BiqCrunch: a semidefinite branch-and-bound method for solving binary quadratic problems

Nathan Krislock, Jérôme Malick, Frédéric Roupin

BiqCrunch: a semidefinite branch-and-bound method for solving binary quadratic problems

NATHAN KRISLOCK, Northern Illinois University
JEROME MALICK, CNRS, Lab. J. Kunztmann
FREDERIC ROUPIN, LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité

This paper presents *BiqCrunch*, an exact solver for binary quadratic optimization problems. *BiqCrunch* is a branch-and-bound method that uses an original, efficient semidefinite-optimization-based bounding procedure. It has been successfully tested on a variety of well-known combinatorial optimization problems, such as Max-Cut, Max-k-Cluster, and Max-Independent-Set. The code is publicly available online; a web interface and many conversion tools are also provided.

CCS Concepts: •**Mathematics of computing** → **Solvers; Combinatorial optimization; Semidefinite programming; Quadratic programming; Integer programming;** •**Theory of computation** → **Branch-and-bound;**

Additional Key Words and Phrases: Binary quadratic programming, exact resolution, NP-hard, semidefinite relaxations, quasi-Newton

ACM Reference Format:
Nathan Krislock, Jérôme Malick, and Frédéric Roupin, 2016. BiqCrunch: a semidefinite branch-and-bound method for solving binary quadratic problems. *ACM Trans. Math. Softw.* 0, 0, Article 0 (2016), 23 pages.
DOI: 0000001.0000001

1. INTRODUCTION: BINARY QUADRATIC PROBLEMS AND SOLVERS

1.1. Binary quadratic optimization problems

We consider binary quadratic optimization problems, i.e., (nonconvex) optimization problems with a quadratic objective, quadratic constraints, and 0–1 variables. A binary quadratic problem with m_I inequality constraints and m_E equality constraints has the following mathematical formulation:

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

where the S_i's are real symmetric $n \times n$ matrices (possibly $S_i = 0$), the s_i's are vectors in $\mathbb{R}^n$, and the a_i's are real numbers. Many optimization problems in the sciences, operations research, or engineering are expressed as binary quadratric problems, such as, in medicine [Iasemidis et al. 2001], in physics [Liers et al. 2005], in space allocation

This work is partly supported by the French ANR Project "GeoLMI" and a Northern Illinois University Research & Artistry Award.
Authors' addresses: Nathan Krislock, Department of Mathematical Sciences, Northern Illinois University, DeKalb, IL, USA, nkrislock@niu.edu; Jérôme Malick, CNRS, Lab. J. Kunztmann, Grenoble, France, jerome.malick@univ-grenoble-alpes.fr; Frédéric Roupin, LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité, France, frederic.roupin@lipn.univ-paris13.fr.

DOI: 0000001.0000001

[Anjos and Vannelli 2008], in computer vision [Joulin et al. 2010], or in computational biology [Engau et al. 2012].

Three examples of classical combinatorial optimization problems that can be expressed as problem (1) are Max-Cut, Max-k-Cluster, and Max-Independent-Set. In the Max-Cut problem (see, e.g., [Goemans and Williamson 1995; Rendl et al. 2010]), we are given an edge-weighted graph with n vertices, and the objective is to maximize the total weight of the edges between a subset of vertices and its complement; this problem can be stated as:

$$(\text{Max-Cut}) \quad \begin{array}{ll} \text{maximize} & \sum_{ij} w_{ij} z_i (1 - z_j) \\ \text{subject to} & z \in \{0,1\}^n. \end{array} \tag{2}$$

In the Max-k-Cluster problem, we are given an edge-weighted graph with n vertices and a natural number k, and the objective is to find a subgraph of k nodes having maximum total edge weight; this problem can be stated as:

$$(\text{Max-}k\text{-Cluster}) \quad \begin{array}{ll} \text{maximize} & \frac{1}{2} \sum_{ij} w_{ij} z_i z_j \\ \text{subject to} & \sum_{i=1}^{n} z_i = k \\ & z \in \{0,1\}^n. \end{array} \tag{3}$$

