Branch-and-Cut-and-Price for Multi-Agent Pathfinding

Edward Lam[1,2] , **Pierre Le Bodic**[1] , **Daniel Harabor**[1] and **Peter J. Stuckey**[1,2]
[1]Monash University, Melbourne, Australia
[2]CSIRO Data61, Melbourne, Australia
{edward.lam, pierre.lebodic, daniel.harabor, peter.stuckey}@monash.edu

Abstract

There are currently two broad strategies for optimal Multi-agent Pathfinding (MAPF): (1) search-based methods, which model and solve MAPF directly, and (2) compilation-based solvers, which reduce MAPF to instances of well-known combinatorial problems, and thus, can benefit from advances in solver techniques. In this work, we present an optimal algorithm, BCP, that hybridizes both approaches using branch-and-cut-and-price, a decomposition framework developed for mathematical optimization. We formalize BCP and compare it empirically against CBSH and CBSH-RM, two leading search-based solvers. Conclusive results on standard benchmarks indicate that its performance exceeds the state-of-the-art: solving more instances on smaller grids and scaling reliably to 100 or more agents on larger game maps.

1 Introduction

Multi-agent Pathfinding (MAPF) is an NP-hard graph optimization problem that involves finding a minimum-cost set of paths for a team of cooperating agents. Each agent must move from its start position to its goal position such that no two agents occupy the same vertex at the same time or cross an edge in opposite directions at the same time.

One popular technique for optimal MAPF is Conflict-based Search (CBS) [Sharon *et al.*, 2015]. CBS is a two-level search algorithm. 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 terminates when it expands a collision-free leaf.

Optimal MAPF can also be solved using mixed integer programming (MIP) by modeling it as a multi-commodity flow problem [Yu and LaValle, 2013]. The 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 MIP solvers are mature industry-grade software supported by decades of academic research.

This paper proposes an optimal MAPF algorithm, called BCP, that combines the strengths of CBS (namely, decomposition and domain-specific reasoning) with the search performance of MIP. The algorithm is based on branch-and-cut-and-price (also abbreviated BCP), a general framework that decomposes a difficult optimization problem into a set of easier problems and then merges the individual results together in a branch-and-bound search tree [Desrosiers and Lübbecke, 2010; Lübbecke and Desrosiers, 2005; Desaulniers *et al.*, 2005]. BCP is also a two-level algorithm. At the low level, it solves a series of single-agent pathfinding problems using dedicated algorithms, much like CBS. At the high level, it uses MIP to assign paths to agents and resolve conflicts. The high-level problem remains NP-hard but possesses structure that can be exploited by modern MIP technologies.

The contributions of this paper are (1) a novel model of MAPF based on the BCP framework, (2) extensions of the basic model that drastically improve its performance, and (3) experimental results that highlight the advantages of the approach. The experiments compare two versions of BCP against two leading CBS variants: CBSH [Felner *et al.*, 2018] and CBSH-RM [Li *et al.*, 2019]. Results on 1,350 standard benchmarks across two small grids and two large game maps indicate that BCP exceeds state-of-the-art performance: (1) optimizing 99% of the instances compared to 85% by CBSH-RM, (2) solving 96% of the hardest instances with 100 agents compared to 34% by CBSH-RM, and (3) achieving an average run-time of 5 seconds compared to 8 seconds by CBSH-RM on the 1,143 instances solved by both algorithms. The remainder of this paper describes BCP and presents the experiments in detail.

2 Background

This section summarizes CBS and the BCP technique applied to MAPF.

2.1 Conflict-based Search

CBS breaks the MAPF problem into one high-level search and multiple single-agent low-level search problems. Initially, each agent plans its optimal path independently using

low-level A*. If none of these paths conflict, then an optimal solution to the MAPF problem is found. Otherwise, the high-level problem selects a conflict between two agents a_1 and a_2, and splits the current problem into two subproblems. In the first subproblem, agent a_1 is prevented from using the conflicting resource (vertex or edge at some time), and its optimal path is resolved under this new constraint using low-level search. In the second subproblem, agent a_2 is similarly constrained and resolved. The high-level search continues by selecting a subproblem with the least possible cost, and either finding that it has no conflicts and hence produces an optimal solution, or that it must be split again into another two subproblems. This process continues until a solution is found.

2.2 Branch-and-Cut-and-Price

The origins of the BCP method can be traced back more than half a century to early works in linear programming (LP) [Dantzig and Wolfe, 1960; Gilmore and Gomory, 1961; Gilmore and Gomory, 1963]. Today, its theoretical foundations are rigorously proven and well-understood [Desrosiers and Lübbecke, 2010; Lübbecke and Desrosiers, 2005; Desaulniers *et al.*, 2005; Barnhart *et al.*, 1998]. Several basic concepts of MIP that develop into BCP are reviewed below. Interested readers are advised to consult [Rader, 2010] for an introduction to MIP, and then the previous references for a formal treatment of the BCP technique.

