Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

New Valid Inequalities in Branch-and-Cut-and-Price for Multi-Agent Path Finding

Edward Lam,[1,2] **Pierre Le Bodic**[1]
[1]Monash University, Melbourne, Australia
[2]CSIRO Data61, Melbourne, Australia
{edward.lam, pierre.lebodic}@monash.edu

Abstract

BCP, a state-of-the-art algorithm for optimal Multi-agent Path Finding, uses the branch-and-cut-and-price framework to decompose the problem into (1) a master problem that selects a set of collision-free low-cost paths, (2) a pricing problem that adds lower-cost paths to the master problem, (3) separation problems that resolve various kinds of conflicts in the master problem, and (4) branching rules that split the nodes in the high-level branch-and-bound search tree. This paper focuses on the separation aspects of the decomposition by introducing five new classes of fractional conflicts and valid inequalities that remove these conflicts to tighten the linear programming relaxation in the master problem. Experimental results on 12820 instances across 16 maps indicate that including the five families of inequalities allows BCP to solve an additional 585 instances, optimize the same instances 41% faster, and solve 2068 more instances than CBSH-RM and 157 more than Lazy CBS.

Introduction

Given a group of cooperating agents, each with a start and goal position, the Multi-agent Path Finding (MAPF) problem attempts to find a path for all agents from their start positions to their goal positions such that the paths are free of collisions and some measure of overall cost is minimized.

Recently, Lam et al. (2019) introduced BCP, an optimal MAPF algorithm that uses the branch-and-cut-and-price framework from mathematical programming to decompose MAPF into a number of easier subproblems. BCP is a two-level algorithm that consists of four key components: (1) a pricing problem that generates paths for each agent independently, (2) separation problems that resolve several kinds of conflicts, (3) a master problem that assembles the paths and conflict resolutions together in a linear programming problem, and (4) branching rules that partition the search space in the high-level branch-and-bound search tree.

Empirical results indicate that CBSH-RM (Li et al. 2019) outperformed a basic implementation of BCP. However, the addition of valid inequalities—special constraints that reason about rectangle and corridor conflicts—allowed BCP to consistently solve instances far larger than CBSH-RM.

This paper introduces five new families of valid inequalities, each reasoning about a different kind of conflict. Like the rectangle and corridor reasoning before, these five types of conflicts are redundant in the sense that BCP is capable of solving MAPF without them; but their inclusion allows BCP to solve ever larger instances. Comprehensive experiments on 12820 instances across 16 maps from two sets of standard benchmarks indicate that equipping BCP with the five new families of inequalities enables it to solve an additional 585 instances, and outperform CBSH-RM and Lazy CBS (Gange, Harabor, and Stuckey 2019) by solving 6375 instances in total; compared to 4307 by CBSH-RM and 6218 by Lazy CBS. Adding the five classes of cuts also permits BCP to optimize the same instances in 41% less time. The remainder of this paper formalizes the five new families of cuts and analyzes the experimental results in detail.

Problem Definition

This paper considers the same grid-based problem as Lam et al. (2019). The notation is reviewed as follows.

Let $L \subseteq \mathbb{Z}_+ \times \mathbb{Z}_+$ be the set of all locations on the grid. A *location* $l \in L$ is a pair of space coordinates $l = (x, y) \in \mathbb{Z}_+ \times \mathbb{Z}_+$. A location $l_2 = (x_2, y_2)$ is a *neighbor* of $l_1 = (x_1, y_1)$ if and only if $|x_2 - x_1| + |y_2 - y_1| = 1$. The problem is defined on a time-expanded directed acyclic graph $G = (V, E)$. A vertex $v \in V$ is a pair $v = (l, t)$, where $l \in L$ is a location and $t \in \mathbb{Z}_+$ is a time step. There exists an edge $e = (v_1, v_2) = ((l_1, t_1), (l_2, t_2)) \in E$ if $t_2 = t_1 + 1$ and either $l_1 = l_2$ (a *wait* action) or l_2 is a neighbor of l_1 (a *move* action). Denote the *reverse* of an edge e as $e' = ((l_2, t_1), (l_1, t_2))$.

Define A as the set of agents. 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 stays 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 $((l_t, t), (l_{t+1}, t+1))$ for all $t \in \{0, \ldots, k-2\}$. The *cost* c_p of a path is equal to its length.

An optimal solution to MAPF is a set of paths, one for each agent $a \in A$, that minimizes the sum of path lengths such that each vertex in V is visited at most once, and each edge in E and its reverse are traversed at most once, thus avoiding *vertex conflicts* and *edge conflicts*, respectively.

Related Work

This section reviews several optimal algorithms for MAPF.

Conflict-Based Search

Conflict-based search (CBS), designed by Sharon et al. (2015), is a tree-search framework for MAPF. CBS begins with a binary search tree that only contains the root node. At the root node, CBS finds the shortest-distance path for every agent independently using A*. If there are no conflicts among the paths, this set of paths is optimal and CBS terminates. Otherwise, CBS selects a conflicting (time-indexed) vertex or edge between two agents a_1 and a_2, and splits the current node into two children nodes. In the first child, agent a_1 is prevented from using the conflicting resource by calling A* to find a replacement path for a_1 that avoids the resource. In the second child, agent a_2 is similarly constrained and its path is replanned.

