First Steps with COIN-OR

François Margot[1]

January 2011

5. Cbc Basics

Contents

[1]Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213-3890, U.S.A. Email: `fmargot@andrew.cmu.edu` .

1 Overview

The goal is to study some simple elements of the COIN-OR branch-and-cut code `Cbc`. A reader interested in using `Cbc` as a black box, can just read the first four sections of this document. The remaining sections deal with the use of the `Cbc` library through a `C++` code.

Section 2 covers the download and installation of the `Cbc` package and Section 3 deals with the generation of the html documentation and customization. Section 4 shows how to to use the code as a black box through the command line interface.

Section 5 gives instruction to compile the code of the project `PROJ5`. Section 6 describes the most important class of `Cbc`, the `CbcModel` class. Section 7 shows how to use the `Cgl` cut generators, while Section 8 presents some of the predefined heuristics and gives a simple example of a user-defined heuristic. Section 9 deals with some of the predefined node selection rules and shows how the user can easily define his own rule. Finally, Section 10 and Section 11 discuss the selection of branching variables for strong branching and the selection of the best of those after optimization for the candidates is done. Section 10 presents some candidate selection rules used during the strong branching phase of the code. A simple example of a user-defined selection rule is also given. Section 11 surveys an example using pseudo-costs and priorities associated with the variables.

These sections can be used to customize most of the behavior of Cbc. However, unless the problem to be solved requires heavy customization, in general one wish to experiment with a slight variation of the default Cbc code. Section 12 show how to use callback and `CbcEvent` to do that, starting from the default settings and altering the behavior of Cbc by adding customization through code and command line parameters.

A more detailed presentation of `Cbc` is available on the web [8].

2 Download and Installation

This section covers the installation of the software `Cbc` on a machine running `Linux Fedora 12` with the `tc` shell. If you run another shell or use another Linux distribution, some of the Linux commands might be slightly different. Note that this project can also be compiled under `Cygwin` on `Windows` machines.

There are more than one way to get code from COIN-OR. Besides the use of tar balls described below, it is also possible to use the `Subversion` versioning software (`svn`) [11]. While using `svn` is more flexible, it is also more complex. For more information about using `svn` go to the COIN-OR help pages [6].

1. Go in your main directory.

2. Download the `Cbc` package tar ball: Using a web browser, go to the COIN-OR download site `http://www.coin-or.org/download`, click on `Cbc` and download the most recent tar ball of `Cbc`. At the time of this writing, this is `Cbc-2.6.1.tgz`.

3. Decompress the tar ball

   ```
   % tar -xvf Cbc-2.6.1.tgz
   ```

4. Go in the newly created directory `Cbc-2.6.1`:

   ```
   % cd Cbc-2.6.1
   ```

5. Create a build directory `build` and go there:

   ```
   % mkdir build
   % cd build
   ```

6. Configure the package according to your system:

   ```
   % ../configure -C >& last_configure.txt
   ```

   If the last lines of the file `last_configure.txt` do not contain:

   ```
   configure: Main configuration of Cbc successful
   ```

   then the configuration failed and you might need to provide information to the configuration script. The Trac pages [6, 7] might help you figure out what is wrong.

   It is advisable to use `Cbc` with `Lapack` and `Blas` for better numerical stability. The configure script installing a COIN-OR package does its best to find `Lapack` and `Blas` libraries on your system. These are usually, but not always, located in `/usr/lib` or `/usr/lib64`. To check if the configure script was successful in finding these libraries, use from `build`:

   ```
   % grep lapack last_configure.txt
   ```

   If you get a line like:

```
checking whether -llapack has LAPACK...  no
```

then configure was not able to find the Lapack library and you are advised to add it manually, although the code will work even if you do not do it. Please refer to Section 4 (and more specifically Section 4.1) of the first project (proj1.pdf) for more detailled instructions.

7. Compile the code:

```
% make
```

8. Test the code:

```
% make test
```

If the last lines of the output do not contain (timing may vary, of course):

```
cbc_clp solved 2 out of 2 and took 3.74443 seconds.
```

then something is wrong. See the Trac pages [3, 4, 6, 7] for help.

