sbb

COIN-OR Simple Branch-and-Cut

Additional Notes for sbb
COIN-OR Tutorial
CORS/INFORMS Joint Meeting, Banff, May, 2004

Lou Hafer
Computing Science
Simon Fraser University

May 14, 2004

The material on these pages augments the tutorial slides and provides some pointers to additional resources.

1 Branching Concepts

A branching object has:

- One or more *feasible regions*.

  For a binary variable, the feasible regions would be the values 0 and 1. For a general integer variable, the feasibile regions would be all integer values between the lower and upper bounds. For a set of binary variables forming a clique, it would be an assignment of values to the variables such that exactly one variable had the value 1 and all others were 0.

- A way to *evaluate the infeasibility* of a solution.

  For an integer variable x with value x^* in the optimal solution to a relaxation, the infeasibility is usually defined as $\max(x^* - \lfloor x^* \rfloor, \lceil x^* \rceil - x^*)$.

  `Sbb` adopts the convention that the infeasibility of a branching object should be scaled to the range $[0, .5]$ so that infeasibility can be compared between different branching objects.

- Some set of *branching actions* which, when executed, will force the relaxation closer to feasibility.

  These are the actions taken to restrict a relaxation. For an integer variable, the common branching actions for a standard two-way branch are

to tighten the upper bound to $\lfloor x^* \rfloor$ in one branch and the lower bound to $\lceil x^* \rceil$ in the other. For a clique of size k, the branch would be k-ary, with one member variable set to 1 in each branch.

- A method to *rank the desireability* of a branch alternative *vs.* other branching alternatives, so that it's possible to specify a preferred branching direction.

  This is in the context of a single branching object. A general integer branching object might use reduced costs to estimate the cost of forcing $x = x^*$ to $\lfloor x^* \rfloor$ *vs.* $\lceil x^* \rceil$.

2 Customizing `Sbb`

If you plan on developing custom methods for branching or node selection, you should do '`make doc`' to extract the documentation embedded in the code and format it as HTML files (*cf.* §4). Read the detailed descriptions (the link labelled '`More...`') for the top level classes; this is the single best source of conceptual overview information about `Sbb`[1]. The detailed descriptions for `SbbObject` and `SbbCompareBase` provide the overview information presented in the tutorial. Additional details are provided in the documentation for the derived classes and their member functions.

[1] And `OSI`, and many of the specific solver interfaces. Hey, if you're doing serious development, you're going to have to read at least some of the code in any event. Make your life easy and start with the embedded documentation.

3 A Minimal Branch-and-Bound Example

Here's a listing of the file `dsbbclp.cpp`, a minimal branch-and-bound solver, with comments. This version uses the `clp` lp solver.

```
/*
  dsbb: dead simple branch-and-bound
  A *really* minimal example of using the sbb library.
*/
#include <iostream>
#include "OsiSolverInterface.hpp"
#define COIN_USE_CLP
#include "OsiClpSolverInterface.hpp"
#include "SbbModel.hpp"
/*
  Here are the basic steps you need to perform in order to solve a MIP:

  Step 1: Make a solver interface.
  Step 2: Create a model. The solver interface is held as an attribute
          of the model.
  Step 3: Load a problem. Here I've used the solver's readMps() routine.
          You can also build the problem from scratch using other OSI
          solver interface routines.
  Step 4: Invoke the solver to do the initial optimization.
  Step 5: Perform branch-and-bound search.
  Step 6: Extract the solution. The code here just checks that a
          solution was found and prints the objective.

          If you want to use cut generators, heuristics, or otherwise
          tweak the search, the code would be interleaved with steps
          2, 3, and 4. There are more complex examples in the COIN/Sbb
          directory tree.
*/

int main (int argc, const char *argv[])
{
  OsiSolverInterface *osi = new OsiClpSolverInterface ; // Step 1
  SbbModel *model = new SbbModel(*osi) ;                // Step 2
  model->solver()->readMps("p0033") ;                   // Step 3
  model->initialSolve() ;                               // Step 4
  model->branchAndBound() ;                             // Step 5
  if (model->bestSolution())                            // Step 6
  { std::cout << "Best solution "
              << model->solver()->getObjValue() << std::endl ;
    return (0) ; }
  else
  { std::cout << "No integer solution found." << std::endl ;
    return (1) ; } }
```

The above file is part of an example that can be found in the COIN distribution in the directory `Examples/Sbb`. The example also includes `dsbbdylp.cpp`, an alternate version which uses the `dylp` solver, and a makefile, `Makefile.dsbb`. These are described in a bit more detail below.

