Coin Branch and Cut
A tutorial

John Forrest

July 18 2006

Outline of Cbc tutorial

Background

Some concepts

Stand-alone solver and AMPL interface

Example C++ code

Less structured part:

- Q & A
- More examples
- Future -
  - What can I do for you?
  - What can you do for Cbc

Background

Clp released – Osi interface needs branchAndBound

- I bet myself I could code in a day – failed – took 10 hours

Slowly became more complex – moved to project as SBB (Simple Branch and Bound – partly to fool IBM)

Supposedly solver independent – uses OsiSolverInterface

Renamed to CBC (Coin Branch and Cut) as there was an SBB

Complex linkages in Coin project

CBC- Coin linkages

Uses -

- OsiSolverInterface – Open Solver Interface
  - OsiClp (and sometimes knows about OsiClp and Clp)
    - Clp which also uses Coin
  - OsiDylp
  - OsiCbc – in a circular way – big mistake
- Cgl – Cut Generator Library – very important
- So normally – Cbc, Osi, OsiClp, CoinUtils, Cgl!
- Some overhead due to being solver agnostic.

Some concepts

Virtual Branching classes

- Integer
- Special Ordered Sets
- Follow on (useful in air-crew scheduling and/or column generation)
- N way
- Lotsizing (will go through as example)
- BranchOnMany (lopsided branching with cut)

Classes

CbcModel – contains model

CbcBranch…. to define variable discontinuity

CbcNode for variable at a node – just tuning

CbcTree organizes tree – from SBB could be improved

CbcCompare to choose node in tree – easy to modify

CbcCutGenerator links to Cgl cut generators

CbcHeuristic heuristic – easy to add new ones

CbcStrategy to try and contain a default strategy

CbcMessage and CbcEventHandler – advanced use

CbcModel class

CbcModel is main class with which user communicates

- Is passed an OsiSolverInterface solver
- Which it clones – so after that access by
  - model->solver()
- Cut generators are added to model (again cloned)
- Heuristics are added to model (cloned)
- Cut generators and heuristics can also be added by CbcStrategy which can be passed to model
  - Strategy checks for duplicate cut generators

CbcCompare

CbcCompareDefault is fairly simple and probably could be improved.

- Very simple to code – e.g. Important code for breadth first search is:
- bool CbcCompareObjective::test(CbcNode *x, CbcNode *y) { return x->objectiveValue()> y->objectiveValue();}
- Which returns true if node y better than node x

Question – has anyone here written their own version?

Main Cgl cut generators

CglClique

CglDuplicateRow – normally just used in preprocessing

CglFlowCover

CglGomory

CglKnapsackCover – would be good to get SOS

CglMixedIntegerRounding

CglProbing

CglRedSplit

CglTwomir

Preprocessing

CglPreProcess – also called from CbcStrategy

- Normal Coin presolve (but knowing about integers)
- Probing to strengthen coefficients in situ
- Duplicate rows out
- Produces a stack of problems which are unwound at end
- Can find some integers and some SOS
- Needs more e.g. Symmetry breaking

Standalone Solver

- Fairly primitive – glad if someone would make more elegant
- Command line and/or interactive
- Double parameters
- Int parameters
- Keyword parameters
- Actions
- Documented?
- Undocumented??
- Can produce reference list of parameters/actions
  - Of course this uses an undocumented option

Double parameters

- AllowableGap – stop if distance between LB and UB less
- Cutoff – cutoff all nodes with objective > this
- Increment – at a solution set cutoff = current + this
- IntegerTolerance – treat variables as integer if close enough
- RatioGap – as allowable gap but as fraction of continuous objective
- Seconds – treat as maximum nodes after this time

Int parameters

- CutDepth – only generate cuts at multiples of this
- LogLevel – increases amount of printout (0==off)
- MaxNodes – stop after this many nodes
- PassCuts – number of cut passes at root
- PassFeasibilityPump –
- SlogLevel - printout for underlying solver
- StrongBranching – number of candidates for strong branching
- TrustPseudoCosts - how many strong branches before trust calculated pseudo costs