Modeling

To solve a problem using MIP, a formal mathematical model of the problem must first be created. A *model* consists of an *objective function* that measures the quality of a solution, *variables* that represent decisions, and *constraints* that express relationships between the variables. A *solution* assigns a value to each variable such that their values respect the constraints. Variables can be categorized as *integer* or *continuous*. Integer variables must take a discrete, integral value in a solution, while continuous variables can take any value (integral or fractional). A solution is *optimal* if the objective function attains a global minimum.

The MIP *master problem* of the BCP MAPF algorithm selects a minimum cost subset of paths from an extremely large set of paths subject to three *classes* of constraints: (1) exactly one path is selected per agent, (2) every vertex is visited in at most one of the selected paths, and (3) every edge is traversed in at most one of the selected paths. Each path is associated with an integer variable taking value 0 and 1, representing the proportion that the path is selected. A solution assigns 0 or 1 to every variable subject to the constraints of the three constraint classes.

Branch-and-bound

NP-hard MIP problems are commonly solved using the branch-and-bound (BB) algorithm. BB sufficiently enumerates all solutions in a search tree by solving a *relaxation* problem at every node. BB begins with a tree containing only the root node, where it proceeds to solve the relaxation; usually but not always, an LP relaxation.

An *LP relaxation* of a MIP model is an identical problem but drops, or *relaxes*, the requirement that all integer variables take integral values. In other words, the integer variables are allowed to take fractional values, but otherwise preserves all constraints and variables. Solving this LP relaxation, which can be done in polynomial time, typically results in a *fractional solution*, i.e., a solution in which at least one integer variable takes fractional value, as opposed to an *integer solution*, in which all integer variables take integral values.

The BB algorithm uses this fractional solution to *branch*. A *branching rule* is a subroutine that partitions the solution space of the LP relaxation into two disjoint sets such that a fractional value appears in neither. BB then solves the LP relaxation over the two partitions, and the process repeats. In simpler terms, a branching rule selects an integer variable taking fractional value in a solution, and creates two children nodes in which this variable cannot take fractional value. The fractional solution, and all others with the same fractionality, are removed at the expense of two more nodes in the search tree. For example, consider a fractional solution in which an agent selects two paths, each with 0.5 proportion. One path traverses edge (i, j) and the other path doesn't (perhaps traversing edge (i, k)). Summed over all paths, edge (i, j) is selected with proportion 0.5. A basic branching rule will fix the edge (i, j) to proportion 0 in one child, forbidding the edge, and fix the edge to proportion 1 in the other child, forcing the edge. (The branching rule does not specify anything about (i, k).) Then, (i, j) will appear with integral proportion (0 or 1) in every solution to the LP relaxation in both children. The search is now one step closer towards selecting all edges, and hence all paths, with integral proportion.

Eventually, solving the LP relaxation naturally returns an integral solution, so branching will cease, resulting in a leaf node. At this point, the integer variables take integral values, so this solution is valid for the original MIP problem. It is accepted as the new best solution if it is better than the current best solution, i.e., its objective value is lower than that of the incumbent solution, if any.

A benefit of solving a relaxation at every node is that its optimal objective value provides a *lower bound* to the objective value of every solution in the entire subtree, and hence, can be used to prune the subtree if this lower bound is higher than the current best solution, i.e., the *upper bound*. In other words, any solution in this subtree cannot be better than the solutions already found, and hence, this subtree need not be explored. Therefore, higher lower bounds from *tight* relaxations can tremendously improve the speed of the search.

Even though a branching rule defines how to create two children nodes, it does not specify which node should be solved next. That is the role of a *node selector*. The standard node selector in BB prefers nodes in a best-first manner (lowest lower bound) but occasionally performs depth-first diving.

Because all integer variables will take integral values at some depth in the search tree, BB will terminate in finite but exponential time due to the combinatorics of branching. Since branching removes fractional solutions, never integer solutions, and bounding removes suboptimal solutions, never optimal solutions, BB is correct and optimal.

Branch-and-price

Recall that the master problem selects paths from a large set P of paths. We might imagine that all possible paths for every

agent must be enumerated in P, from which it selects a subset of optimal paths. Since MAPF allows cycling and waiting, an infinite number of paths is possible. However, based on mathematical arguments, a finite, usually exponential-size subset P' of P is sufficient to prove optimality; all other paths cannot appear in an optimal solution. This is obvious: for example, in a MAPF problem with one agent, nonsensically long paths looping round and round cannot contribute to an optimal solution since a shorter path without the loops exists.

Branch-and-price (BP) is a variant of BB that omits some or all variables/paths initially from the master problem, and reinstates them during the search. Omitting variables is equivalent to fixing their value to zero. Because some variables have been fixed, part of the LP solution space is not searched. Since an optimal LP solution could lie in this part, the optimal objective value to this LP problem no longer provides a valid lower bound at this node.