In the Max-Independent-Set (MIS) problem (see, e.g., [Zhao et al. 1998]), we are given a graph $G = (V, E)$ with vertex weights w_i, and the objective is to maximize the total weight of the vertices in an independent set (a set S of vertices having no two vertices joined by an edge in E); this problem can be stated as:

$$(\text{MIS}) \quad \begin{array}{lll} \text{maximize} & \sum_i w_i z_i & \\ \text{subject to} & z_i z_j = 0, & \forall (i,j) \in E \\ & z \in \{0,1\}^n. & \end{array} \tag{4}$$

These three problems, and more generally binary quadratic problems, are NP-hard and are often difficult to solve in practice.

This article introduces *BiqCrunch*, an exact solver for general binary quadratic (*biq*) optimization problems. Extensive numerical experiments show that *BiqCrunch* is the current state-of-the-art for several difficult binary quadratic optimization problems. The source code is available online and distributed under the GNU General Public License, version 3.

The remainder of the introduction sketches the existing solvers and the contributions of *BiqCrunch*. The mathematical foundations of *BiqCrunch* are presented in Section 3, its algorithmic description in Section 4, and finally advanced techniques for improving its performance in Section 5. Further information is available on the *BiqCrunch* website:

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

1.2. Existing solvers for binary quadratic optimization

Binary quadratic programming is included in the broader class of mixed-integer nonlinear programming [Bussieck et al. 2010; D'Ambrosio and Lodi 2011; Burer and Letchford 2012]. Thus problem (1) could be handled directly by using mixed-integer nonlinear programming solvers, such as the commercial solvers BARON [Sahinidis 2013], LocalSolver, Gurobi, and IBM/CPLEX, as well as the noncommercial solvers SCIP [Achterberg 2009] and Bonmin [Bonami et al. 2008]. However these mixed-integer nonlinear programming solvers do not fully exploit the quadratic form of the objective function and the constraints in problem (1), except in preprocessing phases.

In contrast, another widely used technique for solving binary quadratic problems is to add linearization variables to formulate problem (1) as a binary linear programming problem; see, e.g., [Sherali and Adams 1990]. The advantage of this approach is

the possibility of using all the available efficient tools for integer linear programming. However, for hard combinatorial problems, it is often necessary to go beyond standard linear bounds and work with tighter bounds. For example, for graph problems that are very sparse, linear-based solvers that take advantage of the sparsity and the geometric properties of underlying problems usually perform well; however for small dense problems, they can perform poorly.

The quadratic nature of the objective function and the constraints of problem (1) implies that we can use semidefinite relaxations of problem (1) to get tight bounds (see, e.g., [Shor 1987; Goemans and Williamson 1995; Poljak et al. 1995; Lemaréchal and Oustry 1999]). Currently, there are three types of semidefinite-based solvers for binary quadratic problems. The first type is semidefinite branch-and-bound methods specialized for solving specific subclasses of problems (1), such as the semidefinite solver of [Armbruster et al. 2012] for graph bisection problems, and the *Biq Mac* solver of [Rendl et al. 2010] for the Max-Cut problem (2). The second type of semidefinite-based solvers is the quadratic convex reformulation for mixed-integer quadratic problems [Billionnet and Elloumi 2007; Billionnet et al. 2009; Galli and Letchford 2014] which uses semidefinite bounds at the root node to give a boost to linear programming based branch-and-bound methods.

The third type of semidefinite-based solvers are standard branch-and-bound methods replacing linear programming solvers with semidefinite programming solvers, such as SCIP-SDP [Mars 2013; Gally et al. 2016]. SCIP-SDP solves general mixed-integer semidefinite programming (MISDP) problems, which implies that it is able to solve generic binary quadratic problems after making a suitable transformation of the problem to an MISDP. SCIP-SDP uses a standard branch-and-bound approach where bounds are obtained by solving the SDP relaxation that is obtained by simply relaxing the integer constraints—this SDP relaxation is then solved by a standard SDP solver, such as an interior-point method. SCIP-SDP must use several safe-guards against failures to solve the SDP relaxation due to the loss of strict feasibility that can occur when branching, and is limited to solving only small to medium sized problems.

