CBC

COIN-OR Branch-and-Cut

Short Guide to the Command Line Interface

Prof. Haroldo Gambini Santos

June 28, 2011

Contents

Chapter 1

Before we start

This guide is intend to show the basic usage and tuning of `CBC`: the COIN-OR Branch-and-cut standalone executable which is called by the command line. If you do not have it installed in your computer you can grab it accessing the project page[1] or, more easily, from the CoinAll[2] binary distribution which includes ready to use executables for the major operating systems.

Although a basic knowledge of Integer Programming is assumed, concepts are briefly explained whenever it is possible.

CREDITS: a large part of the content of this guide was obtained from the `cbc` advanced command line help which you can access by typing "`verbose 15`" followed by "?" in the `cbc` interactive mode. Thanks also to the nice folks at `cbc@list.coin-or.org`.

[1] https://projects.coin-or.org/Cbc/
[2] http://www.coin-or.org/download/binary/CoinAll/

Chapter 2

Quick start

Once you have installed `CBC`, you can open the command line interface[1] of your favorite operating system and type:

```
1  cbc air03.lp solve solu sol.txt
```

to load problem air03.lp, solve it and save the best solution in a file named sol.txt.

An example of a customized `cbc` execution is given bellow. In this case, the parameter `cuts` receives value `Off` and the parameter `passF` receives value 100 before the beginning of the solution process:

```
1  cbc air04.lp cuts off passF 100 solve solu sol.txt
```

By calling only `cbc` without parameters you will enter in interactive mode.

```
1 Cbc version 2.6, build Dec 4 2010
2 CoinSolver takes input from arguments ( - switches to stdin)
3 Enter ?  for list of commands or help
4 Coin:
```

To solve air04.lp as we did before but using interactive mode you can enter the following commands:

[1] *Terminal* in linux and *Command Prompt* in Windows

```
1 Coin:  import air04.lp
2 Coin:  cuts off
3 Coin:  passF 100
4 Coin:  solve
5 Coin:  solu sol.txt
```

Chapter 3

Terminal Output

To solve your problem cbc uses a bag of tricks. It has to dynamically decide how much processing power will be used in different algorithms/search strategies. To keep you informed of the successes (or failures) of these attempts it continually prints messages containing details about the current search status. Understanding these messages is a key step to pinpoint which are the main difficulties in solving your problem. Once you have this information in your hands you can start to tune CBC so that it will perform better considering the type of problem you are working on.

```
1 Cbc version 2.6, build Nov 30 2010
2 command line - cbc air03.lp solve solu sol.txt (default strategy 1)
3 Continuous objective value is 338864 - 0.15 seconds
4 Cgl0003I 0 fixed, 0 tightened bounds, 0 strengthened rows, 4
  substitutions
5 Cgl0004I processed model has 120 rows, 8456 columns (8456 integer)
  and 71651 elements
6 Objective coefficients multiple of 2
7 Cutoff increment increased from 1e-05 to 1.998
8 Cbc0038I Pass 1:  suminf.  8.84615 (35) obj.  346406 iterations 116
9 Cbc0038I Pass 2:  suminf.  6.50000 (13) obj.  368168 iterations 85
10 ...
11 Cbc0038I Pass 12:  suminf.  0.00000 (0) obj.  349282 iterations 129
12 Cbc0038I Solution found of 349282
13 ...
```

Output at line 5 indicates that cbc successfully solved the linear programming relaxation of your problem. The objective value of this solution provides a dual bound (338864), which is an optimistic estimate for the optimal solution value.

After solving the linear relaxation, cbc starts the pre-processing phase. On line 5 we can see the outcome of pre-processing. Ideally our pre-processed model is a simpler and smaller problem.

Message on line 5 and all messages starting with `Cbc0038I`, indicate that `cbc` is searching for an initial integer feasible solution using the Feasibility Pump [CITE] method. After twelve passes, it found a solution with value 349282. We now have a valid solution for the problem and valid bounds for the optimal solution value: $[338864, 349282]$. The performance of `cbc` will depend on how close are these bounds. A special attention should be given to messages indicating the progresses in tightening these bounds, which will be discussed in the next paragraphs.

```
14 ...
15 Cbc0038I Full problem 120 rows 8456 columns, reduced to 0 rows 0
   columns
16 Cbc0012I Integer solution of 340160 found by DiveCoefficient after
   11 iterations and 0 nodes (4.07 seconds)
17 ...
18 Cbc0031I 3 added rows had average density of 648
19 Cbc0013I At root node, 3 cuts changed objective from 338864 to
   340160 in 2 passes
20 Cbc0014I Cut generator 0 (Probing) - 0 row cuts average 0.0
   elements, 160 column cuts (160 active) in 0.840 seconds - new
   frequency is 1
21 Cbc0014I Cut generator 1 (Gomory) - 4 row cuts average 1257.2
   elements, 0 column cuts (3 active) in 0.020 seconds - new frequency
   is -100
22 Cbc0014I Cut generator 2 (Knapsack) - 0 row cuts average 0.0
   elements, 0 column cuts (0 active) in 0.020 seconds - new frequency
   is -100
23 Cbc0014I Cut generator 3 (Clique) - 10 row cuts average 3.7
   elements, 0 column cuts (0 active) in 0.000 seconds - new frequency
   is 1
24 Cbc0014I Cut generator 6 (TwoMirCuts) - 0 row cuts average 0.0
   elements, 0 column cuts (0 active) in 0.030 seconds - new frequency
   is -100
25 ...
26 Result - Finished objective 340160 after 0 nodes and 11 iterations -
   took 4.75 seconds (total time 5.06)
27 Total time 5.14
```

