BAC : A BCP based Branch-and-Cut Example

François Margot[1]

May 2003, Updated January 2006

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 Introduction

This paper is an introduction to the Branch-and-Cut-and-Price (`Bcp`) software available in the COIN 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.

[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.

The reader is assumed to be familiar with the Branch-and-Cut process and its standard terminology. (An excellent introduction can be found in [4, 6, 7, 8].) Basic knowledge of C++ is also required.

The code of the example `BAC` is based on the example `BranchAndCut` written by L. Ladányi and available in the COIN repository. These two examples complete each other, `BAC` illustrating the customization of the branching and the generation of the initial LP directly in the code while `BranchAndCut` shows how to use the predefined cut generators and how to define parameters. The structure of the two examples is quite similar and a new user is probably better of studying first the `BAC` example and then move on to the `BranchAndCut` example. Other examples are available in the COIN repository: `Cgl1` and `Cgl2` (illustrate the use of cut generators), `MaxCut` (a very efficient Branch-and-Cut code for solving Maximum Cut problems) [3], `Mkc` (a Branch-and-Price code for solving Multiple Knapsack with Color problems) [5], `VolLp` and `Volume-LP` (volume algorithm for solving LPs), and `VolUfl` (approximate solution of Uncapacited Facility Location problems).

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 describes the integer linear problem solved in the example. Section 3 gives a general overview of `Bcp` and other packages. Section 4 covers the installation, compilation and compilation flags of the example. Section 5 covers the generation of the html documentation. Section 6 lists two data structures defined in `Bcp` that are used in the example: vectors and matrices. Section 7 describes the three types of `Bcp` variables and constraints. Section 8 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 9 describes how the user can define objects associated with the nodes of the enumeration tree that depend on the parent node. Finally, Section 10 discusses briefly the parameter file for `Bcp`.

The description in this paper corresponds to the code available on the COIN repository on January 1, 2006. Since COIN and `Bcp` are continuously in development, there is no guarantee that all details of the description will be accurate, even in a near future. The paper will hopefully be kept up to date and its most recent version will be available in the COIN repository.

Questions or bugs related to `Bcp` or `COIN` in general should be directed to the bugs reporting site of the COIN repository (see FAQs), and other mailing lists available there. Questions or bugs related specifically to the `BAC` example

should be directed to

`fmargot@andrew.cmu.edu` .

2 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 4 for the possible compilation flags).

3 General

`Bcp` is a collection of classes and functions 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 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 BAC tar ball from the COIN 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 from the COIN repository for proper "parallel" packing and unpacking procedures.

The source files are split into a number of subdirectories:

- BAC/include contains all of the include files (*.hpp).
- BAC/TM contains the code for the tree manager (BB_tm.cpp).
- 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 9), and the initialization of the process (BB.cpp, BB_init.cpp).

4 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. Some variables might need to be defined, 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 COIN for files related to COIN, Bcp, or Clp.)

The file Makefile.bc has a line starting with `USER_DEFINES =` . Five flags can be put on this line :

- `MPS` : Read the problem from the file bac.mps (MPS format). If neither this flag nor the flag `LP` is set, the problem is constructed from scratch in the code.
- `LP` : Read the problem from the file bac.lp (LP format). If neither this flag nor the flag `MPS` is set, the problem is constructed from scratch in the code.
- `HEUR_SOL` : Generate heuristic solutions (by simple rounding of the LP solution).
- `CUSTOM_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).
- `USER_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 9.

The object files (*.o) and executable (`bcps`) are in a subdirectory of `BAC` whose name depends on your system. For this document, the name of this subdirectory is assumed to be `SYST`. If the flag(s) used are modified in the file Makefile.bc, make sure that the three files `TM/BB_tm.cpp`, `LP/BB_lp.cpp`, and `Member/BB_init.cpp` are recompiled. This can be achieved by using `make rmo` 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 either by typing `./SYST/bcps` or `./SYST/bcps bb.par`. The second command makes the code read the parameter file bb.par. (Parameters will be discussed below in Section 10.) 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.

5 Documentation

The COIN libraries used by the example are divided into several modules corresponding to subdirectories of the directory `COIN`. Each module has its own Makefile and documentation. The documentation is html based and can be accessed through any browser. Assuming that the software `Doxygen` [2] is installed on your machine (it is usually available on Linux systems, but MacIntosh and Windows versions are available under the GNU GPL license), simply type

```
make doc
```

in the main directory of the module to build its documentation. If `Doxygen` is not available on your machine, you can still get access to some of the html documentation in the COIN repository [1] under the heading "Documentation".

The modules we are interested in are: `COIN/Cgl` (cut generators), `COIN/Clp` (LP solver), `COIN/Coin` (misc.), and `COIN/Osi` (interface with LP solver). Go in each of these directories and build the documentation. For the documentation of the module `Bcp` and `BAC` example itself, type `make doc` in the `BAC` directory. This creates a subdirectory `BAC/Doc` with the documentation. Note that the files specific to the `BAC` examples found in the directories `include`, `LP`, `Member`, and `TM` are commented through `Doxygen` but accessing the actual files directly might be more useful.

