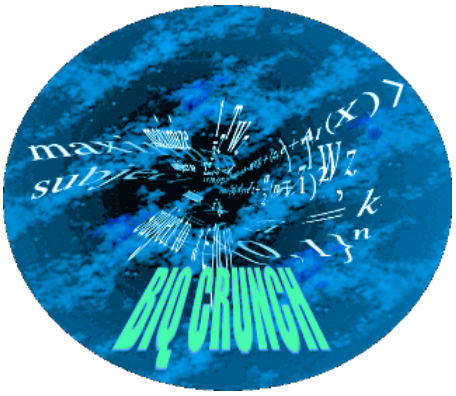
BiqCrunch 2.0 User Guide



Nathan Krislock, Jérôme Malick, Frédéric Roupin

May 2, 2016

Summary

BiqCrunch is a semidefinite-based solver for binary quadratic problems. It uses a branch-and-bound method featuring an improved semidefinite bounding procedure [8], combined with a polyhedral approach (see [5, 4] for details).

BiqCrunch is written in `C` and `Fortran` and uses the external library *L-BFGS-B* [2] for quasi-Newton bound-constrained optimization and the branch-and-bound framework *BOB* [7]. *BiqCrunch* uses specific `BC` input files, and an LP format conversion tool is provided.

People involved with the development of BiqCrunch are

- Nathan Krislock (krislock@math.niu.edu)
- Jérôme Malick (jerome.malick@inria.fr)
- Frédéric Roupin (Frederic.Roupin@lipn.univ-paris13.fr)

Project contributors:

- Marco Casazza: *BiqCrunch* web site, documentation
- Geoffrey Kozak: heuristics for the Maximum Independent Set Problem

Contents

1 BiqCrunch

BiqCrunch is released under the GNU Public License, version 3.0, as open source software available for non-commercial use. *BiqCrunch* is available at:

http://www-lipn.univ-paris13.fr/BiqCrunch/

New features of this release :

- Performance improvement: see the new numerical results on the website.
- The code has been refactored, cleaned, simplified, and formatted to improve readability and maintainability.
- The Goemans-Williamson heuristic is now the default generic heuristic.
- New parameters `soln_value_provided` and `soln_value` can be used to provide the objective value of a known feasible solution. This makes it possible to use *BiqCrunch* to prove the optimality of a solution obtained by a heuristic, for example.
- New `branchingStrategy` parameter can be used to select the branching strategy (three strategies are available).
- Local-search routine (can be disabled with parameter `local_search`) that is called whenever the heuristic finds a feasible solution.
- New `seed` parameter for seeding the random number generator.
- Improved output format for both the terminal output and the more detailed output file.
- For a given problem, user can take advantage of specific constraints to fix the values of variables when branching (a user-function must be defined).

1.1 Installation

Extract the files from the archive "`biqcrunch.tar.gz`". To compile and test *BiqCrunch* go to the `src` directory then run the following commands:

```
$ make
$ source runtests.sh
```

This will produce specific binary files for each problem subdirectory located in the `problems/` directory. This allows one to execute *BiqCrunch* with a specific heuristic for each problem. If your problem does not appear, it is always possible to use the generic version of the solver in the `problems/generic/` directory, or add your own heuristic (details are given in this documentation). Installation from the source files requires either LAPACK or the Intel MKL libraries. If MKL is available then it will be used by default.

1.2 Usage

To run *BiqCrunch* just use the specific problem version that can be found in the corresponding subdirectory in `problems/`.

```
$ ./biqcrunch [-v (0|1)] <INSTANCE> <PARAMETERS>
```

The parameter `-v` is the verbosity of *BiqCrunch* and `<INSTANCE>` is the input file in the `BC` format. If the `-v` flag is missing then *BiqCrunch* will use the non-verbose option. Be cautious: the verbose option can produce large output files for some problems. The more verbose option is mainly useful for testing different parameters values by giving additional information during the evaluation of each node of the search tree. During the solving process, some information will be displayed on screen. In particular, each time a better feasible solution is found, the node number and the new value will be provided.

At the end of the command a parameters file is required (note that several files are provided for different problems). A complete description of these parameters is given in Section 2.

1.3 Conversion tools

BiqCrunch uses a specific input file format (see the next section for a complete description). Nevertheless an LP file format conversion tool (`tools/lp2bc.py`, written in Python) is provided. Moreover, for each problem, some specific tools are also provided to convert standard instances (e.g., a graph generated by rudy) to `BC` files. To get usage information, just run the corresponding tool without parameters. All these tools are written in standard C and can be compiled in a straightforward manner.