1.3. BiqCrunch, a free solver for binary quadratic problems

In this article, we introduce *BiqCrunch*, an open-source code for solving binary quadratic optimization problems to optimality. *BiqCrunch* is a branch-and-bound algorithm using generic or specific heuristics to compute lower-bounds and an original adaptive bounding procedure to compute upper-bounds. The bounding procedure automatically adjusts several parameters to efficiently produce a wide range of tightness levels from rough bounds to tight semidefinite-quality bounds.

BiqCrunch is of particular interest for solving hard problems which are very difficult to solve using linear-bounds. *BiqCrunch* therefore complements the currently available software packages mentioned in the previous section. Generally speaking, the set of problems for which linear-bounds underperform are the problems best-suited for the *BiqCrunch* solver. Compared to other semidefinite-based solvers, *BiqCrunch* offers a flexible and efficient bounding procedure that can produce a range of bounds with a varying degree of tightness.

The *BiqCrunch* solver is available as:

— an open-source code for solving problem (1);
— specific versions of the software for different standard combinatorial problems;
— a simple online interface.

The *BiqCrunch* solver is written in C (and uses a Fortran library). The distribution also includes converters and heuristics written in C and Python. The code is developed using established numerical tools, namely: basic linear algebra functions in LAPACK

[Anderson et al. 1999] or the Intel Math Kernel Library (MKL), the nonlinear optimization routine L-BFGS-B [Zhu et al. 1997; Morales and Nocedal 2011], and the branch-and-bound platform BOB [Le Cun et al. 1995].

We have conducted extensive computational tests on classical NP-hard combinatorial problems, known to be difficult to solve even for many medium-sized instances. Results on Max-Cut and Max-k-Cluster are available in [Krislock et al. 2014] and [Krislock et al. 2016], respectively, and the *BiqCrunch* website reports the latest results available on other problems. These computational results provide strong evidence that *BiqCrunch* is among the best solvers for solving to optimality combinatorial optimization problems that can be formulated using quadratic terms. For example, for Max-Cut, *BiqCrunch* (or more precisely its precursor code) has been compared to the state-of-the-art *Biq Mac* solver, and has been shown to have a more efficient bounding procedure (in that it attains tighter upper bounds in much less time; see [Krislock et al. 2014, Fig.1]) and to be more robust to solve problems exactly (see [Krislock et al. 2014, Fig.2]).

1.4. Outline of the paper

The goal of this paper is to accompany the public release of the *BiqCrunch* code by providing a complete description of the solver and how to use it.

We first present some basic information and examples on how to use *BiqCrunch* in Section 2; a complete description is available in the user manual that is distributed with *BiqCrunch*. The mathematical foundations of *BiqCrunch* are presented in Section 3 where we will recall the standard strengthened semidefinite bounds for problem (1) and, motivated by the desire to have semidefinite quality bounds without the inherent computational cost of the standard bounds, we will describe the original semidefinite bounds that are used in *BiqCrunch*. The two main algorithmic ingredients are then described in Section 4: the generic heuristic for computing feasible solutions (i.e., lower bounds) and the efficient procedure for computing upper bounds. In addition, we provide an analysis of the theoretical convergence of the semidefinite bounding procedure. Finally, in Section 5, we discuss the parameters of the code and the advanced use of *BiqCrunch*.

2. *BIQCRUNCH* IN PRACTICE, EXAMPLES, ILLUSTRATIONS

The latest version of the *BiqCrunch* code is available from the *BiqCrunch* webpage. Installation instructions are included with the source code. We have made the installation straightforward, only requiring a C compiler, a Fortran compiler, and either LAPACK or the Intel MKL.

