, consider a path of length k, as defined in Section 2. Using this path means that agent a departs its start location s_a at time 0 (i.e., the path starts at the vertex $(s_a, 0)$), reaches its goal location g_a at time $k-1$ (i.e., the path ends at the vertex $(g_a, k-1)$) and then remains at its goal location indefinitely. However, at some time $t \geq k$, another agent may still be in motion and may want to cross the

goal location g_a of agent a. Hence, it is not sufficient to plan a path for agent a from $(s_a, 0)$ to $(g_a, k-1)$ because a vertex conflict at (g_a, t) may occur (i.e., after the path ends at time $k-1$).

To handle this situation, the pricing problem operates on an auxiliary graph $\mathcal{G}_\perp = (\mathcal{V}_\perp, \mathcal{E}_\perp)$. Let $\perp$ be a dummy goal vertex that is accessible only from the actual goal location g_a. Define the set of vertices $\mathcal{V}_\perp = \mathcal{V} \cup \{\perp\}$ and the set of edges $\mathcal{E}_\perp = \mathcal{E} \cup \{((g_a, t), \perp) : t \in \mathcal{T}\}$. The edge $((g_a, t), \perp)$ is used to indicate that the agent reaches its goal location at time t and then waits indefinitely. The pricing problem attempts to find a path on $\mathcal{G}_\perp$ with negative reduced cost from vertex $(s_a, 0)$ to vertex $\perp$. For every $t \in \mathcal{T}$, the edge $((g_a, t), \perp)$ has reduced cost

$$\sum_{i \in \mathcal{I}_{(g_a,t)}} -\pi_i$$

where $\mathcal{I}_{(g_a,t)} \subset \mathcal{I}$ are the constraints that include any wait edge $((g_a, t'), (g_a, t'+1))$ where $t' \in \{0, \ldots, t\}$. All penalties due to finishing at time t and then waiting at the goal location indefinitely are penalized solely using the edge $((g_a, t), \perp)$.

It is easy to see this is valid using an example. For some time t, consider two vertex conflict constraints at $(g_a, t-1)$ and (g_a, t) with dual variables π_1 and π_2. According to Constraint (5), these two constraints include the incoming edges $((g_a, t-2), (g_a, t-1))$ and $((g_a, t-1), (g_a, t))$ respectively. Should the pricing problem find a path that ends at $(g_a, t-2)$, it will not incur the penalty $-\pi_1$ because the path never traverses $((g_a, t-2), (g_a, t-1))$, even though the agent waits at g_a. Hence, $-\pi_1$ must be added to the reduced cost of edge $((g_a, t-2), \perp)$. Similarly, the penalty $-\pi_2$ is also added to the reduced cost of $((g_a, t-2), \perp)$ because the path does not traverse $((g_a, t-1), (g_a, t))$. In contrast, should the pricing problem find a path that ends at (g_a, t), the penalty $-\pi_1$ is not added to any edge with $\perp$ because, to reach (g_a, t), the agent must have used some edge leading to (g_a, t), which may or may not be $((g_a, t-1), (g_a, t))$. If it is $((g_a, t-1), (g_a, t))$, then $-\pi_2$ is paid using Equation (19).

The final consideration is the goal conflict constraints. Goal conflicts reason about the occurrence of events along a path, rather than about a set of edges. There is currently no mechanism in the pricing problem to penalize the occurrence of the events corresponding to $W_t^p = 1$ and $Q_{l,t}^p = 1$. Hence, the A* algorithm must be modified to handle goal conflict constraints.

Let $\mathcal{J}$ be the set of goal conflict constraints (Constraint (17)) existing in the master problem and let $\mu_j \leq 0$ be the dual variable of constraint $j \in \mathcal{J}$. When pricing a_{pass}, the path must be penalized by $-\mu_j \geq 0$ if it visits the goal location l of constraint $j \in \mathcal{J}$ at or after time t. The penalty is incurred exactly once, no matter how many times the path visits l after t. Therefore, placing a penalty on the five

incoming edges to (l, t') for all $t' \geq t$ is not correct because entering and exiting l repeatedly will penalize the path multiple times. Instead, a state σ_j, initialized to 0, is introduced to the A* algorithm to track the occurrence of the event and penalize the path once if it visits l at time t or later. Upon expanding the vertex (l, t') for any $t' \geq t$, the penalty is incurred and σ_j is set to 1 to indicate that the penalty has been paid. Future expansions through (l, t'') for any $t'' > t'$ will not incur the penalty again because $\sigma_j = 1$. When pricing a_{goal}, all edges leading to $\perp$ at or before time t incurs the penalty $-\mu_j$.