1.4 Input format

BiqCrunch solves any problem that can be stated as

$$\begin{cases} \text{maximize} & x^T S_0 x + s_0^T x & \\ \text{subject to} & x^T S_i x + s_i^T x \leq a_i, & i \in \{1, \ldots, m_I\} \\ & x^T S_i x + s_i^T x = a_i, & i \in \{m_I + 1, \ldots, m_I + m_E\} \\ & x \in \{0,1\}^n & \end{cases} \tag{1.1}$$

where $x^T S_i x + s_i^T x$ is a quadratic function with integer coefficients, for $i = 0, \ldots, m_I + m_E$, and a is an integer vector. The problem has to be written in `BC` format which uses a sparse representation, similar to the sparse `SDPA` format. The objective function and constraint coefficients are described as $(n+1) \times (n+1)$ matrices, linear terms being stored in the last line/column: $Q_i = \begin{bmatrix} S_i & \frac{s_i}{2} \\ \frac{s_i^T}{2} & 0 \end{bmatrix}$.

Some input files for *BiqCrunch* are provided for several problems (see section Examples).

1.4.1 Example

Model

$$\begin{aligned} \text{maximize} \quad & 20x_1x_3 + 26x_1x_4 + 23x_2x_3 + 8x_2x_5 + 32x_3x_4 + 13x_4x_5 \\ \text{subject to} \quad & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\ & 12x_1x_3 + 24x_1x_4 + 14x_2x_3 + 16x_2x_5 + 28x_3x_4 + 12x_4x_5 \leq 30 \\ & x \in \{0,1\}^5 \end{aligned}$$

LP file

```
Maximize
  20 x1*x3 + 26 x1*x4 + 23 x2*x3 +
   8 x2*x5 + 32 x3*x4 + 13 x4*x5

Subject to
  x1 + x2 + x3 + x4 + x5 = 3

  12 x1*x3 + 24 x1*x4 + 14 x2*x3 +
  16 x2*x5 + 28 x3*x4 + 12 x4*x5 <= 30

Binary
  x1 x2 x3 x4 x5

End
```

BC file (generated by lp2bc.py)

```
# List of binary variables:
#    1:  x1
#    2:  x2
#    3:  x3
#    4:  x4
#    5:  x5
1 = max problem
2 = number of constraints
2 = number of blocks
6, -1
3.0 30.0
0 1 1 3 10.0
0 1 1 4 13.0
0 1 2 3 11.5
0 1 2 5 4.0
0 1 3 4 16.0
0 1 4 5 6.5
1 1 1 6 0.5
1 1 2 6 0.5
1 1 3 6 0.5
1 1 4 6 0.5
1 1 5 6 0.5
2 1 1 3 6.0
2 1 1 4 12.0
2 1 2 3 7.0
2 1 2 5 8.0
2 1 3 4 14.0
2 1 4 5 6.0
2 2 1 1 1.0
```

1.5 Output

BiqCrunch provides detailed information during the solving process and an output file is generated using the name of the input file (same directory).

Example of screen output: `./biqcrunch -v 1 example.bc biq_crunch.param`

```
Output file: example.bc.output
Input file:  example.bc
Parameter file: biq_crunch.param
```

```
Node 0 Feasible solution 43
Nodes = 3
Root node bound = 47.43
Maximum value = 43
Solution = { 1 2 3 }
CPU time = 0.0045 s
```