9. To install the include files in the directory Cbc-2.6.1/build/include/coin, the libraries in the directory Cbc-2.6.1/build/lib, and the executables in the directory Cbc-2.6.1/build/bin use:

```
% make install
```

Note that the use of the directory Cbc-2.6.1/build is not absolutely necessary and it is possible to build the code from Cbc-2.6.1 or from any other directory of your choosing. See the first project (proj1.pdf) for more information.

The Cbc-2.6.1/build/bin directory now contains two executables: clp, and cbc. The code clp is a linear programming solver and the code cbc is a branch-and-cut code studied in Section 4 and beyond and works with clp.

The Cbc-2.6.1/build/lib directory contains, among others, the libraries libCbc and libOsiCbc and Cbc-2.6.1/build/include/coin contains all the header files. The directory Cbc-2.6.1/Cbc/examples contains several example codes that might be helpful for understanding a specific feature of Cbc. These can be compiled by typing in Cbc-2.6.1/build/Cbc/examples:

```
% make DRIVER=example_name
```

where example_name is one of: allCuts, barrier, CbcBranchFollow2, CbcBranchLink, cbc_driverC_sos, CbcBranchUser, CbcCompareUser, CbcSolver2, CbcSolver3, CbcSolverLongThin, ClpQuadInterface, crew, driver2, driver3, driver4, driver, fast0507b, fast0507, gear, hotstart, link, longthin, lotsize, minimum, modify, nway, qmip, qmip2, quad, quad2, repeat, sample1, sample2, sample3, sample4, sample5, simpleBAB, sos, sudoku.

Note that when linking your own code with libraries located in directory Cbc-2.6.1/build/lib, you should always use the header files in Cbc-2.6.1/build/include/coin, not the header files that you can find in other subdirectories of Cbc-2.6.1.

In the rest of this document, we will use SrcCbc as a shorthand for the directory Cbc-2.6.1/Cbc/src and we will look at header files there, but this is just for convenience, as this directory contains all the files we are interested in.

3 Documentation and Custom Configuration

The Trac pages for Cbc are located at [3, 4]. Additional information, access to mailing lists, and instructions for reporting bugs can be found there.

Please refer to the first project (see proj1.pdf) for the generation of the html documentation and useful flags for the configure scripts.

After generating the html documentation, open in a browser the file Cbc-2.6.1/build/doxydoc/html/index.html, click on the link Classes on top of the page and make a bookmark reference to that page. It will be referred to as DocCbc.

4 Command Line Interface

1. Go to the directory PROJ5.

The stand-alone code cbc can be run using:

2. % ~/Cbc-2.6.1/build/bin/cbc dcmulti.mps

It reads the file dcmulti.mps (an integer linear program in MPS format) and solves the corresponding ILP using Cbc and Clp, using all defaults of cbc. (Cbc can also read input files in LP format.) Some of the parameters of cbc can be controlled using the command line interface. To enter the command line mode, use:

3. `% ~/Cbc-2.6.1/build/bin/cbc -`

You should get the prompt "`Coin:`". Entering

4. `Coin:  ?`

gives a list of commands that `cbc` understands. Some of them, for example, are:

- `option`?: short help for command `option`;
- `import name`: read MPS or LP file `name`;
- `export name`: write MPS file `name`;
- `gomory option` where `option is one of`:
  - `on`: use Gomory cuts;
  - `off`: do not use Gomory cuts;
  - `root`: use Gomory cuts only at the root;
  - `ifmove`: use while Gomory cuts improve bound; this is the default setting;
  - `forceOn`: force use at every node;
- `roundingHeuristic on/off`: use/turn off rounding heuristic;
- `branchAndCut`: solve the ILP by branch-and-cut;
- `logLevel k`: set the level of output to k;
- `solution sol.txt`: print the optimal solution in file `sol.txt`. To print on screen use `stdout` as file name;
- `quit`: quit the program.