The sum of path costs of all agents at any given node provides a lower bound to any solution in its subtree. The tree search continues by selecting a node with the least lower bound, and either finding that it has no conflicts, and, hence produces an optimal solution, or that it must be split again into two more children nodes. This process continues until a solution is found. The first solution CBS finds is optimal, and hence, it is not an anytime algorithm.

Building on the basic CBS algorithm, Felner et al. (2018) developed CBSH, which uses heuristics to better select nodes in the search tree. Later, Li et al. (2019) invented rectangle symmetry reasoning for their CBSH-RM algorithm, which dramatically improved the performance of CBSH.

Gange, Harabor, and Stuckey (2019) proposed Lazy CBS, which implements conflict analysis from Boolean satisfiability (SAT) (Marques Silva and Sakallah 1996) and constraint programming (CP) (Ohrimenko, Stuckey, and Codish 2009). Lazy CBS records the change in costs from forcing an agent to avoid a resource upon branching. It then analyzes this information to learn that certain combinations of branchings can never be optimal, and consequently prunes all nodes, even those from disparate parts of the search tree, that contain these branchings.

Compilation-Based Solvers

MAPF can be reduced to an instance of another problem, such as CP (Ryan 2010), answer set programming (Erdem et al. 2013), SAT (Surynek et al. 2016b; 2016a) and mixed-integer programming (MIP) (Yu and LaValle 2013). A model of MAPF must first be created in order to solve it using such technology. SAT models consist of variables that store values representing the actions of the agents, and clauses that express relationships between the variables. CP and MIP models contain similar variables but use constraints, instead of clauses, 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 existing solvers. Furthermore, by formulating MAPF as an instance of another problem, the latest advances in solver techniques are immediately applicable. All that is needed is a "good" model be created, leaving the solving process to specialized software packages.

Branch-and-Cut-and-Price

Branch-and-cut-and-price is a general framework for decomposing a combinatorial optimization problem into a sequence of easier subproblems (Desrosiers and Lübbecke 2010; Lübbecke and Desrosiers 2005; Desaulniers, Desrosiers, and Solomon 2005; Barnhart et al. 1998). Lam et al. (2019) applied branch-and-cut-and-price to MAPF and named their implementation BCP. Note that while BCP is a MIP decomposition technique, it is different than a direct MIP compilation because instantiating BCP for MAPF is not as simple as writing a MIP model and inputting it to a MIP solver, as we will see in the next section and throughout the paper.

BCP

BCP consists of four main components: a master problem, separation problems, a pricing problem, and branching rules.

Master Problem

Given a set of possible paths for every agent, the master problem uses LP to minimize the sum-of-costs by selecting a path or a set of fractional paths for every agent such that the paths are (fractionally) free of collisions.

Assume that every agent $a \in A$ has a pool P_a of candidate paths. 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. Because the master problem is solved using LP, λ_p can take fractional values. The master problem begins as the LP model

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

subject to

$$\sum_{p \in P_a} \lambda_p = 1 \qquad \forall a \in A, \tag{2}$$

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

Objective Function (1) minimizes the total cost of the selected paths. Constraint (2) ensures that every agent uses exactly one path. Constraint (3) are the non-negativity constraints, which disallow negative proportions of a path.

As currently stated, the master problem faces two issues. First, it can select paths that exhibit vertex and edge conflicts since they are not yet disallowed. Second, $\bigcup_{a \in A} P_a$ may not yet contain the paths of an optimal solution. BCP solves the first problem by separation: it checks for conflicts in the subset of paths selected by the master problem and resolves them by adding constraints to the master problem. The second problem is solved by pricing: BCP solves a pricing problem to fill P_a with increasingly shorter paths, if they

exist. BCP iteratively calls the separation and pricing problems to incrementally build the LP model until there are enough constraints and paths to prove optimality. Because the master problem is solved using LP, fractional λ_p values in the solution must be removed by branching.

Resolving Edge Conflicts

Let $y_e^p \in \{0, 1\}$ be a constant that indicates if path p traverses edge $e \in E$. Also define $y_e^p = 0$ if $e \notin E$. Edge conflicts are resolved by dynamically adding the constraints

$$\sum_{a \in A} \sum_{p \in P_a} (y_e^p + y_{e'}^p) \lambda_p \leq 1 \qquad \forall e \in E. \quad (4)$$

Resolving Vertex Conflicts

Let $v = ((x, y), t) \in V$. For all $t \in \{1, \ldots\}$, define

$$x_v^p = y_{(((x-1,y),t-1),v)}^p + y_{(((x+1,y),t-1),v)}^p + y_{(((x,y-1),t-1),v)}^p + y_{(((x,y+1),t-1),v)}^p + y_{(((x,y),t-1),v)}^p \in \{0, 1\}$$

as a constant that indicates whether path p visits vertex $v \in V$. Vertex conflicts are resolved by adding the constraints

$$\sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p \leq 1 \qquad \forall v = (l, t) \in V : t \in \{1, \ldots\}. \quad (5)$$

Constraints are not necessary at time 0 because the start locations are assumed to be unique.

