BaPCod — a generic branch-and-price code

Ruslan Sadykov*[1,2] and François Vanderbeck[3]

[1]Bordeaux-Sud-Ouest Inria Research Centre, 200 avenue de la Vieille Tour, 33405 Talence, France
[2]Institut de Mathématiques de Bordeaux, Université de Bordeaux, 351 cours de la Libération, 33405 Talence, France
[3]Atoptima SAS, 16 place Sainte Eulalie, 33000 Bordeaux, France

December 15, 2022

Abstract

This document presents a user guide for BaPCod version 0.74[1], a C++ library implementing a generic branch-cut-and-price solver. We give guidelines for installing BaPCod, using its modelling language, BaPCod parameterization, retrieving BaPCod statistics, and understanding BaPCod output. We also present the VRPSolver extension of BaPCod which allows one to model and efficiently solve a large number of vehicle routing and related problems.

BaPCod was developed in Bordeaux University and Bordeaux Research Center of Inria, France.

*Corresponding author: ruslan.sadykov@inria.fr

[1]The most recent version of this user guide is available on the BaPCod website: https://bapcod.math.u-bordeaux.fr

Contents

1 Introduction

BaPCod is a prototype academic code that solves Mixed Integer Programs (MIP) by application of a Dantzig-Wolfe reformulation technique. The reformulated problem is solved using the branch-cut-and-price algorithm which includes the column generation procedure to solve the linear relaxation in each node of the branch-and-bound tree. The specificity of this prototype is to offer a "black-box" implementation of the method:

1. the input is the set of constraints and variables of the MIP in its natural/ compact formulation;
2. the user specifies which of these constraints and variables define the subsystems on which the decomposition is based (it is handy to test different decompositions);
3. the reformulation is automatically generated by the code, without any input from the user to define master columns, their reduced cost, pricing problem, or Lagrangian bound;
4. a default column generation procedure is implemented that relies on an underlying LP/MIP solver to handle master and subproblem but the user can define a specific solver for the pricing problem;
5. a branching scheme that preserves the pricing problem structure is offered by default, it runs based on priorities and directives specified by the user on the original variables;
6. the user can specify custom cut generation callbacks and custom branching callbacks;
7. preprocessing, restricted master and diving primal heuristics, some stabilization techniques, and strong branching are available for use;
8. VRPSolver extension can be used, which includes a resource constrained shortest path problem (RCSP) solver, and some families of robust and non-robust cut separation and branching functors; these components can be used to devise state-of-the-art branch-cut-and-price algorithms for vehicle routing and related problems. VRPSolver extension is distributed in the compiled form.

Readers of this user guide are supposed to be familiar with the theory of Dantzig-Wolfe reformulations, the column generation procedure, and the branch-cut-and-price method (see for example [3] for an introduction).

The source code of BaPCod can be obtained on its web-page `https://bapcod.math.u-bordeaux.fr` after accepting the Inria licence for academic use. The following contacts can be used to communicate with the authors of the code:

ruslan.sadykov@inria.fr — issues with the code, critical bugs, user guide, general issues;

laurent.facq@math.u-bordeaux.fr — issues with installation and running of demos.

1.1 Code structure

We suppose that the code is cloned or extracted to the `BapcodFramework` folder. The main folder structure is the following

```
BapcodFramework
-- Applications
-- Bapcod
-- CMake
-- CMakeLists.txt
-- Documentation
-- Demos
-- LICENCE.pdf
-- README.md
-- Scripts
-- Tools
```

`Applications` folder should contain user applications, no applications are provided by default. We explain how to create an application in Section 1.3. `Bapcod` folder contains the C++ source code of BaPCod. `Cmake` folder and file `CMakeLists.txt` contain CMake scripts necessary for compiling and linking BaPCod, its demos and user applications. `Documentation` folder contains the source of the present user guide. `Demo` folder contains demos which come together with BaPCod source code. `LICENCE.pdf` and `README.md` files contain the licence and the installation instructions. The licence stipulates that you can use BaPCod for free for academic purposes. `Scripts` folder contains some scripts necessary for BaPCod installation and other tasks. `Tools` folder contains the archived source code of two third-party open-source libraries, namely `Boost 1.76` and `LEMON 1.3.1`. BaPCod is dependent on them, as well as on the Cplex library, which can be obtained for free for academic use. `Tools` folder also contains the RCSP library, which is necessary for using the VRPSolver extension. RCSP library is pre-compiled for three major operating systems (Mac OS, Linux, and Windows).

1.2 Installation

The installation instructions are given in the `README.md` file.

1.3 Creating a new application

A demo or an application has the following structure by default

```
<ApplicationOrDemoName>
-- CMakeLists.txt
-- config
-- data
-- include
-- src
-- tests
```

File `CMakeLists.txt` contains CMake instructions necessary for compiling and linking the application or demo. `config` folder contains configuration files.

There are usually two configuration files: one contains BaPCod parameters, and another contains application-specific parameters. `data` folder contains instance files. `include` folder contains header files. `src` folder contains source files. `tests` folder contains files and scripts to run non-regression tests. It is highly advised to create several non-regression tests for every user application. Non-regression tests usually verify that the solution value obtained after the run coincides with the optimal value or the dual bound value.

A standard way to create a new application is by modification of a similar available demo. If there is no demo which is similar to the intended application, please request the authors of the code to produce a similar demo. One advantage of this method is to provide for the user a working tree that already contains the files needed by `cmake` for the compilation and pre-filled configuration and test files. Another advantage is to provide for the user the code structure that is easier to modify when making up the new application rather than starting from scratch.

First, copy the folder with the corresponding demo to the `BapcodFramework/Applications` folder. Then modify the folder name to the application name, and add the new folder name to file `BapcodFramework/Applications/CMakeLists.txt` inside `add_subdirectories()` so that the makefile is produced next time your run `cmake`.

Afterwards, the code of the new application should be modified according to the problem one wants to solve. It is advised to change the names of all classes in the application code or to change the name of the namespace used in the code, in order to avoid a clash between different demos and applications. Usually, to adapt the code to the new application, one should change the class with application-specific parameters, the data class, functions to read the data, the model and the callbacks (pricing, cut generation, branching), if applicable.

After the code modification is complete, the makefile of the application should be generated. For that, run the following commands from the `BapcodFramework` folder (on Mac OS and Linux)

```
cd build
cmake ..
```

On Windows

```
cmake -G "Visual Studio 16 2019" -A x64 -B "build"
```

These commands should also be run each time you add or delete header or source files for an application.

1.4 Running a demo or application

To run the application or demo, execute the following commands on Linux or MacOS from the `BapcodFramework` folder

```
cd build/Demos/<DemoName> # for a demo
cd build/Applications/<AppName> # for a application
make -j
bin/<DemoOrAppName> -b <BaPCod_config> -a <app_config> -i <instance>
```

On Windows, run the following commands from the `BapcodFramework` folder

```
cmake --build build --config Release --target <DemoOrAppName>
cd build/Demos/<DemoName> # for a demo
cd build/Applications/<AppName> # for an application
bin/Release/<DemoOrAppName>.exe -b <BaPCod_config> -a <app_config> -i <instance>
```

Here `<AppName>` is the name of the application, `<BaPCod_config>` is the (relative) path to the configuration file with BaPCod parameters, `<app_config>` is the (relative) path to the configuration file with application-specific parameters (if such parameters exists), and `<instance>` is the (relative) path to the date instance file. You can specify additional BaPCod and application-specific parameters in the command line using the format `--<paramName> <value>` . If different values are given for the same parameter in the configuration file and in the command line, the value given in the command line has more priority. See Section 3 for an overview of BaPCod parameters. In addition, in the command line one can specify `-t <tree_file>` parameter to change the default (relative) path (which is `BaPTree.dot`) to the file where the branch-and-bound tree information will be stored in .DOT format for later visualisation.

It is highly advised to use a version control system (for example `Git`) for user applications. It is standard to create a different repository for every user application.

1.5 Citing BaPCod

If you use BaPCod, please cite the present document:
Ruslan Sadykov and Francois Vanderbeck. BaPCod— a generic branch-and-price code. *Technical report HAL-03340548*, Inria Bordeaux Sud-Ouest, 2021.

In addition, if you use the following components of BaPCod, we encourage you to cite the corresponding papers listed below.

- If you use stabilization (automatic dual price smoothing stabilization is activated by default), please cite paper [11]:
  Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and Francois Vanderbeck. Automation and combination of linear-programming based stabilization techniques in column generation. *INFORMS Journal on Computing*, 30(2):339–360, 2018.

- If you use primal heuristics, please cite paper [17]:
  Ruslan Sadykov, Francois Vanderbeck, Artur Pessoa, Issam Tahiri, and Eduardo Uchoa. Primal heuristics for branch-and-price: the assets of diving methods. *INFORMS Journal on Computing*, 31(2):251–267, 2019.

- If you use the VPRSolver extension, please cite paper [12]:
  Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and Francois Vanderbeck. A generic exact solver for vehicle routing and related problems. *Mathematical Programming*, 183:483–523, 2020.

2 Modelling language

This section overviews the modelling language, i.e. the C++ interface for BaPCod. To use this interface, one needs to include the corresponding header file:

```
#include "bcModelingLanguageC.hpp"
```

BaPCod models are similar to Mixed Integer Programs (MIPs): the user needs to define (continuous or integer) variables, linear constraints and the objective function. Models are defined using of *original* variables, i.e. variables of the original compact formulation. There are however following differences.

The user needs to specify the decomposition: i.e. which constraints are dualized (remain in the master formulation) and which constraints are imposed when solving the subproblem formulations (or pricing problems). Therefore, at least two formulations should be defined: master one and at least one subproblem formulation. Each subproblem formulations has integer bounds specifying how many solutions of this subproblem (i.e. how many columns generated by this pricing problem) can participate in the global solution of the model. These bounds are useful to define identical subproblems.

One can also use BaPCod to solve a MIP without applying decomposition; this may be useful when benchmarking a decomposition against the original formulation. For this, one should define a single formulation and solve it (instead of solving the model).

Each variable belongs to a formulation. Subproblem constraints may involve only variables belonging to the corresponding subproblem formulation. On the contrary, master constraints may involve variables belonging both to the master formulation (we call them *pure master* variables) and to the subproblem formulations. Coefficients of subproblem variables in the master constraints determine the coefficients of columns in these constraints.

The user has a possibility to define additional functors (C++ classes) to improve the performance of BaPCod. Functors are enhanced versions of callbacks. A functor may have several functions which are called in different stages of the solution process. The main functor is the pricing one which may be defined for a subproblem formulation for managing the corresponding pricing problem. Another important functor is to separate robust cutting planes. Such cutting planes are defined similarly to standard constraints using original variables. The user can also define a functor for generation of robust branching constraints. Defining non-robust cutting plains and branching constraints is also possible but reserved for an advanced use. The VRPSolver extension predefines a certain number of functors which are used for solving vehicle routing and related problems. This extension is reviewed in Section 6.

2.1 Environment handler

Every time BaPCod is used, one should first declare an environment handler of type `BcInitialisation` using constructor

```
BcInitialisation(int argc, char *argv[],
                 std::string config_filename = "config/bcParams.cfg");
```

Here `argc` and `argv` are the standard parameters of the `main()` function for command line arguments, and `config_filename` is the default (relative) path to the configuration file with BaPCod parameters (for the case `-b` parameter does not present in the command line).