In consideration of these four points, the reduced cost $\bar{c}_p$ of a path p with length k can now be defined. That is, $\bar{c}_p$ is the total reduced cost of p including all relevant penalties. Formally,

$$\bar{c}_p = c_p - \pi_a - \sum_{e \in p} \left(1 - \sum_{i \in \mathcal{I}} \pi_i \alpha_{i,e}^a\right) - \sum_{i \in \mathcal{I}_{(g_a, k-1)}} \pi_i - \sum_{j \in \mathcal{J}: a = a_{\text{pass}}} \mu_j Q_{l,t}^p - \sum_{j \in \mathcal{J}: a = a_{\text{goal}}} \mu_j W_t^p,$$

where the first summation is over the edges in path p. From linear programming theory, p may appear in a better solution if $\bar{c}_p < 0$. If so, p is added to $\mathcal{P}_a$, leading to another round of solving the master problem, pricing new paths and resolving conflicts. If the pricing problem cannot find a path p with $\bar{c}_p < 0$ for any agent $a \in \mathcal{A}$, then no path can improve upon the current master problem solution, which is declared (fractionally) optimal. The reduced costs allows agents to cooperate to decrease the total cost. An agent can use a resource (vertex, edge or goal location) in use by another agent if it overall gains no less than the other agents lose.

For a partial path p ending at vertex $v \in \mathcal{V}_\perp$ with reduced cost $\bar{c}_p$, the A* algorithm requires a function, called the heuristic $h(v)$, that computes a lower bound on the cost to-go from the current vertex v to $\perp$. We use a heuristic based on two segments: from vertex v to the goal location g_a and then from g_a to $\perp$. For any $v = (l, t)$, define $h(l)$ as the the distance of the non-time-indexed shortest path from l to g_a, which is precomputed. As the distance is equal to time, $h(l)$ also gives the minimum number of timesteps to reach g_a from l. Next, define $h_\perp(t) = \min_{t' \in \{t, \ldots, T-1\}} \{t' - t + \sum_{i \in \mathcal{I}_{(g_a, t')}} -\pi_i\}$ as a lower bound on the cost of waiting at g_a from time t to time t' and then entering $\perp$ at time t'. We use $h(v) = h(l) + h_\perp(t + h(l))$, which consists of moving from l to the goal location g_a and then waiting to enter the dummy end location until the time that minimizes the penalties for waiting at the goal location (if any). Then, a lower bound f_p on the total reduced cost of p (i.e., from the start location to $\perp$) is given by $f_p = \bar{c}_p + h(v)$, comprising the reduced cost from the starting location to the current vertex plus a lower bound on the cost from the current vertex to the end.

In the ordinary A* algorithm (outside BCP), if two partial paths p_1, p_2 end at a common vertex v with lower bounds f_1, f_2 on their total cost and $f_1 \leq f_2$, then p_1 is said to dominate p_2 and p_2 can be discarded because every extension of p_2 to the goal cannot be better than the same extension of p_1. (In the case of $f_1 = f_2$, then one of the two paths must be kept.) In BCP, because the goal conflict constraints require states σ_j to track whether a penalty has been paid, the dominance condition must also consider these states.

Figure 13a shows two partial paths ending at (B,2) and their costs at each step along the path. There is an edge penalty of 0.5 for moving from B to A at time 0. There is also a goal conflict penalty of 1 for passing through location C at or after time 1. The top (blue) partial path has cost 2.5, which includes the edge penalty. The bottom (red) partial path has cost 3, which includes the goal penalty. According to the usual dominance rule, the bottom path has higher cost than the top path and can be discarded. However, this is not correct when using goal conflict constraints. Figure 13b shows these two partial paths extended two steps to the goal at D. The bottom path has previously passed through the goal conflict location C at time 1 and has already paid the penalty whereas the top path has not. The top path now incurs the penalty and reaches (D,4) with cost 5.5. The bottom path reaches (D,4) with cost 5 and hence should not have been dominated at (B,2).

To correctly account for the goal conflicts, the following dominance rule is required:

$$f_1 + \sum_{\substack{j \in \mathcal{J}: \\ \sigma_j^1 = 0 \wedge \\ \sigma_j^2 = 1}} -\mu_j \leq f_2$$

where σ_j^1 indicates whether p_1 has passed the goal location of the goal conflict constraint $j \in \mathcal{J}$. Recall that $\mu_j \leq 0$ for all $j \in \mathcal{J}$. This dominance rule says that, even after path p_1 pays the penalty of all goal conflicts that it has not yet paid but that path p_2 has already paid, its cost is still lower than p_2, then it dominates p_2. This dominance rule is also used in branch-and-cut-and-price algorithms for other problems (e.g., Equation (39) of Jepsen et al. (2008)). According to this dominance rule, neither path in Figure 13a dominates the other and hence both must be extended towards the goal.

