AAP_BP: A COIN/BCP Branch and Price Example

Matthew V. Galati*

Revised November 30, 2003

Abstract

COmputational INfrastructure for Operations Research - Lehigh University

This document will serve as a simple example for using `COIN/BCP` to solve an integer linear program (ILP) using branch and price (BP).

Topic:	Integer Linear Program, Column Generation
Modules:	BCP, OSI, Coin, CLP
Source:	`AAP_BP.tar.gz`
Data:	AAP Data - included in `AAP_BP.tar.gz`
Author:	Matthew Galati
Document:	AAP_BP: A COIN/BCP Branch and Price Example

1 Introduction

This document will serve as a simple example for using `COIN/BCP` to solve an integer linear program (ILP) using branch and price (BP). The problem we will focus on is the Axial Assignment Problem [2]. The descriptions in this paper are for illustrative purposes only and are not meant to be computationally efficient.

2 Branch and Price for the Axial Assignment Problem

2.1 Axial Assignment Problem

The *Axial Assignment Problem* (AAP) is that of finding a minimum-weight clique cover of the complete tri-partite graph $K_{n,n,n}$. Let I, J and K be three disjoint sets with $|I| = |J| = |K| = n$ and set $H = I \times J \times K$. Then, AAP can be formulated as the following binary integer program:

*Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, `magh@lehigh.edu`, `http://sagan.ie.lehigh.edu/mgalati`

$$\min \sum_{(i,j,k)\in H} c_{ijk}x_{ijk}$$

$$\sum_{(j,k)\in J\times K} x_{ijk} = 1 \ \forall i \in I$$

$$\sum_{(i,k)\in I\times K} x_{ijk} = 1 \ \forall j \in J \tag{1}$$

$$\sum_{(i,j)\in I\times J} x_{ijk} = 1 \ \forall k \in K \tag{2}$$

$$x_{ijk} \in \{0,1\} \qquad \forall (i,j,k) \in H \tag{3}$$

2.2 Dantzig-Wolfe Formulation

Balas and Saltzman [1] consider the use of the classical Assignment Problem (AP) as a relaxation of AAP in the context of Lagrangian Relaxation. We will use these same ideas in the context of *column generation* or *Dantzig-Wolfe decomposition.*

Define the AP polytope by removing constraint (1) as $\mathcal{P}' = conv(\mathcal{F}')$, where $\mathcal{F}' = \{x \in \mathbb{R}^H : x \text{ satisfies } (1)-(3)\}$. Observe, that the cost of an assignment is $c_s = \sum_{(i,j,k)\in H} s_{ijk}c_{ijk}$, $\forall s \in \mathcal{F}'$.

Then, the Dantzig Wolfe (DW) LP can be written as:

$$z_{DW} = \min \sum_{s\in\mathcal{F}'} c_s\lambda_s$$

$$\sum_{s\in\mathcal{F}'} \sum_{(j,k)\in J\times K} s_{ijk}\lambda_s = 1 \ \forall i \in I \tag{4}$$

$$\sum_{s\in\mathcal{F}'} \lambda_s = 1 \tag{5}$$

$$\lambda_s \geq 0 \qquad \forall s \in \mathcal{F}'$$

where

$$x_{ijk} = \sum_{s\in\mathcal{F}'} s_{ijk}\lambda_s \tag{6}$$

2.3 Column Generation Algorithm

Since there is an exponential number of possible columns in the Dantzig-Wolfe LP, we will use a column generation algorithm. Let $\mathcal{F}'_t \subseteq \mathcal{F}'$ represent a subset of the feasible points in $\mathcal{P}'$, i.e. assignment points. At iteration t, the *restricted master problem* is the Dantzig-Wolfe LP above with $\mathcal{F}' = \mathcal{F}'_t$. Associate a vector of dual variables u with constraint (4) and α with the convexity constraint (5). Let $(\hat{u}, \hat{\alpha})$ be the optimal dual solution to the restricted master problem. We then solve a subproblem to search for the column from $\mathcal{F}' \setminus \mathcal{F}'_t$ with the most negative reduced cost. The reduced cost of a variable x_{ijk} is calculated as

$$r_{ijk} = c_{ijk} - \hat{u}_i - \hat{\alpha} \tag{7}$$

Hence, we can write the column generation subproblem as an optimization problem over the AP polytope as

$$
\begin{aligned}
z_{SP}(\hat{u}, \hat{\alpha})) = \min \quad & \sum_{(j,k) \in J \times K} c'_{jk} x_{jk} \\
& \sum_{k \in K} x_{jk} && = 1 \ \forall j \in J && (8) \\
& \sum_{j \in J} x_{jk} && = 1 \ \forall k \in K && (9) \\
& x_{jk} \in \{0, 1\} && \forall (j,k) \in J \times K
\end{aligned}
$$

where $c'_{jk} = \min_{i \in I}\{c_{ijk} - \hat{u}_i\}$. Then, if $z_{SP}(\hat{u}, \hat{\alpha}) \geq \hat{\alpha}$, then no columns in $\mathcal{F}' \setminus \mathcal{F}'_t$ have negative reduced cost and the solution to the restricted master is optimal. Otherwise, we add the column x_{ijk}, corresponding to the optimal solution of $z_{SP}(\hat{u}, \hat{\alpha})$, to $\mathcal{F}'_{t+1}$ and iterate.