Once *BiqCrunch* has been installed, it can be run from the command-line as follows.

```
$ biqcrunch [-v 1] <INSTANCE> <PARAMETERS>
```

The optional parameter `-v` is the verbosity; `<INSTANCE>` is the input file in the *BiqCrunch* format; `<PARAMETERS>` is a parameters file which can be one of the files provided with the code, or a user's own file.

This section provides some information about the format of the input file (in Section 2.1) and examples on how to use *BiqCrunch* (in Sections 2.2 and 2.3). We refer to the user manual for complete information on installing and running *BiqCrunch*, and to Section 5 for a discussion of the parameters.

2.1. Matrix formulation and input file format

We describe briefly here the matrix formulation of the binary quadratic problem (1) on which the *BiqCrunch* input file format is based. First we introduce the usual inner product of two matrices and the associated norm (sometimes called the Frobenius

norm), respectively defined by

$$\langle X, Y \rangle = \operatorname{trace}(X^T Y) = \sum_{ij} X_{ij} Y_{ij} \qquad \text{and} \qquad \|X\|_F = \sqrt{\langle X, X \rangle} = \sqrt{\sum_{ij} X_{ij}^2}.$$

Since $z^T S_i z = \langle S_i, zz^T \rangle$, this inner product allows us to rewrite the quadratic terms $z^T S_i z + s_i^T z$ of problem (1) as linear terms $\langle Q_i, Z \rangle$ where

$$Z = \begin{bmatrix} zz^T & z \\ z^T & 1 \end{bmatrix} \qquad \text{and} \qquad Q_i = \begin{bmatrix} S_i & \frac{1}{2} s_i \\ \frac{1}{2} s_i^T & 0 \end{bmatrix}.$$

Thus, the binary quadratic problem (1) can be reformulated as:

$$\begin{cases} \text{maximize} & \langle Q_0, Z \rangle & \\ \text{subject to} & \langle Q_i, Z \rangle \leq a_i, & i \in \{1, \ldots, m_I\} \\ & \langle Q_i, Z \rangle = a_i, & i \in \{m_I + 1, \ldots, m_I + m_E\} \\ & Z = \begin{bmatrix} zz^T & z \\ z^T & 1 \end{bmatrix}, \ z \in \{0,1\}^n. & \end{cases} \tag{5}$$

Note that the objective function and the constraints are now linear with respect to Z, and that the only non-convexity of the problem lies in the form of Z, which is a rank-one matrix with 0–1 entries.

BiqCrunch requires the objective value of (5) to be integer for any feasible solution. This corresponds to having integers on the diagonal of Q_0 and integers divided by two on the off-diagonal entries of Q_0. *BiqCrunch* takes advantage of this feature by pruning the branch-and-bound search tree when the computed bound is strictly less than $\beta + 1$, where β is the objective value of the current best feasible solution; see Section 4.1. To use *BiqCrunch* with fractional data, one should first multiply the coefficients by the smallest common denominator to make them integers.

The matrix formulation in problem (5) is used in the input format of the solver. The *BiqCrunch* format is similar to the widely used sparse `SDPA` format in semidefinite optimization; see [Yamashita et al. 2012]. Roughly speaking, it consists of specifying general parameters (m, n, type of constraints, etc.) and describing the matrices Q_i in a sparse matrix format. The *BiqCrunch* solver stores the input problem matrices in this sparse format in memory to keep its memory requirements small. The main difference between the *BiqCrunch* format and the sparse `SDPA` format is that the first line of a *BiqCrunch* input file indicates if the problem is a maximization problem (using $+1$) or a minimization problem (using -1). Moreover the *BiqCrunch* format uses a block of size $n + 1$ to represent the positive semidefinite matrix and a diagonal block of slack variables (for inequality constraints). The *BiqCrunch* file format is fully described and illustrated in the user manual. We also give an example in the next section.