The set P' is initialized with a few paths (possibly zero) and is iteratively filled by calling a *pricer*, an algorithm that generates better paths and adds them to P' for the master problem to select. The pricer identifies omitted variables that enlarge the solution space in the direction that must be explored, or guarantees that there are none. If variables are found, they are added to the master problem, which is solved again. Note that the master problem may or may not choose the new variables. This proceeds until a sufficient portion of the original solution space is considered, at which point the pricer proves that none of the omitted variables could improve the current LP objective value even if included. Hence, the current LP optimal objective value (with many variables remaining omitted) provides exactly the same bound as a problem that includes all variables, which means BP enumerates the same solutions as BB, despite not considering all variables. This implies the correctness and optimality of BP.

Branch-and-cut

Solutions with a vertex (resp. edge) collision are removed by a constraint stating that the vertex (resp. edge) can be used by at most one path. Even though the time horizon extends to infinity, there is no need to check for conflicts at vertices and edges not currently in use or those at some distant time after all agents have reached their target. Therefore, only a finite, usually exponential-size subset of all collision constraints need to appear in the master problem.

Branch-and-cut (BC) extends BB by omitting constraints from the master problem and adding them during the search. After solving the LP relaxation, BC calls a procedure known as a *separator* for every class of constraints (e.g., vertex collision and edge collision). A separator checks the paths chosen by the master problem and determines whether these paths exhibit a conflict of its class. It either adds to the master problem one or more constraints violated in the current LP solution, or concludes that all constraints of its class are satisfied. The new constraints will prohibit more than one agent from using the vertex or edge in all future solutions. This process repeats until no violated constraints are found.

Vertex collision and edge collision are two classes of constraints categorized as *problem* constraints. Problem constraints are necessary and sufficient to correctly model the problem. A class of constraints can also be categorized as *redundant*. Redundant constraints are not necessary for modeling and solving the problem, but rather, they tighten the LP relaxation by cutting off fractional solutions, leading to a stronger lower bound and/or an integral LP solution in the next iteration; subsequently resulting in the subtree being pruned by bound or integrality much earlier than otherwise.

If a class of problem constraints and its separator are correct, its constraints will only remove portions of the LP solution space that do not contain solutions valid according to the problem definition. If a class of redundant constraints and its separator are correct, its constraints will only remove fractional solutions, never integer solutions. Therefore, BC (with many constraints remaining omitted) obtains the same optimal solutions as BB (with all constraints included).

Branch-and-cut-and-price

If variables are omitted, adding a constraint could cut off all currently known paths for an agent. Therefore, the pricer needs to be called again after a separator adds constraints in order to find alternative paths that satisfy the new constraints, or to ensure that the existing paths remain feasible and optimal. The culmination of all the previous components leads to the framework named BCP, which retains the same correctness and completeness guarantees as BC and BP.

To summarize, the master problem, which selects paths from P', the pricer, which adds better paths to P', and the separators, which resolve conflicts on the paths selected by the master problem, are repeatedly solved at a node until the chosen paths are fractionally free of conflicts and fractionally optimal. If any path is fractionally chosen, a branching rule creates two children nodes that do not contain the fractionality, bringing the search closer towards an integer solution. Assuming the correctness and completeness of all the components described previously, BCP the framework and the MAPF algorithm are correct and complete: the algorithm will find an optimal solution in finite but exponential time if and only if one exists (excluding implementation issues such as bugs and numerical instability, as with any other approach).

For many graph-related optimization problems, especially the Vehicle Routing Problems which bear many similarities with MAPF, models with an exponential number of variables and constraints can be mathematically proven to have an LP relaxation tighter than all other known models (e.g., [Letchford and Salazar-González, 2006]), and solving these exponential-size MIP models with BCP far outperforms solving smaller polynomial-size MIP models with BB (e.g., [Fukasawa *et al.*, 2006]).

3 BCP-B

This section formalizes the MAPF problem and presents a basic, minimally-working algorithm, called BCP-B, that uses the BCP technique for optimal MAPF. This basic variant is extended in the next section.

3.1 Problem Definition

Define a *location* $l = (x, y) \in \mathbb{Z}_+ \times \mathbb{Z}_+$ as a pair of space coordinates, and L as the set of all locations. A location $l_2 = (x_2, y_2)$ is a *neighbor* of $l_1 = (x_1, y_1)$ 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 $y_2 = y_1$.

Let a *vertex* $v = (l, t) \in L \times \mathbb{Z}_+$ be a pair of a location and a time coordinate, and let V be the set of vertices. Define E as the set of edges, where an *edge* $e = (v_1, v_2) = ((l_1, t_1), (l_2, t_2))$ is a pair of vertices such that $t_2 = t_1 + 1$ and either $l_1 = l_2$ or l_2 is a neighbor of l_1. That is, an edge indicates a wait action or connects two adjacent locations in one timestep. Furthermore, denote the *reverse* of an edge e as $e' = ((l_2, t_1), (l_1, t_2))$. Define a directed graph $G = (V, E)$. Since there is no time horizon and edges connect vertices with increasing time, G is infinite but acyclic.