Note that this command line interface is not a very robust code, but the problem is only with the interface, not with the underlying library. It is quite easy to make act in strange ways (for example, try to import and solve `p0033.mps` and then import and solve `dcmulti.mps`), but the problem is only with the interface, not with the underlying library. To run `cbc` with its default settings, use:

```
Coin:  import dcmulti.mps
```

```
Coin:  branchAndCut
```

To prevent `cbc` to use Gomory cuts, use:

```
Coin:  import dcmulti.mps

Coin:  gomory off

Coin:  branchAndCut
```

Note that it is also possible to issue a single command collecting all the options that you want to set. For example the command

```
% ~/Cbc-2.6.1/build/bin/cbc -import p0033.mps \
              -gomory off -branchAnd -solution stdout
```

loads `p0033.mps`, turns off Gomory cuts generation, solve the problem and print the solution to the screen.

Do the following:

5. Write down the cpu time taken to solve `dcmulti.mps` with the default settings.

6. By changing the default settings for Gomory cuts, Mixed Integer Rounding cuts, Probing cuts, Rounding heuristic and strong branching what improvement in term of running time can you get when solving `dcmulti.mps`? Experiment with four or five settings.

7. Set the logLevel to value k for $0 \leq k \leq 4$ and try to understand what is in the output in each case.

5 Compiling the Project Code

The files `proj5.cpp`, `p5_driver4.cpp` and related code can be compiled by typing in the directory `PROJ5`:

8. `% make`

If the compilation fails, you might have to modify some of the entries in `PROJ5/makefile`. Here are some modifications that might be needed:

1) If you placed the `build` directory somewhere else than in the recommended `$(HOME)/Cbc-2.6.1/build`, replace in the makefile `CBC = $(HOME)/Cbc-2.6.1/build` by the correct full path.

2) If you get an error message at run time saying that some shared COIN-OR library can not be loaded, you might need to add the path to the `Cbc-2.6.1/build/lib/coin` directory into the environment variable `LD_LIBRARY_PATH`.

3) If the configure script used in step 6 of Section2 was able to find `Lapack` and `Blas` libraries on your machine you might need to define additional environment variables, as described in Section 4.1 of `proj1.pdf`. To check if the configure script indeed found `Lapack`, use:

```
% grep lapack last_configure.txt
```

If `Lapack` was found, this should print a line resembling on of the following:

```
checking whether -llapack has LAPACK...  yes
checking whether user supplied LAPACKLIB="/usr/lib/liblapack.so.3"
works...  yes
```

The code can be run using:

9. `% ./proj5`

and enter, for example, `p0033.mps` when prompted for an input file. The code currently asks for the input file name and accepts either an LP file (the extension of the file name must be ".lp") or an MPS file (the extension of the file name must be ".mps"). It then reads the input file, solves the ILP, and prints the optimal solution. The code can be compiled with the flag `-DTRACE` (see file `PROJ5/makefile`, line starting with `DEFS =`) or without it. Removing it will remove most of the default output. Additional flags are `-DPRIORITIES`, `-DPSEUDO` , and `-DPSEUDODYN` whose use and meaning will be explained in Section 11. The code `p5_driver4` is an example of customization using many of the default setting of `Cbc` and is described in Section 12.

6 Class CbcModel

The class `CbcModel` is the central class of `Cbc`. A model holds the description of the problem, and, among others, the branching decision rules, the cut generators, and the heuristics to use during the solution process. It also gives access to the result of the optimization.

The simplest way to create a `CbcModel` object is to read an LP or MPS file in an `OsiSolverInterface` object and then to create the model, as done in `PROJ5/proj5.cpp`. It is of course possible to build the representation of the problem in the `OsiSolverInterface` object from memory instead of reading a file, similarly to what was done earlier in projects 2 and 3.