To write a *BiqCrunch* file, a user would need to have a good understanding of the SDP relaxation and how to write it in `SDPA` format. This was a major barrier to being able to use *BiqCrunch* before we created an `lp2bc` converter. To simplify the use of *BiqCrunch*, we provide two types of converters to the *BiqCrunch* format:

(1) A converter from the so-called LP format to the *BiqCrunch* format: `lp2bc`.
(2) Specific converters for each problem class; for example `mc2bc` to convert Max-Cut problems and `kc2bc` to convert Max-k-Cluster problems.

These conversion tools are described in the user manual. In the next two sections, we give examples of the use of *BiqCrunch* to solve a generic problem specified in the LP format, and a Max-Cut problem specified in a standard sparse format.

Let us emphasize that *BiqCrunch* works directly on problem (5) given by the Q_i's and a_i's. This is in contrast with CPLEX (version 12.1) and Gurobi (version 5.6), which both

preprocess the entries and in particular convexify binary quadratic problems to work with positive semidefinite matrices. Such automatic reformulations or convexifications are not always efficient as they can negatively affect the solution process. (Smarter convexifications use semidefinite optimization, as exploited in [Billionnet and Elloumi 2007; Billionnet et al. 2009]). In *BiqCrunch* there is no reformulation phase and no data preprocessing phase. The user has complete control over the problem and the way it is modeled. We give some advice on how to enhance the problem formulation in Section 5.3.

2.2. Example with the LP converter

We give an illustration of running *BiqCrunch* on a simple test problem, using the converter from the human-readable LP format of IBM/CPLEX to the *BiqCrunch* format. Consider the binary quadratic problem

$$\begin{cases} \text{maximize} & z_1 z_2 + 2 z_1 z_3 \\ \text{subject to} & z_1 + z_2 + z_3 \leq 2 \\ & (z_1, z_2, z_3) \in \{0,1\}^3 \end{cases}$$

whose optimal solution is $(1, 0, 1)$. We describe this problem in the LP format as a file called `example.lp` containing the following lines:

```
maximize
    z1*z2 + 2 z1*z3
subject to
    z1 + z2 + z3 <= 2
binary
    z1 z2 z3
end
```

The `lp2bc` converter is called by the command line

```
$ lp2bc.py example.lp > example.bc
```

and generates the following file in *BiqCrunch* format[1]:

```
# List of binary variables:
#    1:  z1
#    2:  z2
#    3:  z3
1 = max problem
1 = number of constraints
2 = number of blocks
4, -1
2.0
0 1 1 2 0.5
0 1 1 3 1.0
1 1 1 4 0.5
1 1 2 4 0.5
1 1 3 4 0.5
1 2 1 1 1.0
```

[1] The *BiqCrunch* format is similar to the sparse SDPA format and defines the problem by specifying the coefficient matrices Q_i, which constraints are inequalities, and the right-hand-side values a_i of all the constraints. For a complete description of the *BiqCrunch* format, see the *BiqCrunch* User's Guide which is available on the *BiqCrunch* website.

We solve now the problem using *BiqCrunch* (with default parameters) by executing

```
$ biqcrunch example.bc generic.param
```

We obtain the following command-line output:

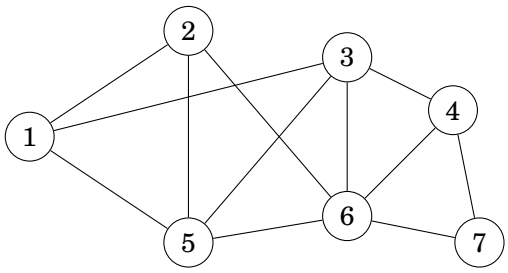
```
Output file: example.bc.output
Input file:  example.bc
Parameter file: generic.param
Node 0 Feasible solution 2
Nodes = 1
Root node bound = 2.84
Maximum value = 2
Solution = { 1 3 }
CPU time = 0.0074 s
```

This output reports the result of running of *BiqCrunch* on this instance. At the root node of the branch-and-bound search tree, the computed bound is 2.84, and, since the optimal value is integer, this gives an effective upper-bound of 2. The generated solution was $z_1 = z_3 = 1$ and $z_2 = 0$ with objective value 2, which proves that it is an optimal solution. Thus there was only one node in the branch-and-bound search. More detailed output information is given in the output file.

