BAC : A BCP based Branch-and-Cut Example

François Margot[1]

May 2003, Updated March 2008

Abstract

This paper is an introduction to the Branch-and-Cut-and-Price (`BCP`) software from the user perspective. It focuses on a simple example illustrating the basic operations used in a Branch-and-Cut: cuts and heuristic solutions generation, and customized branching.

[1]Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213-3890, U.S.A. Email: `fmargot@andrew.cmu.edu` .
Work initiated in 2003 while visiting the Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

Contents

1 Introduction

This paper is an introduction to the branch-and-cut-and-price (BCP) software available in the COIN-OR repository [1]. Its scope is rather limited as its goal is to allow a new user to develop quickly his first application. The perspective is from a user point of view, skipping implementation details and options that are irrelevant for developing a simple application (i.e. "the less I know about BCP, the better"). In particular, it focuses only on setting up a branch-and-cut algorithm, as adding the column generation process should not be too difficult once the branch-and-cut part is understood and set up.

All parts of the example were written for illustration purposes and were chosen to be mathematically as simple as possible. No claim is made regarding the efficiency or style of the code of the examples. Some operations could be done more efficiently by using additional features of BCP, at the cost of clarity. Learning how to use additional features and facilities of BCP can be done later on.

The reader is assumed to be familiar with the branch-and-cut process and its standard terminology. (An excellent introduction can be found in [10, 13, 16, 18].) Basic knowledge of C++ is also required.

The code of the example BAC is based on the example BranchAndCut written by L. Ladányi that was available in the COIN-OR repository. Additional introductory material for BCP and COIN-OR in general is available from [14]. In particular, it provides the SHELL example that can be used as template for developping a new application.

Three other examples for column generation with BCP are available in the directory Bcp/examples of the BCP package: MCF-1, MCF-2, and MCF-3. Limited documentation for these three examples can be found in their respective directories. More involved codes based on BCP can be found in the Applications directory in the root directory of the BCP package: MaxCut (an efficient branch-and-cut code for solving Maximum Cut problems) [9], Csp (a branch-and-cut-and-price code for solving cutting stock problems) [12], and Mkc (a branch-and-cut-and-price code for solving Multiple Knapsack with Color problems) [11].

This document can be read without knowledge of the BAC code itself, but it is probably better to have access to the code while reading.

Section 2 covers the installation of the COIN-OR package BCP and Section 3 the generation of its html documentation, including the documentation of the BAC example. Sections 4 and 5 explain how to customize the installation of BCP by adding support for the debugger, and Cplex. Section

6 gives a general overview of BCP and other packages. Section 7 covers the installation and compilation flags of the BAC example. Section 8 lists two data structures defined in BCP that are used in the example: vectors and matrices. Section 9 describes the integer linear problem solved in the BAC example. Section 10 describes the three types of BCP variables and constraints. Section 11 covers the three main classes of the example: the class BB_prob (description of the problem), BB_TM (tree manager) and BB_LP (operations at the nodes of the tree). Section 13 describes how the user can define objects associated with the nodes of the enumeration tree that depend on the parent node. Section 14 explains the cryptic part of the output and Section 15 discusses briefly the parameter file for BCP. Finally, Section 16 gives pointer to useful modifications of the code to improve its performances when doing pure branch-and-cut.

The description in this paper corresponds to the code in the tar ball Bcp-1.1.2.tgz. Questions or bugs related to BCP, BAC or COIN-OR in general should be directed to the bugs reporting sites of the COIN-OR repository and other mailing lists and Trac pages available there [1, 2, 3, 6].

2 BCP Download and Installation

This section covers the installation of the software BCP on a machine running Linux Fedora 8 with the tc shell and having the commercial software Cplex installed. If Cplex is unavailable, simply skip the corresponding installation steps. If you run another shell or use another Linux distribution, some of the Linux commands might be slightly different. Note that this project can be compiled under Cygwin on Windows machines, provided that a small modification of the project makefile is done (as indicated in the makefile).

Below, the symbol ">" replaces the Linux prompt.

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) [17]. While using svn is more flexible, it is also more complex. For more information about using svn go to the COIN-OR help pages [4].

1. Go in your main directory.

2. Download the BCP package tar ball: Using a web browser, go to the COIN-OR download site [7], click on BCP and download the most recent tar ball of BCP. At the time of this writing, this is Bcp-1.1.2.tgz.

3. Decompress the tar ball

   ```
   > tar -xvf Bcp-1.1.2.tgz
   ```

4. Go in the newly created directory `Bcp-1.1.2`:

   ```
   > cd Bcp-1.1.2
   ```

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

   ```
   > mkdir build
   > cd build
   ```

6. Configure the package according to your system and install the example codes in `Bcp-1.1.2/build/Bcp/examples`:

   ```
   > ../configure -C
   ```

   If the last few lines of the output do not contain:

   ```
   configure:  Main configuration of Bcp successful
   ```

   then the configuration failed and you might need to provide information to the configuration script. The Trac pages [4, 5] might help you figure out what is wrong.

7. Compile the code:

   ```
   > make
   ```

8. To install the include files in the directory `Bcp-1.1.2/build/include`, the libraries in the directory `Bcp-1.1.2/build/lib`, and the executables in the directory `Bcp-1.1.2/build/bin` use:

   ```
   > make install
   ```

Note that the use of the directory `Bcp-1.1.2/build` is not absolutely necessary and it is possible to build the code from `Bcp-1.1.2` or from any other directory of your choosing. The only difference is that Step 6 must then be replaced with the appropriate command launching the configure script located in `Bcp-1.1.2`. The advantage of using a separate build directory instead of using `Bcp-1.1.2` is that it is then easy to have several versions of the code simultaneously by creating separate build directories for each version, as we will see below. It makes it also easy to just delete the `build` directory to start configuration from scratch, as a last resort.