Useful methods[1]:

- `branchAndBound()`: Use branch-and-cut to optimize the problem contained in the model;
- `setIntegerTolerance()`: Set tolerance for deciding if a number is integer or not;
- `setMaximumSeconds()`: Set the time limit.
- `setNumberStrong()`: Set the number of candidates considered during strong branching;
- `setLogLevel()`: Set the level of the output.
- `setPrintFrequency()`: Set the frequency for printing the status line for the optimization;
- `getObjValue()`: Give the value of the best known feasible solution;
- `bestSolution()`: Give a pointer on the best known feasible solution;
- `phase()`: Get the phase number of the optimization;
- `setBestSolution()`: Set the best known solution and its value;
- `setBestObjectiveValue()`: Set the objective value to beat;
- `addCutGenerator()`: Add a `Cgl` cut generator (see Section 7);
- `addHeuristic()`: Add a heuristic (see Section 8);
- `setNodeComparison()`: Set the rule for selecting the next node to process (see Section 9);
- `setBranchingMethod()`: Set the rule used to select the best candidate during strong branching (see Section 10);

[1] `SrcCbc/CbcModel.hpp`, `cSrcCbc/CbcModel.cpp` or `DocCbc->CbcModel`.

- `addObjects()`: Add to the model objects describing the integrality of the variables (see Section 11);

Do the following:

10. Find out how to set a limit on the number of cutting passes at the root node. Modify the file `PROJ5/proj5.cpp` to limit the number of such passes to 10.

11. Find out how to specify an upper bound on the optimal solution value (for a minimization problem). Modify the file `PROJ5/proj5.cpp` so that the user is prompted for giving such an upper bound at the beginning of the execution.

7 Pre-defined Cut Generators

The `Cgl` library contains a number of cut generators that can easily be added to the model. Assuming that the `CbcModel` object has name `model`, the key method is `model.addCutGenerator(generator, mode, name);` where

- `generator` is a pointer on a `Cgl` generator, such as `CglGomory` for Gomory cuts or `CglKnapsack` for knapsack covers (see `DocCbc` for the complete list of available generators);

- `mode` is an integer controlling how often the generator is called. If `mode` is set to k with

    i) $k = 0, 1$: the generator is called at every node;
    ii) $k > 1$: the generator is called every k nodes;
    iii) $-98 \leq k < 0$: the generator is called every $(-k)$ nodes but it may be switched off by `Cbc`;
    iv) $k = -99$: the generator is only called at the root node;
    v) $k < -99$: the generator is not used.

- `name` is a string of characters that will be associated with the generator and used when output is written.

There are additional parameters to the method `addCutGenerator()` but the above ones are sufficient for most applications. The file `PROJ5/proj5.cpp` gives an example for using the Probing cut generator and the Gomory cut generator, including parameter setting for the generators.

Do the following:

12. Add a Mixed Integer Rounding cut[2] generator to the model defined in `PROJ5/proj5.cpp`. Set the maximum number of aggregations to 10. Make sure that the generator will be called every 10 nodes, leaving to `Cbc` the option to turn it off at any time. Don't forget to add the required `include` statements.

8 Heuristics

Some of the predefined heuristics are:

- Rounding: class `CbcRounding`[3];
- Relaxation Induced Neighborhood Search (RINS): class `CbcHeuristicRINS`[4] (see [2]);
- Feasibility Pump: class `CbcHeuristicFPump`[5] (see [10]);
- Greedy: class `CbcHeuristicGreedy`[6];
- Pivot and Fix: class `CbcHeuristicPivotAndFix`[7];
- Diving: classes `CbcHeuristicDiveCoefficient`[8], `CbcHeuristicDiveFractional`[9], CbcHeuristicDiveGuided[10], `CbcHeuristicDiveVectorLength`[11],

[2] `Cbc-2.6.1/Cgl/src/CglMixedIntegerRounding/CglMixedIntegerRounding.hpp`, `Cbc-2.6.1/Cgl/src/CglMixedIntegerRounding/CglMixedIntegerRounding.cpp` or `DocCbc->CglMixedIntegerRounding`.

[3] `SrcCbc/CbcHeuristic.hpp`, `SrcCbc/CbcHeuristic.cpp` or `DocCbc->CbcRounding`.

[4] `SrcCbc/CbcHeuristicRINS.hpp`, `SrcCbc/CbcHeuristicRINS.cpp` or `DocCbc->CbcHeuristicRINS`.

[5] `SrcCbc/CbcHeuristicFPump.hpp`, `SrcCbc/CbcHeuristicFPump.cpp` or `DocCbc->CbcHeuristicFPump`.