Define A as the set of agents, where every *agent* $a \in A$ has a *start location* $s_a \in L$ and a *goal location* $g_a \in L$, which may coincide. A *path* p of length $k \in \mathbb{Z}_+$ for agent $a \in A$ 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 $((l_t, t), (l_{t+1}, t + 1)) \in E$ for all $t \in \{0, \ldots, k - 2\}$. The path p *visits* the vertices (l_t, t) where $t \in \{0, \ldots, k - 1\}$ and continues to visit vertices at its goal location after the path concludes, i.e., p also visits the vertices (g_a, t) for all $t \in \{k, \ldots\}$. The path p *traverses* the edges $((l_t, t), (l_{t+1}, t + 1))$ where $t \in \{0, \ldots, k - 2\}$. The *cost* c_p of a path is equal to its length.

Then, an optimal solution to the MAPF problem is a set of paths, one path for each agent $a \in A$, that minimizes the sum of path lengths such that each (time-indexed) vertex is visited at most once, and each (time-indexed) edge and its reverse are traversed at most once. These two conditions are respectively referred to as vertex conflicts and edge conflicts.

3.2 Master Problem

For every agent $a \in A$, assume the existence of a large pool P_a of candidate paths. The master problem chooses one path for every agent such that the sum of path costs is minimized. Additional constraints that prevent conflicts are added dynamically whenever a conflict is detected.

For all $a \in A, p \in P_a$, define $\lambda_p \in [0, 1]$ as a *variable* representing the proportion of selecting path p. A solution to the master problem maps every λ_p to a value between 0 to 1.

The initial master problem (before adding additional constraints) is specified as the linear program:

$$\min \sum_{a \in A} \sum_{p \in P_a} c_p \lambda_p \tag{1}$$

subject to

$$\sum_{p \in P_a} \lambda_p \geq 1 \qquad \forall a \in A, \quad (\alpha_a) \tag{2}$$

$$\lambda_p \geq 0 \qquad \forall a \in A, p \in P_a. \tag{3}$$

Objective Function (1) minimizes the total cost of the selected paths. Constraint (2) ensures that every agent uses at least one path. By minimizing the total cost, no agent will use more than one path in an optimal solution. Let $\alpha_a \geq 0$ be the dual variable of Constraint (2) as defined by LP duality. Constraint (3) are non-negativity constraints, which disallow negative proportions of a path.

As currently stated, the master problem allows both vertex and edge conflicts. These are resolved by adding constraints found using the separators described as follows.

3.3 Resolving Vertex Conflicts

Define $x_v^p \in \{0, 1\}$ as a *constant* that indicates if path p visits vertex $v \in V$. The separator for vertex conflicts begins by calculating the number x_v of times v is used:

$$x_v = \sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p. \tag{4}$$

The variable x_v is positive if and only if at least one path p that uses v (i.e., the constant $x_v^p = 1$) is selected (i.e., the variable $\lambda_p > 0$). A vertex conflict occurs at v if and only if $x_v > 1$. The separator then removes every solution that uses v more than once by adding the constraint

$$\sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p \leq 1 \qquad \forall v \in V. \quad (\beta_v) \tag{5}$$

Denote its dual variable as β_v. From duality theory, $\beta_v \leq 0$.

3.4 Resolving Edge Conflicts

Edge conflicts are settled in a similar manner. Let $y_e^p \in \{0, 1\}$ be a constant that indicates if path p traverses edge $e \in E$. The separator first calculates the number y_e of times e is used:

$$y_e = \sum_{a \in A} \sum_{p \in P_a} y_e^p \lambda_p.$$

An edge conflict occurs at e whenever $y_e + y_{e'} > 1$. The conflict is resolved by adding the constraint

$$\sum_{a \in A} \sum_{p \in P_a} (y_e^p + y_{e'}^p) \lambda_p \leq 1 \qquad \forall e \in E. \quad (\gamma_e) \tag{6}$$

Let the dual variable of Constraint (6) be $\gamma_e \leq 0$.

3.5 Finding Better Paths

The pool P_a of candidate paths for any agent a is incrementally filled by a pricer that either (1) finds one or more paths that could potentially improve the current master problem solution, or (2) declares that no improving path exists.

For every agent a, the pricer must solve the single-agent shortest path problem with a modified objective function. The output of the pricer is at least one path p for agent a with cost $c_p > 0$. Within the pricer, $x_v^p \in \{0, 1\}$ and $y_e^p \in \{0, 1\}$ are binary decision variables, not constants, that respectively indicate whether p visits vertex v and traverses edge e. The modified objective function is

$$\min \bar{c}_p := c_p - \alpha_a - \sum_{v \in V} \beta_v x_v^p - \sum_{e \in E} \gamma_e (y_e^p + y_{e'}^p), \tag{7}$$

which can be interpreted as follows. Recall from Sections 3.2 to 3.4 that $\alpha_a \geq 0$, $\beta_v \leq 0$ and $\gamma_e \leq 0$. If a chooses to use path p, it pays a penalty $-\beta_v \geq 0$ if it visits a conflicting vertex v and pays $-\gamma_e \geq 0$ if it traverses a conflicting edge e. The term α_a offsets the cost c_p of p to measure whether

the new path is cost-effective compared to all existing paths of a. From LP theory, p can contribute to a better solution if and only if $\bar{c}_p < 0$. In this case, p is added to P_a, leading to another round of separation and pricing. If the pricer cannot find a path p with $\bar{c}_p < 0$ for any agent $a \in A$, then no path can improve upon the current master problem solution, which is declared (fractionally) optimal.