Example of output file: `./biqcrunch -v 1 example.bc biq_crunch.param`

```
*****************************************************************************
*                          BIQ CRUNCH 2.0 Solver                            *
*****************************************************************************
| Copyright(C) 2010-2016 N. Krislock, J. Malick, F. Roupin                  |
| BIQ CRUNCH uses L-BFGS-B by C. Zhu, R. Byrd, J. Nocedal and BOB 1.0 by PNN |
| Team of PRiSM Laboratory.                                                 |
|                                                                           |
| L-BFGS-B is distributed under the terms of the New BSD License. See the   |
| website http://users.eecs.northwestern.edu/~nocedal/lbfgsb.html for more  |
| information. BOB is free software. For more information visit the website |
| http://www.prism.uvsq.fr/~blec/index.php?p=./rech/so&lang=fr              |
*****************************************************************************
Input file: example.bc
Solving as a MAXIMIZATION problem
Problem Size = 5 Number of equalities = 7 Number of inequalities = 1
Using Generic heuristic
BiqCrunch Parameters:
              alpha0 = 0.100000
          scaleAlpha = 0.500000
            minAlpha = 0.000050
                tol0 = 0.100000
            scaleTol = 0.950000
              minTol = 0.010000
            withCuts = 1
             gapCuts = -0.050000
                cuts = 500
             minCuts = 50
            nitermax = 2000
            minNiter = 12
            maxNiter = 100
             scaling = 1
                root = 0
              heur_1 = 1
              heur_2 = 1
              heur_3 = 1
 soln_value_provided = 0
          soln_value = 0
          time_limit = 0
   branchingStrategy = 1
                seed = 2016
               NBGW1 = 1
               NBGW2 = 10
        local_search = 1
Heuristic 1:  Beta updated => 43

*****************************************************************************
                                    Node 1
*****************************************************************************
Problem size 5
Fixed variables:
=============================================================================
Iter    Time    gap  alpha    tol  nbit  Enorm  Inorm  ynorm  minCut  NCut  NSub  NAdd
=============================================================================
   1     0.0    4.6  1e-01  1e-01    24  5e-02  0e+00  5e+01  -3e-01    13  -0    +13
   2     0.0    4.5  5e-02  1e-01     5  5e-02  9e-02  5e+01  -1e-01    31  -0    +18
   3     0.0    4.4  3e-02  9e-02     7  4e-02  3e-02  5e+01  -7e-02    41  -0    +10
   4     0.0    4.4  1e-02  9e-02     8  5e-02  3e-02  5e+01  -3e-02    41  -0    +0
   5     0.0    4.4  6e-03  8e-02     4  6e-02  7e-02  5e+01  -2e-01    60  -0    +19
   6     0.0    4.4  3e-03  8e-02     6  2e-02  0e+00  5e+01   3e-03    60  -0    +0
   7     0.0    4.4  2e-03  7e-02     5  2e-02  5e-02  5e+01  -5e-02    60  -0    +0
   8     0.0    4.4  8e-04  7e-02     6  2e-02  6e-02  5e+01  -6e-02    60  -0    +0
   9     0.0    4.4  4e-04  7e-02     6  2e-02  6e-02  5e+01  -6e-02    60  -0    +0
  10     0.0    4.4  2e-04  6e-02     6  2e-02  6e-02  5e+01  -6e-02    60  -0    +0
  11     0.0    4.4  1e-04  6e-02     7  2e-02  6e-02  5e+01  -6e-02    60  -0    +0
  12     0.0    4.4  5e-05  6e-02    13  4e-02  0e+00  5e+01     nan    60  -0    +0
=============================================================================
Giving up! Bound = 47.43, Iter = 12, alpha = 5e-05, tol = 6e-02, cuts = 60, time = 0.0
=============================================================================
Depth = 0, Bound = 47, Best = 43

Branching on x[3] = 0.04
Fixing x[3] = 0

*****************************************************************************
                                    Node 2
*****************************************************************************
Problem size 4
Fixed variables: (x[3],0)
=============================================================================
Iter    Time    gap  alpha    tol  nbit  Enorm  Inorm  ynorm  minCut  NCut  NSub  NAdd
=============================================================================
   1     0.0    2.2  1e-01  1e-01    12  5e-02  0e+00  4e+01  -1e-01     6  -0    +6
   2     0.0    0.8  5e-02  1e-01    22  1e-01  0e+00  5e+01     nan     6  -0    +0
=============================================================================
```

```
Prune! Bound = 43.78, Iter = 2, alpha = 5e-02, tol = 1e-01, cuts = 6, time = 0.0
================================================================================
Depth = 1, Bound = 43, Best = 43

Fixing x[3] = 1

********************************************************************************
                                    Node 3
********************************************************************************
Problem size 4
Fixed variables: (x[3],1)
================================================================================
Iter    Time    gap  alpha    tol  nbit  Enorm  Inorm  ynorm  minCut  NCut  NSub  NAdd
================================================================================
   1     0.0    0.9  1e-01  1e-01    24  3e-01  0e+00  6e+01     nan     0  -0    +0
================================================================================
Prune! Bound = 43.90, Iter = 1, alpha = 1e-01, tol = 1e-01, cuts = 0, time = 0.0
================================================================================
Depth = 1, Bound = 43, Best = 43

Nodes = 3
Root node bound = 47.43
Maximum value = 43
Solution = { 1 2 3 }
CPU time = 0.0045 s
```

BiqCrunch can be also used as a simple semidefinite solver to get a bound (further details are given about parameters in Section 2). Hereafter, the same small example is run with the parameter file `bound.param` (provided in the `biqcrunch/` folder). In the first test, all the heuristics are disabled and so no lower bound (given by a feasible solution) is available. In the second run, the heuristics are enabled (see Section 2.1).