Alternatively, one can declare an environment handler using constructor without parameters.

```
BcInitialisation();
```

In this case case, all parameters are initialized with default values.

The environment handler has several useful methods.

```
UserControlParameters & param();
```

This method allows one to obtain the object with basic BaPCod parameters. One then can read and modify these parameters. The available parameters are reviewed in Section 3.

```
void bcReset();
```

This method should necessarily be called between two calls to `BcModel::solve()` function (of a model associated with this environment handler) in order to reset internal counters.

Finally, for obtaining the statistics of the execution on can use the methods

```
double getStatisticValue(const std::string & statName);
long   getStatisticCounter(const std::string & statName);
double getStatisticTime(const std::string & statName);
```

to get values of individual statistics (records, counters, and timers). See Section 4 for an overview of BaPCod statistics.

2.2 Model handler

The model handler is used to define and solve BaPCod models. Every model should be associated with a BaPCod environment handler, specified in the model constructor:

```
BcModel(const BcInitialisation & bapcodInit,
        const std::string & modelName = "Model");
```

The main method of the handler solves the model:

```
BcSolution BcModel::solve();
```

It returns the best found solution. The solution returned is disaggregated by default, see Section 2.6 for details.

Solution management callback can be attached to a model:

```
void attach(BcSolutionFoundCallback * solutionFoundCallbackPtr);
```

This callback is called every time a solution is found which is considered to be feasible by the model. This callback can be used to check the feasibility of the solution (and thus the correctness of the model) and possibly show and/or store it for future use.

2.3 Formulation handlers

Formulation handler `BcFormulation` serves to manage formulations. The master formulation can be created or obtained using the constructor

```
BcMaster(BcModel & model, const std::string & name = "master");
```

Subproblem formulations are handled in an array of type `BcColGenSpArray`. This array is created or obtained using the constructor

```
BcColGenSpArray(BcModel & model, const std::string & name = "colGenSp");
```

`BcColGenSpArray` has two operators `operator()` and `operator[]` to access individual formulations in the array. These operators take integer indices as parameters. The first operator creates the formulation with given indices if it does not exists. If no index is given, 0 is used by default. The second operator does not create formulation; if the formulation with given indices does not exist, an error occurs. For example, the following code creates a model with the master formulation and two subproblem formulations.

```
BcInitialisation bcInit(argc, argv);
BcModel model(bcInit);
BcMaster master(model);
BcColGenSpArray colGenSp(model);
colGenSp(0);
colGenSp(1);
```

The master formulation is created automatically if `BcColGenSpArray` is constructed.

The master formulation of a subproblem one can be obtained with method

```
const BcFormulation BcFormulation::master() const;
```

It returns `BcMaster` handler which inherits from `BcFormulation`. If the formulation is not a subproblem one, the returned master formulation is not defined. The same method, defined for `BcModel`, allows one to retrieve the master formulation of a model.

The list of subproblem formulations of a master one can be obtained with method

```
const std::list< BcFormulation > & colGenSubProblemList() const;
```

If the formulation is not the master one, the returned list is empty.

For a subproblem formulation, one may define multiplicity bounds, i.e. the minimum and maximum number of solutions (i.e. columns) from this formulation which can participate in the global model solution. These bounds will define convexity constraints in the restricted master problem. The bounds are set with operators `operator>=`, `operator<=`, and `operator==`. For example, the code

```
colGenSp[0] >= 1;
colGenSp[1] <= 2;
```

sets lower bound one for the subproblem formulation with index 0 (by default, the lower bound is equal to 0), and upper bound two for the subproblem formulation with index 1 (by default, the upper bound is equal to ∞).

For a subproblem formulation, one may also define the fixed cost, i.e. the value which will be added to the coefficient in the objective function of every column coming from this subproblem. The method is

```
void BcFormulation::setFixedCost(const double & value);
```

By default, the fixed cost is equal to zero. For example, in the bin packing model, the fixed cost of the knapsack subproblem may be set to 1.0. The same `setFixedCost` method is defined for `BcColGenSpArray`. It sets the same fixed cost for all subproblem formulations in the array.

The master formulation can be initialized with a set of columns using two methods of `BcFormulation`:

```
void initializeWithSolution(BcSolution & sol);
void initializeWithColumns(BcSolution & sol);
```

These two methods should be called after completely defining the master and subproblem formulations. The solution passed should be disaggregated (see Section 2.6). The first method updates the global solution of the model and adds its subproblem solutions as columns to the restricted master problem (if parameterized accordingly, see Section 3.2). The second method adds columns corresponding to subproblem solutions in the provided solution chain `sol` to the restricted master problem (independently of the parameterization and without updating the global solution of the model).

If one wants to solve a MIP model without applying decomposition, a single formulation should be declared using constructor

```
BcFormulation(BcModel & model, const std::string & name = "directForm");
```

Then, instead of solving the model, one should solve this formulation using method

```
BcSolution BcFormulation::solve();
```

2.4 Variables and constraints

Variables and constraints are defined using arrays:

```
BcVarArray(const BcFormulation & formulation, const std::string & name);
BcConstrArray(const BcFormulation & formulation, const std::string & name = "");
```

Every array of variables or constraints should belong to a formulation, either to the master or the a subproblem one. Note that these constructors are used in two ways. First, one can create a new array of variables or constraints before solving the model. After starting the solution process, one can retrieve the existing array by passing its name as the parameter. This functionality is used to retrieve existing variables and constraints in user-defined functors.

The same operators `operator()` and `operator[]` as for formulations are used to access individual elements in the array. Again, the first operator creates the element with given indices if it does not exist, the second operation raises an error in this case. Note that the syntax of these operators is different for multi-dimensional indices:

```
BcVarArray xVar(colGenSp[0], "X");
xVar(0,0);
xVar[0][0] <= 1;
```

This code creates variable $x_{0,0}$ belonging to the subproblem formulation with index 0, and then sets the upper bound of this variable to 1. The type of individual variables is `BcVar`, and the type of individual constraints is `BcConstr`.

For an array of variables or constraints, one can define index characters:

```
xVar.defineIndexNames(MultiIndexNames('i','j'));
```

This characters are used in names of individual variables. For example, the name of variable `xVar[0][0]` will be `"Xi0j0"`.

For an array of variables, their type is set using method `type`:

```
xVar.type('B');
```

`'B'` stands for binary, `'I'` for integer, and `'C'` for continuous. By default, variables are continuous. The same method is also defined for individual variables.

For an array of variables, or individual variables, one can define their bounds using operators `operator<=` and `operator>=`. For subproblem variables these bounds are *local* and valid only when solving the subproblem. In the global model solution, the total value of a subproblem variable can violate its local bounds. To set global bounds on subproblem variables, one should define the corresponding master constraints.

For an array of constraints, or individual constraints, one can define their sense and right-hand size using operators `operator<=`, `operator>=`, and `operator==`. For example the code

```
BcConstrArray setPackConstr(master(), "SPC");
setPackConstr(0) <= 1;
```

creates a less-or-equal constraint with the right-hand-size value equal to 1.

For an array of constraints, or individual constraints, one can indicate whether they will be used in preprocessing, using method

```
void toBeUsedInPreprocessing(bool flag);
```

To set coefficients of variables in constraints, one may use operators `operator+=`, `operator+`, `operator-=`, and `operator-`. For example, the code

```
BcConstrArray setCovConstr(master(), "SCC");
setCovConstr.defineIndexNames(MultiIndexNames('k'));
setCovConstr(0) >= 2;
BcVarArray yVar(master(), "Y");
setCovConstr[0] += xVar[0][0] + 2 * y[0];
setCovConstr[0] -= 0.5 * xVar[0][0];
```

defines constraint $\frac{1}{2}x_{0,0} + 2y_0 \geq 2$ with name `"SCCk0"`.

2.5 Objective function

The objective function of a model or a formulation (when solving a MIP) can be obtained using the constructors

```
BcObjective(BcModel & model);
BcObjective(BcFormulation & bcForm);
```

The sense and integrality of the objective function is determined using the method

```
void setMinMaxStatus(const BcObjStatus::MinMaxIntFloat & newObjectiveSense);
```

Possible arguments are `BcObjStatus::minInt` and `BcObjStatus::minFloat`. One may not use maximization objective with BaPCod for the moment, as there are known bugs when it is used. One can transform maximization objective to minimization by multiplying it by -1. By default, the objective function is "float", which mean it can take any value. If it is known that the objective function value of any feasible solution is integer, one should give this information to the solver by setting the objective function type to "integer". In this case, the lower bound will be rounded up before checking whether a branch-and-bound node should be pruned, making the search tree smaller.

Coefficients of the variables in the objective function are set in the same way as for constraints. For example, the code

```
BcObjective objective(model);
objective.setMinMaxStatus(BcObjStatus::minInt);
objective += yVar(0) + 2 * yVar(1);
```

sets the objective function to $\min y_0 + 2y_1$ and states that it can take only integer values. Both master and subproblem variables may participate in the objective function.

The initial upper bound for the objective function value (i.e. the cut-off value) can be set using operator `operator<=`. For example, the code

```
objective <= 100;
```

sets the cut-off value to 100, meaning that all branch-and-bound nodes will be pruned as soon as the lower bound becomes strictly greater than 99 (as the objective function is known to be integer as indicated above). Setting initial upper bound to a value which is as close as possible to the optimum value is important to decrease the size of the search tree.

Another important method sets the initial coefficient of artificial variables in the objective function:

```
void setArtCostValue(const double & cost);
```

In some cases, BaPCod may wrongly declare the model infeasible if the value of this coefficient is below the optimum value. Therefore, a good practice is to set this coefficient to a known upper bound on the optimum solution value. Decreasing the value of this coefficient may help to avoid numerical issues when they are present.

2.6 Solution handler

In BaPCod modelling interface, solutions are stored using handlers of type `BcSolution`. This handler may contain either a single solution or several solutions in a chain. The constructor of this handler is

```
BcSolution(const BcFormulation & formulation);
```

Each solution belongs to a formulation. A solution of a subproblem formulation defines the corresponding column in the restricted master problem. To set a value of a variable in the solution, operators `operator=` and `operator+=` is used. For example, the code

```
BcSolution bcSol(colGenSp[0]);
x[0][0] = 2;
bcSol += x[0][0];
```

defines a solution $x_{0,0} = 2$ (all other variables take value 0) for the subproblem formulation with index 0.

To add a solution to the solution chain, one can use method

```
BcSolution & appendSol(BcSolution & sol);
```

This method is useful, when one needs to pass several solutions. For example, the user-defined pricing problem solver may return several solutions so that multiple columns are added to the restricted master LP in a single column generation iteration.

To read all solutions in a solution chain, the following methods are useful:

```
BcSolution next() const;
bool defined() const;
```

One can obtain next solution in a loop until the current solution is not defined.