BCP-B implements the pricer by adapting an existing A* code to the modified path cost. Every edge e initially has cost 1. The penalty $-\gamma_e$ is added to the cost of e and e', and $-\beta_v$ is added to all five incoming edges to v. Then, the A* algorithm finds a path p and outputs

$$\hat{c}_p := c_p - \sum_{v \in V} \beta_v x_v^p - \sum_{e \in E} \gamma_e (y_e^p + y_{e'}^p).$$

The output path p is accepted into P_a if $\hat{c}_p < \alpha_a$.

3.6 Resolving Fractional Solutions

The master problem solves a linear program, which typically produces fractional solutions, e.g., given an agent $a \in A$ and two paths $p_1, p_2 \in P_a$, we can have $\lambda_{p_1} = \lambda_{p_2} = 0.5$. Whenever the pricer declares a solution with fractional values is optimal, branching must proceed to resolve the fractionality. At the very least, branching must split the problem into two subproblems such that the current fractional solution appears in neither subproblem.

BCP-B branches on an agent-vertex pair, stipulating that an agent a must visit a vertex v in one child and must not visit v in the other child. The branching rule first computes the constants x_v as for Equation (4) and builds the set

$$U_v = \{a \in A : \sum_{p \in P_a} x_v^p \lambda_p > 0\}$$

of agents using the vertex v. At this stage, all violated Constraint (5) appear in the master problem, and hence, $x_v \leq 1$ for all $v \in V$. The branching rule then selects a vertex

$$v = \arg\min_{(l,t) \in V} \{t : 0 < x_v < 1 \wedge |U_v| \geq 2\}.$$

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

$$a = \arg\min_{a \in U_v} \{c_p : p \in P_a \wedge \lambda_p > 0\}.$$

That is, it chooses an agent a that is (fractionally) using a path visiting v and has a path of the shortest length among all paths used by all agents visiting v. The branching rule then creates two children nodes. In one child, a must visit v. In the other child, a must not visit v. This is in turn enforced in the master problem and the pricer at each child.

The master, separation and pricing problems are then solved in both subproblems and the process repeats. Node selection and node pruning occur as in a regular branch-and-bound. Globally optimal solutions appear in the leaves of the search tree, where all variables have integral value.

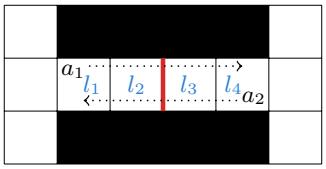

Figure 1: An example of a corridor conflict.

4 BCP

BCP-B includes Constraints (2), (3), (5) and (6), which are problem constraints necessary and sufficient for correctly modeling MAPF. BCP tightens the LP relaxation of BCP-B using redundant constraints that resolve corridor conflicts and rectangle symmetry conflicts. BCP also branches on path length in addition to the vertex branching in BCP-B.

4.1 Resolving Corridor Conflicts

Corridor conflicts can appear when two agents fractionally cross a space of unit height and some length in opposite directions. In an integer solution, BCP-B forbids all types of conflicts, namely, vertex conflicts and edge conflicts, via Constraints (5) and (6). However, these constraints may not be violated by two agents fractionally crossing a corridor.

Consider two agents a_1 and a_2 trying to cross the corridor in Figure 1. Even though the illustration portrays a corridor as a space surrounded by side obstacles, a corridor conflict can occur anywhere two agents cross, regardless of obstacles. Agent a_1 has two paths $p_{1,1} = (l_1, l_2, l_3, l_4)$ and $p_{1,2} = (l_1, l_1, l_2, l_3, l_4)$, which differs by a wait at time 0. Agent a_2 also has two paths $p_{2,1} = (l_4, l_3, l_2, l_1)$ and $p_{2,2} = (l_4, l_3, l_3, l_2, l_1)$. Assume that a master problem solution assigns value 0.5 to all four paths. Vertex conflicts do not occur in this solution since $(l_2, 2)$ and $(l_3, 2)$ are each used with value 1. Similarly, edge conflicts do not appear because $((l_2, 1), (l_3, 2))$ and its reverse, and $((l_2, 2), (l_3, 3))$ and its reverse are each used with value 1.