Message at line 6 indicates the success of the DiveCoefficient heuristic in finding a better feasible solution. We now have tighter bounds: [338864 , 340160].

To improve the dual bound, `cbc` adds a series of cuts to improve the for-

mulation. Messages starting at 6 contain details of the progress achieved in generating series of cuts to remove fractional solutions. In this case 3 cuts suffice to close the gap and produce the best possible dual bound: 340160. We are now sure that the previously found solution is the optimal one. Last messages of this log inform how each one of the active `cbc` cut generators performed. By these messages we can observe that only Clique and Gomory cuts where useful. Gomory cuts, as usual, produced cuts which are much denser. Finally message at line 6 announces the end of the search, which took only 5 seconds. In this problem, `cbc` performed noticeably well: a good feasible solution and a good dual bound were obtained at root node, so that no further exploration in the branch-and-bound tree was needed. The following output refers to a harder problem, air04.lp.

```
28 ...
29 Cbc0010I After 200 nodes, 46 on tree, 56212 best solution, best
   possible 55800.3 (148.79 seconds)
30 Cbc0016I Integer solution of 56174 found by strong branching after
   116654 iterations and 263 nodes (152.34 seconds)
31 Cbc0038I Full problem 615 rows 7673 columns, reduced to 109 rows 110
   columns
32 Cbc0012I Integer solution of 56138 found by RINS after 124176
   iterations and 300 nodes (156.76 seconds)
33 Cbc0038I Full problem 615 rows 7673 columns, reduced to 1 rows 2
   columns
34 Cbc0010I After 300 nodes, 43 on tree, 56138 best solution, best
   possible 55805.8 (156.94 seconds)
35 Cbc0038I Full problem 615 rows 7673 columns, reduced to 142 rows 143
   columns
36 Cbc0010I After 400 nodes, 55 on tree, 56138 best solution, best
   possible 55863.5 (166.19 seconds)
37 ...
```

At line 7 `cbc` informs that the search has advanced 200 nodes in the tree. Frequent messages indicate how many nodes were explored and how many nodes still open and need to be explored. Whenever a new integer solution is found (line 7), information about its value and which method found it is printed.

Chapter 4

Tunning

To modify `cbc` default settings, we can specify options before finally call the solve command. A command line can include as many options as you want, in the format:

```
1 cbc air04.lp ...  option1 parameter 1 option2 ...  solve solu
  sol.txt
```

Next subsection will briefly present some of the most important `cbc` options and parameters which can impact the search process. Options are presented in the following format:

ShortName : DataType : DefaultValue : LongName

ShortName indicates the abbreviation necessary to specify the option; DataType indicates the domain of possible values for the parameter and default value indicates the value which is automatically selected on `cbc` startup. Finally, LongName indicates the option parameter name.

4.1 Pre-processing

Prep : PrepChoice : sos : Preprocess

Preprocessing tries to reduce the problem size and strengthen the formulation. Although it may slow down the start of the search, the resulting model may be much easier to solve.

Decides which preprocessing will be used. Values for `PrepChoice` are:

`off` : turns of pre-processing

`equal` : turns $\leq$ clique constraints into equality clique constraints

`sos` : creates Special Ordered Sets [CITE] to improve branching

PassP : Integer : 5 : PassPresolve

Maximum number of passes for presolve.

4.2 Cut Generation

CBC cut generator can be configured to be applied only at root node or in the entire search tree. More flexible ways are also allowed.

Cuts : Logical : On : CutsOnOff

Cuts on (Cuts Off) activates (deactivates) all cuts at once.

Clique : CutAppChoice : IfMove : CliqueCuts

Determines the application of Clique cuts. Possible values for `CutAppChoice` are:

`Off` : never try this cut;

`Root` : cuts applied only at root node;

`IfMove` : cuts will be used of they succeed on improving the dual bound;

`ForceOn` : forces the use of the cut generator at every node.

Lift : CutAppChoice : Off : liftAndProjectCuts

Determines the application of Lift-and-Project cuts.

Mixed : CutAppChoice : IfMove : MixedIntegerRounding-Cuts

Determines the application of MIR - Mixed Integer Rounding cuts.