Because the pricing problem is solved using A* with an admissible heuristic, a path with negative reduced cost will be found if and only if it exists. Therefore, the pricer is correct and complete.

(a) Two partial paths ending at (B,2).

(b) The two partial paths extended two steps to D.

Figure 13: Two partial paths demonstrating the dominance condition for goal conflict constraints.

*4.4.2. Caching Solutions in A**

The pricer calls A* to generate paths repeatedly throughout the search tree. Even if an agent is not in conflict, the pricer needs to call A* on this agent to verify that changes to the selection of paths for the other agents do not lead to an improved path for this agent. This is very costly, especially if the agent is so far away from the conflicts that resolving these conflicts has no effect on the agent. We mitigate this issue using a caching procedure that recalls past solutions for this agent.

The A* search implementation is modified to mark all data it uses (the edge penalties and the penalties of the events of the goal conflict constraints) during the search. If A* fails to find a new path, these data are then stored in a database. In the next call to A* for the same agent, the new edge penalties and the penalties of the goal conflict constraints are queried in the database. If the penalties are the same or worse, then no improving path is possible for this agent, and hence, A* does not need to be called.

4.5. Resolving Fractionalities

The master problem is a linear programming problem, which often produces fractional solutions. For example, given an agent $a \in \mathcal{A}$ and two paths $p_1, p_2 \in \mathcal{P}_a$, we can have $\lambda_{p_1} = \lambda_{p_2} = 0.5$. Whenever the pricer declares that a solution with at least one variable taking a fractional value is optimal, branching must proceed to resolve the fractionality. Branching splits the current node in the search tree into two children nodes such that the current fractional solution appears in neither children.

4.5.1. Branching on Vertices

The first branching rule branches on an agent-vertex pair, stipulating that an agent $a \in \mathcal{A}$ must visit a vertex $v \in \mathcal{V}$ in one child and must not visit v in the other child. Using Equation (3) and the solution to the master problem, the branching rule first computes the number of times X_v^a that each vertex $v \in \mathcal{V}$ is visited by agent

$a \in \mathcal{A}$, and then calculates the number of times

$$X_v = \sum_{a \in \mathcal{A}} X_v^a$$

that each vertex $v \in \mathcal{V}$ is visited by all agents. Because branching occurs after all separators have declared that the solution to the master problem exhibits no conflicts, $X_v \leq 1$ for all $v \in \mathcal{V}$.

Next, the branching rule builds the set

$$\mathcal{A}_v = \{a \in \mathcal{A} : X_v^a > 0\}$$

of agents visiting each vertex $v \in \mathcal{V}$. The branching rule then selects a vertex

$$v^* = \underset{v=(l,t) \in \mathcal{V}}{\arg\min} \{t : 0 < X_v < 1 \wedge |\mathcal{A}_v| \geq 2\}$$

that has fractional value and is used by two or more agents, favoring the vertex with the earliest time. Next, it selects an agent

$$a^* = \underset{a \in \mathcal{A}_{v^*}}{\arg\min} \{c_p : p \in \mathcal{P}_a \wedge \lambda_p > 0\}$$

that is (fractionally) using a path visiting v^* and has a path of the shortest length among all used paths visiting v^*. The branching rule then creates two children nodes. In one child, agent a^* must visit v^*. In the other child, a^* cannot visit v^*.

The decisions made by branching must be enforced in the master problem and the pricing problem of each child. If v^* cannot be visited, all paths that visit v^* are disabled in the master problem and the five incoming edges to v^* are removed in the pricing problem, preventing new paths from visiting v^*.

If v^* must be visited, the goal in the A* search for agent a^* is set to the vertex v^*, instead of any vertex associated with the actual goal location g_a. Now the pricer will find a partial path from the starting vertex $(s_a, 0)$ to v^*. Next, the pricer is called again to find a partial path from v^* to any vertex associated with the goal location g_a. Since every vertex is associated with a time, the mandatory vertex v^* of all branching decisions can be ordered in a sequence, and A* is called for every adjacent pair of vertices in the sequence. Upon completion, all partial paths are concatenated to form a path from s_a to g_a via detours to all mandatory vertices v^*.