[6] `SrcCbc/CbcHeuristicGreedy.hpp`, `SrcCbc/CbcHeuristicGreedy.cpp` or `DocCbc->CbcHeuristicGreedy`.

[7] `SrcCbc/CbcHeuristicPivotAndFix.hpp`, `SrcCbc/CbcHeuristicPivotAndFix.cpp` or `DocCbc->CbcHeuristicPivotAndFix`.

[8] `SrcCbc/CbcHeuristicDiveCoefficient.hpp`, `SrcCbc/CbcHeuristicDiveCoefficient.cpp` or `DocCbc->CbcHeuristicDiveCoefficient`.

[9] `SrcCbc/CbcHeuristicDiveFractional.hpp`, `SrcCbc/CbcHeuristicDiveFractional.cpp` or `DocCbc->CbcHeuristicDiveFractional`.

[10] `SrcCbc/CbcHeuristicDiveGuided.hpp`, `SrcCbc/CbcHeuristicDiveGuided.cpp` or `DocCbc->CbcHeuristicDiveGuided`.

[11] `SrcCbc/CbcHeuristicDiveVectorLength.hpp`, `SrcCbc/CbcHeuristicDiveVectorLength.cpp` or `DocCbc->CbcHeuristicDiveVectorLength`.

CbcHeuristicDivePseudoCost[12], CbcHeuristicDiveLineSearch[13].

Assuming that the CbcModel object has name model, the key method to add an heuristic is model.addHeuristic(heur); where heur is a pointer on an object of one of the above classes or on an object of a user defined heuristic class.

To code a new heuristic, a new class has to be derived from the class CbcHeuristic defined in the same files as the Rounding heuristic mentioned above. As a simple example, the files PROJ5/p5_HeuristicRound.hpp and PROJ5/p5_HeuristicRound.cpp implement in class p5_HeuristicRound the simplest rounding heuristic where each integer variable value is rounded to the nearest integer. Besides a number of constructor, destructor, clone, and assignment methods that must be implemented as they are virtual in the base class, the core of the heuristic is implemented in the method p5_HeuristicRound::solution().

Do the following:

13. Add the Feasibility Pump heuristic available in Cbc to the model in PROJ5/proj5.cpp. Set the maximum number of passes to 10. Don't forget to add the required include statements.

9 Node Selection

Some of the predefined node selection rules[14] are:

- Depth-First: class CbcCompareDepth[15];
- Best-First: class CbcCompareObjective[16];
- Hybrid strategy: class CbcCompareDefault[17]. This is the default comparison rule of Cbc. It is based on the infeasibility of the nodes their depths and objective values. See the code for more details.
- Heuristic strategy: class CbcCompareEstimate[18]. Selection based on an estimation of the objective value of the subproblem.

[12] SrcCbc/CbcHeuristicDivePseudoCost.hpp, SrcCbc/CbcHeuristicDivePseudoCost.cpp or DocCbc->CbcHeuristicDivePseudoCost.

[13] SrcCbc/CbcHeuristicDiveLineSearch.hpp, SrcCbc/CbcHeuristicDiveLineSearch.cpp or DocCbc->CbcHeuristicDiveLineSearch.

[14] SrcCbc/CbcCompareActual.hpp, SrcCbc/CbcCompareActual.cpp.

[15] DocCbc->CbcCompareDepth.

[16] DocCbc->CbcCompareObjective.

[17] DocCbc->CbcCompareDefault.

[18] DocCbc->CbcCompareEstimate.

Assuming that the CbcModel object has name model, the key method to set the comparison rule is model.setNodeComparison(compare); where compare is a pointer on an object of one of the above classes or on an object of a user defined comparison class.

To code a new comparison rule, a new class has to be derived from the class CbcCompareBase[19]. For example, files PROJ5/p5_CompareDFS_BFS.hpp and PROJ5/p5_CompareDFS_BFS.cpp implement class p5_CompareDFS_BFS corresponding to the following selection rule: Start with Depth-First-Search (DFS) until either two feasible solutions have been found or one thousand nodes have been processed. Then, switch to Breadth-First-Search (BFS).