Finding Shorter Paths

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

For every agent a, the pricing problem uses A* to solve the single-agent shortest path problem with a modified objective function that measures the cost-effectiveness of a using a resource (vertex or edge) against all other agents using the same resource in the current solution of the master problem. Whenever a constraint of the form

$$\sum_{a \in A} \sum_{p \in P_a} \left(\sum_{e \in E_a} y_e^p \right) \lambda_p \leq k, \quad (6)$$

with $E_a \subseteq E$, $k \in \mathbb{Z}_+$ and LP dual variable $\pi \leq 0$, is added to the master problem, the objective function in the pricing problem is modified to penalize agent a traversing any edge $e \in E_a$ with a cost equal to $-\pi \geq 0$. In particular, this occurs whenever adding one of Constraint (4) or (5) to the master problem. We refer the reader to (Lam et al. 2019) for a detailed presentation.

Branching Rules

Unlike other approaches such as CBS, BCP uses LP to solve the master problem, and hence, frequently encounters *fractional solutions*, in which at least one path is selected with a non-integer proportion. For example, given an agent $a \in A$ and two paths $p_1, p_2 \in P_a$, we can have $\lambda_{p_1} = \lambda_{p_2} = \frac{1}{2}$. The master problem often finds fractional solutions as they usually have a lower sum-of-costs than a solution with only integral proportions. The role of the branching rules is to split the problem into two subproblems such that the current fractional solution appears in neither subproblem, so that eventually, the leaves of the search tree only contain integral solutions. We use the same branching rules as Lam et al. (2019).

Fractional Conflicts

As currently described, BCP will solve MAPF optimally. However, it contains many fractional solutions, which weaken the lower bound. Fractional solutions can be removed by targeting specific kinds of fractional conflicts, which are conflicts not forbidden by Constraints (5) or (4) under some fractional assignment. For instance, a vertex can be used by two paths, each assigned a proportion of $\frac{1}{2}$; therefore, Constraint (5) is satisfied, and the fractional assignment is valid for the master problem (but not MAPF).

Fractional conflicts are removed by adding redundant constraints, called *valid inequalities* or *cuts*, to the master problem. Cuts are never violated in an integral solution valid for MAPF, but merely prune fractional solutions. Valid inequalities are constraints no different to the constraints enforcing vertex and edge conflicts, and they are found by solving separation problems in exactly the same way.

BCP currently reasons about two types of fractional conflicts: rectangle (Li et al. 2019) and corridor. We refer the reader to the original work for further details. The main contributions of this article are five new classes of fractional conflicts designed to further improve the LP lower bound.

Five New Classes of Fractional Conflicts

This section presents the main contributions. The first four types of fractional conflicts are resolved using constraints of the form given in Constraint (6), whereas the last constraint has a different structure.

Wait-Edge Conflicts

An edge conflict (Constraint (4)) permits an edge or its reverse to be used by at most one agent. Wait-edge conflicts generalize the edge conflicts by also prohibiting a wait within the same constraint. Figure 1 shows an edge $e = ((l_1, t), (l_2, t + 1))$, its reverse $e' = ((l_2, t), (l_1, t + 1))$ and another edge $e_{\text{wait}} = ((l_1, t), (l_1, t + 1))$ that is incompatible with both e because of the vertex conflict at (l_1, t) and with e' because of the vertex conflict at $(l_1, t+1)$. Since the three edges are mutually incompatible, at most one from $\{e, e', e_{\text{wait}}\}$ can be used. Then, the wait-edge conflict constraints are

$$\sum_{a \in A} \sum_{p \in P_a} \left(y_e^p + y_{e'}^p + y_{e_{\text{wait}}}^p \right) \lambda_p \leq 1 \qquad \forall e. \quad (7)$$

Example 1. Table 1 provides an example of a fractional solution removed by a wait-edge conflict constraint. Suppose that agents 1, 2 and 3 are using paths p^1, p^2 and p^3 respectively, each with fractional value $\frac{1}{2}$. Under the "Variables"

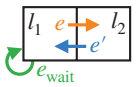

Figure 1: A wait-edge conflict.

Time	Variables			Constraint LHS			
τ	p^1	p^2	p^3	$(5)\|_{l_1}$	$(5)\|_{l_2}$	$(4)\|_e$	$(7)\|_e$
t	l_1	l_2	l_1	1	$\frac{1}{2}$	1	$\frac{3}{2}$
$t+1$	l_2	l_1	l_1	1	$\frac{1}{2}$	–	–

Table 1: Example of a violated wait-edge constraint.

heading, the table shows the location visited by each path at times t and $t+1$: p^1 moves from l_1 to l_2 using e, p^2 moves from l_2 to l_1 using e', and p^3 stays at l_1.

The values of the left-hand side (LHS) of Constraints (4), (5) and (7) at different times and locations are listed under the heading "Constraint LHS". From the table, we can read that p^1 and p^3 both use location l_1 at time $\tau = t$. As both paths are equal to $\frac{1}{2}$, the LHS of Constraint (5) is equal to 1, as shown under $(5)|_{(l_1,\tau)}$.

The table shows that Constraints (4) and (5) have a LHS less than or equal to 1; so choosing half of p^1, p^2 and p^3 satisfies the vertex and edge conflict constraints, but Constraint (7) is violated with a LHS of $\frac{3}{2}$. Removing wait-edge conflicts tightens the master problem because it removes fractional solutions that are otherwise feasible.

Constraint (7) contains all the terms in Constraint (4); hence, it is strictly stronger and can completely replace Constraint (4).

Unlike Constraint (4), 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_{wait}.