If you have trouble compiling, installing or linking your code, you might need to follow the steps listed in Section 5. Note that when linking your own code with libraries located in directory `Bcp-1.1.2/build/lib`, you should always use the header files in `Bcp-1.1.2/build/include`, not the header files that you can find in other subdirectories of `Bcp-1.1.2`.

3 Documentation

The Trac page for `Bcp` is located at [3]. Additional information, access to mailing lists, and instructions for reporting bugs can be found there.

If you have `Doxygen` [8] available on your machine, you can build the html documentation by typing, in the directory `Bcp-1.1.2/build`:

```
> make doxydoc
```

Then open the file `Bcp-1.1.2/doxydoc/html/index.html` with a browser and click on the link `Classes` on top of the page. In your browser, make a bookmark reference to that page. This page will be referred to as `DocBcp` in the future.

Note that the above command creates the documentation for all the projects in the `BCP` package. If you prefer to generate the documentation only for a subset of these projects, you can skip some directories by editing the file `Bcp-1.1.2/doxydoc/doxygen.conf` (by setting the value of the `EXCLUDE` variable, for example) and reissuing the above command.

If `Doxygen` is not available on your machine, the documentation is available at [7].

4 Custom Configuration of BCP

All of this section is optional and can be skipped with no harm. We nevertheless recommend to do an installation of a version of the code compiled with the debugger option. **If, for some reason, you prefer not to do that**, please copy `Bcp-1.1.2/build` to `Bcp-1.1.2/buildg` using the following command from the directory `Bcp-1.1.2`:

```
> cp -r build buildg
```

The above command should only be used by people wishing to skip this section.

Assuming that you were successful in installing the code, following the instructions above, we can now start from scratch and add customization.

For example, we want to compile everything with the debugger option and add access to `Cplex`. If you do not have `Cplex`, skip the part of the text related to it.

To be able to use `Cplex`, its location on the system should be given to the configure script. For simplicity, we use environment variables for this.

The configure script can be customized using a list of parameters. To see a short list, type in `Bcp-1.1.2/build`:

```
> ../configure --help
```

To see the full list, type:

```
> ../configure --help=recursive
```

The parameters to the script can be passed either through the command line, or with a configuration file. We use the latter in this document. For more information about configuration options or use of the command line see [4].

1. Go to `Bcp-1.1.2`, create subdirectories `buildg`, create a `share` directory with a configuration file `config.site` and go to `buildg`:

   ```
   > cd ~/Bcp-1.1.2
   > mkdir buildg
   > cd buildg
   > cp -r ../BuildTools/share .
   ```

2. Open the file `Bcp-1.1.2/buildg/share/config.site` with your favorite editor.

In the remainder of this document, any mention of `config.site` refers to the file opened in Step 2 above and we use `buildg` as a shorthand for `Bcp-1.1.2/buildg`.

4.1 Using `Cplex`

Define the environment variable `ENV_CPLEX_LIB` and set it to point to the directory containing the `Cplex` library. It is assumed that its name is either `libcplex.a` or `libcplex.so`. Define the environment variable `ENV_CPLEX_H` and set it to point to the directory containing the `Cplex` header file (its name should be `cplex.h`).

For example, on my machine:

```
> setenv ENV_CPLEX_LIB \
  /usr/ilog/cplex110/lib/x86_rhel4.0_3.4/static_pic/

> setenv ENV_CPLEX_H /usr/ilog/cplex110/include/ilcplex
```

Note that these two commands need to be issued at the beginning of each session. You might want to put them in your `.tcshrc` file so that they are automatically executed when you log in.

Finally add the following lines in `config.site`. Make sure to type them without extra space and with the quotation signs.

```
with_cplex_lib="-L$ENV_CPLEX_LIB -lcplex -lpthread"

with_cplex_incdir=$ENV_CPLEX_H
```

4.2 Using the Debugger

Add the following line in `config.site`. Then the whole package will be compiled for use with a debugger.

```
enable_debug=yes
```

4.3 Compiling and Installing

It remains to configure, compile and install the code in `buildg`. This is done by following steps 6 to 8 of Section 2 from the directory `buildg`.

5 Re-configuring

If you want to recompile a package after changing options of the configure script (i.e. modifying `config.site`), you must use in the corresponding build directory the five commands:

```
> make uninstall

> make distclean

> ../configure -C

> make

> make install
```

The first command erases the installed libraries and header files. The second command erases all makefiles, object files and any information created by the original configure script. These two commands then essentially resets the package to its initial state. If you only want to recompile the code without reconfiguring, there is no need to use the first command and you may use `make clean` instead of `make distclean`. If all else fails, erase the directory `Bcp-1.1.2/build` and continue with steps 5 to 8 of Section2.

6 Overview of BCP

`BCP` is a collection of classes and methods handling the enumeration tree, constraints, and variables. It needs an LP solver, but the LP solver is not part of `BCP`. The interface between `BCP` and the solver is handled by the COIN-OR `Osi` (Open Solver Interface) library. The advantage of using `Osi` is that replacing an LP solver by another one requires only small (ideally zero) changes in the code. The example is written to run with the LP solver named `Clp`. All the code that you need is included when you download the code from the COIN-OR repository [1].

`BCP` handles only minimization problems.