Example of screen file: `./biqcrunch -v 1 example.bc bound.param`

```
Output file: example.bc.output_1
Input file:  example.bc
Parameter file: bound.param
Nodes = 1
Root node bound = 47.32536
Best value = inf
Gap = 100.00%
CPU time = 0.0037 s

Output file: example.bc.output_2
Input file:  example.bc
Parameter file: bound.param
Node 0 Feasible solution 43
Nodes = 1
Root node bound = 47.32536
Best value = 43
Gap = 10.06%
CPU time = 0.0038 s
```

2 Advanced usage

BiqCrunch will not modify your model before solving it, so be cautious when modeling your problem. The semidefinite bound and the efficiency of the solver could be affected. In particular, if one wants to get a stronger underlying semidefinite relaxation, some extra constraints (redundant in the initial 0-1 model) should be added in the model. A small example illustrating this point is provided in `/problems/generic/examples/`.

2.1 BiqCrunch Parameters

In most cases, *BiqCrunch* will be efficient using the default parameters. Nevertheless, to improve the performance of *BiqCrunch* for some problems it is sometimes a good idea to adjust the values of some of the key parameters. These values (especially for the bounding parameters) can have a huge impact on the efficiency of the solver. For more details the reader is referred to [4, 8].

General parameters

root: 1 to stop the algorithm after the evaluation of the root node (default=0). Thus *BiqCrunch* can be used as a simple solver to test a relaxation (computational time and gap). This option is also useful when adjusting parameters by inspecting several output files for the root node (verbosity command line option `-v 1` should be used in that case);

time_limit: maximum running time in seconds. Set to 0 for no time limit (default=0). If the solver stops before solving the problem exactly then the final gap (between the worst bound in the search tree and the value of current best feasible solution found) will be provided;

heur_1: enable (1) or disable (0) the heuristic called at the beginning of the execution (default=1);

heur_2: enable (1) or disable (0) the heuristic called after each call of `L-BFGS-B` during the computation of the bound (default=1);

heur_3: enable (1) or disable (0) the heuristic called after the evaluation of a node (default=1);

seed: random number generator seed. In particular, this seed is used for the Goemans-Williamson heuristic (default heuristic) (default=2016);

local_search: enable (1) or disable (0) a simple "1-opt" local-search. This local-search routine returns a solution that is locally optimal starting from a feasible solution provided by each heuristic;

branchingStrategy: (0) Branch on least-fractional variable, (1) Branch on most-fractional variable, (2) Branch on variable that is closest to one (default=1);

soln_value_provided: user is providing a known feasible solution value (default=0);

soln_value: the value of a known feasible solution (default=none).

Bounding parameters

alpha0: starting value of alpha (default=1e-1);

scaleAlpha: scaling value of alpha (default=0.5). This parameter controls the rate at which alpha decreases;

minAlpha: minimum value of alpha (default=5e-5);

tol0: starting value of tolerance (default=1e-1);

scaleTol: scaling value of tolerance (default=0.95);

minTol: minimum value of the tolerance (default=1e-2);

gapCuts: minimum violation value to add a cut (default=-5e-2);

withCuts: 1 to add triangle inequalities during the computation of the bound, 0 to compute the bound without the triangle inequalities (default=1);

cuts: maximum number of inequalities to add at each iteration (default=500);

minCuts: alpha and tolerance will be reduced when the number of added triangle inequalities is below this value (default=50);

nitermax: maximum number of iterations of the `L-BFGS-B` solver (default=2000);

minNiter: minimum number of `L-BFGS-B` calls (default=12);

maxNiter: maximum number of `L-BFGS-B` calls (default=100);

scaling: 1 to scale the constraints (default=1).

2.2 Instance syntax

A `BC` instance begins with some optional lines of comments, which are strings preceded by a semicolon or by a asterisk:

```
<COMMENT> ::= ; <STRING> | * <STRING>
```

The first line gives the problem type : -1 for minimization, 1 for maximization. This can be followed by other characters ignored by BiqCrunch.

```
<#MIN/MAX> ::=  <-1> | <1> <STRING>
```

The next line defines the number of constraints, which is a positive integer such that `<INT>`$= m_I + m_E$.

```
<#CONSTRAINTS> ::= <INT> | <INT> <STRING>
```

Similar to the SDPA format, we define the number of blocks of the matrices of the input file. As seen before, this line also admits characters after the definition.

```
<#BLOCKS> ::= <INT> | <INT> <STRING>
```

In the BC format an instance can have 1 or 2 blocks depending on the model: if the model contains no inequality constraints, <INT> must be equal to 1; if the model has inequality constraints <INT> must be equal to 2.