To retrieve the variables with non-zero values in a solution, one can use methods

```
void extractVar(std::set< BcVar > & varSet);
void extractVar(const std::string & genericName, std::set< BcVar > & varSet);
```

The first method retrieves all non-zero variables in the solution. The second one retrieves only variables in the array with the given name. Once the non-zero variables are extracted, values of particular variables can be obtained using method

```
double BcVar::solVal() const;
```

Solutions of the master formulation can be standard (or *aggregated*) and *disaggregated*. A standard solution has values of all variables, both master and subproblem ones, in the aggregated form. This means that the value of a subproblem variable in this solution is the weighted sum of values of this variable in solutions corresponding to columns, where weights are values of the columns in the master solution. A disaggregated solution is a chain of solutions, where the first solution contains values of pure master variables, and all other solutions in the chain correspond to columns in the master solution. For every such solution in the chain, one can retrieve the subproblem solution it belongs to and its multiplicity, i.e. the value of the corresponding column in the master solution:

```
BcFormulation formulation() const;
int getMultiplicity() const;
```

By default, the solution returned by the solver is disaggregated.

2.7 Branching

One can use three types of branching in BaPCod: branching on variables, branching on constraints, and custom "algorithmic" branching (with user-defined functors).

To branch on variables, one needs to define their branching priorities. For that, the following methods exists for an array of variables

```
const BcVarArray & priorityForMasterBranching(double priorityValue);
const BcVarArray & priorityForSubproblemBranching(double priorityValue);
const BcVarArray & priorityForRyanFosterBranching(double priorityValue);
```

If the branching priority value is non-positive, the corresponding branching strategy is not applied. Default priority values for master branching, subproblem branching, and Ryan&Foster branching are 1.0, 0.1, and -1.0. The corresponding variable branching strategies are: 1) branching in master on the total value of the variable; 2) Vanderbeck branching on bounds of subproblem variables, proposed in [18]; 3) Ryan& Foster branching, proposed in [15]. It suffices to use only the first branching strategy if the upper bound of all subproblem formulations is one, i.e. in the absence of identical subproblems. Otherwise, if all subproblem variables are binary, Ryan& Foster branching may be used. In the most general case (identical subproblems with general integer subproblem

variables), the second branching strategy should be used to ensure the exact solution. If the pricing functor is used for a subproblem, it should explicitly support Ryan& Foster branching in order to use it. The pricing functor needs to support arbitrary bounds on subproblem variables in order to use Vanderbeck branching. Finally, we advise to use Vanderbeck branching only if it is really needed, as we will not likely be able to correct possible bugs in the implementation. For variables which belong to the master formulation, only the first branching strategy is possible.

To branch on constraints, one should define array(s) of branching constraints using constructor

```
BcBranchingConstrArray(const BcFormulation & formulation,
                       const std::string & name,
                       const SelectionStrategy & priorityRule
                             = SelectionStrategy::MostFractional,
                       const double & priorityLevel = 1.0);
```

Here the `priorityLevel` is the branching priority value of all constraints in this array. After creating the array, branching constraints can be defined in the same way as standard constraints, see Section 2.4. The only difference is that the sense and the right-hand-size of branching constraints are ignored. Branching constraints are not added to the formulation in the beginning of the solution process. During branching, the left-hand-side value v of each branching constraint is computed using the current solution of the restricted master LP. In the case v is fractional, two constraints with the same left-hand-size are generated for two child nodes of the branch-and-bound tree: one is less-or-equal to $\lfloor v \rfloor$, and the second is greater-or-equal to $\lceil v \rceil$.

To add a custom "algorithmic" branching strategy one should define a functor which inherits from class `BcDisjunctiveBranchingConstrSeparationFunctor` and associate it to an array of branching constraints (such an array should not contain any pre-defined branching constraints) using method

```
const BcBranchingConstrArray &
      attach(BcDisjunctiveBranchingConstrSeparationFunctor
             * separationRoutinePtr);
```

The functor should implement the following operator

```
bool operator()(BcFormulation formPtr,
                BcSolution & primalSol,
                const int & candListMaxSize,
                std::list<std::pair<BcConstr, std::string>>
                & retBrConstrList);
```

This function receives the master formulation handler, the solution handler which contains the current solution of the restricted master LP, and the maximum number of branching constraints which will be processed (others will be ignored). All branching constraints generated should be added to list `retBrConstrList` of pairs. The first member of the pair is the branching constraint generated, and the second member is its description string. The description string is used to recognise the same branching constraint when collecting the branching history.

The branching history is used to select good branching candidates in the strong branching. In the function, the branching constraints should be defined in the same way as above (their sense and right-hand-side are ignored). The function should return `true` if and only if at least one branching constraint has been added to `retBrConstrList`.

All defined branching strategies are applied in decreasing order of their branching priority values. If no branching candidate is found for branching strategies with a certain priority value, strategies with the next lower priority value will be tried. If strong branching is used (see Section 3.6 for parameterization), branching strategies with lower priority are tried only if the number of generated branching candidates is lower than the required number of candidates. One particularity is that, if at least one branching candidate is found, then branching strategies with lower priority are tried only if their priority value is not smaller than the priority value of the found candidates rounded down.

2.8 Separation of cutting planes

A family of cutting planes can be defined using constructor

```
BcCutConstrArray(const BcFormulation & formulation,
                 const std::string & name,
                 const char & type = 'F',
                 const double & rootPriorityLevel = 1.0,
                 const double & nonRootPriorityLevel = 1.0);
```

A family of cutting plains is similar to an array of constraints. The difference is that the cuts are added during the solution process by the cut separation functor. Two important parameters of the constructor are `type` and `priorityLevel`. "Facultative" cuts (type `'F'`) are cuts which are separated only for fractional solutions. "Core" cuts (type `'C'`) are separated both for fractional and integer solutions, similarly to lazy constraints. Families of cuts are separated in the decreasing order of their priority level. A family of cuts with the next lower priority level is separated only if no cuts from families of higher priority level could not be separated or if the tailing-off condition for cuts with the previous higher priority level was reached. The parameterisation of the tailing-off condition in given in Section 3.3. Priority level of the cut families may be different in the root node. The following method can be used to set the root priority level:

```
void setRootPriorityLevel(const double & rootPriorityLevel);
```

To add a custom cut separation procedure for a family of cuts, one should define a functor which inherits from class `BcCutSeparationFunctor` and associate with the cut family using the method

```
const BcCutConstrArray & attach(BcCutSeparationFunctor
                                * separationRoutinePtr);
```

The functor should implement the following operator

```
int operator()(BcFormulation formPtr,
               BcSolution & primalSol,
               double & maxViolation,
               std::list< BcConstr > & cutList);
```

This function receives the master formulation handler, the solution handler which contains the current solution of the restricted master LP. Parameter `maxViolation` should be ignored. All generated cutting planes should be added to list `cutList`. The cutting planes are defined in the same way as normal constraints. This functor should return the number of cutting planes added to `cutList`.

Separation of non-robust cuts and taking into account non-robust cuts when solving the pricing problem is reserved for advanced use. Please contact the authors if you want to use this functionality.

2.9 Pricing functor

To solve the pricing problem by a user-defined algorithm, one should define a pricing functor which inherits from class `BcSolverOracleFunctor` and attach it to the corresponding subproblem formulation using method

```
const BcFormulation & attach(BcSolverOracleFunctor * oraclePtr);
```

For a pricing functor, one may implement several functions which are called in different moments of the solution process. The main function `operator()` serves to solve the pricing problem in each iteration of the column generation procedure:

```
bool operator()(BcFormulation spForm,
                int colGenPhase,
                double & objVal,
                double & dualBound,
                BcSolution & primalSol);
```

This function receives the subproblem formulation handler. All the information needed for solving the pricing problem can be retrieved using this handler. The only additional information passed by function arguments is `colGenPhase`, which is the current column generation phase. Phase zero is the exact phase, and phases one and above are heuristic phases. More information about the column generation phases is given in Section 3.2. The function should return the best solution value found in `objVal`, the lower bound on the solution value in `dualBound`, and the best found solution in `primalSol`. If `colGenPhase` is equal to 0 and the pricing problem is solved to optimality, then values of `objVal` and `dualBound` should all be equal. If `colGenPhase` is equal to 0 then `dualBound` should be equal to a valid lower bound on the optimal solution of the pricing problem, otherwise the optimality of the solution found by BaPCod is not guaranteed. If multiple solutions are found, they can be added to `primalSol` by using method `appendSol()`, see Section 2.6. The function should return `true` if and only if at least one solution to the pricing problem has been found.

To obtain current information about variables of the subproblem formulation, one needs to retrive these variables (`BcVar` handlers) using subproblem formulation handler `spPtr`, the `BcVarArray` constructor and operator `operator[]` (see Section 2.4). The following methods of `BcVar` are used:

```
double BcVar::curCost() const;
double BcVar::curLb() const;
double BcVar::curUb() const;
```

The first method retrieves the current reduced cost of the variable, the other two retrieve the current lower and upper bounds on the values of this variable in any feasible solution of the pricing problem. These bounds may be changed by the preprocessing or by branching constraints (if Vanderbeck branching is used, see Section 2.7). If the algorithm for solving the pricing problem does not support modified bounds on variable values, the preprocessing should be turned off (or at least the preprocessing of subproblem formulations, see parameterisation in Section 3.1), and Vanderbeck branching should not be used.

The pricing problem solved by the user-defined functor should not take into account the dual values of the master convexity constraints. Therefore, a solution to the pricing problem with a negative value does not necessarily corresponds to a column with a negative reduced cost. The solution value which corresponds to the zero reduced cost in the current column generation iteration can be obtained by the method

```
double BcFormulation::zeroReducedCostThreshold() const;
```

A useful function which may be implemented for the pricing functor is

```
bool prepareSolver();
```

This function is called after building the model and may be used to initialize the pricing functor. The function should return `true` if and only if the pricing functor initialization succeeds.

Two more useful functions of the functor are

```
BcSolverOracleInfo * recordSolverOracleInfo(const BcFormulation spPtr);
bool setupNode(BcFormulation spPtr, const BcSolverOracleInfo * infoPtr);
```

The first function is called at the end of a node in the branch-and-bound tree. It can be used to save the current state of the pricing functor to a structure or class which inherits from `BcSolverOracleInfo`. The second function is called before treating any node in branch-and-bound tree except the root node. This function receives as an argument the pointer to the structure created by the first function at the end of the parent node. So, these two functions are typically used to ensure that the state of the pricing functor is restored to the state of the parent node when the solution process goes from one node in the branch-and-bound tree to another. The second function is also used to retrieve the current set of active Ryan&Foster constraints For this, the following method of `BcFormulation` is used:

```
void getRyanAndFosterBranchingConstrsList(
     std::list<BcRyanAndFosterBranchConstr>
                              & ryanFostBranchConstrList) const;
```

Function `setupNode()` should return `true` if and only if the pricing problem becomes infeasible (for example because of non-compatibility of Ryan&Foster branching constraints).

Other functions of the pricing functor are reserved for advanced use. Please contact the authors if you want to use such functionality as the subproblem variable fixing based on reduced costs, enumeration of proper columns (for example, enumeration of elementary routes), a dynamic adjustment of the subproblem relaxation, or influencing cut separation from the pricing problem.

3 BaPCod configuration

This section lists the most important parameters of BaPCod. These parameters should be put to the BaPCod configuration file, see Section 1.4. If a parameter is missing in the configuration file, BaPCod will its default value, shown below.