4.5.2. Branching on Path Length

Bounding the length of all paths for an agent at a particular node in the search tree bounds its cost in the entire subtree below. In particular, fixing the length of all

paths for an agent fixes its cost, regardless of the vertices it visits. In contrast, fixing a vertex to be used or unused only indirectly affects the cost by rerouting the agent. Therefore, it is beneficial to fix path lengths early in the search. BCP employs a tiered branching rule that first branches on path length and only branches on vertices (as in Section 4.5.1) once every path used by an agent has the same length.

The second branching rule begins by calculating the set

$$\mathcal{S} = \{a \in \mathcal{A} : \exists p_1, p_2 \in \mathcal{P}_a, \lambda_{p_1} > 0, \lambda_{p_2} > 0, c_{p_1} \neq c_{p_2}\}$$

of agents fractionally using paths with different costs (i.e., different lengths). From $\mathcal{S}$, the branching rule chooses an agent a and a path p of smallest cost that is fractionally used, i.e.,

$$(a^*, p^*) = \underset{a \in \mathcal{S}, p \in \mathcal{P}_a}{\arg\min}\{c_p : \lambda_p > 0\}.$$

It then creates two children nodes. In the left child, a^* only uses paths with cost less than or equal to c_{p^*}. In the right child, a^* only uses paths with cost greater than or equal to $c_{p^*} + 1$.

After making one of these decisions, it must be enforced in the master problem and pricing problem. In the case of requiring the cost of all paths of agent a^* to be less than or equal to c_{p^*}, the goal in the A* search is set to any vertex $(g_a, t) \in \mathcal{V}$ where $t \in \{0, \ldots, c_{p^*}\}$. (Recall from Section 2 that the length is the number of vertices whereas the cost is the number of edges.) Using the heuristic in A*, all expansions with a time greater than c_{p^*} are also discarded and not added to the priority queue for future expansion. A similar process occurs in the case of requiring paths whose cost is greater than or equal to $c_{p^*} + 1$. The goal is set to any vertex $(g_a, t) \in \mathcal{V}$ where $t \in \{c_{p^*} + 1, \ldots, T - 1\}$.

After each branching decision, the A* search will be required to find paths whose cost lies within increasingly smaller intervals. BCP switches to the vertex branching rule described in Section 4.5.1 when $\mathcal{S} = \emptyset$, signifying that all agents are using paths of the same length.

Recall that BCP uses fractional solutions (i.e., relaxation) to find lower bounds. At any node in the branch-and-bound search tree, if the node lower bound is higher than the global upper bound, the node is pruned, regardless of whether the solution is fractional, because solutions in the subtree below can only have the same or higher cost. The left child of this branching rule drives the search towards paths of shortest length. The right child coerces the lower bound up. In conjunction, this branching rule aims to quickly find good solutions (i.e., tight upper bounds) in the left child and then prove their optimality by proving suboptimality in the right child.

5. Experiments

This section evaluates the empirical performance of BCP in two experiments. BCP uses SCIP 7.0.3 (Gamrath et al., 2020) for the integer programming branch-and-bound tree search, Gurobi 9.1 for solving the linear programming master problem and a custom implementation of the A* algorithm for the pricing problem. The source code is available online.[1]

The experiments are run on an Intel Xeon Platinum 8260 CPU at 2.4 GHz with a time limit of five minutes and a memory limit of 8 GB. All algorithms are single-threaded.

5.1. Comparison Against Other Algorithms

The first experiment evaluates two variants of BCP against two other leading MAPF algorithms. The first variant of BCP is a minimally-working baseline, which only contains the vertex conflict constraints (Section 4.3.2), edge conflict constraints (Section 4.3.3) and vertex branching (Section 4.5.1). The second variant is the full algorithm, comprising all of the improvements detailed in Section 4. Specifically, it includes all valid inequalities (Sections 4.3.4 to 4.3.12 and 4.3.14), A* solution caching (Section 4.4.2) and path length branching Section 4.5.2. The two versions of BCP are compared against CBSH2-RTC,[2] a new variant of CBS that adds generalized rectangle reasoning and generalized corridor reasoning, and the latest version of Lazy CBS,[3] which includes rectangle reasoning and enhancements to its underlying constraint programming solver.

The algorithms are evaluated on two sets of standard benchmarks. The first contains 670 instances across two maps representative of warehouses.[4] There are between 10 and 30 instances for any given number of agents. The second consists of 3760 instances across 14 maps from the Moving AI repository (Sturtevant, 2019), consisting of 10 instances for any given number of agents. This collection includes many maps. Each map contains scenarios categorized as `even` or `random` according to the distribution of the start and goal positions. Due to the large number of instances, the algorithms are compared using the `random` instances from a subset of 14 maps selected to include a wide variety of structures. This experiment spans 4430 instances over 16 maps in total.