Wait-Delay Conflicts

A wait-edge conflict spans all agents due to the summation over $a \in A$ in Constraint (7). Contrastingly, wait-delay conflicts only cover a pair of agents $a_1, a_2 \in A$, $a_2 \neq a_1$. Consider the example in Figure 2. Agent a_2 is attempting to visit $l_1 = (x,y)$ at times t or $t+1$ 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

$$\begin{aligned} E_2 = \{&((l_1,t-1),(l_1,t)),\\ &(((x-1,y),t-1),(l_1,t)),\\ &(((x+1,y),t-1),(l_1,t)),\\ &(((x,y-1),t-1),(l_1,t)),\\ &(((x,y+1),t-1),(l_1,t)),\\ &(((x-1,y),t),(l_1,t+1)),\\ &(((x+1,y),t),(l_1,t+1)),\\ &(((x,y-1),t),(l_1,t+1)),\\ &(((x,y+1),t),(l_1,t+1))\} \cap E. \end{aligned}$$

The first five edges of E_2 lead into (l_1,t), and the last four edges lead into $(l_1,t+1)$ from an adjacent location; hence,

Time	Variables			Constraint LHS			
τ	p^1	p_1^2	p_2^2	$(5)\|_{l_1}$	$(5)\|_{l_2}$	$(4)\|_{l_1,l_2}$	$(8)\|_{e_1}$
$t-1$	l_1	l_2	l_1	1	$\frac{1}{2}$	1	–
t	l_1	l_1	l_2	1	$\frac{1}{2}$	1	$\frac{3}{2}$
$t+1$	l_1	l_2	l_1	1	$\frac{1}{2}$	–	–

Table 2: Example of a violated wait-delay constraint.

agent a_2 can use at most one edge from E_2. Note that 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 $e_2 \in E_2$ is incompatible, the wait-delay conflict constraints are

$$\sum_{p \in P_{a_1}} y_{e_1}^p \lambda_p + \sum_{p \in P_{a_2}} \sum_{e_2 \in E_2} y_{e_2}^p \lambda_p \leq 1 \qquad \forall a_1, a_2, e_1. \quad (8)$$

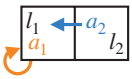

Figure 2: A wait-delay conflict.

Example 2. Table 2 reports a fractional solution cut by the wait-delay constraints but not the vertex and edge constraints. Let $l_2 = (x+1,y)$ be the location east of l_1, as shown in Figure 2. Agent 1 uses path $p^1 = ((l_1,t-1),(l_1,t),(l_1,t+1))$ with proportion $\frac{1}{2}$. Agent 2 uses paths $p_1^2 = ((l_2,t-1),(l_1,t),(l_2,t+1))$ and $p_2^2 = ((l_1,t-1),(l_2,t),(l_1,t+1))$, each with proportion $\frac{1}{2}$. The path p_1^2 uses the third edge listed in E_2, and p_2^2 uses the seventh edge in E_2. The wait-delay constraint for e_1 is violated with a LHS of $\frac{3}{2}$, but the vertex conflict constraint for $(l_1,t-1)$, (l_1,t) and $(l_1,t+1)$ are not violated, nor the edge conflict constraint for $((l_1,t-1),(l_2,t))$ and $((l_1,t),(l_2,t+1))$.

Exit-Entry Conflicts

Exit-entry conflicts are very similar to wait-delay conflicts. Figure 3 shows an agent $a_1 \in A$ 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 $a_2 \in A$, $a_2 \neq a_1$, with a set of edges

$$\begin{aligned} E_2 = \{&((l_1,t),(l_1,t+1)),\\ &((l_1,t),((x_1-1,y_1),t+1)),\\ &((l_1,t),((x_1+1,y_1),t+1)),\\ &((l_1,t),((x_1,y_1-1),t+1)),\\ &((l_1,t),((x_1,y_1+1),t+1)),\\ &((l_2,t),(l_2,t+1)),\\ &(((x_2-1,y_2),t),(l_2,t+1)),\\ &(((x_2+1,y_2),t),(l_2,t+1)),\\ &(((x_2,y_2-1),t),(l_2,t+1)),\\ &(((x_2,y_2+1),t),(l_2,t+1)),\\ &((l_2,t),(l_1,t+1))\} \cap E. \end{aligned}$$

Time	Variables			Constraint LHS		
τ	p^1	p_1^2	p_2^2	$(5)\|_{l_1}$	$(5)\|_{l_2}$	$(9)\|_{e_1}$
t	l_1	l_1	l_2	1	$\frac{1}{2}$	$\frac{3}{2}$
$t+1$	l_2	l_1	l_2	$\frac{1}{2}$	1	–

Table 3: Example of a violated exit-entry constraint.

The first five edges in E_2 exit (l_1, t), the next five edges enter $(l_2, t+1)$, and the last edge is the reverse of e_1. Note that some of the edges in E_2 can be duplicates, which are removed by the union operator. All edges in E_2 use the same time step, 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 constraints are

$$\sum_{p \in P_{a_1}} y^p_{e_1} \lambda_p + \sum_{p \in P_{a_2}} \sum_{e_2 \in E_2} y^p_{e_2} \lambda_p \leq 1 \qquad \forall a_1, a_2, e_1. \quad (9)$$

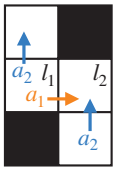

Figure 3: An exit-entry conflict.