3.1 Main parameters

```
GlobalTimeLimitInTick = 2147483645 # Time limit in ticks
```

If you want to set the time limit in seconds, use the parameter

```
GlobalTimeLimit = 0 # Time limit in seconds to solve the model
```

Parameter `GlobalTimeLimitInTick` is used only if `GlobalTimeLimit = 0`.

```
MipSolverMaxBBNodes = 2000000 # max. nodes number for the MIP solver
MipSolverMaxTime = 360000 # time limit in seconds for the MIP solver
MipSolverMultiThread = 0 # number of threads for the MIP solver
```

These options are valid for the underlying MIP solver, which is used to solve the pricing problems (if BaPCod is parameterized for that), the restricted master problem as a MIP in the corresponding heuristic, and the enumerated master (if the pricing functor supports subproblem solution enumeration). If the value of parameter `MipSolverMultiThread` is equal to `0` then the number of threads is determined automatically by the underlying MIP solver. BaPCod itself does not use parallelisation. Therefore, setting parameter `MipSolverMultiThread` to `1` makes the whole solution process to use a single thread.

```
OptimalityGapTolerance = 1e-6
```

If the relative gap between lower and upper bound is below this value, the column generation procedure terminates. Also, the node is pruned in the branch-and-bound tree if the relative difference between the global upper bound and the lower bound of the node is below this tolerance.

```
ApplyPreprocessing = true # use pre-processing or not
PreprocessVariablesLocalBounds = true # adjust bounds of subprob. vars or not
```

By default, BaPCod pre-processes both master and subproblem formulations. This procedure adjusts bounds of variables and deactivates redundant constraints. The second parameter determines whether bounds of subproblem variables are adjusted or not. This parameter should be set to `false` if a user-defined pricing problem functor is used and it does not support changing bounds on values of subproblem variables. The performance of the diving heuristic may be significantly reduced if the preprocessing in the subproblems is switched off.

```
MaxNbOfBBtreeNodeTreated = 100000
treeSearchStrategy = 1
OpenNodesLimit = 1000 # max. number of nodes in breadth-first strategy
```

The first parameter here limits the total number of explored nodes in the BaPCod branch-and-bound tree. There are primary and secondary branch-and-bound trees. The second parameter can take two values: `0` ("breadth-first" exploration strategy for the primary tree, i.e. the open node with the smallest lower bound is considered next), and `1` ("depth-first' exploration strategy for the primary tree, the open node with the largest depth is considered first). Parameter `OpenNodesLimit` sets the maximum number of open nodes in the primary tree. When this limit is reached, newly created nodes are pushed to the secondary branch-and-bound tree, which is always explored in the "depth-first" manner. We return to the primary tree when the secondary one becomes empty.

```
DEFAULTPRINTLEVEL = -1 # verbosity of the BaPCod output
```

Possible values here are `-2` (no output except errors and important warnings), `-1` (reduced output, one line per 10 column generation iterations), `0` (standard output, one line per one column generation iteration). Positive values are not recommended as the output quickly becomes overwhelming. More details about output are given in Section 5.

```
solverName = CPLEX_SOLVER # underlying LP and MIP solver
```

Can be set to `CLP_SOLVER` if BaPCod was appropriately configured by Cmake (see `README.md` file for instructions). Note that certain BaPCod features cannot be used with CLP solver, as the latter is not a MIP solver. Also the overall performance may be degraded.

3.2 Column generation parameters

```
SolverSelectForMast = a # algorithm to solve the restricted master LP
```

Possible values here are `a` (automatic LP solver), `p` (primal simplex method), `d` (dual simplex method), `b` (barrier method without crossover), and `c` (barrier method with crossover).

```
colGenSubProbSolMode = 2 # algorithm to solve the pricing problems
```

Possible values here are `2` (MIP solver) and `3` (user-defined pricing functor). By default, the pricing is solved by the MIP solver, even if the pricing functor is defined. In the case `colGenSubProbSolMode = 3`, the constraints of subproblem formulations are ignored, and thus their definition may be skipped.

```
mastInitMode = 3 # restricted master problem initialization mode
```

Possible values here are `1` (global artificial variables), `3` (local artificial variables), `4` (columns from the initial solution provided by the user), `5` (columns from the initial solution and global artificial variables), `6` (columns from the initial solution and local artificial variables).

```
ArtVarPenaltyUpdateFactor = 2.0 # artificial vars cost update factor
ArtVarMaxNbOfPenaltyUpdates = 5 # max. number of updates of these costs
```

If artificial variables participate in a solution of the master problem, their coefficients in the objective function are multiplied by the value of parameter `ArtVarPenaltyUpdateFactor`. The maximum number of such multiplications is defined by parameter `ArtVarMaxNbOfPenaltyUpdates`. If this number is reached, BaPCod switches to pure Phase 1 of the column generation procedure (the objective function is changed to minimizing the sum of values of artificial variables). If the artificial variables cannot be pushed out of the solution in pure Phase 1, the master problem is declared to be infeasible.

```
MaxNbOfStagesInColGenProcedure = 1 # number of col. gen. phases
```

Column generation phases are used to specify several algorithms for solving the pricing problems (only in the case the pricing functor is defined). Usually, during phase zero, the pricing problems are solved exactly, and the larger is the phase number, the "lighter" is the heuristic algorithm applied. The stages are solved successively, from phase `MaxNbOfStagesInColGenProcedure`-1 to phase zero. Column generation procedure passes to phase $k-1$ once phase k has converged.

```
GenerateProperColumns = false # generate only "proper" columns or not
```

If this parameter is set to true, the pricing problem is restricted to generate only so-called "proper" columns, i.e. columns which respect bounds on the subproblem variables. In this case, BaPCod will generate an error if the pricing problem generates a non-proper column. Generating only proper columns usually improves the Lagrangian dual bound produced by column generation. Setting this parameter to true is necessary if one uses generic subproblem branching (Vanderbeck branching). Setting this parameter to true is also necessary if one uses a diving-based primal heuristic (unless a heuristic pricing oracle generating proper columns is provided).

```
InsertAllGeneratedColumnsInFormRatherThanInPool = true
```

If this parameter is true, all generated columns are inserted directly in the restricted master LP, otherwise only the one with the smallest reduced cost (among columns corresponding to solutions of the same subproblem formulation) is inserted.

```
InsertNewNonNegColumnsDirectlyInFormRatherThanInPool = true
```

If this parameter is true, all generated columns are inserted directly in the restricted master LP, otherwise only columns with negative reduced cost are inserted.

```
ColumnCleanupThreshold = 10000
ColumnCleanupRatio = 0.66
```

Once the number of columns in the restricted master LP exceeds `ColumnCleanupThreshold`, only `ColumnCleanupRatio` part of them (with smallest reduced cost) remain, and the others are removed. The columns participating in the basis of the restricted master LP are never removed.

```
ReducedCostFixingThreshold = 0.9
```

This is the parameter to determine how often the reduced cost fixing procedure of the pricing functor is called. It is called if the current integrality gap is less than `ReducedCostFixingThreshold` part relative to the integrality gap when the reduced cost fixing procedure was called the last time. If the value is equal to `0.0`, no reduced cost fixing is performed. If the value is equal to `1.0`, reduced cost fixing is called after each convergence of the column generation prodedure.

3.3 Cut generation parameters

```
MasterCuttingPlanesDepthLimit = 1000 # max. tree depth for cut generation
MaxNbOfCutGeneratedAtEachIter = 1000 # max. number of cuts added per cut round
```

These are main parameters to determine when cut separation routines are called (set the first parameter to `-1` to switch off cut generation) and how much cuts are added at each cut separation round (the upper limit on the overall number of cuts from all cut separators).

```
CutCleanupThreshold = 1
```

If the number of cuts reaches this threshold, all non-active cuts are removed from the restricted master LP.

```
CutTailingOffThreshold = 0.02
CutTailingOffCounterThreshold = 3
```

These parameters are used to control the tailing-off condition of cut separation. The tailing-off counter is initialized with zero in the beginning of each branch-and-bound node. After a cut generation round, if the relative decrease of the integrality gap is smaller than the value of `CutTailingOffThreshold`, then the tailing-off counter is increased by one. If the previous relative integrality gap is more than 10%, then the decrease calculated is relative from the lower bound absolute value multiplied by 0.1. When the tailing-off counter reaches the value of `CutTailingOffCounterThreshold`, the tailing-off condition is activated: the cut separators with smaller priority are called if they are defined, or branching is performed.

3.4 Stabilization parameters

Implementation of stabilization techniques in BapCod follows the paper [11]. Please cite it if you use stabilization. By default, only the automatic dual price smoothing stabilization is activated.

Dual price smoothing parameters are the following.

```
colGenDualPriceSmoothingAlphaFactor = 1.0
colGenDualPriceSmoothingBetaFactor = 0.0
```

These two parameters correspond to parameters α and β introduced in the paper. The first parameter corresponds to Wentges smoothing [19] and the second parameter corresponds to directional smoothing. Value `0.0` means that the technique is not used. Value `1.0` means that the technique is used with automatic parameter setting. Any value in $(0, 1)$ fixes the corresponding parameter to this value.

Piecewise linear penalty function stabilization [2] parameters are the following.

```
colGenStabilizationFunctionType = 0
colGenProximalStabilizationRule = 1
StabilFuncKappa = 1.0
```

The first parameter sets the stabilization function type: `0` (penalty function stabilization is not used), `2` (3-piecewise linear function is used), `3` (5-piecewise linear function). The second parameter switches between the "curvature mode" (value `0`) and "explicit mode" (value `1`), see [11] for details. The third parameter sets the value for parameter κ introduced in the paper. The penalty function stabilization should be used with caution as it may deteriorate the column generation performance. We advice to use 3-piecewise linear function stabilization in "explicit mode" (only in the case of severe convergence problems). Parameter κ is very dependent on the problem at hand and even on the instance. Its value may vary broadly, from 0.001 to 1000 and sometimes even more.

Additional stabilization parameters are

```
colGenStabilizationMaxTreeDepth = 10000
StabilizationMinPhaseOfStage = 0
```

One can limit the stabilization use only to nodes at maximum depth `colGenStabilizationMaxTreeDepth` in the branch-and-bound tree. One can also limit the stabilization use only to column generation phases with number `StabilizationMinPhaseOfStage` and above.

3.5 Primal heuristic parameters

Implementation of primal heuristics in BapCod follows the paper [17]. Please cite it if you use heuristics. No heuristic is activated by default.

Parameters for the restricted master heuristic are the following

```
MaxTimeForRestrictedMasterIpHeur = -1
CallFrequencyOfRestrictedMasterIpHeur = 0
MIPemphasisInRestrictedMasterIpHeur = 1
PolishingAfterTimeInRestrictedMasterIpHeur = -1
```

The first parameter sets the maximum time in seconds for the MIP solver called to solve the restricted master MIP. The second parameter sets the frequency of the heuristic. Its value should be `1` to call it at every node of the branch-and-bound tree. The heuristic is called only at the root node if the value of the second parameter is not positive. The last two parameters correspond to parameters `CPX_PARAM_MIPEMPHASIS` and `CPX_PARAM_POLISHAFTERTIME` of the Cplex MIP solver. The restricted master heuristic cannot be used with CLP solver.