[1] `https://github.com/ed-lam/bcp-mapf`

[2] `https://github.com/Jiaoyang-Li/CBSH2-RTC`

[3] `https://bitbucket.org/gkgange/lazycbs/`

[4] The warehouse instances are provided by Jiaoyang Li.

Figure 14 plots the percentage of solved instances against the number of agents. The percentage is computed across a varying number of instances for the warehouse maps but exactly 10 instances for the Moving AI maps. The improvements to BCP clearly enable substantially better performance compared to the baseline algorithm. Therefore the remainder of this section focuses on the full algorithm.

BCP mostly outperforms Lazy CBS and CBSH2-RTC on the two warehouse maps, losing to CBSH2-RTC on two instances of `10x30-w5` with 24 agents and to Lazy CBS on one instance of `31x79-w5` with 48 agents. BCP solves 634 of the 670 warehouse instances, compared to 559 by Lazy CBS and 593 by CBSH2-RTC. CBSH2-RTC has improved substantially since the last evaluation almost two years ago (Lam and Le Bodic, 2020), when it performed poorly on the warehouse maps and could not solve any of the `31x79-w5` instances with 52 agents.

Lazy CBS has also made massive improvements since the last comparison. On the two city maps `Berlin_1_256` and `Paris_1_256`, Lazy CBS is now pushing beyond 300 agents. BCP is behind on `Berlin_1_256` but slightly ahead on `Paris_1_256`. In total, BCP solves 504 instances of the city maps, while Lazy CBS solves 510 and CBSH2-RTC solves 406.

The performance of the algorithms vary drastically on the `brc202d`, `orz900d`, `w_woundedcoast`, `ost003d` and `lt_gallowstemplar_n` maps from the Dragon Age series of games. BCP dominates on these five game maps, sometimes quite significantly. Lazy CBS struggles on the first three of these five maps, which contain narrow hallways, and has regressed on `orz900d`, where it previously solved several instances with 10 or fewer agents. Overall, BCP, Lazy CBS and CBSH2-RTC respectively solve 592, 277 and 364 instances of the 1500 instances.

The next four maps are structured. The first two are empty 8×8 and 32×32 grids. In the last two maps, 10% and 20% of the passable cells in `empty-32-32` are replaced with obstacles. On `empty-8-8`, where contention is high, BCP solves almost all instances. On the larger `empty-32-32`, Lazy CBS completely dominates BCP and CBSH2-RTC. As more obstacles are added, its lead drops considerably, losing to BCP with 20% obstacles. Of the 1060 instances on these four maps, BCP solves 491 instances, Lazy CBS solves 551 instances and CBSH2-RTC solves 415 instances.