`BCP` can be used for developing applications running on parallel machines. This comes with an additional cost for dealing with the passing of messages between processors (the "packing" and "unpacking" procedures present in many `BCP` classes). Since different processors are not assumed to share memory, it is impossible to use pointers to pass information between them. However, if the application runs only on a non parallel machine, the code can be simplified by the use of pointers in the packing and unpacking procedures. This is what is done in this example, and thus the example will not run on parallel machines. See other examples (`MCF-1, MCF-2, MCF-3` for proper "parallel" packing and unpacking procedures.

Four examples are installed in `Bcp-1.1.2/buildg/Bcp/examples`: `MCF-1`, `MCF-2`, `MCF-3` and `BAC`. The `MCF` examples are multicommodity flow examples using column generation, while `BAC` is an example for branch-and-cut. In the reminder of this document, we use `BAC` as a shorthand for `Bcp-1.1.2/buildg/Bcp/examples/BAC`. The source files of the `BAC` example are split into a number of subdirectories:

- `BAC/include` contains all of the header files.
- `BAC/TM` contains the code for the tree manager (`BB_tm.cpp`); it also contains the body of the `main()` procedure.

- BAC/LP contains the code for the operations at the node level (BB_lp.cpp).
- BAC/Member contains the code for classes in between TM and LP. In the example, it contains the code for handling cuts (BB_cuts.cpp), user data (BB_user_data.cpp) (see Section 13), and the initialization of the process (BB.cpp, BB_init.cpp).

7 BAC Compilation and Execution

Normally, typing make in the BAC directory is all that is needed to install the package, provided that the gnu make is available through that command and that the COIN-OR package BCP has been installed in the directory ∼/Bcp-1.1.2/buildg. If this is not the case, some modifications might be required, see the file INSTALL in the directory BAC. (When the location of files is mentioned below, the given path always starts implicitly from the directory BAC for files specific to the example or it starts with Bcp-1.1.2 or build for files related to BCP or other COIN-OR projects.)

The file Makefile has a line starting with "USER_DEFINES = ". Three flags can be put on this line :

- -DHEUR_SOL : Generate heuristic solutions (by simple rounding of the LP solution).
- -DCUSTOM_BRANCH : Use a customized branching strategy (branching on the first non-integer variable). If this flag is not set, the code uses the default branching of BCP (strong branching).
- -DUSER_DATA : The code associates with each node of the branch-and-cut tree a vector containing the indices of the variables set to 0 by branching decisions leading to the node. To avoid cluttering the presentation of the basic features with details related to user data, the reader interested in using user data will find all the relevant material in Section 13.

The executable (bac) is created in the directory BAC. If the flag(s) used are modified in the file Makefile, make sure to use make clean before issuing the make command. (The first line of the output gives the flags that were used when compiling Member/BB_init.)

Once the program is compiled, it can be run from the BAC directory by typing either ./bac or ./bac bb.par. If the last lines of the output obtained from using ./bac do not contain:

`TM: Default BCP_tm_user::display_feasible_solution() executed.`
then something is wrong. See the Trac pages [2, 3, 4, 5] for help.

The second command makes the code read the parameter file `bb.par`. (Parameters are discussed below in Section 15.) The obvious difference between the two commands is in the amount of output that is produced. The first command yields a detailed output. The second command prevents the code of diving and uses Depth-First-Search instead of Best-Bound in the selection of the next node to process.

If the LP file `bac.lp` is in the directory from which `bac` is run, that file is used as input file. Otherwise, the problem is constructed in memory. Comments in the code[1] show how to use MPS files instead of LP files.

Do not run the code with an LP file other than `bac.lp`, as the code likely will crash. To develop a new application, use the `SHELL` example from [14].

8 Data Structures

A few data structures are available within `BCP` and COIN-OR. Two of them will appear in the example: a class implementing vectors and a class implementing matrices.

The class `BCP_vec` is an implementation of an array with elements of generic type `T` with facilities for resizing. If `V` is a `BCP_vec<T>`, the following methods are used:

- `BCP_vec<T> V(`k`)` : creates a vector with k entries of type <`T`>.
- `V.size()` : returns the number of elements stored in `V`.
- `V[`i`]` : access the element stored at position i.
- `V.push_back(x)` : inserts element `x` to the end of the vector; if the space allocated to `V` is filled, then `V` is resized before inserting `x`.
- `V.clear()` : removes all entries in `V`.
- `CoinFillN(V,` n`, elem)` : put `elem` in the first n entries of `V`.

See the documentation `DocBcp->BCP_vec` for the full description of this class.

[1] `TM/BB_tm.cpp` in method `BB_tm::readInput()`.

The class `CoinPackedMatrix` implements a two dimensional matrix stored either by rows or by columns. For a `CoinPackedMatrix` `M` stored by rows, the following methods are used in the examples:

- `M = new CoinPackedMatrix(false,` a`,` b`)` : creates a matrix stored by rows (first Boolean parameter `false`). Roughly speaking, the parameter $0 \leq a \leq 1$ is a percentage of extra space to be allocated when a reallocation of the matrix occur: When trying to insert k new rows in a matrix having m rows and allocated space for p rows, with $k+m > p$, the matrix will be reallocated to store $(k+m)(1+a)$ rows. The parameter b is similar, for reallocation when columns are added. Note that if a matrix stored by columns is created using `CoinPackedMatrix(true,` a`,` b`)`, then a is used for the reallocation of columns and b for the reallocation of rows.

- `submatrixOf(M, nb_ind, v_ind)` : extracts from `M` the submatrix formed by the rows of `M` with indices in the vector `v_ind` (having `nb_ind` entries).

- `M.times(v, v_res)` : Computes the matrix-vector product (`M v`) and puts the result in `v_res`.