The third entry of the instance describes the size of the blocks of the matrices.

```
<SIZE> ::= <INT_1>           | {<INT_1>}           |
           <INT_1>, -<INT_2> | {<INT_1>, -<INT_2>}
```

If the problem has no inequalities, then the size of the first block of the matrices (equal to $n+1$) is provided (<INT_1>). If the problem contains inequalities then the size of the second block must also be given (<INT_2>). It always starts with a *minus* before <INT_2> to indicate that the values of the blocks are only on the diagonal of the matrix, and it is equal to m_I. In the next line, the right-hand side values of the constraints are given as a sequence of values.

```
<RIGHT-HAND_SIDE> ::= <REAL_k> | <REAL_k> <RIGHT-HAND_SIDE>
```

The number of values must be equal to $m_I + m_E$ and <REAL_k> must be the right-hand side value of constraint k.

Finally, all the matrices that describe the objective function and the left-hand side of the constraints must be provided. The first matrix corresponds to S_0, the objective function. Each line represents to a non-zero element of the matrix.

```
<OBJ_MATRIX_EL> ::= 0 1 <INT_1> <INT_2> <REAL>
```

where the first (0) and second number (1) respectively mean that this line concerns the objective function matrix and the first block. <INT_1> and <INT_2> are the row and the column of the non-zero element of the matrix and <REAL> is its value. <INT_1> and <INT_2> must be greater than 0 and less or equal to $n+1$. Similarly, for each constraint k the non-zero coefficients of the matrix S_k is given in sparse format:

```
<CONS_MATRIX_EL> ::= <INDEX_k> 1 <INT_1> <INT_2> <REAL>
```

where <INDEX_k> must be equal to k.

In the case constraint j is an inequality, one has to provide the value of the second block of the matrix:

```
<INEQ_MATRIX_EL> ::= <INDEX_j> 2 <INT> <INT> <REAL>
```

where <INDEX_j> must be equal to j, <INT> $\in \{1, \ldots, m_I\}$ is the counter for the inequality, and <REAL> is either 1.0 for a $\leq$ inequality or -1.0 for a $\geq$ inequality.

2.3 Heuristics

BiqCrunch comes with specific heuristics for:

- *generic problems*;
- *k-cluster problem*;
- *max-cut problem*;
- *max-independent-set problem*.

One can also add their own heuristics. There are several folders `problems/<PROBLEM>` that refer to different optimization problems and a `problems/user` directory where it is possible to write a new heuristic. One can add new heuristics for *BiqCrunch* by simply creating more `problems/<PROBLEM>` folders. To add a heuristic for *BiqCrunch* put a new `heur.c` file inside the corresponding problem directory. An example of `heur.c` is provided in the `problems/user` directory.

2.3.1 Generic heuristics

BiqCrunch offers generic heuristics which are useful when the user prefers not writing their own specific heuristic. These generic heuristics can be used for any binary quadratic problem and can be found in `src/rounding.c` (as well as the simple one-opt local search algorithm). During the computation of the bound of the node (heur_2), and after the evaluation of each node (heur_3), *BiqCrunch* uses the celebrated Goemans-Williamson heuristic.

By adding the corresponding calls in `heur.c` file, the user can also use a variant of the classical randomized rounding heuristic [9, 10] that rounds to 0 or 1 the variables according to the probability provided by the fractional SDP solution. Indeed one has $0 \leq x_i \leq 1$ for any feasible solution of the SDP relaxation (see [8] for details about the relaxations used). The rounding is done by comparing each x_i to a fixed $\alpha = x_j$, for $j = 1, \ldots, n$. Then *BiqCrunch* tests if the resulting 0-1 vector is feasible for the combinatorial problem, and updates the best current feasible solution if a better feasible solution is found. Afterwards, *BiqCrunch* generates an additional 100 random binary vectors by comparing each fractional x_i to a different random value γ, and again updates the best current feasible solution if a better feasible solution is found.

At the root node we generate a random vector of values in the interval $[0, 1]$ and then we apply the variant of randomized algorithm described before. To call or create new heuristics for a specific problem, one has to modify the `heur.c` source file in the corresponding subdirectory in `problems` (e.g., `problems/max-cut/heur.c`).