The class p5_CompareDFS_BFS has a data member called do_DFS which indicates if DFS or BFS should be used. Besides a number of constructor, destructor, clone, and assignment methods that must be implemented as they are virtual in the base class, the core of the rule is implemented in the method p5_CompareDFS_BFS::test().

The method p5_CompareDFS_BFS::newSolution() is called each time a new feasible solution is found. It is used to switch to DFS as soon as two feasible solutions have been found.

The method p5_CompareDFS_BFS::every1000Nodes() is called every thousand nodes, starting from the root node. It is used to switch to DFS if one thousand nodes have been processed.

Do the following:

14. Try to understand the code for the default node selection of Cbc. Good luck.

10 Best Candidate Selection Rule

After strong branching on several candidate is done, a selection rule for the branching variable is required. Some of the predefined candidate selection rules are:

- Default selection: class CbcBranchDefaultDecision[20]. Selection is based on four different criteria. See the code for more details;

[19] SrcCbc/CbcCompareBase.hpp, SrcCbc/CbcCompareBase.cpp or DocCbc->CbcCompareObjective.

[20] SrcCbc/CbcBranchActual.hpp, SrcCbc/CbcBranchActual.cpp or DocCbc->CbcBranchDefaultDecision.

- Alternative selection: class `CbcBranchDynamicDecision`[21]. Selection is based on infeasibility until a feasible solution is found by search, then switch to selection based on change in objective value.

Assuming that the `CbcModel` object has name `model`, the key method to set the candidate selection rule is `model.setBranchingDecision(bdec);` where `bdec` is a pointer on an object of one of the above classes or on an object of a user defined candidate selection class.

To code a new selection rule, a new class has to be derived from the class `CbcBranchBase`[22]. As a simple example, the files `PROJ5/p5_Branch.hpp` and `PROJ5/p5_Branch.cpp` implement class `p5_Branch` corresponding to the following candidate selection rule: Pick the candidate for which the minimum improvement in the value of the objective function in its two sons is maximum.

Besides a number of constructor, destructor, clone, and assignment methods that must be implemented as they are virtual in the base class, the core of the rule is implemented in the method `p5_Branch::bestBranch()`.

Do the following:

15. Modify the files `PROJ5/p5_Branch2.hpp,cpp` (the class defined in these files is identical to the class `p5_Branch`) to implement a new class `p5_Branch2` for the selection rule of [1]: If branching on variable x_i gives an improvement of q_1 in the first son and an improvement q_2 in the second son, the score associated with branching on x_i is

$$\frac{5}{6}\min\{q_1, q_2\} + \frac{1}{6}\max\{q_1, q_2\}\ .$$

The index of the branching variable is chosen so that it maximizes the above value. Use your new class instead of `p5_Branch` in the model defined in `PROJ5/proj5.cpp`.

11 Variable Selection for Strong Branching

Without going into too much details, it might be useful to understand how integrality conditions are used and stored in `Cbc`. For each integer vari-

[21] `SrcCbc/CbcBranchDynamic.hpp`, `SrcCbc/CbcBranchDynamic.cpp` or `DocCbc->CbcBranchDynamicDecision`.

[22] `SrcCbc/CbcBranchBase.hpp`, `SrcCbc/CbcBranchBase.cpp` or `DocCbc->CbcBranchBase`.

able, Cbc associates an object of class CbcObject[23]. An object has methods, among others, to check feasibility and generate branching objects. By default, Cbc associates an object of class CbcSimpleInteger[24] to each integer variable x_i, implementing the usual branching rule "either $x_i \leq k$ or $x_i \geq k+1$" for some integer k.

The simplest way to direct the choice of variables for performing strong branching in Cbc is to define priorities on the integer variables, i.e. to associate an integer p_i with each integer variable x_i. Then, when candidates for strong branching are selected, variable x_i will be preferred to variable x_j if $p_i < p_j$. To use priorities with the code, set the flag -DPRIORITIES on the line starting by "DEFS =" in PROJ5/makefile and recompile using make.