See the documentation `DocCoin->CoinPackedMatrix` for the full description of this class.

9 The Problem

The example solves the following integer linear program with ten binary variables $x_0, \ldots, x_9$ (indices taken modulo 10):

$$
\begin{array}{lllll}
\text{Minimize} & \displaystyle\sum_{i=0}^{9} -x_i & & & \\
\text{s.t.} & x_i + x_{i+1} + x_{i+2} & \leq & 1, & \text{for } i = 0, \ldots, 9, \\
& x_0 + x_1 & = & 1 & \\
& x_i + x_{i+1} & \leq & 1, & \text{for } i = 1, \ldots, 9, \\
& x_1 + x_3 + x_9 & \leq & 1 & \\
& x_0 + x_2 + x_4 & \leq & 1 & \\
& x_0 + x_3 + x_7 & \leq & 1 & \\
& x_1 + x_4 + x_5 & \leq & 1 & \\
& x_5 + x_6 + x_7 & \leq & 1 & \\
& x_0 + x_6 + x_8 & \leq & 1 & \\
& x_i \in \{0, 1\} & & & \text{for } i = 0, \ldots, 9.
\end{array}
$$

The formulation is a little bit silly, as constraints $x_i + x_{i+1} \leq 1$ are implied by $x_i + x_{i+1} + x_{i+2} \leq 1$ and could thus be removed. However, since the purpose of this example is more to illustrate a few features of `BCP` than solving a mathematically interesting problem, this should be good enough.

Despite the apparent triviality of this integer linear program, the output of the code depends on the compilation flags in use (see Section 7 for the possible compilation flags).

10 Types of Constraints and Variables

`BCP` has three types of constraints (or cuts):

- Core constraints come from the initial LP formulation and are present in the LP at every node of the tree.
- Algorithmic constraints are cuts given implicitly by a separation algorithm. Algorithmic constraints, unlike core constraints, might be added or removed from the node LP. The user controls which cuts

are added, but cut removal is done by BCP based on a value called "row effectiveness" (number of consecutive iterations for which the corresponding slack variable is zero, for example). Limited user input (through the parameter file discussed in Section 15) is used for the definition of row effectiveness.

- Indexed constraints are constraints in bijection with a set of integers, such that (user defined) methods to generate the constraint from the corresponding integer and vice-versa are available. Indexed constraints can be seen as a special type of algorithmic cuts, having an extremely compact representation. They might be removed from the node LP similarly to the algorithmic cuts.

A typical use of indexed constraints is in the situation where some of the constraints of the initial formulation are more important than others and the initial formulation has a large number of constraints: Important constraints will become core constraints and the remaining ones will be indexed constraints, stored in an (indexed) array of constraints,

When developing a new application, the first decision to make is how to partition the constraints into the three classes.

Example: For the example described in Section 9, the core constraints are chosen as:

$$\begin{aligned} x_0 + x_1 &= 1 \\ x_i + x_{i+1} &\leq 1, \quad \text{for } i = 1, \ldots, 9. \end{aligned}$$

The indexed constraints are chosen as:

$$\begin{aligned} x_1 + x_3 + x_9 &\leq 1 \\ x_0 + x_2 + x_4 &\leq 1 \\ x_0 + x_3 + x_7 &\leq 1 \\ x_1 + x_4 + x_5 &\leq 1 \\ x_5 + x_6 + x_7 &\leq 1 \\ x_0 + x_6 + x_8 &\leq 1. \end{aligned}$$

Finally, the algorithmic constraints are chosen as:

$$x_i + x_{i+1} + x_{i+2} \leq 1, \quad \text{for } i = 0, \ldots, 9.$$

□

Instead of storing the sense ("$\geq$", "$\leq$", or "=") and right hand side of an inequality, BCP stores a lower bound and an upper bound for each inequality ("ranged" constraints). Setting one of the bounds to $\pm\infty$, or setting both bounds to the same value allows for the three possible senses.

BCP does not have have (yet) the possibility of using global cuts: all cuts passed to BCP are handled as local cuts, valid only in the subtree rooted at the node where the constraint is generated. The user may of course implement pools for holding global cuts, but he will then be responsible for the management of those cuts. While this might be done relatively easily for a non-parallel implementation, this becomes trickier when parallelism is involved.

In the example, coefficients of core and indexed constraints are stored in CoinPackedMatrix objects in the class BB_prob. To store algorithmic cuts, the class BB_cut[2] is used. It implements in a standard way a representation of a cut as its set of nonzero coefficients.

The variables in BCP may also be of one of the three types: core, algorithmic, or indexed. Since we focus here on a branch-and-cut, all variables are core variables. Each variable has an upper and a lower bound, possibly $\pm\infty$. In addition, each variable is labeled as integer, binary or continuous. Variables are internally numbered with integers, starting at 0. When BCP reports information related to variables, it is with respect to its internal numbering.

11 Main Classes: BB_prob, BB_tm, and BB_lp

The main() function is located in the file TM/BB_tm.cpp and uses an object of class BB_init to pass to BCP the user-defined classes as discussed in Section 12.

The most important classes for the example are BB_prob, BB_tm, and BB_lp. The class BB_prob[3] is used for the problem description and contains the definitions for handling core and indexed constraints. This is the class that the user modifies to store additional information about the problem.

11.1 Class BB_tm

The class BB_tm[4] (tree manager) contains a single object of type BB_prob, named desc, holding the description of the problem (defined by the user).

[2] DocBcp->BB_cut.

[3] include/BB.hpp, Member/BB.cpp or DocBcp->BB_prob.

[4] include/BB_tm.hpp, TM/BB_tm.cpp or DocBcp->BB_tm.