Example 3. Table 3 displays a fractional solution invalidated by an exit-entry constraint. Agent 1 uses path $p^1 = ((l_1, t), (l_2, t+1))$. Agent 2 uses paths $p_1^2 = ((l_1, t), (l_1, t+1))$ and $p_2^2 = ((l_2, t), (l_2, t+1))$. All three paths are selected with value $\frac{1}{2}$. The vertex conflict constraint is not violated at l_1 nor l_2, but the exit-entry constraint is violated at e_1.

Two-Edge Conflicts

Figure 4 illustrates three distinct locations l_1, l_2, l_3 such that l_1 and l_3 are neighbors of l_2. Consider two edges $e_1 = ((l_1, t), (l_2, t+1))$ and $e_2 = ((l_2, t), (l_3, t+1))$. An agent $a_1 \in A$ can use at most one of these two edges since they occur at the same time. Denote their reverses 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 $a_2 \in A$, $a_2 \neq a_1$, cannot simultaneously use e_1' or e_2'. The edge e_1' is incompatible with e_1 since it is the reverse, and with e_2 because of the vertex conflict at (l_2, t). For the same reasons, e_2' is incompatible with e_2 and e_1. Using this idea, the two-edge conflict constraints are

$$\sum_{p \in P_{a_1}} (y^p_{e_1} + y^p_{e_2}) \lambda_p + \sum_{p \in P_{a_2}} (y^p_{e_1'} + y^p_{e_2'}) \lambda_p \leq 1$$
$$\forall a_1, a_2, e_1, e_2. \quad (10)$$

Example 4. Table 4 shows a fractional solution removed by a two-edge constraint. Agent 1 uses path $p^1 = ((l_1, t), (l_2, t+1))$. Agent 2 uses paths $p_1^2 = ((l_2, t), (l_1, t+1))$ and $p_2^2 = ((l_3, t), (l_2, t+1))$. All three paths are chosen with value $\frac{1}{2}$. The table shows that the two-edge constraints are violated but not the vertex and edge constraints.

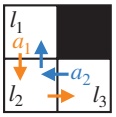

Figure 4: A two-edge conflict.

Goal Conflicts

The previous four types of conflicts reason about a set of incompatible edges traversed by a set of agents; hence, adding one of Constraints (7) to (10) changes the pricing problem with new cost penalties on the edges, as described earlier. Goal conflicts differ in that they reason about events that occur on whole paths, and cannot be expressed using agent-edge pairs, i.e., a constraint in the form of Constraint (6).

Figure 5 shows an example of a goal conflict, in which an agent $a_{\text{pass}} \in A$ is (fractionally) passing through the goal location l of another agent $a_{\text{goal}} \in A$, $a_{\text{goal}} \neq a_{\text{pass}}$, at some time t after a_{goal} has already (fractionally) reached its goal. Goal conflicts are resolved using the constraint

$$\sum_{p \in P_{a_{\text{goal}}}} G^p_{\leq t} \lambda_p + \sum_{p \in P_{a_{\text{pass}}}} 1^p_{l, \geq t} \lambda_p \leq 1 \qquad \forall a_{\text{goal}}, a_{\text{pass}}, t, \quad (11)$$

where $G^p_{\leq t}$ takes value 1 if path p finishes at time t or earlier, and takes value 0 otherwise, and $1^p_{l, \geq t}$ takes value 1 if path p visits location l at time t or later, and takes value 0 otherwise.

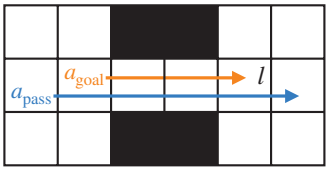

Figure 5: A goal conflict.

Example 5. Consider Figure 5. Agent a_{goal} uses path p_1 that reaches its goal l at time 3, and then waits indefinitely; hence, $G^{p_1}_{\leq 3} = 1$. Agent a_{pass} uses path p_2 that passes through l at time 4; hence, $1^{p_2}_{l, \geq 3} = 1$. The goal constraint for $(a_{\text{goal}}, a_{\text{pass}}, 3)$ is

$$p_1 + p_2 \leq 1,$$

which states that p_1 and p_2 are incompatible since a_{goal} is blocking l from time 3 onwards.

Unlike the previous four types of conflicts, there is currently no mechanism in the pricing problem to penalize the occurrence of the two events corresponding to $G^p_{\leq t} = 1$ and $1^p_{l, \geq t} = 1$. Indeed, implementing this constraint requires deep modifications to the A* pricing algorithm to record a state of whether an event has occurred during the search.

Time	Variables			Constraint LHS					
τ	p^1	p_1^2	p_2^2	$(5)\|_{l_1}$	$(5)\|_{l_2}$	$(5)\|_{l_3}$	$(4)\|_{e_1}$	$(4)\|_{e_2}$	$(10)\|_{e_1,e_2}$
t	l_1	l_2	l_3	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{3}{2}$
$t+1$	l_2	l_1	l_2	$\frac{1}{2}$	1	0	–	–	–

Table 4: Example of a violated two-edge constraint.