Another way to direct the choice of variables for performing strong branching is to define pseudo-costs associated with each integer variables. Although dynamic pseudo-costs (i.e. pseudo-costs that are updated during the search) are more powerful, the example in PROJ5/proj5.cpp just sets static pseudo-costs: For integer variable x_i with objective coefficient c_i, its pseudo-cost for branching up or down is set to the absolute value of c_i.

The predefined class CbcSimpleIntegerPseudoCost[25] is convenient to use pseudo-costs. We just need to construct an object of this class for each integer variable, collect these objects in an array and use the method model.addObjects() to add them to the model.

If we add an object of class CbcSimpleIntegerPseudoCost associated with variable x_i, Cbc recognizes that this is an object of a class derived from CbcSimpleInteger and will replace the existing object by the new one.

To use pseudo-costs with the code, set the flag -DPSEUDO on the line starting by "DEFS =" in PROJ5/makefile and recompile proj5 using make in PROJ5. Note that using both flags -DPRIORITIES and -DPSEUDO is identical to using only the second one, as the priorities are passed to the model before the objects used for pseudo-costs are passed to the model. It is possible to define both priorities and pseudo-costs and the candidate selection will use both in a non trivial way (see method CbcNode::chooseBranch()[26]).

Do the following:

[23] SrcCbc/CbcBranchBase.hpp, SrcCbc/CbcBranchBase.cpp or DocCbc->CbcObject.

[24] SrcCbc/CbcBranchActual.hpp, SrcCbc/CbcBranchActual.cpp or DocCbc->CbcSimpleInteger.

[25] SrcCbc/CbcBranchActual.hpp, SrcCbc/CbcBranchActual.cpp or DocCbc->CbcSimpleIntegerPseudoCost.

[26] SrcCbc/CbcNode.cpp or DocCbc->CbcNode.

16. Use the class `CbcSimpleIntegerDynamicPseudoCost`[27] which implements the dynamic pseudo-costs of [1] instead of the pseudo-costs used in `PROJ5/proj5.cpp`. Just defining objects of that class for each integer variable and adding them to the model is enough. Add your code between

    ```
    #ifdef PSEUDODYN
        ...
    #endif
    ```

    add the flag `PSEUDODYN` (removing `-DPRIORITIES` and `-DPSEUDO` if present) on the line starting with `''DEFS =''` in the makefile and recompile using `make`. Don't forget to add the required `include` statements.

12 Customizing the default `Cbc` code

In this section, we use a combination of the command line options (see Section 4) and the customization covered in the subsequent sections to add customization to the default `Cbc` settings. The code is based on the example `driver4` that can be found in `Cbc-2.6.1/Cbc/examples`. The main part of the code is in file `p5_driver4.cpp`. The code expects the name of the input file (either an LP or MPS file) followed by the `Cbc` command line options that should be passed to `Cbc`, in the same format as seen in Section 4.

For example, you can use:

```
% ./p5_driver4 p0033.mps -logLevel 2 -branchA
```

The code in `p5_driver4.cpp` reads the input file, asks for an upper bound value that will be passed to `Cbc`, creates a `CbcModel`, adds to it an event handler, and call `CbcMain1()` with the command line options given to `p5_driver4`. Finally, it prints the best known solution in file `f_sol.xxx`. Customization is done in two ways: Using the event handler and callback. These two options are described in more detail in the next two sections.

The files `p5_miscMeth.hpp` and `p5_miscMeth.cpp` contain several simple methods: `printFileName()` to print the name of the input file and upper bound, `printInfoLine()` to print the best known solution value, the current LP relaxation bound, the cpu time, node number, final `Cbc` status

[27] `SrcCbc/CbcBranchDynamic.hpp`, `SrcCbc/CbcBranchDynamic.cpp` or `DocCbc->CbcSimpleIntegerDynamicPseudoCost`.

and secondary status[28], `printSol()` to print the best known solution and `getSolverPtr()` to get a pointer on the solver of a `CbcModel`. These method are useful to perform the tasks listed below.