Parameters for the variants of the diving heuristic are the following

```
DivingHeurUseDepthLimit = -1
CallFrequencyOfDivingHeur = 0
```

The first parameter sets the maximum depth in the branch-and-bound tree for using the diving heuristic. If its value is negative, diving heuristic is not used. The second parameter is equivalent to `CallFrequencyOfRestrictedMasterIpHeur`.

```
RoundingColSelectionCriteria = 4 6 9
```

This parameter determines the criteria for column selection for rounding. This parameter should be initialized with a chain of integers separated by spaces. Each integer corresponds to a certain criterion. A criterion is used only if all previous ones could not select the column for rounding. The criteria are:

2 - highest priority (a column from a higher priority subproblem is preferred)

4 - smallest distance to the closest non-zero integer

5 - distance to the closest non-zero integer weighted by the column cost

6 - closest value to its round-up

9 - least column cost

```
FixIntValBeforeRoundingHeur = true
```

If set to true, then any column with integer value in the solution will be fixed before rounding a non-integer column. Otherwise, integer columns will be ignored (and thus may take different values later in the dive).

```
MaxNbOfCgIteDuringRh = 5000
```

This parameter limits the number of column generation iterations in each node of the diving heuristic.

```
MaxLDSbreadth = 0
MaxLDSdepth = 0
```

These parameters correspond to parameters `maxDiscrepancy` and `maxDepth` in the paper. If their values are positive, they serve to control the diving heuristic with Limited Discrepancy Search.

```
DivingHeurStopsWithFirstFeasSol = false
```

If set to true, the diving heuristic will stop as soon as it finds the first feasible solution (this behaviour corresponds to the "diving for feasibility" heuristic in the paper).

```
DivingHeurPreprocessBeforeChoosingVar = false
```

If set to true, the preprocessing will be launched after rounding of each candidate column (thus the diving will be slower). If preprocessing determines infeasibility, the candidate will be discarded and the next one will be considered. When this parameter is false, a dive is stopped, if the preprocessing determines the infeasibility.

```
StrongDivingCandidatesNumber = 1
```

If the value of this parameter is greater than `1`, the strong diving heuristic will be activated. This parameter corresponds to parameter `maxCandidates` in the paper.

```
EvalAlgParamsInDiving =
```

This is an optional parameter sequence to set the behaviour of the column and cut generation procedure at every node of the diving heuristic. The instantiation of this parameter is similar to the instantiation of the parameters for strong branching phases (described in Section 3.6). If this parameter sequence is empty, the same parameters are used as for the column at cut generation in the main branch-and-bound tree.

The following parameters are for the local search heuristic (corresponds to the diving heuristic with restarts in the paper).

```
LocalSearchHeurUseDepthLimit = -1
LocalSearchColSelectionCriteria =
LocalSearchHeurUseDepthLimit = 2
MaxFactorOfColFixedByLocalSearchHeur = 0.8
MaxLocalSearchIterationCounter = 3
```

The first parameter sets the maximum depth in the branch-and-bound tree for using the local search heuristic. If its value is negative, the heuristic is not used. The second parameter can be set in the same way as the parameter `RoundingColSelectionCriteria` above. The third parameter sets the maximum depth in the branch-and-bound tree for using the local search heuristic. The last two parameters correspond to parameters `fixRatio` and `numIterations` in the paper.

Also, the following parameters described above have an impact on the local search heuristic : `MaxNbOfCgIteDuringRh`, `DivingHeurPreprocessBeforeChoosingVar`, and `FixIntValBeforeRoundingHeur`.

3.6 Strong branching parameters

The strong branching with phases is implemented in BapCod. Each phase has its proper parameters given as a sequence of numbers separated by spaces :

```
StrongBranchingPhaseOne =
StrongBranchingPhaseTwo =
StrongBranchingPhaseThree =
StrongBranchingPhaseFour =
```

The parameter sequence is empty if the corresponding strong branching phase is not active, i.e. is not used. For each active phase the order of parameters is the following

1. Boolean indicating where this phase is exact or not. In the exact phase the column and cut generation parameters do not change. The exact phase should always be the last active phase to guarantee the optimality of the final solution. Candidates in an exact phase are selected using the minimum estimated tree size rule (see [6] for a method to estimate the tree size). Candidates in a non-exact phase are selected using maximum product rule, i.e. according to the product of lower bound increases in the child nodes.

2. The maximum number of candidates evaluated.

3. Tree size ratio to stop : the maximum number of candidates evaluated in this phase does not exceed this value multiplied by the estimated size of the subtree rooted at the father node. If there is no father node (i.e. for the root), the latter value is equal to infinity.

The next parameters are given only for a non-exact phase.

4. Maximum number of column generation iterations. If this parameter is zero, no column generation is performed, i.e. only re-optimization of the restricted master LP is performed.

The next parameters are given only for a non-exact phase with column generation.

5. Minimum phase for the column generation
6. Minimum number of cut generation rounds
7. Maximum number of cut generation rounds
8. Boolean indicating whether the reduced cost fixing is performed.
9. The frequency of column generation output, i.e. the number of column generation iteration between two consecutive lines in the output. If the frequency is zero, then only one summary line is shown.

The following is an example of instantiation of the strong branching parameters.

```
StrongBranchingPhaseOne = false 100 0.5 0
StrongBranchingPhaseTwo = false 5 0.1 100 1 0 0 false 0
StrongBranchingPhaseThree = true 1
```

Here, during the first phase, at most $\min\{100, 0.5e\}$ candidates will be evaluated, where e is the estimated subtree size of the father node, and only restricted master will be solved for them without column generation. During the second phase, the best $\min\{5, 0.1e\}$ candidates from the first phase will be evaluated. At most 100 column generation iterations will be performed for each node of each branching candidate. During column generation, all stages except stage zero will be considered. No cut generation and no reduced cost fixing will be performed. No column generation statistics will be shown during this phase (only one summary line). In the third phase, all branches of the best candidate from the second phase will be evaluated exactly. The purpose for this immediate evaluation is to estimate the subtree size rooted at the current node. The fourth phase is not defined and thus it is not active.

Finally, the following parameter is used to activate the simplified setting of strong branching parameters:

```
SimplifiedStrongBranchingParameterisation = false
```

If it is set to `true`, then the strong branching parameters are set to

```
StrongBranchingPhaseOne = false <p1> <p2> 0
StrongBranchingPhaseTwo = false <p3> <p4> 10000 1 0 0 false 0
StrongBranchingPhaseThree = true 1
StrongBranchingFour =
```

where `<p1>`, `<p2>`, `<p3>`, and `<p4>` are set by the following parameters:

```
StrongBranchingPhaseOneCandidatesNumber = 100    # <p1>
StrongBranchingPhaseOneTreeSizeEstimRatio = 0.3 # <p2>
StrongBranchingPhaseTwoCandidatesNumber = 3      # <p3>
StrongBranchingPhaseTwoTreeSizeEstimRatio = 0.1 # <p4>
```

4 BaPCod statistics

This section overviews the statistics which can be retrieved after solving the model (see Section 2.1) for the methods to use.

4.1 Timers

The values of all timers are in ticks. To obtain the time in seconds, the value should be divided by 100.

bcTimeMain - overall solution time

bcTimeBaP - total branch-cut-and-price time (excludes formulations building time and the final solution disaggregation)

bcTimeMIPSol - solution time when just a MIP formulation is solved without decomposition (excludes formulations building time)

bcTimeRootEval - solution time of the root (excluding primal heuristics and branching)

bcTime1stLP - solution time of the first column generation convergence at the root (before cut separation)

bcTimeColGen - total column generation time (all time to solve the master problem including generation of columns)

bcTimeMastMPsol - total time taken for solving the restricted master LPs by the LP solver

bcTimeCgSpOracle - total time taken for generation of columns (computing the reduced cost of subproblem variables, solving the pricing problems, generating the columns from subproblem solutions, and adding columns to the restricted master LP including computation of column coefficients in the master constraints)

bcTimeSpUpdateProb - total time taken to compute the reduced cost of subproblem variables

bcTimeSpMPsol - total time taken to solve the pricing problems

bcTimeSetMast - total time to update the formulations of the master problem and subproblems before solving a node

bcTimeSepFracSol - total branching time, i.e. generation of branching candidates (evaluation of branching candidates in strong branching is not included)

bcTimeCutSeparation - total time for cut separation

bcTimeAddCutToMaster - total time for adding cuts to the restricted master LP (essentially computing coefficients of master variables and columns in the cuts)

bcTimeRedCostFixAndEnum - total time for reduced cost fixing and enumeration

bcTimeEnumMPsol - total time to solve enumerated MIPs

bcTimeSBphase1 - total time for evaluating branching candidates during phase one of strong branching

bcTimeSBphase2 - total time for evaluating branching candidates during phase two of strong branching

bcTimePrimalHeur - total time for running primal heuristics

Some times may "intersect". For example, primal heuristic time includes a part of column generation time (in the case of diving heuristics). Evaluation time for branching candidates also includes a part of column generation time and a part of solving the restricted master LPs by the LP solver.

4.2 Records and counters

bcRecRootDb - the lower bound obtained at the root node (rounded up if the objective function is integer)

bcRecRootLpVal - the value of the master problem solution at the root node (not rounded even if the objective function is integer)

bcRecBestDb - final lower bound obtained by the branch-and-bound (rounded up if the objective function is integer)

bcRec1stLPDb - lower bound obtained by the column generation procedure at the root (before cut separation)

bcRecBestInc - best upper bound (value of the best feasible solution found) obtained by the branch-and-bound (the solution is optimal if this value is equal to **bcRecBestDb**)

bcCountCg - total number of iterations in the column generation procedure

bcCountCol - total number of generated columns (not all columns are necessarily added to the restricted master LP)

bcCountNodeProc - total number of processed nodes in the branch-and-bound tree

bcCountCutInMaster - total number of cuts added to the restricted master LP

bcCountMastSol - total number of times the restricted master LP has been solved by the LP solver

bcCountSpSol - total number of times the pricing problems have been solved

bcCountMastIpSol - total number of times the restricted master has been solved as a MIP

5 BaPCod output

See Section 3.1 for the parameter to change the verbosity of the BaPCod output. It is possible to suppress all output except important warnings and errors. In the case the output is not suppressed, BaPCod will output the following information.

At the beginning BaPCod prints its version:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              BaPCod v063,  2/09/2021, © Inria Bordeaux, France
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

At the beginning of every branch-and-bound node, BaPCod outputs the following information

```
*********************************************************************************************
**** BaB tree node (N° 5, parent N° 3, depth 2)
**** Local DB = 831.794, global bounds : [ 831.794 , 878.08 ], TIME = 1h22m49s40t = 496940
**** 3 open nodes, 24679 columns (8375 active), 426 dyn. constrs. (199 active),
     559 art. vars. (332 active)
*********************************************************************************************
```

The first line prints the index of the current node (indices are given in the order of node creation), index of the father node, and the depth of the current node. The second line outputs the lower bound value of the current node, the global lower and upper bounds (valid for the model), and the current elapsed time. The third line outputs the current number of open branch-and-bound nodes (i.e. which were created but not yet pruned), and the aggregated information about the restricted master problem. This information includes the total number of columns in the memory (number of columns present in the current restricted master LP), the total number of cuts and branching constraints in the memory (number of cuts and branching constraints present in the current restricted master LP), the total number of artificial variables in the memory (the number of artificial variables present in the current restricted master LP).