These fractional solutions naturally arise very frequently in the LP relaxation. Pruning them is critical to tightening the lower bound, and hence, improves the performance of BCP. Corridor conflicts are removed using the constraint

$$\sum_{p \in P_{a_1}} y^p_{((l_1,t),(l_2,t+1))} \lambda_p + \sum_{p \in P_{a_1}} y^p_{((l_1,t+1),(l_2,t+2))} \lambda_p + \sum_{p \in P_{a_2}} y^p_{((l_2,t),(l_1,t+1))} \lambda_p + \sum_{p \in P_{a_2}} y^p_{((l_2,t+1),(l_1,t+2))} \lambda_p \leq 1 \quad \forall (a_1, a_2, l_1, l_2, t), \quad (8)$$

where $a_1, a_2 \in A$, $l_1, l_2 \in L$ and $t \in \mathbb{N}$ such that $a_1 \neq a_2$, and l_1 and l_2 are neighbors. Constants y_e^p are defined as in Section 3.4.

The correctness of Constraint (8) is reasoned as follows. In an integer solution, if a_1 uses $((l_1, t), (l_2, t+1))$, it arrives at l_2 at $t+1$, and hence, cannot simultaneously use $((l_1, t+1), (l_2, t+2))$. Also, a_2 cannot use $((l_2, t), (l_1, t+1))$ because it would incur an edge conflict, nor use $((l_2, t+1), (l_1, t+2))$ because it would incur a vertex conflict at $(l_2, t+1)$. By symmetry, at most one of those four edges can be used, which is

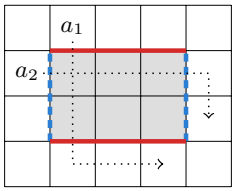

Figure 2: An example of a rectangle symmetry conflict.

precisely Constraint (8). In a sense, this constraint combines Constraints (5) and (6) for two specific agents.

Let $\pi_{a_1,a_2,l_1,l_2,t} \leq 0$ be the dual variable of Constraint (8). Adding one of Constraint (8) to the master problem will also modify the objective function of the pricer by appending

$$-\pi_{a_1,a_2,l_1,l_2,t} y^p_{((l_1,t),(l_2,t+1))} - \pi_{a_1,a_2,l_1,l_2,t} y^p_{((l_1,t+1),(l_2,t+2))}$$

to Objective Function (7) when pricing agent a_1, and

$$-\pi_{a_1,a_2,l_1,l_2,t} y^p_{((l_2,t),(l_1,t+1))} - \pi_{a_1,a_2,l_1,l_2,t} y^p_{((l_2,t+1),(l_1,t+2))}$$

when pricing agent a_2.

4.2 Resolving Rectangle Symmetry Conflicts

Rectangle symmetry conflicts were recently introduced by [Li *et al.*, 2019]. The idea is that any 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 2. If the two agents use a single-agent-optimal path, the agents will always conflict somewhere within the rectangle. Let the edges of the two sides used by a_1 be E_1, shown in thick red lines. Let the edges of the two sides used by a_2 be E_2, shown in dashed blue lines. In BCP, these rectangle symmetries are removed using the constraint

$$\sum_{p \in P_{a_1}} \sum_{e \in E_1} y^p_e \lambda_p + \sum_{p \in P_{a_2}} \sum_{e \in E_2} y^p_e \lambda_p \leq 3 \;\; \forall (a_1, E_1, a_2, E_2), \quad (9)$$

where $a_1 \neq a_2$, and $E_1 \subset E$ and $E_2 \subset E$ are the edges timed precisely for entering and exiting the rectangle in an optimal-length path (no delay in the interior). Constraint (9) states that at most three of the four sides can be used at those exact timesteps. The correctness of this constraint relies on the proof given in [Li *et al.*, 2019].

Let $\rho_{a_1,E_1,a_2,E_2} \leq 0$ be the dual variable of Constraint (9). Adding one of Constraint (9) will modify the objective function of the pricer by appending, respectively, the terms

$$-\sum_{e \in E_1} \rho_{a_1,E_1,a_2,E_2} y^p_e \quad \text{and} \quad -\sum_{e \in E_2} \rho_{a_1,E_1,a_2,E_2} y^p_e$$

to Objective Function (7) when pricing agents a_1 and a_2.

4.3 Branching on Path Length

Bounding the length of all paths for a particular agent bounds its cost in the entire subtree. In particular, fixing the length of an agent fixes its cost, regardless of the vertices it visits. Contrastingly, 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 3.6) once every path used by an agent has the same length.

The branching rule begins by calculating the set S of agents fractionally using paths of different lengths; defined as

$$S = \{a \in A : \exists p_1, p_2 \in P_a, \lambda_{p_1} > 0, \lambda_{p_2} > 0, c_{p_1} \neq c_{p_2}\}.$$

From S, it chooses an agent a and a path p of shortest length that is fractionally used, i.e.,

$$(a, p) = \underset{a \in S, p \in P_a}{\arg\min} \{c_p : \lambda_p > 0\}.$$

It then creates two children nodes. In one child, a only uses paths with length less than or equal to c_p. In the other child, a only uses paths with length greater than or equal to $c_p + 1$. Branching on path lengths removes the current fractional solution and cannot remove an integer solution.

5 Experiments

This section compares the performance of BCP-B and BCP against CBSH and CBSH-RM.