12.1 Class `p5_CbcEventHandler`

The class `p5_CbcEventHandler`[29] is derived from class `CbcEventHandler`[30] and the only non trivial method is `p5_EventHandler::event()`. That function is called from `Cbc` each time some predefined CbcEvent[31] occurs. The events we are interested here are `node`, `solution`, and `heuristicSolution`. A `node` event occurs at the end of the processing of a node, a `solution` event when a feasible solution is found by the LP relaxation, and a `heuristicSolution` event when a feasible solution is found by one of the heuristics. By adding code in `p5_EventHandler::event()`, one can direct `Cbc` to perform operations each time one of these events occur. The method should return a `CbcAction`[32] and can be either `noAction` (i.e. continue regular operations), `stop`, `restart`, or `restartRoot` which are self-explanatory.

As an illustration, try to implement the following:

17. Print a line info in file `f_res.xxx` (using a call to `printInfoLine()`) and print on the screen the current best known solution value of the preprocessed problem each time a feasible solution is found.

18. Print the number of integer variables with fractional value in the solution of the final LP relaxation at each node of the tree.

12.2 Callback

It is possible to pass a callback method to `Cbc`. An example of such a method is `callback()`[33] which has a parameter `whereFrom` that can have any value between 1 and 5 with the following meaning:

- 1: After solving the initial LP relaxation;
- 2: After preprocessing;

[28] See `SrcCbc/CbcModel.hpp` or `DocCbc->CbcModel` for a description of status and secondary status.

[29] See files `p5_EventHandler.hpp`, `p5_EventHandler.cpp`.

[30] `SrcCbc/CbcEventHandler.hpp`, `SrcCbc/CbcEventHandler.cpp` or `DocCbc->CbcEventHandler`.

[31] `SrcCbc/CbcEventHandler.hpp`, `DocCbc->CbcEventHandler`.

[32] `SrcCbc/CbcEventHandler.hpp` or `DocCbc->CbcEventHandler`.

[33] See files `p5_callback.hpp` and `p5_callback.cpp`.

- 3: Just before starting the branch-and-cut;
- 4: Just after finishing the branch-and-cut (before post processing);
- 5: After post processing.

The method `callback()` is called by `Cbc` at each of these steps, with the corresponding value for `whereFrom`. It is thus possible to direct `Cbc` to perform operations depending on that value.

As an example, try to implement the following:

19. Direct `Cbc` to use the node comparison rule `p5_CompareDFS_BFS`, to use the Feasibility Pump heuristic at every node of the tree with a maximum of 10 passes, to use the `CbcHeuristicDiveFractional` heuristic at every node of the tree with a maximum time of 5 minutes, no limit on iteration number, fixing all integer variables with current integer value, and a maximum of 5000 nodes, as well as the simple rounding heuristic `p5_HeuristicRound` at every node of the tree. All this can be implemented in the method `setMySelection()`[34] that is called by `callback()` just before the start of the branch-and-cut.
20. Make a call to `printLineInfo()` to print the final information line at the end of the solution process.

References

[1] T. Achterberg, T. Koch, A. Martin, "Branching Rules Revisited", *Operations Research Letters* 33 (2005), 42–54.

[2] Danna E., Rothberg E., Le Pape C., "Exploring Relaxation Induced Neighborhoods to Improve MIP Solutions", *Mathematical Programming* 102 (2005), 71–90.

[3] `http://www.coin-or.org/projects/Cbc.xml`

[4] `https://projects.coin-or.org/Cbc`

[5] `http://www.coin-or.org/download/source`

[6] `https://projects.coin-or.org/CoinHelp`

[7] `https://projects.coin-or.org/CoinHelp/wiki/user-troubleshooting`

[8] J. Forrest, R. Lougee-Heimer, "CBC User Guide", `http://www.coin-or.org/Cbc/cbcuserguide.html`

[34]See file `p5_callback.cpp`.

[9] `http://www.stack.nl/~dimitri/doxygen/`

[10] M. Fischetti, F. Glover, A. Lodi, “The feasibility pump” *Mathematical Programming* 104 (2004), 91–104.

[11] `http://subversion.tigris.org/`