During the column generation procedure, BaPCod outputs the following information

```
#<DWph=2> <it=  1> <et=4621.25> <Mt= 0.66> <Spt= 1.66> <nCl=  3> <al=0.50> <DB=  817.8515>
 <Mlp=  830.2749> <PB=878.08035>
```

This information includes the current column generation phase (printed only if the number of phases is more than one), the current column generation iteration number, the current elapsed time, the time in seconds to solve the restricted master LP in the current iteration, the total time in seconds taken for generating columns in the current iteration (includes the time to solve the pricing problems), the number of columns added to the restricted master LP in the current iteration, the current dual pricing smoothing stabilization parameter α (if dual pricing smoothing stabilization is activated), the value of the Lagrangian lower bound at the current iteration, the value of the current restricted master LP (upper bound on the solution value of the master problem), and the value of the current best known feasible solution of the model (or the initial cut-off value given if not yet improved). If artificial variables are present in the current solution of the restricted master LP, then character **#** is printed in the beginning of the line.

In some cases, one line is printed per several column generation iterations. This can be recognized if the iteration numbers are not consecutive in two consecutive lines. In this case the time for solving the restricted master LP, the time to generate columns, and the number of generated columns are aggregated for all iterations since the iteration of the previous line (or since the beginning of the column generation procedure).

If some cuts are added during a cut separation round, BaPCod outputs the following information

```
----- Add fac. cuts : R1C(291) SSI(3), max.viol = 0.578033, aver.viol = 0.260438,
      sep/add took 1.43/2.13 sec. -----
19840 columns (7441 active), 797 dyn. constrs. (441 active), 1266 art. vars. (574 active)
```

This information includes whether facultative (`fac.`) or core (`core`) cuts were added, the number of added cuts separately for every family, the maximum violation among added cuts, the average cut violation among added cuts, and the times spent for separation and adding cuts to the restricted master LP in seconds. If `zero.viol = #` is shown, then there are some generated but non-violated cuts. These may indicate the presence of a bug in the cut separation procedure (unless non-violated cuts are intentionally generated). The same aggregated information about the restricted master problem as in the beginning of the branch-and-bound node is also printed.

If cut separation routines are called, but no cuts were generated, BaPCod outputs

```
----- no cuts found
```

After a cut separation round and subsequent column generation procedure, BaPCod outputs the relative change of integrality gap (together with the lower bound value before the previous cut round):

```
Gap improvement since the last cut separation : 0.0144399 (845.956)
```

After reaching the tailing-off condition, BaPCod outputs either

```
----- Cut separators priority level decreased to 1 -----
```

if cut separators with lower priority exist, or otherwise

```
----- Cut generation is stopped due to tailing off -----
```

After evaluation of a branching candidate in strong branching, BaPCod outputs the following information

```
SB phase 1 cand. 9 branch on var U_1_OmastV (lhs=0.2362) : [  834.1127,   862.0468],
                                                           score = 261.81 (h)  <et=3303.55>
```

This information includes the number of the current strong branching phase, index of the branching candidate, description of the branching candidate (here we have branching on the value of variable U_1), the left-hand-side value of the branching candidate (current value of the branching variable or the left-hand-side value of the branching constraint), solution values for the restricted master LP for both branches, the score of the branching candidate (the larger is the score, the better it is) and the current elapsed time. If the branching candidate was included to the current set of candidates because of its good previous performance according to the branching history, then character `(h)` is printed (otherwise the branching candidate was generated according to the branching strategy used).

At the end of the solution process, the final values for global lower and upper bounds are printed together with the overall solution time:

```
*********************************************************************************************
Search is finished, global bounds : [ 835.602 , 835.602 ], TIME = 1h48m56s75t = 653675
*********************************************************************************************
```

If values of global bounds are equal, then the best found solution is optimal.

6 VRPSolver extension

VRPSolver extension includes an implementation of the pricing functor which allows the user to define the subproblems as resource constrained shortest path problems in graphs. The functor implements the bucket-graph based labeling algorithm from paper [16] for solving the pricing problem, as well as the corresponding bucket arc elimination procedure (i.e. reduced cost fixing procedure), and the elementary route enumeration procedure [1]. VRPSolver extension also implements cut separation functors for rounded capacity cuts [7] and limited memory rank-1 packing cuts [9], as well as packing set based Ryan-and-Foster branching, and branching over accumulated resource consumption [10]. We strongly advise to read paper [12] before using VRPSolver extension. Please, cite this paper if your are using the VRPSolver extension. For the moment, the extension can be obtained in the compiled form by request from the corresponding author.

To use VRPSolver functors, one needs to include the corresponding header file:

```
#include "bcModelRCSPSolver.hpp"
```

6.1 VRPSolver pricing functor

To create such pricing functor one should use the constructor

```
BcRCSPFunctor(const BcFormulation & spForm);
```

Afterwards, this functor should be attached to the subproblem formulation as described in Section 2.9 (`BcRCSPFunctor` inherits from `BcSolverOracleFunctor`).

The information about the resource-constrained path structure should be given by defining a graph handler of type `BcNetwork`:

```
BcNetwork(BcFormulation & bcForm, int numElemSets = 0,
          int numPackSets = 0, int numCovSets = 0);
```

Here `bcForm` is the subproblem to which we associate the graph. Other three arguments are the number of elementarity sets, the number of packing sets, and the number of covering sets. These sets are used to express elementarity, packing, and covering constraints over the arcs and/or nodes of graphs (see [12]). The distance matrix for elementarity sets can be passed using the following method of `BcNetwork`:

```
void setElemSetsDistanceMatrix(
     const std::vector<std::vector<double>> & matrix);
```

Vertices and arcs of the graph can be defined using the following methods of `BcNetwork`:

```
const BcVertex createVertex();
const BcArc createArc(int tail, int head, double originalCost);
```

Here `tail` and `head` are the indices of the tail and head vertices of the arc, and `originalCost` is the "pure" cost of the arc. The coefficient of a column representing a path in a graph in the objective function is determined not only by the "pure" costs of its arcs, but also by objective function coefficients of subproblem variables mapped to the arcs (see below how to map arcs to variables). An important notice here is that a special "cost" subproblem variable should be communicated to the pricing functor if at least one arc has a non-zero "pure" cost:

```
void BcRCSPFunctor::setPureCostBcVar(BcVar bcVar);
```

The value of this variable in the subproblem solution representing a path will be set to the total "pure" cost of arcs which form the path. To define the arc cost one can alternatively map a variable to a subproblem variable (see below) and define a coefficient of this variable in the objective function (see Section 2.5).

The source and the sink of the graph can be defined using the following methods of `BcNetwork`:

```
void setPathSource(const BcVertex & vertex);
void setPathSink(const BcVertex & vertex);
```

Indices of vertices and arcs are always assigned in the order of creation of vertices and arcs (starting from 0). These indices can be retrieved using methods

```
int BcVertex::ref() const;
int BcArc::ref() const;
```

Vertices and arcs can be retrieved from their indices using the following methods of the handler `BcNetwork`:

```
const BcVertex getVertex(int id) const;
const BcArc getVertex(int id) const;
```

One can add a vertex or an arc to an elementarity, packing, or covering set using the following methods defined both for `BcVertex` and `BcArc`:

```
void setElementaritySet(int elemSetId);
void setPackingSet(int packSetId);
void setCoveringSet(int covSetId);
```

The set indices here should be not less than 0 and smaller than the corresponding number of elementarity, packing, or covering sets given in the constructor of `BcNetwork`.

An arc can be mapped to a variable belonging to the same subproblem using the following methods of `BcNetwork`:

```
void addVarAssociation(const BcVar & newvar, const double coeff);
void arcVar(const BcVar & newvar);
```

The first method defines a mapping with an arbitrary coefficient, whereas the second method defines a standard mapping introduced in [12]. The standard mapping has coefficient 1.0. The coefficient of a column respresenting a path in a graph in a master constraint is determined by the scalar product of the vector of coefficients of subproblem variables in this constraint and the sum of vectors of mapping coefficients for the arcs which form the path.

A resource can added by defining the resource handler using the constructor

```
BcNetworkResource(const BcNetwork & bcNetwork, int id);
```

Here `id` is the index of the ressource. All resources should have different indices. By default, ressources are secondary and disposable (see [12] for the definition of resource types). To define a main resource or a non-disposable resource, please use the following methods of `BcNetworkResource`:

```
void setAsMainResource();
void setAsNonDisposableResource();
```

Consumption of a resource for an arc can be defined using the following method of `BcNetworkResource`:

```
void setArcConsumption(const BcArc & bcArc, double consumption);
```

Lower and upper bounds for the accumulated resource consumption can be defined on vertices and/or on arcs using the following method of `BcNetworkResource`:

```
void setArcConsumptionLB(const BcArc & bcArc, double consumptionLB);
void setArcConsumptionUB(const BcArc & bcArc, double consumptionUB);
void setVertexConsumptionLB(const BcVertex & bcVertex, double consLB);
void setVertexConsumptionUB(const BcVertex & bcVertex, double consUB);
```

At most 20 resources can be defined, among which at most two can be main. The main resource with the smallest index is used to determine the threshold for the bi-directional labeling algorithm which solves the pricing problem (see [16]).

There is a possibility to define special resources for which the accumulated resource consumption can be either 0 or 1 (i.e. binary resource). These resources are declared implicitly by defining consumption of such resources on arcs using the following method of `BcArc`:

```
void addBinaryResourceConsumption(int binaryResId, int consumption)
```

and by defining accumulated resource consumption bounds using the following methods of `BcVertex`:

```
void setBinaryResourceConsumptionLB(int binaryResId, int lowerBound);
void setBinaryResourceConsumptionUB(int binaryResId, int upperBound);
```

Here `lowerBound` and `upperBound` can take either value 0 or value 1. It makes sense to only define non zero values for the (accumulated) resource consumption (bounds). The index (id) of a binary ressource can be between 0 and 511. Again, by default every binary resource is disposable. To declare a binary resource non-disposable, the following method of `BcNetwork` can be used:

```
void setBinaryResourceNonDisposable(const int binaryResId);
```

Besides solving a BaPCod model with RCSP functors in the standard way, one can also enumerate all feasible paths in directed graphs associated with subproblems:

```
BcSolution BcModel::enumerateAllColumns(int & nbEnumColumns);
```

The total number of enumerated columns is returned in `nbEnumColumns`, it is equal to -1 if enumeration did not succeed. The enumerated solutions are returned in the solution chain, see Section 2.6 for details. This method should be used for very small instances mainly for debugging and teaching purposes.