3 BCP Implementation

`BCP` provides a framework for doing branch, cut and price. The user is responsible for describing the structure of the application to be solved, as well as, routines for doing variable and/or cut generation.

In order to implement a branch and price algorithm using `BCP`, there are a minimal set of functions that must be written by the user. In the following section we will describe each of these functions as they have been used in the `AAP_BP` example. The application source code is divided into the following directories:

- `LP`: methods used by the linear programming process,
- `TM`: methods used by the tree manager,
- `Member`: miscellaneous or common methods shared by processes,
- `include`: the source header files,
- `Data`: some data instances for AAP,
- `Run`: a run directory which includes a sample parameter file.

3.1 Data Structures

`BCP` is written to allow for running in parallel. The serial version emulates a parallel environment. Because of this, every data structure that is passed between processes must be packed into a buffer. For most built-in types, `BCP_buffer` has an overloaded `pack` and `unpack` routine. For every user-defined class, we must write our own `pack` and `unpack` routine. Typically, these consist of a collection of calls to `BCP_buffer::(un)pack()` on each data member of the class.

- `Member/AAP.cpp`
  The `AAP` class simply defines a storage container for an instance of AAP. This includes the dimension of the problem n and the costs c_{ijk}. It also includes some helper functions for indexing and debugging.

- `include/AAP_user_data.hpp`
  This class, `AAP_user_data`, is derived from `BCP_user_data` and is meant to store extra information that is to be sent between nodes after branching. In our case, since we branch on the original variables x_{ijk} and not the master variables λ_s, we need to keep track of which variables have been fixed to 0 or 1 at a particular node. We need this information so that we can enforce these conditions in our column generation subproblem.

- `include/AAP_var.hpp`
  Each variable (a member of $\mathcal{F}'$), represents an assignment between sets J and K. This class, `AAP_var`, is derived from `BCP_var_algo`, which is a generic class for algorithmic variables. We use algorithmic variables (as opposed to indexed variables) since there are potentially too many to give an index to each assignment point. The storage of an assignment is done using a vector of the (i, j, k) indices and its cost.

3.2 Initialization

- `Member/AAP_init.cpp`
  In order to initialize the interface to BCP, we have to create an instance of each process that will be used. In this case we simply initialize:

  - `BCP_user_init()`: the user interface,
  - `lp_init()`: the LP process, and
  - `tm_init()`: the TM process.

  In `tm_init()`, in addition to initializing the TM process, we read in the parameter file and any additional command line arguments. We also read in the datafile which describes the `AAP` instance.

3.3 Parameters

- BCP Parameters
  A description of the various BCP parameters that can be used for the LP process and the TM process can be found in the Doxygen notes on the COIN website. In `Run/par.par` there is a list of those parameters which must be set for our application.

  - `BCP_LpValueIsTrueLowerBound` is set to 0, since we are generating columns and cannot say that the intermediate LP solutions give a true lower bound.
  - `BCP_SolveLpToOptimality` is set to 1, since we need the optimal dual variables to generate columns. We do not want the LP solver to stop once it has reached the dual objective limit.
  - `BCP_DoReducedCostFixingAtZero` is set to 0, since we do not want to allow for reduced cost fixing while we are generating columns.
  - `BCP_DoReducedCostFixingAtAnything` is set to 0, since we do not want to allow for reduced cost fixing while we are generating columns.

- `TM/AAP_tm_param.cpp` - `AAP_datafile`: The location of the AAP instance data file.

- `LP/AAP_lp_param.cpp` - `AAP_StrongBranch_Can`: The number of candidates to produce for strong branching.

3.4 Tree Manager

- `TM/AAP_tm.cpp::initialize_core()`:
  In this method, we describe which variables and which constraints will be *core*. If a variable or constraint is core, then it can never be removed from the LP. We have no core variables. The core constraints for the restricted master problem are (4) and (5).

- `TM/AAP_tm.cpp::create_root()`:
  In this method, we describe those variables that will be used to construct the initial LP. In order to do this, we need to start with an initial set of feasible columns. To keep it simple, we choose the columns $x_{iii} = 1 \; \forall i \in I$.