2.3. Example with the Max-Cut converter

We give an illustration of using *BiqCrunch* to solve a Max-Cut problem on a simple graph, using the converter `mc2bc` to create a *BiqCrunch* input file from a graph file. Let us consider the following graph, drawn in the figure and described in a file `graph.txt`. The first line of `graph.txt` records the number of nodes and number of edges in the graph and the following lines record the list of edges, each written as the triple $i\ j\ w_{ij}$. Note that each edge in this graph has a weight of 1.

```
7 12
1 2 1
1 3 1
1 5 1
2 5 1
2 6 1
3 4 1
3 5 1
3 6 1
4 6 1
4 7 1
5 6 1
6 7 1
```

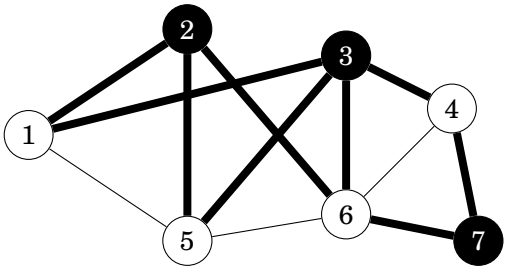


Using the `mc2bc` converter, we compute a maximal cut for this graph using *BiqCrunch* as follows:

```
$ mc2bc.py graph.txt > maxcut.bc
$ biqcrunch maxcut.bc maxcut.param
```

BiqCrunch returns the following output giving us a maximum cut which we have represented in the figure. The nine edges between the black nodes and the white nodes are a maximal cut.

```
Output file: maxcut.bc.output
Input file:  maxcut.bc
Parameter file: maxcut.param
Node 0 Feasible solution 9
Nodes = 1
Root node bound = 9.91
Maximum value = 9
Solution = { 1 4 5 6 }
CPU time = 0.0075 s
```



3. MATHEMATICAL FOUNDATIONS OF *BIQCRUNCH*

In this section, we give a theoretical description of the bounds used by *BiqCrunch*. We start by recalling some basic facts about semidefinite bounds in Section 3.1, then we present the special semidefinite bounds used by *BiqCrunch* in Section 3.2.

We refer the interested reader to the books [Saigal et al. 2000] and [Anjos and Lasserre 2012] for more information, including historical perspectives, about semidefinite programming in the context of combinatorial optimization.

3.1. Semidefinite relaxation

We first introduce some notation and briefly describe the standard semidefinite bound for the binary quadratic problem (1) written as its matrix form as problem (5). The presentation of bounds in the next section becomes more straightforward when the problem is reformulated using $\{-1,1\}$ variables. We apply the change of variables between $z \in \{0,1\}^n$ and $x \in \{-1,1\}^n$ defined by $z = \frac{1}{2}(x+e)$, where e is the vector of all ones. This can be written as

$$\begin{bmatrix} z \\ 1 \end{bmatrix} = U \begin{bmatrix} x \\ 1 \end{bmatrix} \qquad \text{with} \qquad U = \begin{bmatrix} \frac{1}{2}I_n & \frac{1}{2}e \\ 0 & 1 \end{bmatrix}$$

and therefore, in the matrix formulation (5)

$$Z = UXU^T \qquad \text{where} \qquad X = \begin{bmatrix} xx^T & x \\ x^T & 1 \end{bmatrix}.$$

Since U is invertible, this transformation works the opposite way as well. Observe now that the binary constraints $x_i \in \{-1,1\}$ (or $x_i^2 = 1$) can be formulated as $\operatorname{diag}(xx^T) = e$. A formulation in $\{-1,1\}$ variables of problem (5) is

$$\left\{ \begin{array}{ll} \text{maximize} & \langle U^T Q_0 U, X \rangle \\ \text{subject to} & \langle U^T Q_i U, X \rangle \leq a_i, \ i \in \{1, \ldots, m_I\} \\ & \langle U^T Q_i U, X \rangle = a_i, \ i \in \{m_I+1, \ldots, m_I+m_E\} \\ & \operatorname{diag}(X) = e \\ & X = \begin{bmatrix} xx^T & x \\ x^T & 1 \end{bmatrix}. \end{array} \right. \tag{6}$$

In *BiqCrunch*, the matrices $U^T Q_i U$ are formed directly from Q_i when reading the data, without explicitly forming the matrix U.