6.2 VRPSolver cut separation functors

The rounded capacity cuts [7] require an undirected graph $G = (V \cup \{0\}, E)$ with special "depot" node 0. There should be exactly one binary variable x_e associated with every edge $e \in E$. A positive demand d_v should be associated with every non-depot node $v \in V$, and a positive capacity C should be defined. A rounded capacity cut is defined for a subset $S \subseteq V$ of nodes. It states that there should be enough paths passing by set S to satisfy path capacity C:

$$\sum_{e=(i,j):\ |\{i,j\}\cap S|=1} x_e \geq 2 \cdot \left\lceil \sum_{i \in S} d_i / C \right\rceil .$$

To add the functor for separation of rounded capacity cuts (the separation algorithm follows paper [8]), on should use the constructor

```
BcCapacityCutConstrArray(const BcFormulation & formulation,
                         const int & maxCapacity,
                         const std::vector<int> & demands,
                         const bool & isFacultative = true,
                         const bool & equalityCase = true,
                         const int & twoPathCutsResId = -1,
                         const double & rootPriorityLevel = 1.0,
                         const double & nonRootPriorityLevel = 1.0);
```

Here `formulation` is the master formulation handler, `maxCapacity` is the value for capacity C, `demands` is the vector d of demand values. Argument `isFacultative` determines whether cuts are facultative (separated only for fractional solutions) or core (separated also for integer solutions). Argument `twoPathCutsResId` should be equal to `-1`. The remaining two arguments define the root and non-root priority level (see Section 2.8). This cut separator is introduced in [12].

The construction of undirected graph $G = (V \cup \{0\}, E)$ is based on all directed graphs defined for subproblems of the BaPCod model. Vertices in

directed graphs defined for VRPSolver pricing functors are "projected" into vertices of graph G. The projection is defined based on packing sets (`equalityCase=true`) or on covering sets (`equalityCase=false`), and on vector of demands. The first case (`equalityCase=true`) should be used when exactly one vertex in every packing set should be visited exactly once (all other vertices in the set should not be visited). The second case (`equalityCase=false`) should be used when at least on vertex in every packing set should be visited at least once (all other vertices in the set may or may not be visited). A vertex i in a directed VRPSolver graph is projected into vertex $v \in V$ if i belongs to packing/covering set v and d_v is positive. Otherwise, i is projected into depot vertex 0. An edge (v, v') belongs to E if there exists an arc (i, j) in a directed graph of VRPSolver such vertex pair $\{i, j\}$ projects into vertex pair (v, v'). In this case, we say that arc (i, j) projects into edge (v, v'). A BaPCod variable is *appropriate* for an edge $e \in E$ if it binary, mapped with coefficient one only to (some or all) arcs (i', j') or (j', i') in directed graphs of VRPSolver which projects into edge e, and not mapped to any other arc with any coefficient.

Separation of rounded capacity cuts can be used only if

1. packing or covering sets (depending on argument `equalityCase`) are defined on vertices, i.e. no arc in any directed VRPSolver graph belongs to packing/covering set;
2. for each VRPSolver graph, each its arc projected to an edge (and not to a node) in graph G is mapped to exactly one appropriate BaPCod variable.

Appropriate variables are used to generate rounded capacity cuts. Note that during projection we loose information about arc direction. Nevertheless, the generated cuts remain valid. Moreover, separation of rounded capacity cuts as core cuts is sufficient to replace the capacity resource in VRPSolver graphs if

- we are in the equality case (`equalityCase=true`);
- for every arc in VRPSolver directed graphs there exists an arc in the opposite direction;
- all vertices in VRPSolver directed graphs except the source and the sink belong to a packing set;
- all demands are positive and equal to all vertices belonging to the same packing set;
- path capacity C is equal for all VRPSolver directed graphs.

In this case, declaration of the capacity resource may be skipped in VRPSolver directed graphs. This may increase the performance when the capacity resource is not tight (as for some Solomon instances of the Vehicle Routing Problem with Time Windows).

To add the functor for separation of limited-memory rank-1 cuts, on should use the constructor

```
BcLimMemRankOneCutConstrArray(const BcFormulation & formulation,
                              const double & rootPriorityLevel = 1.0,
                              const double & nonRootPriorityLevel = 1.0);
```

Here `formulation` is the master formulation handler, and the remaining two arguments define that the root and non-root priority level (see Section 2.8). Limited-memory rank-1 cuts are introduced in [9]. VRPSolver uses the definition of packing/covering sets in the model to separate rank-1 packing/covering cuts (see [16]). VRPSolver pricing functor supports limited-memory rank-1 cuts.

6.3 VRPSolver branching functors

VRPSolver extension includes two functors for separating non-robust branching constraints (these constraints are supported in the pricing functor).

The first functor implementes separation of packing-set-based Ryan&Foster branching constraints. To add this functor, one should use the constructor

```
BcPackSetRyanFosterBranching(const BcFormulation & formulation,
                             const double & priorityLevel = 1.0);
```

Here `formulation` is the master formulation handler, and `priorityLevel` is the branching priority value (see Section 2.7). This functor searches branching candidates, each of which corresponds to a pair of packing sets (see [12] for the definition of packing sets). A branching candidate generates two branching constraints. In the first one, the number of columns corresponding to paths which include arcs (or vertices) belonging to both packing sets is set be equal to one. In the second constraint, the number of such columns is set to be equal to zero.

There is also a possibility to define permanent packing-set-based Ryan&Foster constraints, using the method of `BcNetwork`:

```
void addPermanentRyanAndFosterConstraint(int firstPackSetId,
                                         int secondPackSetId,
                                         bool together);
```

This method adds the following constraint to the subproblems associated with the graphs. In the case `together=false`, the path should include arcs (or vertices) belonging to at most one of packing sets `firstPackSetId` and `secondPackSetId`. In the case `together=true`, the path should include arcs (or vertices) belonging to non of these packing sets or to both of them.

The second functor implementes branching over accumulated resource consumption, introduced in [10]. To add this functor, one should use the constructor

```
BcPackSetResConsumptionBranching(const BcFormulation & formulation,
                                 const double & priorityLevel = 1.0);
```

Here `formulation` is the master formulation handler, and `priorityLevel` is the branching priority value (see Section 2.7). This functor searches branching candidates, each of which corresponds to a packing set p, threshold value τ for accumulated consumption of resource with index 0. A branching candidate generates two branching constraints. The first one forbids all columns corresponding to paths in which an arc (or vertex) belonging to packing set p is visited when the accumulated consumption of resource 0 is less than τ. The

second one forbids all columns corresponding to paths in which an arc (or vertex) belonging to packing set p is visited when the accumulated consumption of resource 0 is greater or equal to τ.

6.4 VRPSolver paramerization

First of all, the following parameters should be set when using VRPSolver extension:

```
MaxNbOfStagesInColGenProcedure = 3
colGenSubProbSolMode = 3
```

The following additional parameters are defined for the VRPSolver extension. Please consult [12] for more explanation about these parameters.

```
RCSPstopCutGenTimeThresholdInPricing = 10
RCSPhardTimeThresholdInPricing = 20
```

"Soft" and "hard" time thresholds in seconds for the labeling algorithm (τ^{soft} and τ^{hard} in the paper).

```
RCSPnumberOfBucketsPerVertex = 25
RCSPdynamicBucketSteps = 1
```

Parameters for calculation of step sizes for buckets (ψ^{buck} and ψ^{reduc} in the paper). Dynamic adjustment of bucket steps is on if `RCSPdynamicBucketSteps` takes value `1` and off if it takes values `0`.

```
RCSPuseBidirectionalSearch = 2
```

Parameter to determine when the bi-directional labeling algorithm is used to solve the pricing problem (ϕ^{bidir} in the paper): `0` — not used, `1` — always used, `2` — used only during the exact column generation phase.

```
RCSPapplyReducedCostFixing = 1
```

Parameter to determine which bucket arc elimination (reduced cost fixing) procedure is used (ϕ^{elim} in the paper): `0` — not used, `1` — standard bucket arc elimination procedure (follows [16]), `2` — light bucket arc elimination procedure (less strong, but faster), `3` — standard arc elimination procedure (similar to [5]), `4` — light arc elimination procedure.

```
RCSPmaxNumOfColsPerExactIteration = 150
RCSPmaxNumOfColsPerIteration = 30
```

Maximum number of generated columns per column generation iteration and per subproblem with the RCSP pricing functor (γ^{exact}, γ^{heur} in the paper). The first parameter is for the exact column generation phase, and the second parameter is for heuristic column generation phases.

```
RCSPmaxNumOfLabelsInEnumeration = 1000000
RCSPmaxNumOfEnumeratedSolutions = 1000000
RCSPmaxNumOfEnumSolutionsForMIP = 10000
RCSPmaxNumOfEnumSolsForEndOfNodeMIP = 0
```

The first two parameters are for the maximum number of labels and paths in the elementary path enumeration procedure (ω^{labels} and ω^{routes} in the paper). The third parameter is for the maximum total number of enumerated paths to trigger the solving of enumerated MIP (ω^{MIP}). The fourth parameter is for the maximum total number of enumerated path to trigger the solving of enumerated MIP at the of a branch-and-bound node. The last parameter is active only if its value is greater than the value of the third parameter. Solving enumerated MIP is not supported if the CLP solver is used instead of CPLEX.

```
RCSPinitNGneighbourhoodSize = 8
RCSPmaxNGneighbourhoodSize = 8
```

Initial and maximum size of ng-sets (η^{init} and η^{max} in the paper).

```
RCSPrankOneCutsMaxNumPerRound = 100
RCSPrankOneCutsMaxNumRows = 5
RCSPrankOneCutsMemoryType = 2
RCSPrankOneCutsLSnumIterations = 1000
```

Parameters for separation of limited-memory rank-1 cuts. First three ones correspond to parameters θ^{num}, θ^{rows}, θ^{mem} in the paper. Possible values for the third parameters are: `0` — automatic selection of node- or arc-memory (the root node may be solved two times in this case), `1` — arc-memory is used, `2` — node-memory is used. The fourth parameters sets the number of iterations in the local search heuristic to separate rank-1 cuts with four rows and more.

```
RCSPallowRoutesWithSameVerticesSet = true
```

Whether multiple paths which pass by the same set of vertices (in different order) can be generated in the same iteration of the column generation procedure. Setting this parameter to `false` may improve convergence (as more diversified set of paths is generated in every column generation iteration) but may slow down the labelling algorithm as the additional check is necessary.

```
RCSPredCostFixingFalseGap = 0.0
```

If the value of this parameter is ≤ 1.0 that it has no effect. If its value is > 1.0 then reduced cost fixing and enumeration are performed with the primal-dual gap is divided by this value, making the whole branch-cut-and-price algorithm heuristic. More about the *false gap mechanism* can be found in paper [14].

```
SafeDualBoundScaleFactor = -1
```

If this parameter is positive, it activates the calculation of the safe dual bound (parameter $\tilde{K}$ in the paper). This may be important to avoid rounding errors when the objective function is to minimize the number of paths in the solution. Safe lower bounds for column generation were introduced in [4].

The following parameters are used to activate the primal heuristic which is based on the elementary path enumeration with false gap. This heuristic was proposed in [13] and was also used in [14].

```
RCSPmaxNumOfLabelsInHeurEnumeration = 0
MaxNumEnumSolsInRestrictedMasterIpHeur = 5000
```

If the first parameter is positive, then the elementary path enumeration is triggered at the end of a some branch-and-bound nodes (depends on parameter `CallFrequencyOfRestrictedMasterIpHeur`). The primal-dual gap is divided by two each time the enumeration does not succeed for at least one subproblem associated to the RCSP pricing functor. After succeeding enumeration, at most `MaxNumEnumSolsInRestrictedMasterIpHeur` columns corresponding to paths with the smallest reduced cost are added to the restricted master, and the latter is solved as a MIP. The restricted master heuristic parameters (see Section 3.5) also apply here. Again, this primal heuristic cannot be used with the CLP solver.

