Supplementary Material for "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

This document describes the intuition and background of the branch-and-cut-and-price technique behind the BCP algorithm for unfamiliar readers.

1. Modeling

To solve a problem using integer programming, the problem must be formally described as a mathematical model. 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. *Solving* a model refers to the process of finding an optimal solution.

The *master problem* of BCP 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 or its reverse are traversed in at most one of the selected paths. Every 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.

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)

2. Branch-and-bound

NP-hard integer programming problems are often solved using the branch-and-bound algorithm. Branch-and-bound sufficiently enumerates all solutions in a search tree by solving a *relaxation* at every node. It begins with a tree containing only the root node, where it proceeds to solve the relaxation; usually but not always, a linear relaxation.

A *linear relaxation* of an integer programming 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 linear 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 branch-and-bound algorithm uses this fractional solution to *branch*. A *branching rule* is a subroutine that partitions the solution space of the linear relaxation into two disjoint sets such that the fractional solution appears in neither. Branch-and-bound then solves the linear relaxation over the two partitions, each in its own child node, and the process repeats. In many cases, a branching rule selects an integer variable taking fractional value in a solution and creates two children nodes in which this variable cannot take this 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 linear 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 linear 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 integer programming model. It is accepted as a new best solution if it is better than the current best solution, i.e., the value of its objective function 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 branch-and-bound 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, branch-and-bound 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, branch-and-bound is correct and optimal.

3. Branch-and-price

The *master problem* of BCP is an integer programming problem that selects paths from a large set P of paths. We might imagine that every possible path for every agent must be enumerated in P, from which it selects a subset of optimal paths. 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 an instance 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 is an extension of branch-and-bound that initially omits some or all variables (i.e., paths in BCP) from the master problem and reinstates them during the search. Omitting variables is equivalent to implicitly fixing their value to zero. Because some variables have been fixed, part of the solution space of the linear relaxation is not searched. Since an optimal solution could lie in this part, the optimal objective value to this linear relaxation 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*. A pricer is an algorithm that generates better paths and adds them to P' for the master problem to select. A pricer solves a *pricing problem*, which is typically an easier combinatorial optimization problem. The pricer identifies omitted variables that enlarge the solution space of the master problem in a 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 objective value even if included. Hence, the current optimal objective value (with many variables remaining omitted) provides exactly the same bound as a problem that includes all variables. This means that branch-and-price enumerates the same solutions as regular branch-and-bound despite not considering all variables. This implies the correctness and optimality of branch-and-price.

4. Branch-and-cut

Solutions with a vertex (respectively edge) collision are removed by a constraint stating that the vertex (respectively edge) can be used in at most one path. Given a finite time horizon, there is a finite number of vertex and edge constraints. However, there are still too many to enumerate explicitly in the master problem. Instead, a subset of useful constraints are dynamically added as necessary.

Branch-and-cut extends branch-and-bound by omitting constraints and adding them during the search. After solving the linear relaxation, branch-and-cut calls an algorithm 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 solution or concludes that all constraints within 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 *valid inequalities* or *cuts*. Valid inequalities are not necessary for modeling and solving the problem, but rather, they tighten the linear relaxation by cutting off fractional solutions, leading to a stronger lower bound and/or an integral solution appearing earlier; subsequently resulting in the subtree being pruned by suboptimality 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 solution space that do not contain solutions valid according to the problem definition. If a class of cuts and its separator are correct, its constraints will only remove fractional solutions, never integer solutions. Therefore, branch-and-cut (with many constraints remaining omitted) obtains the same optimal solutions as branch-and-bound (with all constraints included).

5. 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 branch-and-cut-and-price, which retains the same correctness and completeness guarantees as branch-and-cut and branch-and-price.

BCP deploys the branch-and-cut-and-price technique to solve MAPF exactly. 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, the branch-and-cut-and-price framework and the BCP 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).

Branch-and-cut-and-price is used successfully for many graph optimization problems. Two problems that bear many similarities to MAPF are the multi-commodity network flow (Barnhart et al., 2000) and the vehicle routing problems (e.g., Costa et al. (2019)). Unintuitively, branch-and-cut-and-price models with an exponential number of variables and constraints can be mathematically proven to have a linear relaxation tighter than all other known models (e.g., Letchford and Salazar-González (2006)), and solving these exponential-size integer programming models using branch-and-cut-and-price often far outperforms solving smaller polynomial-size models (e.g., Costa et al. (2019); Fukasawa et al. (2006)).

References

Barnhart, C., Hane, C.A., Vance, P.H., 2000. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. Operations Research 48, 318–326.

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

Fukasawa, R., Longo, H., Lysgaard, J., de Aragão, M.P., Reis, M., Uchoa, E., Werneck, R.F., 2006. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. Mathematical Programming 106, 491 – 511.

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