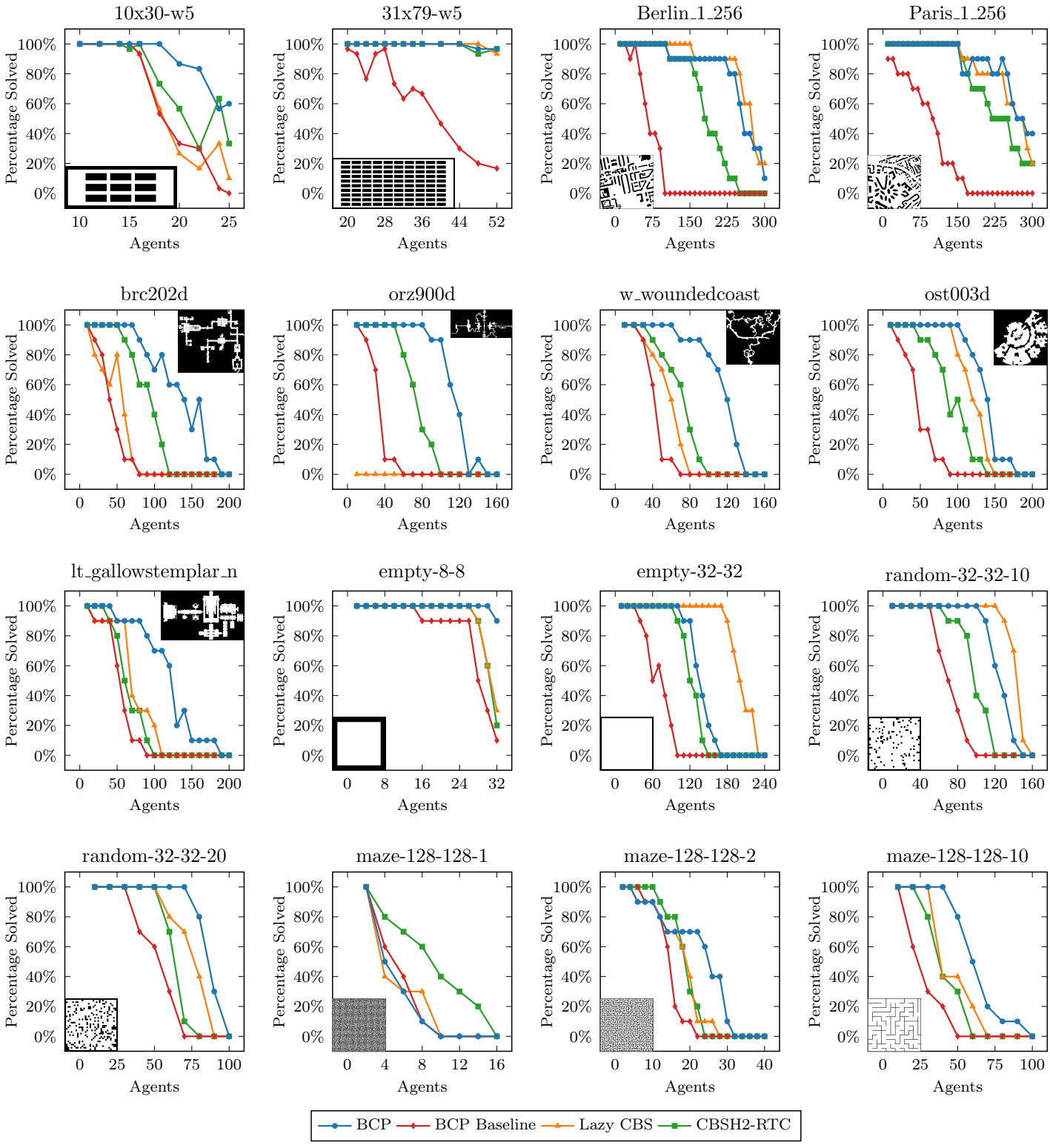

Figure 14: Success rate of the algorithms by map. Higher is better.

The last three maps are mazes with a corridor width of 1, 2 and 10 cells respectively. Last year, all three algorithms perform similarly poorly on the extremely difficult `maze-128-128-1`. Since then, CBSH2-RTC has improved significantly and now vastly outperforms BCP and Lazy CBS due to the introduction of corridor-target symmetry handling. On the maze with corridor width 2, CBSH2-RTC performs best with fewer agents whereas BCP scales better to more agents. BCP is superior on the maze with corridor width 10. In total, BCP, Lazy CBS and CBSH2-RTC respectively solve 181, 142 and 161 of the 600 maze instances.

All three algorithms clearly scale to high numbers of agents considered out-of-reach just a few years ago but none of the algorithms closes the MAPF problem. Overall, BCP retains its lead on Lazy CBS and CBSH2-RTC by solving 2402 of the 4430 instances, compared to 2039 by Lazy CBS and 1939 by CBSH2-RTC. The gap between the three approaches is closing, particularly because new developments in one technique are later ported to other algorithms.

A few general conclusions can be drawn from these experiments. BCP maintains consistent high-performance across all maps. It is competitive with the other two world-leading algorithms Lazy CBS and CBSH2-RTC in all maps and besting them in most cases. The performance of Lazy CBS highly depends on the structure of the map. On the sparse maps (`Berlin_1_256`, `Paris_1_256` and `empty-32-32`), Lazy CBS performs exceedingly well, matching BCP despite being much simpler. In contrast, on the maps with high contention due to small spaces or long corridors, Lazy CBS sometimes performs poorly (`10x30-w5` and `brc202d`) or even fails catastrophically (`orz900d`); although it is competitive on several of these maps (`w_woundedcoast`, `maze-128-128-1` and `maze-128-128-2`). CBSH2-RTC has improved significantly since the previous comparison when it greatly lagged behind BCP for all maps except `maze-128-128-1` and `maze-128-128-2` where it matched BCP. CBSH2-RTC is now unchallenged on `maze-128-128-1`.

5.2. Comparison of the Improvements

The second experiment compares the impact of each improvement described in Section 4. Each of the thirteen improvements is removed from the full BCP algorithm in turn. Due to the large number of configurations, this experiment is run on half of the 4430 instances (2215 instances in total).