6.5 VRPSolver output

To activate additional output of the VRPSolver extension, one needs to set parameter

```
DEFAULTPRINTLEVEL = 0
```

During the initialization of each RCSP pricing functor, the following information is shown:

```
RCSP solver info : symmetric case is detected for graph G_0!
Bidirectional border value is initialised to 100
RCSP solver info : number of forw. reachable buckets /
                   buck. strongly connected components is 3856( 97.5709% ) / 1931( 50.0778% )
```

The information whether the symmetric case is detected, the initial value for the bi-directional threshold, and the initial number of reachable buckets and the number of strongly connected components in the bucket graph (together with the percentage from the total number of buckets). See paper [16] for an additional explanation.

After each execution of the exact bucket-graph based labeling algorithm (see paper [16]) to solve the RCSP pricing problem, some statistics are shown:

```
RCSP exact solver info for graph G_0 : TT = 0.092659, pt = 0.000274, dt = 0.079908,
ct = 0.009966, ndl = 18', bdl = 466', odl = 15',  odf = 136', bsi = 1', cnt = 22',
bdch = 3616', odch = 280', lcp = 189', #sols = 150
```

This statistics include:

- the index of the subproblem to which the directed graph is associated (`G_0` means index 0);

- `TT`: the total labelling time in seconds
- `pt`: preparation time in seconds (retrieving information about reduced costs and non-robust cuts)
- `dt`: dynamic programming time in seconds (extension of labels and dominance checks)
- `ct`: concatenation time in seconds (in the bi-directional labeling)
- `ndl` : number of non-dominated labels in thousands
- `bdl` : number of labels dominated by labels in the same bucket, in thousands
- `odl` : number of labels dominated by labels in different buckets, in thousands
- `odf` : number of times the function which checks the dominance between a label and all labels in a different bucket is called, in thousands
- `bsi` : number of times a bi-directional solution is inserted to the pool of solutions of the labeling algorithm, in thousands
- `cnt` : number of times a pair of forward and backward labels is tried for concatenation, in thousands
- `bdch` : number of dominance checks between labels in the same bucket, in thousands
- `odch` : number of dominance checks between labels in different buckets, in thousands
- `lcp` : number of times a label is copied in the memory, in thousands
- `#sols` : number of solutions generated by the labeling algorithm

Every time the bucket arc elimination (reduced cost fixing) procedure is launched, the following information is printed:

```
Reduced cost fixing for graph G_0... 40752 buck. arcs remain (79.1% from prev., 3.69% from max.)
 + 41953 jump buck. arcs (3.8% from max.)
 TT = 2.81583, pt = 0.450316, dt = 1.11438, ct = 1.24473, ndl = 307', bdl = 1252', odl = 30',
 odf = 1139', lpcb = 1790', cnt = 20528', bdch = 40847', odch = 28678', lcp = 9921'
Labels distribution in buckets (bucket size) : largest - 507, top 0.1% - 472, top 0.5% - 376,
 top 2% - 277, top 10% - 114, top 50% - 9
Run forward enumeration with border = 100... succeeded! lpt = 1.53273, ndl = 95', dl = 6',
 lpcb = 912', dch = 125'
Run enumeration concatenation with time limit 30.2322 sec.... succeeded!
Sorting and storing enumerated solutions ... done!
 TT = 3.1909, pt = 0, dt = 1.53273, ct = 0.937832, ndl = 95', bdl = 6', lpcb = 912', cnt = 11817',
 bdch = 125', odch = 0', lcp = 0'
Enumeration succeeded!, number of elem. solutions is 103246
Estimating inspection time...     TT = 0.061083, performed by inspection with 103246 solutions
Inspection time is low enough. Pricing will be done by inspection.
Removed 5445 columns (not in the enumerated set) from the formulation
Regenerated 2114 columns with the 'enumerated' flag
```

First, the number of remaining bucket arcs after the reduced cost fixing procedure is printed for each direction together with the percentage from the number of bucket arcs before the procedure and from the maximum possible number of bucket arcs. Then the statistics of the reduced cost fixing procedure are printed (similarly to the standard labelling algorithm). The time to determine whether a bucket arc can be eliminated or not is included in the concatenation time. Then, the labels distribution statistics is shown, i.e. the number of labels in the largest bucket, and the minimum number of labels in 0.1%, 0.5%, 2%, 10%, and 50% of largest buckets. Afterwards, the elementary path enumeration procedure is attempted, and its statistics are shown (which are again similar to to the standard labelling statistics, except that there is no buckets). If enumeration procedure succeeds, the number of elementary solutions is shown. Then the inspection of these solutions (to calculate their reduced cost) is tried. If the inspection time is sufficiently low, the pricing problem passed to the "enumerated mode". Then VRPSolver outputs the number of columns removed from the restricted master (because they do not correspond to any elementary solution in the enumerated set) and the number of columns regenerated (their coefficients in master constrains may change as rank-1 cuts has full memory in the enumerated mode).

When the elementary path enumeration procedure does not succeed, the ratio between the final number of non-extended labels and the total number of non-dominated labels is shown:

```
Run forward enumeration with border = 100... not succeeded (ratio 0.513939)
```

The closer this ratio to zero, the closer is the enumeration procedure to be successful.

If the reduced cost fixing procedure is not called, then this information is shown:

```
Full reduced cost fixing is not called (gap ratio is 0.959188)
Dynamic params and stats :  aver.obdmd = 44.8224, aver.buck.num. = 51, nbR1C = 274
                                                                    with avMem = 11.3285
```

This information contains the ratio of the current primal-dual gap in comparison with the primal-dual gap when the reduced cost fixing procedure was called the last time (this ratio should be below the value of parameter `ReducedCostFixingThreshold` to trigger the procedure). The information in the next line contains the average value for the maximum depth when checking domination between labels in different buckets, the average number of buckets per vertex, the number of active rank-1 cuts, and their average memory size.

When separating rank-1 cuts, the number of generated cuts for each number of rows is shown together with the statistics for maximum and average violation:

```
3 Rank-1 1-rows pack. cuts added ,  max viol. = 0.0275816, aver. viol. = 0.0243375
88 Rank-1 3-rows pack. cuts added ,  max viol. = 0.104368, aver. viol. = 0.0337835
Building structures for heuristic rank-1 packing cut separation...done!
96 Rank-1 4-rows pack. cuts added ,  max viol. = 0.0791168, aver. viol. = 0.0348517
100 Rank-1 5-rows pack. cuts added ,  max viol. = 0.100438, aver. viol. = 0.0599266
```

7 Perspectives

BaPCod is a prototype (proof-of-concept) academic code. Bugs and numerical issues are expected. Thus, BaPCod is distributed for an academic use without

any warranty. The user support is not guaranteed. Although critical issues and bugs will be addressed by request, possibly with a delay. Corrections and extensions of this user guide will also be done by request. No major developpement of BaPCod is planned in the future. On the other side, the work on the VRPSolver extension will likely continue. We plan to extend modelling capabilities, implement additional cut separation functors, and to improve the overall efficiency of VRPSolver.

Acknowledgements

We appreciate financial support from Inria, Université de Bordeaux, Institute de Mathématiques de Bordeaux, and CNRS (Centre Nationale de la Recherche Scientifique, France).

We would like to thank many people who contributed to the design, implementation, and maintenance of BaPCod and VRPSolver extension: Issam Tahiri, Laurent Facq, Boris Detienne, Aurélien Froger, François Clautiaux, Pierre Pesneau, Artur Pessoa, Eduardo Uchoa, Halil Şen, Jinil Han, Céline Saubatte, Franck Labat, Romain Leguay, Teobaldo Bulhões, Guillaume Marquez, Eduardo Queiroga, Adeline Fonseca. The names are listed in no particular order.

References

[1] Roberto Baldacci, Nicos Christofides, and Aristide Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115:351–385, 2008.

[2] Hatem M.T. Ben Amor, Jacques Desrosiers, and Antonio Frangioni. On the choice of explicit stabilizing terms in column generation. *Discrete Applied Mathematics*, 157(6):1167 – 1184, 2009.

[3] Jacques Desrosiers and Marco E. Lübbecke. Branch-price-and-cut algorithms. In *Wiley Encyclopedia of Operations Research and Management Science*. American Cancer Society, 2011.

[4] Stephan Held, William Cook, and Edward C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4(4):363–381, 2012.

[5] Stefan Irnich, Guy Desaulniers, Jacques Desrosiers, and Ahmed Hadjar. Path-reduced costs for eliminating arcs in routing and scheduling. *INFORMS Journal on Computing*, 22(2):297–313, 2010.

[6] O Kullmann. *Handbook of Satisfiability*, chapter Fundaments of branching heuristics, pages 205–244. IOS Press, Amsterdam, 2009.

[7] G. Laporte and Y. Nobert. A branch and bound algorithm for the capacitated vehicle routing problem. *Operations-Research-Spektrum*, 5(2):77–85, Jun 1983.

[8] Jens Lysgaard, Adam N. Letchford, and Richard W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100(2):423–445, Jun 2004.

[9] Diego Pecin, Artur Pessoa, Marcus Poggi, and Eduardo Uchoa. Improved branch-cut-and-price for capacitated vehicle routing. *Mathematical Programming Computation*, 9(1):61–100, 2017.

[10] Artur Pessoa, Ruslan Sadykov, and Eduardo Uchoa. Solving bin packing problems using vrpsolver models. *Operations Research Forum*, 2(2):20, 2021.

[11] Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. Automation and combination of linear-programming based stabilization techniques in column generation. *INFORMS Journal on Computing*, 30(2):339–360, 2018.

[12] Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. A generic exact solver for vehicle routing and related problems. *Mathematical Programming*, 183:483–523, 2020.

[13] Artur Pessoa, Eduardo Uchoa, and Marcus Poggi de Aragão. A robust branch-cut-and-price algorithm for the heterogeneous fleet vehicle routing problem. *Networks*, 54(4):167–177, 2009.

[14] Eduardo Queiroga, Ruslan Sadykov, and Eduardo Uchoa. A popmusic matheuristic for the capacitated vehicle routing problem. *Computers & Operations Research*, page 105475, 2021.

[15] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, pages 269 – 280. North-Holland, Amsterdam, 1981.

[16] Ruslan Sadykov, Eduardo Uchoa, and Artur Pessoa. A bucket graph–based labeling algorithm with application to vehicle routing. *Transportation Science*, 55(1):4–28, 2021.

[17] Ruslan Sadykov, François Vanderbeck, Artur Pessoa, Issam Tahiri, and Eduardo Uchoa. Primal heuristics for branch-and-price: the assets of diving methods. *INFORMS Journal on Computing*, 31(2):251–267, 2019.

[18] François Vanderbeck. Branching in branch-and-price: a generic scheme. *Mathematical Programming*, 130(2):249–294, 2011.

[19] Paul Wentges. Weighted dantzig–wolfe decomposition for linear mixed-integer programming. *International Transactions in Operational Research*, 4(2):151–162, 1997.