2.3.2 Heuristic timing

The heuristic function is called during the execution of the branch-and-bound algorithm:

1. at the beginning of the algorithm;
2. during the computation of the bound of each node of the branch-and-bound tree;
3. after the evaluation of each node of the branch-and-bound tree.

This information is saved in the function parameter `heuristic_code`, which can take three different values:

PRIMAL_HEUR: if the function is called at the beginning of the algorithm;

SDP_HEUR: if the function is called during the evaluation of the bound;

ROUNDING_HEUR: if the function is called after the evaluation of a node.

A simple use of this information is shown in the Code 2.1 taken from `heur.c`.

```
double BC_runHeuristic(Problem *P0, Problem *P, BobNode *node, int *x,
                       int heuristic_code)
{
    double heur_val;

    switch (heuristic_code) {
        case PRIMAL_HEUR:
            heur_val = primalHeuristic(P0, x);
            break;
        case SDP_BOUND_HEUR:
            heur_val = sdpBoundHeuristic(P0, node, x);
            break;
        case ROUNDING_HEUR:
            heur_val = roundingHeuristic(P0, node, x);
            break;
        default:
            printf("Choosen heuristic doesn't exist\n");
            exit(1);
    }

    // If x is feasible, perform a local search around x for a better solution
    if (params.local_search && BC_isFeasibleSolution(x)) {
        local_search(node, x, &heur_val, P0);
    }

    return heur_val;
}
```

Code 2.1: Use of the `heuristic_code` information

2.3.3 Additional functions

In addition to the heuristic function, the user must also define `BC_allocHeuristic(...)` and `BC_freeHeuristic(...)` to allocate and free the global dynamic structure. These functions are called by the solver at the beginning and at the end of the execution.

BiqCrunch also provides three useful functions for testing the solution produced with the heuristic:

- **int** `BC_isFeasibleSolution`(**int** `*sol`) which allows the user to test if the solution in the binary vector `sol` is feasible;
- **double** `BC_evaluateSolution`(**int** `*sol`) which returns the value of the objective function computed with the solution `sol`, a binary vector of size `problemSize`.
- **int** `update_best`(**int** `*xbest`, **int** `*xnew`, **double** `*best`, **Problem** `*P0`) which allows the user to check if the new solution `xnew` is feasible for problem `P0`, and to update, if needed, `xbest` and `best`. This function is useful when several heuristics are sequentially used (e.g., `problems/k-cluster/heur.c`).

For each problem, a `BC_FixVariables` function can be defined by the user to take advantage of particular constraints (see the Max-Independent-set example in Section 3). This function must be defined in the corresponding `problems/<PROBLEM>/heur.c` file.

- **void** `BC_FixVariables`(**BobNode** `*node`, **int** `ic`, **int** `xic`). User can set the value of other variables as soon as x[ic]=xic (i.e., 0 or 1) assuming that constraints in problem imply it. The `node` structure contains the full information to set the values: `node->xfixed` is an indicator vector for the fixed variables, `node->sol.X` contains the values of the fixed variables. The parameters `ic` and `xic` are such that `node->xfixed[ic]` = 1 and `node->sol.X[ic]` = `xic`.

2.3.4 Data structures

In this section we describe the data structures used in *BiqCrunch* heuristics, consisting of:

- the `Problem` and `Inequality` structures that contain all the information about the problem instance (see Code 2.2);
- the `Sparse` structure that represents a matrix in sparse format (see Code 2.3);
- the `BobNode` structure which contains the information about a node of the branch and bound tree (see Code 2.5);
- the `BobSolution` structure which contains a binary solution vector (see Code 2.4).

These structures can be modified in order to include additional information for a specific problem (or to increase the maximum problem size).

```
typedef struct Problem {
   double *Q;  // Objective matrix in DENSE format
   Sparse Qs; // Objective matrix in SPARSE format
   int n; // size of Q
   Sparse *As; // list of sparse matrices for the inequality constraints
   double *a; // right-hand-side vector of inequality constraints
   int mA; // number of inequality constraints
   Sparse *Bs; // list of sparse matrices for the equality constraints
   double *b; // right-hand-side vector of equality constraints
   int mB; // number of equality constraints
   int max_problem; // 1 if it is a max problem, and 0 if it is a min problem
} Problem;

typedef struct Inequality {
   int i;
   int j;
   int k;
   int type;
   double value;
   double y;
} Inequality;
```