Strong branching

- Find N variables which look most violated
  - For each do up to K iterations up and down
  - Choose variable which gives max min (or another rule)
  - If objective exceeds cutoff one way – can fix
  - If both ways can kill node
  - Can do faster as we can re-use starting data
  - CbcSimpleInteger, CbcSOS etc
- Can be expensive – Achterberg, Koch and Martin say trust calculated costs after so many tries
  - NumberBeforeTrust
  - CbcSimpleIntegerDynamicPseudoCost only

Keyword parameters (some)

- CostStrategy – just do priorities on costs – crude
- CutsOnOff – normally on – set off then add one by one
- ForceSolution – crash to solution (needs example)
- HeuristicsOnOff – normally on – set off then one by one
- PreProcess – on, off or try to find sos
- SosOptions – whether to ignore sos from AMPL

Cuts

- Options – off, on, root, ifmove
- Clique
- FlowCover
- Gomory
- Knapsack
- MixedIntegerRounding
- Probing
- ReduceAndSplit
- TwoMir

Heuristics

- CombineSolutions – when we have two or more solutions just choose union and preprocess and run for 200 nodes
- FeasibilityPump – Fischetti and Lodi
- Greedy – positive elements and costs – integer elements for == case, any for >= case
- LocalTreeSearch – normally off as not normal heuristic – again Fischetti and Lodi
- Rounding – simplest (and often most powerful). See if you can get solution by rounding expensive way. Also tries with SOS – could be improved.

Actions

- BranchAndCut – does branch and cut
- InitialSolve – just do continuous
- PriorityIn – reads in priorities etc from file
  - Priorities
  - Directions
  - Pseudocosts
  - Initial solutionand how to get there
- Solve – as BranchAndCut if has integers, otherwise just solve
- Strengthen – probably not very useful – produces a strengthened model
- UserCbc – user code (useful with AMPL interface)

AMPL

- See FAQ for how to build
- Build cbc and point to that from AMPL
- Syntax is maxNodes=1000 rather than -maxNodes 1000
- If no solve or branch and cut command will add it
- Rather silent unless log=n set (even log=0)
- If running using xxxx.nl file then stand-alone syntax is
  - Cbc xxxx.nl -AMPL maxNodes=1000 etc
- Priorities, direction, SOS allowed

Undocumented stuff

- Debug – mainly to track down bugs especially in cut generators.
  - Use to create a good solution
  - Then feed back on suspect run
  - Can give false reading on good run (due to strong branching or heuristic)
- Outduplicates – take out duplicate rows and fix variables if possible

Tuning

- Which cuts (if any looked good)
  - Try off or just at root (more nodes but may be faster)
  - Tweak parameters if you think cuts should be generated
- Strong branching
  - Weak point of Cbc – look at output
    - Sometimes essential
    - Sometimes too much effort – try priorities
    - Iterations in hotstart, trust, moreOptions
- Reformulate – e.g. More integers

Code generation ?

- Standalone solver makes it easy to experiment and find fast way of solving problem
- But what if you want to build model rather than read an mps file?
- Or what if you want to set a parameter you can find in CbcModel.hpp but not in solver?
- Up to now it was difficult to transfer settings but ...
- Cpp option – use it before the solve and a file user_driver.cpp will be produced.
- The Makefile in Cbc/examples can be used.
- Not 100%coverage

Lotsizing

Valid lotsizes 0,100, 125, 150, 175, 200 up

1970's situation where valid ranges 0 and 1 to 1000

- Can be modeled with extra constraint and 0-1 variable
- But if we want 2.3 then 0-1 variable would be 0.0023 and would have to be branched on even though 2.3 is valid!
- Semi-Continuous (SC) variables which were general integer variables with two lines of extra code to say feasible if >=1

Lotsizing is just a generalization

Lotsizing 2

Done for IBM Microelectronics

- Using OSL – clumsily
- Influenced design of Cbc branching

Ordered set of valid ranges and/or points

Main work is providing

- Inheriting from CbcObject
  - Infeasiblity()
  - FeasibleRegion()
  - CreateBranch() which constructs a branchingObject
- Inheriting from CbcBranchingObject
  - branch

Advanced use

- Event handler e.g. Stop on max nodes if using too much memory
- OsiAuxInfo class – replaced appData_ in OsiSolver
  - OsiBabSolver is derived from it (and you can derive ..)
  - This allows great control
    - Continue adding cuts if solution
    - Whether we have reduced costs, basis etc
    - Please ask if you need more
  - Currently used for BonMin and next examples

Simple advanced use!