The remaining methods are essentially those for setting up the LP at the root, and for packing and unpacking algorithmic cuts.

The class `BB_tm` is derived from the class `BB_tm_user`[5]. Some of the data members in `BB_tm` are:

- `BB_prob desc` : object holding the description of the problem.
- `double EPSILON` : value used for numerical precision when comparing numbers of type `double`. This value is only for the user calculations, `BCP` having its own parameter for numerical precision. (`BCP` uses the value set for numerical precision in the LP solver.)
- `bool *integer` : `integer[`j`] = true` if and only if variable j is an integer variable;
- `double *clb, *cub` : `clb[`j`]` = lower bound on variable j. `cub[i]: upper bound on variable` j. (Use $\pm$`BCP_DBL_MAX`[6] for unbounded variables.)
- `double *obj` : `obj[j]` = objective function coefficient of variable j.
- `*rlb_core, *rub_core` : `rlb_core[`i`]` = lower bound for core constraint i. `rlb_core[`i`]` = upper bound for core constraint i.
- `*rlb_indexed, *rub_indexed` : `rlb_indexed[`i`]` = lower bound for indexed constraint i. `rlb_indexed[`i`]` = upper bound for indexed constraint i.
- `CoinPackedMatrix *core, *indexed` : matrices holding the coefficients of the core and indexed constraints. Core constraints will be transmitted to `BCP` through the method `initialize_core()` and `BCP` will manage them. Indexed constraints are managed by the user who decides which of them should be added to the formulation at the node level. Once an indexed constraint (or algorithmic cut) is added to the formulation at node S, it will remain in the formulation of all children of S, until deleted by `BCP` (based on row effectiveness).

Prominent methods in `BB_tm` are:

- `readInput()` : reads input data.

[5] `DocBcp->BCP_tm_user`.

[6] `buildg/include/BCP_math.hpp`.

- `pack_module_data()` : packs the data stored in `BB_prob` that the user wants to be available at the nodes of the tree. The corresponding unpacking method `unpack_module_data()` is in the class `BB_lp`[7]. In the example, this method is quite simple, as it simply writes the address of the object `desc` of class `BB_prob`. (This is the type of packing that is impossible to use to run the program on parallel machines).

- `pack_cut_algo()` : encodes algorithmic cuts. It uses the packing method defined in the class BB_cut[8].

- `unpack_cut_algo()` : decodes encoded algorithmic cuts. It uses one of the constructors defined in the class BB_cut [9].

- `initialize_core()` : Transmits core constraints and core variables to `BCP`.

- `create_root` : set up the data at the root node. In this example, this method is really used only when the flag `-DUSER_DATA` is set.

- `display_feasible_solution()` : self explanatory.

11.2 Class `BB_lp`

The class `BB_lp` contains the methods operating at the nodes of the tree: cut generation, branching selection, and heuristic solution generation among others. The main loop (exited by fathoming or branching decision) performs the steps in the following order (steps followed by a "(u)" indicate that the user has an entry point for that step):

1. Initialize the new node (u).
2. Solve the node LP.
3. Test the feasibility of the node LP solution (u).
4. Update the lower bound for the node LP.
5. Fathom the node (if possible).
6. Perform (logical, reduced cost) fixing on the variables (u).

[7] `DocBcp->BB_lp`.
[8] `DocBcp->BB_cut`.
[9] `include/BB_cut.hpp`, `Member/BB_cut.cpp`.

7. Update the row effectiveness records.
8. Generate cuts (u).
9. Generate a heuristic solution (u).
10. Fathom the node (if possible).
11. Decide to branch, fathom, or repeat the loop (u).
12. Add to the node LP the cuts generated during the iteration, if the loop is repeated.
13. Purge the constraint pool.

Note that if, in an iteration, cuts are generated but the decision to branch is taken, then the cuts are discarded unless the next node to be processed is one of the sons of the current node. Also, if variables are successfully fixed in Step 6 and primal feasibility is lost, the control returns to Step 2.

It is important to realize that `BCP` creates a single object of type `BB_lp` for the whole enumeration, instead of creating one per node of the enumeration tree. (This holds for a non-parallel execution; in the parallel case, `BCP` creates one such object per processor used to process nodes.) The data members are of course updated at each node of the tree, but the same object is used throughout the enumeration. This is somewhat counter-intuitive in an object-oriented setting, but is motivated by efficiency reasons: The amount of data that the user needs when processing a node might be quite large. Copying and sending it for each node would be rather inefficient. This implies that variables that the user adds to the description of the class might need to be initialized somewhere else than in the constructor of `BB_lp`. The method `initialize_new_search_tree_node()` described below is available for this.

Some of the data members in class `BB_lp` are:

- `BB_prob *p_desc` : pointer to the object `desc` of class `BB_tm`.
- `MY_user_data *p_ud` : pointer to the object `p_ud` of class `MY_user_data` associated with the node. See Section 13 for a description of this class.
- `int in_strong` : An integer variable having value 1 while `BCP` is performing strong branching and zero otherwise. Its use will be explained below when describing the method `test_feasibility()`.

- `double EPS` : A double holding the value of EPSILON defined in `BB_prob`.
- `BCP_vec<BCP_cut*> algo_cuts` : vector to hold pointers to algorithmic cuts generated but not yet transmitted to `BCP`.
- `BCP_vec<int> violated_cuts` : Vector used to store the indices of the indexed cuts violated by the current LP solution.