The file `dsbbdylp.cpp` shows the changes that are required to use a different OSI solver, `dylp`. There are two:

- The solver-specific `#include` file must be changed from

  ```
  #define COIN_USE_CLP
  #include "OsiClpSolverInterface.hpp"
  ```

  to

  ```
  #define COIN_USE_DYLP
  #include "OsiDylpSolverInterface.hpp"
  ```

  The compile-time symbols `COIN_USE_CLP` and `COIN_USE_DYLP` are used in the context of the full COIN-OR library to control inclusion of solver specific code. Normally they would be part of a `makefile` or equivalent build control file. To keep the example and its accompanying `makefile` as simple as possible, the symbol is hardcoded here.

- The solver-specific constructor must be changed from

  ```
  OsiSolverInterface *osi = new OsiClpSolverInterface ;
  ```

  to

  ```
  OsiSolverInterface *osi = new OsiDylpSolverInterface ;
  ```

Here's a slightly abridged listing for `Makefile.dsbb`, with comments.

```
# Dead simple makefile for dead simple branch-and-bound (dsbb)

# So that standard makefile tricks do not obscure the necessary actions,
# this file is hardwired for locations and programming environment.

# Standard locations for COIN libraries and include files. Edit if you've
# installed COIN somewhere else.

COINHOME := $(HOME)/COIN
COININC := $(COINHOME)/include
COINLIB := $(COINHOME)/lib

# You only need these to build dsbbdylp with the dylp solver, one of the
# alternative solvers with COIN/OSI interfaces.

DYLPHOME := $(HOME)/Bonsai/OsiDylp
DYLPINC := $(HOME)/Bonsai/OsiDylp
DYLPLIB := $(HOME)/Libraries/DylpLib
DYLPLIBINC := $(HOME)/Libraries/DylpLib

# In the Sun Solaris/Workshop environment, CC is the name of the C++
# compiler. Change to suit your environment (g++ if you use GCC under
# Solaris or Linux).

# '-R' is the flag that tells the Solaris linker to add a directory
# to the run-time library search path. Change to suit your environment
# (for GCC, use '-Wl,-R' under Solaris, '-Wl,-rpath' under Linux)

dsbbdylp: dsbbdylp.cpp
        CC -I$(COININC) -I$(DYLPINC) -I$(DYLPLIBINC) \
           -L$(COINLIB) -lSbb -lOsiDylp -lOsiClp -lOsi -lCoin -lClp -lCgl \
           -L$(DYLPHOME) -lOsiDylpSolver.sparc \
           -L$(DYLPLIB) -ldylpstd.sparc \
           -lz \
           -R$(COINLIB) -R$(DYLPHOME) -R$(DYLPLIB) \
           -o dsbbdylp dsbbdylp.cpp

# The clp link requires only components from the COIN software distribution.

dsbbclp: dsbbclp.cpp
        CC -I$(COININC) \
           -L$(COINLIB) -lSbb -lOsiClp -lOsi -lCoin -lClp -lCgl \
           -lz \
           -R$(COINLIB) -R$(DYLPLIB) \
           -o dsbbclp dsbbclp.cpp
```

As the makefile comments state, the reason for including the clp library (`-lClp`) in the link step for `dsbbdylp` is that in my environment both solvers are enabled. Also, COIN makefiles will enable `clp` by default. For efficiency, `sbb` includes a few pieces of code which are specialized for `clp`. Rather than rebuild the `sbb` library without `clp` just for this example, I chose the easy solution and added the `clp` library to the link.

Similarly, `sbb` is a branch-and-cut code, and support for cutting planes is built in by default. The `Cgl` library is needed to satisfy the references, even though the code will not be executed unless cut generators and/or heuristics are actually installed.

4 Resources

Specifically for sbb, be sure to consider:

- The documentation embedded in the code.

  Typing '`make doc`' in the `Sbb` directory of the COIN-OR distribution will use `doxygen` to extract embedded documentation, format it in HTML, and store it in a subdirectory named `Doc`. Point your browser to the file `Doc/html/index.html`.

  If you don't have `doxygen` installed on your system, check the `doxygen` web site, `http://www.stack.nl/ dimitri/doxygen`.

- The samples included with the distribution, in the `Sbb/Samples` directory.

The following are general resoources for COIN-OR:

- COIN-OR website: `www.coin-or.org`
- COIN-OR tutorials site: `http://sagan.ie.lehigh.edu/coin/`

The following are on-line resources for C++:

- *C++ Annotations* by Frank B. Brokken. In the author's words, "... intended for knowledgeable users of C who would like[2] to make the transition to C++."

  `http://www.icce.rug.nl/documents/cplusplus`

- *C/C++ Reference* by Nate Kohl. Strictly reference, but comprehensive.

  `http://www.cppreference.com/`

[2] And just as useful if you're being dragged kicking and screaming out of your nice, comfy C environment into the wholly incomprehensible world of C++.