- `TM/AAP_tm.cpp::init_new_phase()`:
  This method tells `BCP` that we will be generating columns.

3.5 Linear Program

- `LP/AAP_lp.cpp::initialize_solver_interface()`:
  This method returns an instance of an `OsiSolverInterface`. This is an interface to the LP solver that will be used to solve the linear programming relaxations. The decision for which LP solver to use is done at compile time (see the section on *Build Process and Execution*).

- `LP/AAP_lp.cpp::compute_lower_bound()`:
  In this method, we generate a valid lower bound on the master LP. In order to do this, we need to solve the column generation subproblem $z_{SP}(\hat{u}, \hat{\alpha})$. The subproblem is an instance of the Assignment Problem (AP). In the helper function `generate_vars()`, we calculate the reduced costs and construct an instance of AP. In addition, we use a *bigM*-method with the information contained in `AAP_user_data` to enforce the bounds determined by branching. We then solve the subproblem by calling `solve_assignment_problem()`, which constructs the AP as an LP using OSI. Then, any variables generated with negative reduced cost are saved as an `AAP_var` for use in the next method `generate_vars_in_lp()`.

- `LP/AAP_lp.cpp::generate_vars_in_lp()`
  In this method, we look for variables with negative reduced cost to add into the master LP. The work has already been done in `compute_lower_bound()`.

- `LP/AAP_lp.cpp::restore_feasibility()`
  This method is invoked before fathoming a search tree node if the LP has been found infeasible and no new variables were generated. Upon finding the LP infeasible, BCP attempts to produce a set of certificates, or dual rays. These are sent as arguments to `restore_feasibility()`, which we then use in `generate_vars()` in an attempt to find variables that restore feasibility.

- `LP/AAP_lp.cpp::vars_to_cols()`
  This method takes a variable (an assignment point) and expands it into a column in the LP. The assignment point is stored as a vector of (i, j, k) indices in an `AAP_var`. From this, we construct a `BCP_col`.

Another method that a user typically needs to derive is `test_feasibility`. In our case, however, the default implementation, which checks for integrality is sufficient.

3.6 Branching

- `LP/AAP_lp_branch.cpp::select_branching_candidates()`
  We branch using a disjunction on the original variables x_{ijk}. The choice for which variables to select as candidates for branching is determined in the function `branch_close_to_half()`, in which we pick variables with high cost that are closest to 0.5. The number of branching candidates is determined by the parameter `AAP_StrongBranch_Can`.

  In the function `append_branching_vars()`, for each candidate we construct a `BCP_lp_branching_object`. For our case, the branching object simply contains information about which variables we will change the bounds on. Since we are branching on the original variables $x_{ijk} = \sum_{s \in \mathcal{F}'} s_{ijk} \lambda_s$, we have to reset a set of variables in the master formulation. In one child, we set $x_{ijk} = 0$, which forces $\lambda_s = 0$ for all $s \in \mathcal{F}'$ that contain index (i, j, k). In the other child, we set $x_{ijk} = 1$, which forces $\lambda_s = 0$ for all $s \in \mathcal{F}'$ that do not contain index (i, j, k).

- `LP/AAP_lp_branch.cpp::set_user_data_for_children()`
  In this method, we update the user data with the branching information.

4 Build Process and Execution

On most systems, the default `Makefile` should be sufficient. The `COIN` makefiles can be found in `COIN/Makefiles`. The `AAP_BP` makefile is `Makefile.aap`. In order to select which LP solver to use, simply set the `SOLVER` flag. The current setup uses `COIN/CLP` as the LP solver.

The typical steps to build `AAP_BP` are as follows:

1. Download the `COIN` source from here,
2. Download the `AAP_BP` source from here,
3. Edit `COIN/Makefiles/Makefile.location`, if necessary,
4. Edit `Makefile.aap`, if necessary,
5. Run `make`.

This should build `BCP` and `AAP_BP`, creating an executable `bcps` in the directory `$(uname)-$(USER_OPT)`, e.g., `Linux-g`. Then, in order to run the example (from the Run subdir), type `../Linux-g/bcps ParamFile par.par`. Any of the parameters can also be overridden on the command line as well. For example, if you want to run with the settings in the parameter file `par.par`, but you want to use 10 branching candidates, instead of 5, type `../Linux-g/bcps ParamFile par.par AAP_StrongBranch_Can 10`.

References

[1] Balas, E., and Saltzman, M. An algorithm for the three-index assignment problem. *Operations Research 39* (1991), 150–161.

[2] Qi, L., and Sun, D. Polyhedral method for solving three index assignment problems. Working Paper.