After solving the node LP, the following LP termination codes from class `BCP_termcode` [10] might arise:

- `BCP_Abandoned = 0x01`: some unresolved numerical problem happened in the LP solver and the problem was not solved to optimality.
- `BCP_ProvenOptimal = 0x02`: Optimal solution found.
- `BCP_ProvenPrimalInf = 0x04`: Primal infeasible.
- `BCP_ProvenDualInf = 0x08`: Dual infeasible.
- `BCP_PrimalObjLimReached = 0x10`: Not relevant.
- `BCP_DualObjLimReached = 0x20`: Dual bound is higher than the best known feasible solution value. (Possibly not solved to optimality.)
- `BCP_IterationLimit = 0x40`: Number of iterations in the solver hit the given limit. (Possibly not solved to optimality.)
- `BCP_TimeLimit = 0x80`: Time limit exceeded. (Possibly not solved to optimality.)

Note that several return codes can be added together to get the final return code if more than one applies.

Prominent methods in `BB_lp` are:

- `initialize_solver_interface()` : Entry point to communicate with the LP solver at the beginning of the execution (called only once from the root node). In the example, this method is used to turn off the printing of the output of `Clp`.
- `initialize_new_search_tree_node()` : Entry point at the beginning of the processing of a node. The associated LP is set up but not yet solved. Natural place for initializing user defined variables of `BB_lp`.

[10] `DocBcp->BCP_lp_result`.

- `modify_lp_parameters()` : Called each time an LP is solved by the LP solver. Used primarily for changing the maximum number of simplex iterations to perform while doing strong branching. It is also used to set the variable `in_strong` to its correct value and to print the current node LP in the file `lpnode.lp` (by uncommenting a few lines in the method, the problem is printed in file `lpnode.mps`).

- `test_feasibility()` : If the current LP is feasible, the LP solution satisfies in particular all core constraints, but the user has to check himself if the indexed and algorithmic cuts not in the current LP are satisfied too. If some of these cuts are violated, they can be immediately transmitted to `BCP` through the method parameter `cuts`. In the example, an alternative way is used: the two vectors `violated_cuts` and `algo_cuts` are holding indices or pointers to the violated cuts and these will be transmitted to `BCP` in the method `generate_cuts_in_lp()`. If all the indexed and algorithmic cuts are satisfied, the user still has to check if the LP solution satisfies the integrality conditions. This is done by calling the default method `BCP_lp_user::test_feasibility()`. Its return value is either a pointer to the current LP solution (when it is a feasible solution for the initial problem too) or the NULL pointer (otherwise).

  The method `BB_lp::test_feasibility()` is also called while the process is solving LPs for selecting the branching variable during strong branching. This is useful in particular in applications where heuristic solutions are generated during the feasibility check. In the example, the method returns immediately when it is called while doing strong branching (i.e. when called with `in_strong == 1`).

  The method is also called when the LP is infeasible and when other LP termination codes occurred. While this does not make much sense for a branch-and-cut, it does when column generation is possible. In the example, the method returns immediately if the last LP was not solved to optimality.

- `logical_fixing()` : Empty method in the example. Might be useful for other applications.

- `generate_cuts_in_lp()` : Transmits to `BCP` the violated indexed cuts and generated algorithmic cuts. `BCP` can easily use the methods defined in the Cut Generation Library (`Cgl`) included in `COIN`. Among others,

Gomory cuts, Knapsack covers, Lift-and-Project, and Odd Hole cuts are available.

- `generate_heuristic_solution()` : self explanatory. The method is active in the example only if the compilation flag `-DHEUR_SOL` is used. (See Section 7.) In the example, the solution simply rounds the current LP solution and checks if this rounded solution is feasible. The code is of course very similar to the code of the method `test_feasibility()` since the checking of indexed and algorithmic cuts is done in both methods. Introducing methods for those tests would certainly make sense, but this was not done to keep the code as simple as possible. The predefined type `BCP_solution_generic`[11] derived from `BCP_solution` is used to encode the solution

- `select_branching_candidates()` : The method is active in the example only if the compilation flag `-DCUSTOM_BRANCH` is used. (See Section 7.) In the example, the variables are considered in order and branching is performed on the first fractional variable.

- `cuts_to_rows()` : Required method when algorithmic or indexed cuts are used. It describes how to get a row of the constraint matrix from the representation of the cut. If `BB_cut` is used, the two representations are close and this might seem to be a redundant method. However, for some problems, it is possible to store a cut in a compact form avoiding the storage of all its non-zero coefficients. (An extreme example of this is the case of the indexed cuts.) The method generating the coefficients from the compact representation is then necessary.

12 Initialization of User Classes and Class `BB_init`

The two main user-defined classes `BB_tm` and `BB_lp` must be passed to `BCP`. This in done through an additional class, `BB_init`[12] that has only two methods: `lp_init()` returning a pointer on a `BCP_lp_user` object (actually a `BB_lp` object) and `tm_init()` returning a pointer on a `BCP_tm_user` object (actually a `BB_tm` object).

The `main()` procedure, located in `TM/BB_tm.cpp`, simply calls the method `bcp_main()` with a `BB_init` object as parameter. Then `bcp_main()` calls the

[11] `DocBcp->BCP_solution`.

[12] `include/BB_init.hpp`, `Member/BB_init.cpp`, or `DocBcp->BB_init`.

two procedures BB_init::lp_init and BB_init::tm_init to get pointer on objects of these two classes.

Note that BB_init::tm_init() calls BB_tm::readInput() where the problem to be solved is actually set up. If you want readInput() to have access to additional command line arguments than currently (i.e. the parameter file, if any), modify that method as well as BB_init::tm_init() to pass the wanted arguments.