Code 2.2: Problem data structure

```
typedef struct Sparse {
   int *i;
   int *j;
   double *val;
   int nnz;
} Sparse;
```

Code 2.3: Data structure of a sparse matrix

```
/*
 * Maximum number of variables
 */
#define NMAX 1024

/*
 * Solution of the problem.
 * This structure defines the content of a solution of the problem.
 */
typedef struct BobSolution {
   /*
    * Vector X.
    * Binary vector that stores the solution of the branch-and-bound
    * algorithm
    */
   int X[NMAX];
} BobSolution;
```

Code 2.4: Data structure of a solution

```
typedef struct BobNode {
   /*
    * Node information.
    *
    */
   BobTNdInfo BobNdInfo; // (int Size, int Off)
   /*
    * Number of fixed variables.
    * Integer variable that stores the number of fixed variables in the
    * current node.
    */
   int level;
   int xfixed[NMAX];
   BobSolution sol;
   double fracsol[NMAX];
   BobTPri Pri; // (int Eval, int Depth)
} BobNode;
```

Code 2.5: Branch-and-bound tree node data structure

3 Examples

3.1 Max-Cut problem

Given a graph $G = (V, E)$ with edge weights w_{ij} for $ij \in E$ and $w_{ij} = 0$ for $ij \notin E$, *Max-Cut* is the problem of finding a bipartition of the nodes V such that the sum of the weights of the edges across the bipartition is maximized. Let $n = |V|$ be the cardinality of V; we can state *Max-Cut* as

$$\begin{aligned} \text{maximize} \quad & \sum_{i<j} w_{ij} \left(\frac{1 - x_i x_j}{2} \right) \\ \text{subject to} \quad & x \in \{-1, 1\}^n. \end{aligned}$$

We can rewrite the problem of *Max-Cut* as

$$\begin{aligned} \text{maximize} \quad & \frac{1}{4} x^T Q x \\ \text{subject to} \quad & x \in \{0, 1\}^n. \end{aligned}$$

where Q is the Laplacian matrix of the weighted graph G.

3.1.1 Max-Cut heuristic

With *BiqCrunch* we provide the Goemans-Williamson random hyperplane algorithm [3]. The heuristic is run after each node evaluation to get a feasible solution.

3.1.2 Conversion tools

To simplify the creation of the `BC` instances we provide some tools to convert standard instances in sparse format to the `BC` format. All the tools can be downloaded from the BiqCrunch download page.

From Biq to BiqCrunch

To convert binary quadratic problems (e.g., [1]) to a standard `BC` instance we provide the `qp2bc` conversion tool. This tool converts an instance in a standard sparse format to a valid instance for *BiqCrunch*. This tool is written in Python and can be used directly from command line:

```
$ ./qp2bc.py <BIQ_INSTANCE> > <BC_INSTANCE>
```

From Mac to BiqCrunch

To convert *Max-Cut* problems to a standard `BC` instance we provide the `mc2bc` conversion tool. This tool converts an instance in a standard sparse format to a valid instance for *BiqCrunch*. This tool is written in Python and can be used directly from command line:

```
$ ./mc2bc.py <MC_INSTANCE> > <BC_INSTANCE>
```

3.2 k-cluster problem

Given a graph $G = (V, E)$ the k-cluster problem consists of determining a subset $S \subseteq V$ of k vertices such that the sum of the weights of the edges between vertices in S is maximized.

Letting $n = |V|$ denote the number of vertices, and w_{ij} denote the edge weight for $ij \in E$ and $w_{ij} = 0$ for $ij \notin E$, the problem can be modeled as the following 0-1 quadratic problem:

$$\begin{array}{ll} \text{maximize} & \frac{1}{2} x^T W x \\ \text{subject to} & \sum_i x_i = k, \quad x \in \{0,1\}^n. \end{array}$$

where $W = (w_{ij})_{ij}$ is the weighted adjacency matrix of the graph G.

3.2.1 k-cluster heuristic

We use two types of heuristics to find a cluster with exactly k nodes. First, for the initial feasible point (before running the Branch-and-Bound), we use the classical greedy heuristic, since it gives very good feasible solutions: we remove vertices one at a time from the graph by choosing the vertex with the smallest degree (or sum of the weights over the adjacent vertices) at each step. Second, during the evaluation of the bound and after running the bounding procedure on a subproblem having k' nodes added to the cluster, we add the remaining $k - k'$ nodes having the largest fractional values x_i in the SDP solution. More details can be found in [5]. Finally, to improve the solution we use a two-opt algorithm: swap two vertices (one in the k-cluster with one outside) until no progress is made. Finally, we also call the generic heuristic of *BiqCrunch* and keep the best solution. For further details see the `problems/k-cluster/heur.c` file.