Open the file `COIN/Clp/Doc/html/index.html` in a browser and make a bookmark reference to that page. This page will be refereed to as `DocClp` in the future. The "Class List" link on the top of the page give access to the list of files in `COIN/Clp` with a brief description.

Open the corresponding main pages for the modules `Bcp`, `Osi`, `Coin`, and `Cgl`. Bookmark each of them. They will be refereed to as `DocBcp, DocOsi, DocCoin` and `DocCgl` in the future.

6 Data structures

A few data structures are available within `Bcp` and `COIN`. 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 functions 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->Class List->BCP_vec` for the full description of this class.

The class `CoinPackedMatrix` implements a two dimensional matrix stored either by rows or by columns. For a `CoinPackedMatrix` `M` stored by rows, the following functions 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->Class List->CoinPackedMatrix` for the full description of this class.

7 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 10) is used for the definition of row effectiveness.
- Indexed constraints are constraints in bijection with a set of integers, such that (user defined) functions 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 2, 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 ("≥", "≤", 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`[1] 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.

[1] `DocBcp->Class List->BB_cut`.

8 Main classes: BB_prob, BB_tm, and BB_lp

The main classes are BB_prob, BB_tm, and BB_lp. The class BB_prob[2] 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.

The class BB_tm[3] (tree manager) contains a single object of type BB_prob, named desc, holding the description of the problem (defined by the user). The remaining functions 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[4]. Some of the data members in BB_tm are:

- BB 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 $\pm$DBL_MAX 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 function initialize_core() and Bcp will manage

[2] DocBcp->Class List->BB_prob.
[3] include/BB_tm.hpp, TM/BB_tm.cpp or DocBcp->Class List->BB_tm.
[4] DocBcp->Class List->BCP_tm_user.

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 function members in `BB_tm` are:

- `readInput()` : reads input data.
- `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 function `unpack_module_data()` is in the class `BB_lp`[5]. In the example, this function 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 function defined in the class BB_cut[6].
- `unpack_cut_algo()` : decodes encoded algorithmic cuts. It uses one of the constructors defined in the class BB_cut [7].
- `initialize_core()` : Transmits core constraints and core variables to `Bcp`.
- `create_root` : set up the data at the root node. In this example, this function is really used only when the flag `USER_DATA` is set.
- `display_feasible_solution()` : self explanatory.

The class `BB_lp` contains the functions 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):

- Initialize the new node (u).

[5] `DocBcp->Class List->BB_lp`.
[6] `DocBcp->Class List->BB_cut`.
[7] `include/BB_cut.hpp`, `Member/BB_cut.cpp`.

- Solve the node LP.
- Test the feasibility of the node LP solution (u).
- Update the lower bound for the node LP.
- Fathom the node (if possible).
- Perform (logical, reduced cost) fixing on the variables (u).
- Update the row effectiveness records.
- Generate cuts (u).
- Generate a heuristic solution (u).
- Fathom the node (if possible).
- Decide to branch, fathom, or repeat the loop (u).
- Add to the node LP the cuts generated during the iteration, if the loop is repeated.
- 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.

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 function `initialize_new_search_tree_node()` described below is available for this.

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

- `BB *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 9 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 function `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.

Prominent function members 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 function 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`.
- `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.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 function 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 function `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 function `BCP_lp_user::test_feasibility()`. The return value of the function 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 function `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 function returns immediately when it is called while doing strong branching (i.e. when called with `in_strong == 1`).

The function is also called when the LP is infeasible. While this does not make much sense for a Branch-and-Cut, it does when column generation is possible. In the example, the function returns immediately if the last solved LP is infeasible.

- `logical_fixing()` : Empty function 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 functions 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. See the example `BranchAndCut` for more on this.

- `generate_heuristic_solution()` : self explanatory. The function is active in the example only if the compilation flag `HEUR_SOL` is used. (See Section 4.) 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 function `test_feasibility()` since the checking of indexed and algorithmic cuts is done in both functions. Introducing functions 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`[8] derived from `BCP_solution` is used to encode the solution

- `select_branching_candidates()` : The function is active in the example only if the compilation flag `CUSTOM_BRANCH` is used. (See Section 4.)

[8] `DocBcp->Class List->BCP_solution`.

In the example, the variables are considered in order and branching is performed on the first fractional variable.

- `cuts_to_rows()` : Required function 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 function. 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 function generating the coefficients from the compact representation is then necessary.

9 User Data

The class `BCP_user_data`[9] 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` [10] 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.

[9] `DocBcp->Class List->BCP_user_data`.
[10] `DocBcp->Class List->MY_user_data`.

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.

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 class `real_user_data`[11], modify the constructor, the destructor and the method `print()` of the class `real_user_data` in file `Member/BB_user_data.cpp`.

An additional function 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 functions 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 function

- `BB_lp::initialize_new_search_tree_node()`.

10 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->Class List->BCP_tm_par` and `DocBcp->Class List->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

[11] `DocBcp->Class List->real_user_data`.

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 second 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.

Acknowledgments

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

References

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

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

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

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

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

[6] Ladányi L., Ralphs T.K., Trotter L.E., "Branch, Cut, and Price: Sequential and Parallel", in [4], 223-260.

[7] 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.

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