Let $\eta_{a_{\text{goal}},a_{\text{pass}},t} \leq 0$ be the dual variable of Constraint (11). When pricing a_{pass}, the path must be penalized by $-\eta_{a_{\text{goal}},a_{\text{pass}},t} \geq 0$ if and only if it visits l 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 valid since entering and exiting l repeatedly will penalize the path multiple times. Instead, a state is introduced to the A* algorithm to track the occurrence of the event and penalize the path on the first time it visits l at time t or later. Upon expanding the vertex (l,t') for any $t' \geq t$, the penalty is incurred and a state is recorded to mark that the penalty has been paid. Future expansions through (l,t'') for any $t'' > t'$ will not incur the penalty again.

When pricing a_{goal}, a dummy goal vertex $\perp$ is added with edges leading to $\perp$ from the original goal cell. All edges leading to $\perp$ at or before time t incurs a cost $-\eta_{a_{\text{goal}},a_{\text{pass}},t}$.

Experiments

This section presents results from two experiments. The first compares the improvements gained from using the different inequalities, and the second compares BCP against CBSH-RM and Lazy CBS.

Set-up

The following five algorithms are discussed:

- BCP-0 is the base algorithm that only contains Constraints (4) and (5) and none of the valid inequalities. This variant was previously called BCP-B by Lam et al. (2019).
- BCP-2 includes the rectangle and corridor reasoning presented by Lam et al. (2019). This algorithm was then called BCP.
- BCP-7 adds the five new classes of cuts to BCP-2 and is the main contribution of this paper.
- CBSH-RM by Li et al. (2019) is a recent variant of CBS that handles rectangle symmetries.
- A recent version of Lazy CBS by Gange, Harabor, and Stuckey (2019) that considers rectangle symmetries.

BCP is implemented in the MIP solver SCIP 6.0.2 (Gleixner et al. 2018) with CPLEX 12.10 as the LP solver. All algorithms are single-threaded and are run on an Intel Xeon E5-2660 V3 CPU at 2.6 GHz with a time limit of five minutes.

The algorithms are evaluated on two sets of standard benchmarks. The first contains 670 instances across two maps representative of warehouses. The second consists of 12150 instances across 14 maps from the Moving AI repository (Sturtevant 2019). This collection includes 29 maps in total, and 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 span a wide variety of structures. Results for the *even* instances are very similar during preliminary testing, and hence, are not reported here. In total, there are 12820 instances over 16 maps, as shown inset in Figure 6.

Algorithm	Instances Solved	
BCP-0	3942	
BCP-0 + Rectangle	4829	(+887)
BCP-0 + Corridor	4536	(+594)
BCP-0 + Wait-edge	3956	(+14)
BCP-0 + Wait-delay	3939	(–3)
BCP-0 + Exit-entry	4706	(+764)
BCP-0 + Two-edge	4297	(+355)
BCP-0 + Goal	4041	(+99)
BCP-2	5790	(+1848)
BCP-7	6375	(+2433)

Table 5: Number of instances solved by extending BCP-0 with inequalities.

Comparison of the Inequalities

Table 5 lists the number of additional instances solved after adding inequalities to the basic BCP-0. The rectangle and corridor constraints from previous work remain very successful. Including the wait-edge constraints merely solves an extra 14 instances. Adding the wait-delay constraints is detrimental by solving 3 fewer instances. The other constraints make a much larger improvement. The exit-entry constraints perform nearly as well as the rectangle inequalities, which is surprising since comparing Constraints (8) and (9) reveals that the exit-entry constraints are very similar to the wait-delay constraints, which perform the worst of all seven families of cuts. Overall, the five new classes of inequalities enables BCP-7 to solve 585 more instances than BCP-2 and 2433 more than BCP-0, indicating that at least some of the new inequalities are a substantial improvement.

Comparison to CBSH-RM and Lazy CBS

Figure 6 plots the success rate of the algorithms. To illustrate the anytime behavior of BCP, the charts also plot the percentage of instances for which BCP-7 finds a solution and can prove that it is within 1% of the optimal cost.

On the smaller 10x30-w5 warehouse map, BCP-7 outperforms the other algorithms, solving 66 more instances than Lazy CBS. On the larger 31x79-w5 warehouse map, Lazy CBS solves 4 more instances with 48 agents than BCP-7. Of

the 670 warehouse instances, BCP-7 optimizes 620 in total, and 645 to within an optimality gap of 1%. Overall, the additional constraints in BCP-7 enable it to solve many more of the harder warehouse scenarios in comparison to BCP-2, highlighting the improvements gained in using the new constraints.

Lazy CBS achieves stellar performance on the two city maps Berlin_1_256 and Paris_1_256; significantly outclassing all other algorithms on these maps with wide boulevard-like regions. Even accepting a 1% optimality gap in BCP-7 is not enough to compete with the optimal solutions found by Lazy CBS.

The performance of the algorithms vary drastically on the three Dragon Age Origins maps brc202d, orz900d and ost003d. BCP-7 solves the most instances on brc202d, which features many long, straight hallways. CBSH-RM performs remarkably better than Lazy CBS on orz900d, which also contains narrow corridors. On the ost003d map with larger open spaces, CBSH-RM performs poorly, and Lazy CBS outperforms all other algorithms. The next two maps lt_gallowstemplar_n and w_woundedcoast maps, from Dragon Age 2, contains tight doorways and winding hallways. BCP-7 solves the most instances, and Lazy CBS is outperformed by BCP-2. Results on the five Dragon Age maps demonstrate that Lazy CBS underperforms on maps with long corridors, conceding to even BCP-2 and CBSH-RM in some cases.