13 User Data

The class BCP_user_data[13] is used for handling data that the user wants to associate with each node of the tree. For some type of data, this can be done using the class BB_lp, but if the data associated with a node depends on the data of its father, the use of a class MY_user_data[14] derived from BCP_user_data is necessary. The user data used in the example consists in:

- is_processed : indicator for memory management (see below);
- max_card_set_zero : maximum length of vector set_zero;
- set_zero : vector of integers holding the indices of variables set to zero by branching decisions leading to the node;
- card_set_zero : number of entries stored in set_zero.

A sequential view of the operations involving the user data is as follows: The user data for the initial node is created in BB_tm::create_root(). Then the TM sends a node a to LP. The user data of a is packed and sent to LP. There, it is unpacked, the node a is processed, sons are (possibly) generated and their user data is set up. Then the user data of a and of its sons is packed and sent back to the TM. The TM unpacks them, replaces the user data of a by the (possibly) updated one and creates new nodes with associated user data for the sons.

In order to avoid the need to update several packing and unpacking methods when the user data is modified and to allow for the passing of pointer on the data instead of writing it explicitly, the class MY_user_data has two fields: the integer is_processed set to 1 when the corresponding node is processed at the LP level and a pointer on an object of type real_user_data holding the data defined by the user.

[13] DocBcp->BCP_user_data.
[14] DocBcp->MY_user_data.

Note that the way the memory management of the user data is set up implies that the data associated with a node will be destroyed as soon as the node has been sent back to the `TM` after processing. Note also that the passing of pointers makes the code unfit to run on a parallel machine.

To define specific user data, all there is to do is to set the members in the class `real_user_data`[15], and modify the constructor, the destructor and the method `print()` in file `Member/BB_user_data.cpp`.

An additional method is `BB_lp::set_user_data_for_children()`. Its parameters are the selected branching object `best` as well as its index in the list of candidate branching objects. This methods has to set up the user data for the sons that will be generated. Pointers on these objects are stored in the vector `best->user_data()` having as many entries as the number of sons generated by the branching object.

Finally, the entry `is_processed` is set to 1 in the method

- `BB_lp::initialize_new_search_tree_node()`.

14 Output

Running `./bac` from `BAC` gives the following output:

```
Compilation flags:

readInput(): core size: 10   indexed size: 6
############################################################
TM: Starting phase 0
############################################################
 TM: Default init_new_phase() executed.

LP: **** Processing NODE 0 on LEVEL 0 (from TM) ****
 LP: Default purge_slack_pool() executed.

LP: *** Starting iteration 1 ***
LP node written in file lpnode.lp
LP:   Matrix size: 10 vars x 10 cuts
LP:   Solution value: -5.0000 / 2 , 10
 LP: Default display_lp_solution() executed.
  LP : Displaying LP solution (RelaxedSolution) :
  LP : Displaying solution :
  Core  var (internal index:      1                   ) at 1.0000
  Core  var (internal index:      3                   ) at 1.0000
  Core  var (internal index:      5                   ) at 1.0000
  Core  var (internal index:      7                   ) at 1.0000
  Core  var (internal index:      9                   ) at 1.0000
LP:   Row effectiveness: rownum: 10 ineffective: 0
LP:   Number of leftover cuts: 0
```

[15] `DocBcp->real_user_data`.

```
generate_cuts_in_lp(): found 4 indexed cuts
generate_cuts_in_lp(): found 5 algorithmic cuts
LP:   Number of cuts generated in the LP process: 9
LP:   Non-violated (hence removed): 0
LP:   Number of cuts received from CG: 0
LP:   Total number of cuts in local pool: 9
LP:   Number of leftover vars: 0
LP:   Number of vars received from VG: 0
LP:   Total number of vars in local pool: 0
LP:   In iteration 1 BCP generated 9 cuts and 0 vars before calling branch()
 LP: Default select_branching_candidates() executed.
LP:   In iteration 1 BCP added 9 cuts and 0 vars.
 LP: Default purge_slack_pool() executed.

LP: *** Starting iteration 2 ***

...

 LP: Default select_branching_candidates() executed.

LP: Starting strong branching:

LP node written in file lpnode.lp
LP node written in file lpnode.lp
LP:   Presolving: (3,0.3333,-1.0000 /  ) [-2.5000,2,1] [-1.0000,36,6]
 LP: Default compare_presolved_branching_objects() executed.
LP node written in file lpnode.lp
LP node written in file lpnode.lp
LP:   Presolving: (0,0.3333,-1.0000 /  ) [-2.0000,2,2] [-2.0000,2,3]
 LP: Default compare_presolved_branching_objects() executed.
LP node written in file lpnode.lp
LP node written in file lpnode.lp
LP:   Presolving: (4,0.3333,-1.0000 /  ) [-2.6667,2,1] [-1.0000,36,3]
 LP: Default compare_presolved_branching_objects() executed.
LP node written in file lpnode.lp
LP node written in file lpnode.lp
LP:   Presolving: (1,0.6667,-1.0000 /  ) [-2.0000,2,3] [-2.0000,2,2]
 LP: Default compare_presolved_branching_objects() executed.
LP node written in file lpnode.lp
LP node written in file lpnode.lp
LP:   Presolving: (6,0.6667,-1.0000 /  ) [-2.6667,2,2] [-2.0000,2,3]
 LP: Default compare_presolved_branching_objects() executed.
LP:   Deleting 7 rows from the matrix.
 LP: Default set_actions_for_children() executed.

LP:   SB selected candidate 0 out of 5.

LP:   The selected object is: (3,0.3333,-1.0000 /  ) [-2.5000,2,1] [1.000000e+100,36,6]
LP node written in file lpnode.lp

LP: **** Processing NODE 1 on LEVEL 1 (dived) ****
...
```