3.2.2 k-cluster instances and conversion

To obtain *BiqCrunch* input files for the k-cluster problem you can simply use the conversion tool `kc2bc` which can convert instances from a standard sparse format and from the format used in [6] to `BC`. When using the conversion tool, weights can be ignored with a simple flag (thus the graph will be considered unweighted). Note that the conversion tool also adds redundant constraints to the instance to improve the bound obtained during the bounding procedure [11].

3.3 Maximum independent set problem

Consider an undirected graph $G = (V, E)$, where $V = \{1, ..., n\}$ and $|E| = m$. A weight $w_i \in \mathbb{R}$ is assigned to each vertex $i \in V$.

An independent set is a set $S \subseteq V$ such that no two vertices in S are joined by an edge in E. We seek an independent set of maximum total weight in G.

$$\begin{array}{lll} \text{maximize} & \sum_{i=1}^n w_i x_i & \\ \text{subject to} & x_i x_j = 0, & \forall ij \in E, \\ & x \in \{0,1\}^n. & \end{array}$$

For this problem, we take advantage of the constraints by fixing several nodes at the same time in *BiqCrunch* when a decision variable is set to 1. Indeed, if (ic, j) is an edge of G and $x_{ic} = 1$ then $x_j = 0$. The function `BC_FixVariable` (defined in `problems/max-indep-set/heur.c`) is used to fix additional variables whenever one variable

is fixed. Note that in `problems/generic/heur.c` the body of this function is empty. Of course, for other problems with particular constraints, one may modify this function (as is done for the maximum independent set problem) to decrease the size of the subproblems.

```
void BC_FixVariables(BobNode *node, int ic, int xic) {
int j;

   if (xic == 1) {
      for (j = 0; j < BobPbSize; j++) {
         if (j != ic)
            if (getSparseAdjacencyMatrixValue(M_adj, ic, j) == 1.) {
            // (ic,j) belongs to E so X[ic] = 1 => X[j] = 0
                if(!node->xfixed[j]) {
                node->xfixed[j] = 1;
                node->sol.X[j] = 0;
                }
            }
      }
   }
}
```

Code 3.1: Use of the `BC_FixVariable` for the MIS problem

3.3.1 Maximum independent set heuristics

In addition to the Goemans-Williamson heuristic, two standard heuristics are provided in the corresponding `heur.c` file. First, in order to obtain an initial solution before we start the Branch-and-Bound method, we go through the vertices of G from lowest degree to highest degree and add a vertex to the independent set if it has no neighbours in the set. Second, during the evaluation of the bound and after running the bounding procedure, follow the same method as the previous heuristic by trying to add each vertex to the independent set given a different order. This time, the order is not actually given by the increasing degree of the vertices but by the fractional solution provided by *BiqCrunch* (sort the vertices by decreasing fractional value).

Bibliography

[1] Alain Billionnet and Sourour Elloumi. Using a mixed integer quadratic programming solver for the unconstrained quadratic 0-1 problem. *Mathematical Programming*, 109:55–68, 2007. 10.1007/s10107-005-0637-9.

[2] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995.

[3] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995.

[4] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. Improved semidefinite bounding procedure for solving max-cut problems to optimality. *Mathematical Programming*, 143(1-2):61–86, 2014.

[5] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. Computational results of a semidefinite branch-and-bound algorithm for k-cluster. *Computers and Operations Research*, 66:153 – 159, 2016.

[6] Amélie Lambert. A library of k-cluster problems. http://cedric.cnam.fr/~lamberta/Library/k-cluster.html. CNAM-CEDRIC.

[7] Bertrand Le Cun, Catherine Roucairol, and The Pnn Team. Bob: a unified platform for implementing branch-and-bound like algorithms. Technical report, Laboratoire Prism, 1995.

[8] Jérôme Malick and Frédéric Roupin. On the bridge between combinatorial optimization and nonlinear optimization: a family of semidefinite bounds for 0-1 quadratic problems leading to quasi-Newton methods. *Mathematical Programming*, 140(1):99–124, 2013.

[9] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37(2):130–143, October 1988.

[10] Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, December 1987.

[11] Frédéric Roupin. From linear to semidefinite programming: An algorithm to obtain semidefinite relaxations for bivalent quadratic problems. *Journal of Combinatorial Optimization*, 8:469–493, 2004. 10.1007/s10878-004-4838-6.