On the empty 8×8 map empty-8-8, BCP-7 dominates all other approaches. Conversely, Lazy CBS dominates on the larger 32×32 map empty-32-32. The next two maps random-32-32-10 and random-32-32-20 have a given proportion designated as obstacles; they respectively transform 10% and 20% of the cells in empty-32-32 to obstacles. The performance of Lazy CBS drops as the obstacles increase. Lazy CBS performs better on empty-32-32 and random-32-32-10, while BCP-7 performs better on empty-8-8 and random-32-32-20.

All algorithms perform poorly but similarly on the extremely difficult maze map maze-128-128-1. CBSH-RM, Lazy CBS, BCP-2 and BCP-7 respectively solve 57, 53, 57 and 54 of the 250 instances. Surprisingly, CBSH-RM and BCP-2 solve the most instances, indicating that the new constraints guide BCP-7 (by branching) to regions of the search space that do not contain an optimal solution. Tolerating a 1% optimality gap allows BCP-7 claim 70 instances.

CBSH-RM, Lazy CBS, BCP-2 and BCP-7 respectively optimize 323, 405, 404 and 465 test cases on the wider mazes maze-128-128-2 and maze-128-128-10. Despite the larger open spaces on the maze maze-128-128-10, it contains a large number of choke points; enabling BCP-7 to outperform Lazy CBS. Substantially more instances are solved to an optimality gap of 1% by BCP-7 on these two maps.

Of all 12820 instances, 5769 are optimized by both BCP-2 and BCP-7. BCP-2 and BCP-7 respectively solve 3183 and 3759 of these 5769 instances at the root node (i.e., no branching occurred), and the total time to prove optimality is decreased from 143391 to 84518 seconds, a 41% reduction. These results indicate that the new constraints dramatically improve the performance of BCP.

Overall, CBSH-RM, Lazy CBS, BCP-2 and BCP-7 optimize 4307, 6218, 5790 and 6375 of the 12820 instances. These results show that both BCP and Lazy CBS scale well to high numbers of agents considered out-of-reach just a few years ago, but that neither algorithm closes the MAPF problem. The main findings from these experiments are consistent and conclusive:

- CBSH-RM, considered the state-of-the-art until very recently, solves the fewest instances in total, but succeeds in some of the difficult maps with long hallways.
- The performance of Lazy CBS is highly dependent on the structure of the maps. Lazy CBS dominates the other algorithms on the empty and sparse maps, but is challenged by the congested maps with long, narrow hallways.
- BCP-2 maintains consistent performance across all maps, including the maps with long corridors.
- BCP-7 excels at most maps, but especially those with narrow corridors, choke points and specific regions of high contention.
- The five new families of valid inequalities implemented in BCP-7 allow it to solve many more instances and solve the same instances faster than BCP-2.

Conclusion and Future Work

This paper adds five new families of valid inequalities to BCP. Four of the five classes of cuts only modify the edge costs in the pricing problem to penalize agents traversing edges involved in a cut. The fifth family of cuts requires the pricing algorithm to consider state; agents are penalized at the first time they pass through the goal cell of another agent, but accounting for this state does not increase the total time complexity of the pricing algorithm.

A new variant of BCP with the five classes of constraints is compared empirically against CBSH-RM, Lazy CBS, and the previous version of BCP with only the rectangle and corridor fractional conflicts. The new BCP algorithm outperforms the other three algorithms by solving more instances. This result indicates that the new inequalities are worthwhile and enable BCP to maintain its exceptional performance.

With BCP being a recent development, many opportunities for future work are available. Following the success of Lazy CBS, conflict analysis can also be implemented in BCP using logic-based Benders decomposition (Davies, Gange, and Stuckey 2017) and branch-and-check (Lam and Van Hentenryck 2016; Lam and Van Hentenryck 2017). Research directions can also mirror the history of branch-and-cut-and-price algorithms in other domains. The family of Vehicle Routing Problems is the standard test suite for branch-and-cut-and-price algorithms. Implementing some of their confirmed advances (Costa, Contardo, and Desaulniers 2019), such as reduced cost fixing or dual stabilization, in BCP should prove beneficial. Future research can also continue to design more classes of fractional conflicts to further tighten the LP relaxation of the master problem.