Table 1 shows the number of instances solved and the mean run time for each configuration. The greatest degradations come from omitting the path length branching, rectangle conflict constraints and goal conflict constraints. These three additions directly attack the MAPF problem structure by bringing new reasoning, whereas the other conflict constraints simply tighten the master problem by merging different

Configuration	Instances Solved		Average Time	
Full algorithm	1188		149	
Wait-edge conflicts	1187	(-1)	149	(+0)
Two-agent wait-edge conflicts	1185	(-3)	149	(+1)
Corridor conflicts	1184	(-4)	150	(+1)
Wait-corridor conflicts	1183	(-5)	150	(+1)
Wait-delay conflicts	1188	(-0)	149	(+1)
Exit-entry conflicts	1126	(-62)	156	(+7)
Two-edge conflicts	1171	(-17)	151	(+2)
Wait-two-edge conflicts	1178	(-10)	151	(+2)
Rectangle conflicts	896	(-292)	184	(+36)
Step-aside conflicts	1187	(-1)	149	(+0)
Goal conflicts	1000	(-188)	176	(+27)
A* caching	1165	(-23)	151	(+2)
Path length branching	889	(-299)	184	(+35)

Table 1: Number of solved instances and average run time after removing each improvement.

combinations of the vertex conflict constraints and edge conflict constraints.

The remaining constraints make a much smaller contribution. This finding departs from preliminary experiments. During their development, all thirteen improvements were evaluated on a small set of instances and were shown to increase the number of instances being solved. Ironically, recent improvements to the core BCP code (i.e., more efficient data structures, better tuning of parameters, improved primal heuristics, etc.) have substantially diminished the impact of many valid inequalities, which were crucial to performance in earlier work (Lam and Le Bodic, 2020). (Five other valid inequalities were also developed but had no impact or negative impact on performance. These valid inequalities are not reported in this paper but remain accessible in the public code.)

6. Conclusion and Future Work

This paper presents BCP, a state-of-the-art exact algorithm for MAPF that uses the branch-and-cut-and-price framework from mathematical optimization. BCP decomposes the MAPF problem into a master problem that selects one low-cost path for every agent, a pricing problem that generates lower-cost paths for the agents and separation problems that each resolves a different class of conflicts. A minimally-working baseline algorithm is augmented with eleven classes of valid inequalities that dramatically tighten the master problem, a branching rule that directly targets the cost function and a caching technique that reduces the number of calls to the A* shortest path algorithm, where BCP spends the majority of its run time. A further five classes of constraints are also developed but they do not improve performance in preliminary experiments and hence are not reported here.

Empirical results show that BCP displays exceptional performance, sometimes substantially outperforming the two other state-of-the-art algorithms CBSH2-RTC and Lazy CBS. BCP, Lazy CBS and CBSH2-RTC respectively solve 2402, 2039 and 1939 of 4430 instances across 16 maps. These results demonstrate that close collaboration between the mathematical optimization and artificial intelligence communities can yield significant advances.

The experiments also reveal that BCP and CBSH2-RTC exhibit consistent high-performance across all maps, whereas the performance of Lazy CBS is highly dependent on the map structure. Lazy CBS dominates on maps with wide open spaces but fails on maps with long, narrow corridors or choke points. The reasons for these behaviors remain an open question. One hypothesis is that the open nature of the `empty-32-32` and `random-32-32-10` maps leads to many more repeated failures, which Lazy CBS handles well. Consider two agents at the top of the map in conflict and two agents at

the bottom of the map in conflict. In BCP and CBSH2-RTC, every time the conflict between the top pair of agents is resolved at an adjacent location, the conflict for the bottom pair must also be resolved. In Lazy CBS, the conflict for the bottom pair is resolved once, recorded and never seen again, dramatically reducing the amount of computation.

BCP is an anytime algorithm. It can find a sequence of improving solutions before finally proving optimality. However, it is not yet developed for this situation. Future work can consider tuning its numerous parameters to improve its anytime behavior to the detriment of proving optimality.

The foundation of Lazy CBS is its conflict analysis technique from constraint programming. Recent collaborations by the inventors of BCP and Lazy CBS unified the conflict analysis procedure from constraint programming with the strong lower bounds from integer programming (Lam and Van Hentenryck, 2017; Davies et al., 2017; Lam et al., 2020). Future work should attempt to merge BCP and Lazy CBS.

Funding Sources

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Declarations of Interest

None.

References

Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.P., Vance, P.H., 1998. Branch-and-price: Column generation for solving huge integer programs. Operations Research 46, 316–329.

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

Davies, T.O., Gange, G., Stuckey, P.J., 2017. Automatic logic-based Benders decomposition with MiniZinc, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI), pp. 787–793.

Desaulniers, G., Desrosiers, J., Solomon, M.M., 2005. Column Generation. Springer US.