Two : CutAppChoice : Root : TwoMirCuts

Determines the application Two phase Mixed Integer Rounding cuts.

Knapsack : CutAppChoice : IfMove : KnapsackCuts

Determines the application of Knapsack cover cuts.

Flow : CutAppChoice : IfMove : FlowCoverCuts

Determines the application of Knapsack cover cuts.

Rrobing : CutAppChoice : forceOnStrong : ProbingCuts

Activates Probing Cuts. For this cut generator other more aggressive options are available: `forceOn`, `forceOnGlobal`, `forceOnStrong`, `forceOnButStrong` and `strongRoot`.

Residual : CutAppChoice : Off : ResidualCapacityCuts

Residual : CutAppChoice : Off : ResidualCapacityCuts

CutD : Integer : -1 : CutDepth

Allows to activate cuts whenever the depth in the three is a multiple of *CutD*. When `CutD`=-1, `cbc` decides if cuts will be applied or not.

CutL : Integer : -1 : CutLength

Gomory cuts can produce very dense rows which can slowdown the search. This option allows one to limit the maximum number of acceptable columns in gomory cuts. By default, `cbc` decides it (-1). Values greater than 0 indicate:

$0 \leq$ *CutL* $< 10,000,000$: maximum length of *CutL* for cuts generated at root node and in the tree;

CutL $\geq 10,000,000$: allows cuts with unlimited length at root node, with a limit inside the tree. for example: *CutL* =10,000,130 indicate that in the tree only cuts with at most 130 variables will be accepted.

PassC : Integer : -1 : PassCuts

Maximum number of cut passes in the root note. If -1, the following strategy is used, according to the number of columns n:

$n \leq 500$: 100 passes;

$500 < n \leq 5000$: 100 passes, stopping when bound improvements are small;

$n \geq 5000$: 20 passes for larger problems.

4.3 Heuristics

Heuristics are an important feature of `cbc`. In many cases heuristics can significantly speedup the appearance of high quality feasible solutions or, most importantly, avoid the embarrassing result of obtaining no feasible solution in a long `cbc` run. This section presents heuristics available in `cbc`. Most heuristics accept a parameter of type `HeurAppChoice` for controlling its activation. Valid values for this parameter are:

`Off` : never apply;

`On` : applies heuristic after preprocessing;

`Before` : applies heuristic before preprocessing;

`Both` : applies heuristic before and after preprocessing.

Heur : Logical : On : HeuristicsOnOff

Heuristics `On` (Heuristics `Off`) activates (deactivates) all heuristics at once.

Round : HeurAppChoice : On : RoundingHeuristic

This switches on a simple (but effective) rounding heuristic at each node of tree.

Feas : HeurAppChoice : On : FeasibilityPump

This switches on feasibility pump heuristic at root. This is due to Fischetti, Lodi and Glover [CITE] and uses a sequence of LPs to try and get an integer feasible solution.

PassF : Integer : 30 : PassFeasibilityPump

Indicates the maximum number of passes for the Feasibility Pump heuristic. Try higher values if no feasible solution was obtained.

Local : HeurAppChoice : Off : LocalTreeSearch

This switches on a local search algorithm when a solution is found. This is from Fischetti and Lodi and is not really a heuristic although it can be used as one.

PivotAndC : HeurAppChoice : Off : PivotAndComplement

Switches Pivot and Complement heuristic.

PivotAndF : HeurAppChoice : Off : PivotAndFix

Switches Pivot and Fix heuristic.

Combine : HeurAppChoice : On : CombineSolutions

This switches on a heuristic which does branch and cut on the problem given by just using variables which have appeared in one or more solutions. It obviously only tries after two or more solutions.

Combine2 : HeurAppChoice : Off : CombineSolutions

Same as before, but considers only variables which have the same value in two or more solutions.

Rins : HeurAppChoice : On : Rins

Controls the activation of Relaxation Induced Neighborhood Search heuristic.

Rens : HeurAppChoice : Off : Rens

Controls the activation of Relaxation Enforced Neighborhood Search heuristic.

Vnd : HeurAppChoice : Off : VndVariableNeighborhood-Search

Controls the activation of Variable Neighborhood Search heuristic.

DivingG : HeurAppChoice : Off : DivingGuided

Switches Guided Dives heuristic.

DivingP : HeurAppChoice : Off : DivingPseudoCost

Switches Diving heuristic usign pseudocosts.

DivingF : HeurAppChoice : Off : DivingFractional

Switches Diving Fractional heuristic.

DivingS : HeurAppChoice : Off : DivingSome

Switches on a random diving heuristic at various times.

4.4 Limits

Sec : Integer : Infinity : Seconds

Time limit for execution in seconds.

MaxN : Integer : Infinity : MaxNodes

Maximum number of nodes in the search tree. Can be used to avoid very large memory requirements.

MaxS : Integer : Infinity : MaxSolutions

Maximum number of integer solutions. If you are interested in any feasible solution, not necessarily the optimal one, set it to 1.