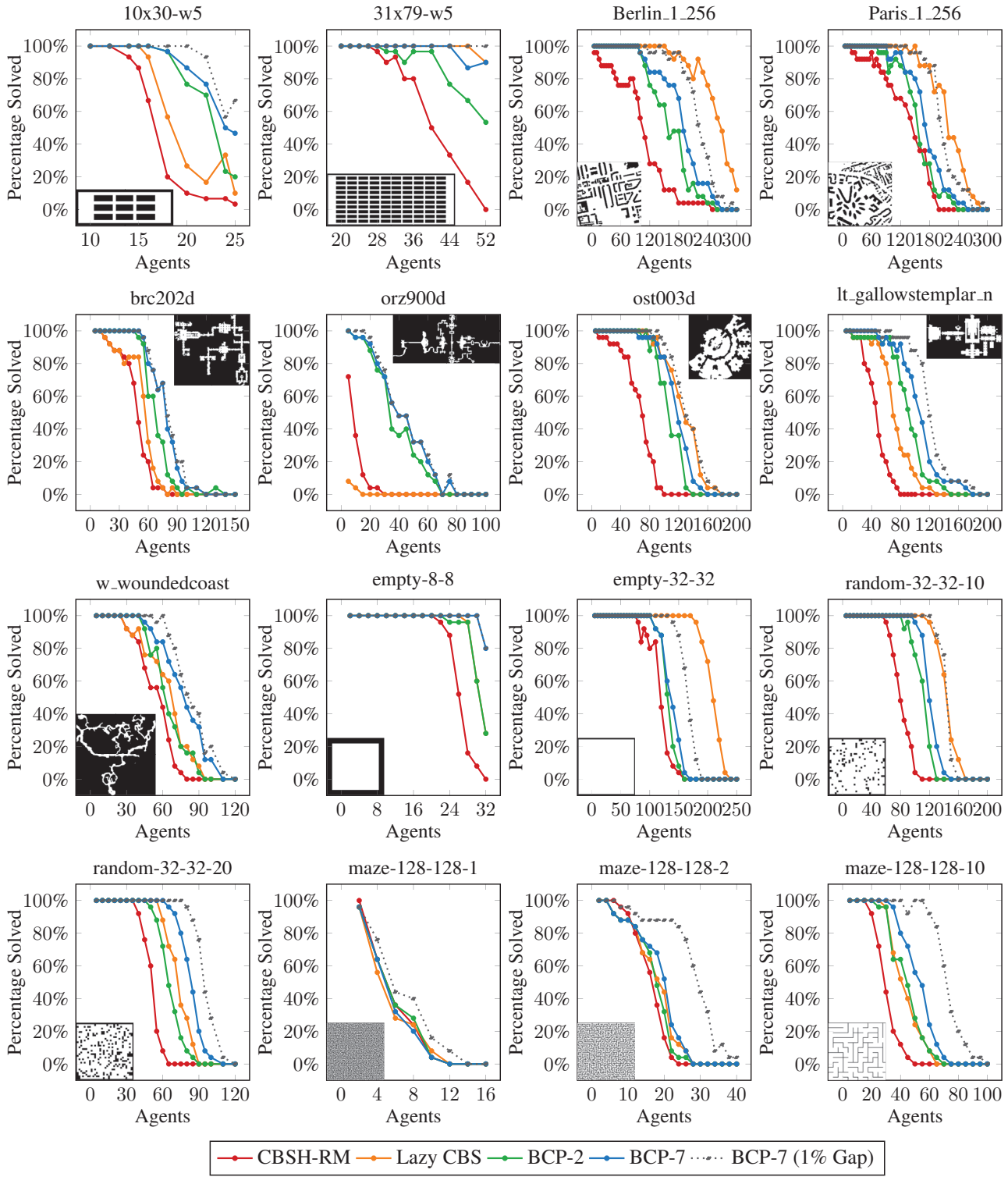

Figure 6: Success rate of the algorithms by map. Higher is better.

Acknowledgements

We would like to thank Jiaoyang Li for providing the code to CBSH-RM and the warehouse test cases, and Graeme Gange for the code to Lazy CBS.

Pierre Le Bodic is supported by the Australian Research Council Discovery Project DP200100025.

References

Barnhart, C.; Johnson, E. L.; Nemhauser, G. L.; Savelsbergh, M. W. P.; and Vance, P. H. 1998. Branch-and-price: Column generation for solving huge integer programs. *Operations Research* 46(3):316–329.

Costa, L.; Contardo, C.; and Desaulniers, G. 2019. Exact branch-price-and-cut algorithms for vehicle routing. *Transportation Science* 53(4):946–985.

Davies, T. O.; Gange, G.; and Stuckey, P. J. 2017. Automatic logic-based Benders decomposition with MiniZinc. In *AAAI*, 787–793.

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

Desrosiers, J., and 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.; and 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'13, 290–296. AAAI Press.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2018. Adding heuristics to conflict-based search for multi-agent path finding. In *ICAPS*, 83–87.

Gange, G.; Harabor, D.; and Stuckey, P. 2019. Lazy CBS: Implict conflict-based search using lazy clause generation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Gleixner, A.; Bastubbe, M.; Eifler, L.; Gally, T.; Gamrath, G.; Gottwald, R. L.; Hendel, G.; Hojny, C.; Koch, T.; Lübbecke, M. E.; Maher, S. J.; Miltenberger, M.; Müller, B.; Pfetsch, M. E.; Puchert, C.; Rehfeldt, D.; Schlösser, F.; Schubert, C.; Serrano, F.; Shinano, Y.; Viernickel, J. M.; Walter, M.; Wegscheider, F.; Witt, J. T.; and Witzig, J. 2018. The SCIP Optimization Suite 6.0. Technical report, Optimization Online.

Lam, E., and Van Hentenryck, P. 2016. A branch-and-price-and-check model for the vehicle routing problem with location congestion. *Constraints* 21(3):394–412.

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

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

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2019. Symmetry-breaking constraints for grid-based multi-agent path finding. In *AAAI*.

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

Marques Silva, J. a. P., and Sakallah, K. A. 1996. GRASP–a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, 220–227. IEEE Computer Society.

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

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

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

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

Surynek, P.; Felner, A.; Stern, R.; and 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)*, 1435–1436.

Surynek, P.; Felner, A.; Stern, R.; and 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)*, 810–818.

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