Desrosiers, J., Lübbecke, M.E., 2010. Branch-price-and-cut algorithms, in: Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Inc.

Erdem, E., Kisa, D.G., Oztok, U., Schüller, P., 2013. A general formal framework for pathfinding problems with multiple agents, in: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI), pp. 290–296.

Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, S., Koenig, S., 2018. Adding heuristics to conflict-based search for multi-agent path finding, in: Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS), pp. 83–87.

Felner, A., Stern, R., Shimony, E., Goldenberg, M., Sharon, G., Sturtevant, N., Wagner, G., Surynek, P., 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges, in: Proceedings of the Symposium on Combinatorial Search (SoCS), pp. 28–37.

Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Le Bodic, P., Maher, S.J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., Witzig, J., 2020. The SCIP Optimization Suite 7.0. Technical Report. Optimization Online. `http://www.optimization-online.org/DB_HTML/2020/03/7705.html`.

Gange, G., Harabor, D., Stuckey, P., 2019. Lazy CBS: Implict conflict-based search using lazy clause generation, in: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS), pp. 155–162.

Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N.R., Holte, R.C., Schaeffer, J., 2014. Enhanced partial expansion A*. Journal of Artificial Intelligence Research 50, 141–187.

Jepsen, M., Petersen, B., Spoorendonk, S., Pisinger, D., 2008. Subset-row inequalities applied to the vehicle-routing problem with time windows. Operations Research 56, 497–511.

Lam, E., Gange, G., Stuckey, P.J., Van Hentenryck, P., Dekker, J.J., 2020. Nutmeg: A MIP and CP hybrid solver using branch-and-check. SN Operations Research Forum 1, 22.

Lam, E., Le Bodic, P., 2020. New valid inequalities in branch-and-cut-and-price for multi-agent path finding, in: Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS), pp. 184–192.

Lam, E., Le Bodic, P., Harabor, D., 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), International Joint Conferences on Artificial Intelligence Organization. pp. 1289–1296.

Lam, E., Van Hentenryck, P., 2017. Branch-and-check with explanations for the Vehicle Routing Problem with Time Windows, in: Principles and Practice of Constraint Programming: 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings, Springer, Cham. pp. 579–595.

Letchford, A.N., Salazar-González, J.J., 2006. Projection results for vehicle routing. Mathematical Programming 105, 251–274.

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

Li, J., Harabor, D., Stuckey, P.J., Ma, H., Koenig, S., 2019. Symmetry-breaking constraints for grid-based multi-agent path finding, in: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI), pp. 6087–6095.

Lübbecke, M.E., Desrosiers, J., 2005. Selected topics in column generation. Operations Research 53, 1007–1023.

Ma, H., Koenig, S., Ayanian, N., Cohen, L., Hoenig, W., Kumar, T.K.S., Uras, T., Xu, H., Tovey, C., Sharon, G., 2017. Overview: Generalizations of multi-agent path finding to real-world scenarios. https://arxiv.org/abs/1702.05515. `arXiv:1702.05515`.

Marques Silva, J.a.P., Sakallah, K.A., 1996. GRASP–a new search algorithm for satisfiability, in: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design, IEEE Computer Society. pp. 220–227.

Ohrimenko, O., Stuckey, P.J., Codish, M., 2009. Propagation via lazy clause generation. Constraints 14, 357–391.

Rader, D.J., 2010. Deterministic operations research: models and methods in linear optimization. John Wiley & Sons.

Ryan, M., 2010. Constraint-based multi-robot path planning, in: 2010 IEEE International Conference on Robotics and Automation, pp. 922–928.

Sharon, G., Stern, R., Felner, A., 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), pp. 173–178.

Sturtevant, N., 2019. Moving AI: Pathfinding benchmarks. https://movingai.com/benchmarks.

Surynek, P., Felner, A., Stern, R., Boyarski, E., 2016a. Boolean satisfiability approach to optimal multi-agent path finding under the sum of costs objective, in: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1435–1436.

Surynek, P., Felner, A., Stern, R., Boyarski, E., 2016b. Efficient SAT approach to multi-agent path finding under the sum of costs objective, in: Proceedings of the European Conference on Artificial Intelligence (ECAI), pp. 810–818.

Torreño, A., Onaindia, E., Komenda, A., Štolba, M., 2017. Cooperative multi-agent planning: A survey. ACM Computing Surveys (CSUR) 50.

Yu, J., LaValle, S.M., 2013. Planning optimal paths for multiple robots on graphs, in: 2013 IEEE International Conference on Robotics and Automation, IEEE. pp. 3612–3617.