5.1 Set-up

BCP-B and BCP are implemented using the MIP solver SCIP 6.0.0 [Gleixner *et al.*, 2018] with CPLEX as the LP solver. Codes for CBSH [Felner *et al.*, 2018] and CBSH-RM [Li *et al.*, 2019] are obtained from their authors. All four solvers are single-threaded and are run on an Intel Xeon E5-2660 V3 CPU at 2.6 GHz with a time-out of five minutes. The solvers are evaluated on the same instances as [Li *et al.*, 2019], which consist of 1,350 standard benchmarks over two small grids and two large game maps sourced from the Moving AI repository [Sturtevant, 2018]. The *20x20* map is an empty 20×20 grid. In *10obs-20x20*, 10% of the empty space is replaced with obstacles. The maps *den520d* and *lak503d* are considerably larger and have structured obstacles. Fifty instances are tested for every different number of agents.

5.2 Results

Figure 3 shows the percentage of instances optimally solved by the four algorithms, categorized by map and the number of agents. BCP displays exceptional performance in comparison to the other three algorithms. It is obvious that the improvements detailed in Section 4 are worthwhile: BCP has the highest success rate, while BCP-B has the lowest, tied with CBSH. BCP optimizes 1,335 of the 1,350 instances compared to 1,145 by CBSH-RM. On the den520d instances with 120 agents, BCP and CBSH-RM respectively solve 49 and 36 instances. On the hardest lak503d map with 100 agents, BCP closes 48 instances in comparison to CBSH-RM at 17.

Of the 1,335 instances optimized by BCP, 829 of them are solved at the root node, i.e. no branching occurred. On the other 506 instances with branching, the root node optimality gap averages 0.1%, with the largest at 1.3%. These results demonstrate that the lower bounds from BCP are extremely tight, making the problems reasonably easy.

BCP found feasible solutions to all 15 instances on which it timed out. On these instances, the optimality gap averages

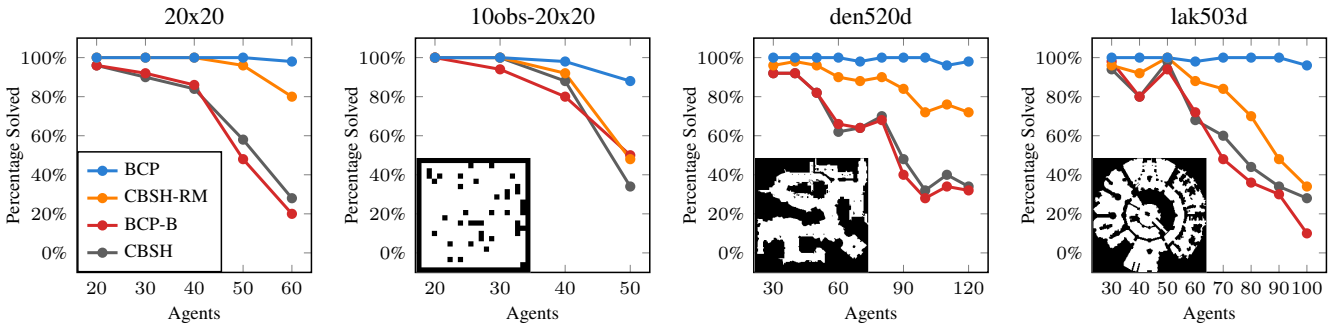

Figure 3: Success rate for each map and number of agents. Higher is better.

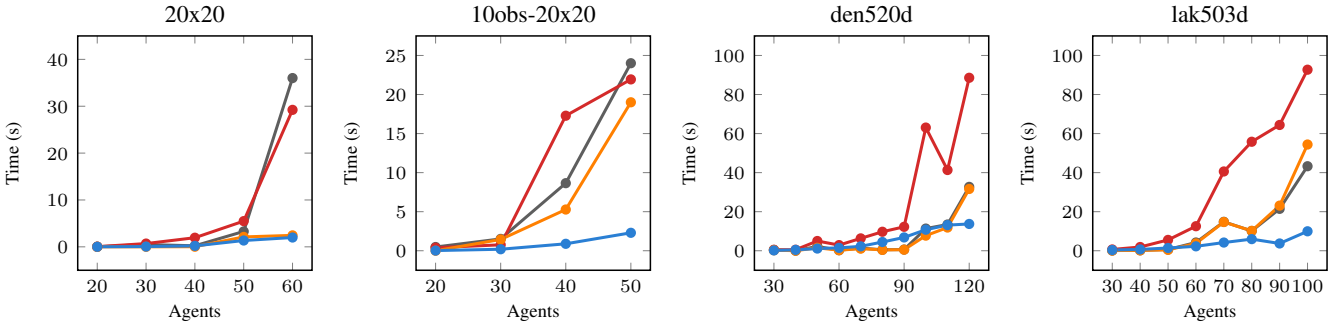

Figure 4: Average run-time computed over the instances solved optimally by all algorithms. Lower is better.