- Integer quadratic constraints
  - Done with putting coefficient on y_{ij}
  - And stored cuts $x_i + x_j - y_{ij} \leqslant 1$
- Problems and solutions
  - Continuous solution may look feasible
    - Pass in OsiBabSolver – type 4
  - Strong branching may get “feasible” solution
    - Pass in a CbcFeasibility object to say NO
  - Might do 100 passes of cuts, exit and think feasible
    - Say cut generator can go on ad infinitum
- examples/qmip2.cpp

Using Clp with Cbc etc

- Cbc - OsiClp – Clp imposes overhead
- Several attempts to improve situation
- Other LP solvers could do same used with Cbc
- Other MIP solvers could use these switches
- Look for specialOptions in .hpp files
  - CbcModel.hpp
  - OsiClpSolverInterface.hpp
  - ClpSimplex.hpp

Dantzig-Wolfe example

- Column generation can lead to much tighter better formulations e.g. Mkc problem(Multi-colored Knapsacks)
- We do branch and bound on master problem where each proposal is an integer solution to subproblem.
- Often after one proposal per subproblem master is integer feasible at root node – we need OsiBabSolver to say that is not really true but we also want to save that solution – one way is to use dummy heuristic.
- Basis handling more complicated – need new class - CoinWarmStartBasisDynamic.

CoinWarmStartBasisDynamic

- Derive from CoinWarmStartBasis
- Number of static rows and columns and status – use CoinWarmStartBasis
- Number and list of identifiers for dynamic variables
- Need a “Diff” but we don't try and do a diff so fairly simple.

ClpDynamicInterface

- Derive from OsiClpSolverInterface
- Main work is in :
  - Initialize
  - Resolve
  - SetBasis
  - GetBasis
  - addProposals

Initialize

- Given master rows marked by -1 find which block all rows and columns are in.
- Create static part of model with convexity rows and artificials to make feasible.
- Each subproblem is a ClpSimplex
- Each block of master rows is a CoinPackedmatrix
- Backward pointers from columns of subproblem to full model
- Initialize proposals_ as empty CoinPackedMatrix

Resolve

- If just doing initial solve etc use OsiClp one
- Otherwise create ClpSimplex copy from static part
- Add in proposals from proposals_ (only those valid at this node+ invalid basic ones with zero bound)
- Solve
- Use D-W to try and improve
  - Use duals
  - Create OsiClpSolverInterface from subproblem
  - Fix those that need to be fixed
  - Solve branch and bound

Resolve 2

- If has negative reduced cost add as proposal
  - Elements, cost, primal solution which created all stored as column of proposals_ CoinPackedMatrix
- When all subproblems solved
  - If sum reduced costs good do another pass < N
- If can't be better than cutoff return as infeasible
- Check if integer solution – if so store
- If at root node do IP on master + current proposals (only if not too many)

Results on mkc

- Unsolved up to a few years ago – still marked as unsolved in miplib3/ miplib.cat!
- Laci and I used similar approach to get optimal solution
- Doesn't need basis handling or dummy heuristic as solves at root node!
- Using Cbc to solve subproblems took 150 minutes on my laptop – with tuning down to 50 minutes.
- Opbdp – implicit enumeration algorithm for pure 0-1 problems with integer coefficients (cvs version)
  - By a strange coincidence there is OsiOpbdpSolver which solves such problems from OsiSolverInterface
    - Can scale to get integer coefficients
    - Can be used to get all feasible solutions.

Referenced code

- Lotsizing – Cbc/examples
  - lotsizeSimple.cpp (simplified lotsize.cpp)
  - CbcBranchLotsizeSimple.?pp (simplified versions of code in Cbc/src)
- Advanced solvers – Osi/src/OsiAuxInfo.?pp
  - qmip2.cpp
- Dynamic matrices/ Dantzig Wolfe solver
  - dynamic2.cpp
  - ClpDynamicInterface.?pp (from OsiClpSolverInterface)
  - CoinWarmStartBasisDynamic.?pp
- Cbc - verbose 11 - ?