Most of the output is self-explanatory. The output of the strong branching operations, however, is somewhat cryptic. The format of:

```
LP:   Presolving: (3,0.3333,-1.0000 /  ) [-2.5000,2,1] [-1.0000,36,6]
```

is the following:

- 3: index of the variable branched on;
- 0.3333: current value of the variable in the LP solution;
- -1.0 / : coefficient of the variable in the objective function;
- -2.5000 : objective value of LP for the first son;
- 2: LP termination code (see Section 11);
- 1: #iterations for solving the LP of the first son;
- -1.0000 : objective value of LP for the second son;
- 36: LP termination code (see Section 11);
- 6: #iterations for solving the LP of the second son;

Note that when the selected object is displayed, the line reads:

```
LP:   The selected object is: (3,0.3333,-1.0000 /  )
                                    [-2.5000,2,1] [1.000000e+100,36,6]
```

Note that the objective value (1.000000e+100) for the LP of the second son has changed. The reason is that these values are changed according to the LP termination code. The termination code for the second son (36, i.e. 0x20 and 0x04) indicates that the LP was solved to optimality and that the value is above the best known integer feasible solution. This implies that if branching on that variable occur, then the second son will be pruned by bound. Changing the value of the LP to 1.000000e+100 (= `BCP_DBL_MAX`) is done to simplify logical tests later. The method `fake_objective_values()`[16] is in charge of these modifications.

If both sons are feasible the line looks like:

```
LP:   Presolving: (0,0.3333,-1.0000 /  ) [-2.0000,2,2] [-2.0000,2,3]
```

where the termination code (2, i.e 0x02) indicates that the optimal solution of the LP was found.

[16] `DocBcp->BCP_lp_brobj`.

15 Parameters

BCP has a large number of parameters that can be modified by the user by using a parameter file. The file bb.par contains a certain number of them, hopefully the ones that a user might want to modify. To get the full list of parameters and their default values, look at DocBcp->BCP_tm_par and DocBcp->BCP_lp_par.

The file bb.par contains succinct explanations for some of the parameters and some default values. For 0/1 parameters, the meaning of only one of the two values is given, this value being the one set by default.

Some of the parameters conflicts with each other and nothing prevents the user from setting conflicting values. The result is unpredictable without looking at the code in detail. For example, setting the parameters

BCP_VerbosityShutUp 1 (to suppress all BCP printed output) and
BCP_TmVerb_BestFeasibleSolution 1 (to print the best solution found)

results in BCP printing the final solution. Another example is that, assuming that BCP uses the default branching strategy, setting the parameters

BCP_MaxPresolveIter -1 (strong branching should not be used)
BCP_StrongBranch_CloseToHalfNum 3 (default)
BCP_StrongBranch_CloseToOneNum 3 (default)

implies that more than one candidate variable is chosen (up to 6 can be selected) and since BCP can not use strong branching to decide on which variable to branch, it selects the first one and raises an error message. (To avoid the error message, the sum of the values of the last two parameters should be 1.)

A third example is the use of Breadth-First or Best-Bound enumeration strategies. Setting

BCP_TreeSearchStrategy 0 (Best-Bound is used)

is not enough, since BCP might decide to dive on certain nodes. Setting

BCP_UnconditionalDiveProbability -1 (no diving)

is still not enough. In addition,

BCP_QualityRatioToAllowDiving_HasUB -1 (no diving when an upper bound is available)
BCP_QualityRatioToAllowDiving_NoUB -1 (no diving when no upper bound is available)

must also be set.

It is possible for the user to define new parameters (for example to pass the name of an input file). Facilities already exist for doing that and the interested reader can look at the example `BranchAndCut` for an illustration.

16 Specific Modifications for Branch-and-Cut

Several minor modifications of the `BCP` code allow for significant performance improvement when doing pure branch-and-cut. A list of suggested modifications is available from [15].

Acknowledgments

I wish to thank Laszlo Ladányi for patiently answering my questions while developing this example.

References

[1] `http://www.coin-or.org`

[2] `https://www.coin-or.org/projects/Bcp.xml`

[3] `https://projects.coin-or.org/Bcp`

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

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

[6] `http://www.coin-or.org/faqs.html`

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

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

[9] Barahona F., Ladányi L., "Branch-and-Cut Based on the Volume Algorithm: Steiner Trees in Graphs and Max Cut", IBM Research report RC22221 (2001).

[10] Jünger M., Naddef D., eds., *Computational Combinatorial Optimization*, *Lecture Notes in Computer Science* 2241, Springer (2001).

[11] Ladányi L., Forrest J.J., Kalagnanam J.R., "Column Generation Approach to the Multiple Problem with Color Constraints", IBM Research report RC22013 (2001).

[12] Ladányi L., Lee J., Lougee-Heimer R., "Rapid prototyping of optimization algorithms using COIN-OR: A case study involving the cutting-stock problem.", *Annals of Operations Research* 139 (2005), 243-265.

[13] Ladányi L., Ralphs T.K., Trotter L.E., "Branch, Cut, and Price: Sequential and Parallel", in [10], 223-260.

[14] `http://wpweb2.tepper.cmu.edu/fmargot/COIN/coin.html`

[15] `http://wpweb2.tepper.cmu.edu/fmargot/COIN/bcp_mod.html`

[16] Padberg M.W., Rinaldi G., "A Branch-and-Cut Algorithm for the Resolution of Large Scale Symmetric Travelling Salesman Problems", *SIAM Review* 33 (1991), 60–100.

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

[18] Wolsey L.A., *Integer Programming*, Wiley (1998).