0.5%, with a maximum of 1.7%. A lower bound 1 lower than the feasible solution is found for 7 of the 15 instances, indicating that their optimal solutions are at most 1 lower than what is already found, if any.

Figure 4 plots the average run-time averaged over the instances solved by all four algorithms. These charts confirm that BCP has similar run-times for the easier instances but substantially shorter run-times for the harder instances.

6 Related Work

Multi-agent Pathfinding is a special case of Multi-agent Planning; a well known problem that is often solved by modeling the set of agents as one large agent with many degrees of freedom [Torreño *et al.*, 2018]. This approach, sometimes called *joint planning*, fails completely even for small instances of MAPF [Standley, 2010]. To improve on joint planning, state-of-the-art methods for optimal MAPF depend on some of the following decompositions and reasoning.

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. OD [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 [Goldenberg *et al.*, 2014] solver, which defers generating all but the most promising successors of a node. OD 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 same methods often exhaust available memory long before finding a solution.

Compilation-based solvers. Algorithms of this type transform MAPF into related problems for which efficient algorithms exist. Examples include SAT [Surynek *et al.*, 2016], ASP [Erdem *et al.*, 2013] and multi-commodity flow [Yu and LaValle, 2013]. These approaches reason about the entire problem at once but use simple time-expanded models and currently without any specialized constraints and reasoning techniques developed specifically for MAPF. Such approaches exhibit state-of-the-art performance but typically do not scale well to large maps with many agents.

Two-level search-based solvers. Algorithms of this type reason about single agents at the low level and about the interactions between single-agent plans at the high level. Perhaps the most prominent example is CBS [Sharon *et al.*, 2015], which includes 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; e.g. lower-bounding heuristics [Felner *et al.*, 2018], branching strategies [Boyarski *et al.*, 2015] and specialized constraints [Li *et al.*, 2019].

7 Conclusion

This paper introduced BCP, a novel algorithm for Multi-agent Pathfinding that substantially improves upon current state-of-the-art methods, including the recent algorithm CBSH-RM. These results demonstrate that close collaboration between the mathematical optimization and artificial intelligence communities can result in significant advances.

References

[Barnhart *et al.*, 1998] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[Boyarski *et al.*, 2015] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *IJCAI*, pages 740–746, 2015.

[Dantzig and Wolfe, 1960] George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[Desaulniers *et al.*, 2005] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon. *Column Generation*. Springer US, 2005.

[Desrosiers and Lübbecke, 2010] Jacques Desrosiers and Marco E. Lübbecke. Branch-price-and-cut algorithms. In *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., 2010.

[Erdem *et al.*, 2013] Esra Erdem, Doga G. Kisa, Umut Oztok, and Peter Schueller. A general formal framework for pathfinding problems with multiple agents. In *AAAI*, pages 290–296, 2013.

[Felner *et al.*, 2017] Ariel Felner, Roni Stern, Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R. Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *SOCS*, pages 29–37, 2017.

[Felner *et al.*, 2018] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *ICAPS*, pages 83–87, 2018.

[Fukasawa *et al.*, 2006] Ricardo Fukasawa, Humberto Longo, Jens Lysgaard, Marcus Poggi de Aragão, Marcelo Reis, Eduardo Uchoa, and Renato F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, 106(3):491 – 511, 2006.

[Gilmore and Gomory, 1961] P. C. Gilmore and Ralph E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[Gilmore and Gomory, 1963] P. C. Gilmore and Ralph E. Gomory. A linear programming approach to the cutting stock problem—part II. *Operations Research*, 11(6):863–888, 1963.

[Gleixner *et al.*, 2018] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, July 2018.

[Goldenberg *et al.*, 2014] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R. Sturtevant, Robert C. Holte, and Jonathan Schaeffer. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, 50:141–187, 2014.

[Letchford and Salazar-González, 2006] Adam N. Letchford and Juan-José Salazar-González. Projection results for vehicle routing. *Mathematical Programming*, 105(2):251–274, 2006.

[Li *et al.*, 2019] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Symmetry-breaking constraints for grid-based multi-agent path finding. In *AAAI*, 2019.

[Lübbecke and Desrosiers, 2005] Marco E. Lübbecke and Jacques Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.

[Rader, 2010] David J. Rader. *Deterministic operations research: models and methods in linear optimization*. John Wiley & Sons, 2010.

[Sharon *et al.*, 2015] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40 – 66, 2015.

[Standley, 2010] Trevor S. Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, pages 173–178, 2010.

[Sturtevant, 2018] Nathan Sturtevant. Moving AI: Pathfinding benchmarks. https://movingai.com/benchmarks, 2018.

[Surynek *et al.*, 2016] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*, pages 810–818, 2016.

[Torreño *et al.*, 2018] Alejandro Torreño, Eva Onaindia, Antonín Komenda, and Michal Stolba. Cooperative multi-agent planning: a survey. *ACM Computing Surveys (CSUR)*, 50(6):84, 2018.

[Yu and LaValle, 2013] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In *2013 IEEE ICRA*, pages 3612–3617, 2013.