Let us now simplify notation by introducing $a \in \mathbb{R}^{m_I}$, $b \in \mathbb{R}^{m_E+n+1}$ and the two mappings

$$A \colon \mathbb{S}^{n+1} \to \mathbb{R}^{m_I} \quad \text{and} \quad B \colon \mathbb{S}^{n+1} \to \mathbb{R}^{m_E+n+1}$$

to gather all the inequality constraints of problem (6) as $A(X) \leq a$ and all the equality constraints as $B(X) = b$ (those defined by $U^T Q_i U$ together with the diagonal con-

straints). Defining $Q = U^T Q_0 U$, problem (1) can be written as the following optimization problem with respect to X positive semidefinite (denoted $X \succeq 0$) and rank-one:

$$\begin{cases} \text{maximize} & \langle Q, X \rangle \\ \text{subject to} & A(X) \leq a \\ & B(X) = b \\ & X \succeq 0 \\ & \operatorname{rank}(X) = 1. \end{cases} \tag{7}$$

The basic semidefinite relaxation of this problem is then obtained by dropping the rank-one constraint on X:

$$\begin{cases} \text{maximize} & \langle Q, X \rangle \\ \text{subject to} & A(X) \leq a \\ & B(X) = b \\ & X \succeq 0. \end{cases} \tag{8}$$

Since problem (1) is equivalent to problem (7), the optimal value of problem (8) provides an upper bound on the optimal value of problem (1). As in linear programming, we can further tighten the bound by adding valid inequalities to problem (7) before relaxing. There exist many problem-dependent or generic inequality families in our framework; see, e.g., the textbook [Deza and Laurent 1997] or early references as [Barahona et al. 1989; Helmberg and Rendl 1998].

In *BiqCrunch*, we use the triangle inequalities, defined for $1 \leq i < j < k \leq n+1$ by

$$\begin{aligned} X_{ij} + X_{ik} + X_{jk} &\geq -1, \\ X_{ij} - X_{ik} - X_{jk} &\geq -1, \\ -X_{ij} + X_{ik} - X_{jk} &\geq -1, \\ -X_{ij} - X_{ik} + X_{jk} &\geq -1. \end{aligned}$$

These inequalities correspond to the fact that for $x \in \{-1, 1\}^{n+1}$, it is not possible to have exactly one of three products $\{x_i x_j, x_i x_k, x_j x_k\}$ equal to -1, nor is it possible to have all three of the products equal to -1. These inequalities are particularly interesting in our framework, for two reasons. There are $4\binom{n+1}{3}$ of them, which is large but still manageable. Given X, we can evaluate all of them and efficiently find the most violated ones. Moreover, they are known to give good results for semidefinite relaxations in general (see, e.g., [Helmberg and Rendl 1998; Roupin 2004; Armbruster et al. 2012]).

Ideally we would like to add all the triangle inequalities to problem (8) to get the tightest possible bound of this type. However, the cost of doing so is very high, so the relaxation incorporating all the triangle inequalities is rarely used (see, e.g., the numerical comparisons of [Roupin 2004]). In *BiqCrunch* we iteratively identify a subset of useful inequalities, as in [Rendl et al. 2010]. The idea is to select the most promising active inequalities using the current approximate solution. The management of inequalities will be precisely described in Algorithm 2.

For a subset of triangle inequalities $\mathcal{I}$, we let $A_{\mathcal{I}} \colon \mathbb{S}^n \to \mathbb{R}^{|\mathcal{I}|}$ be the corresponding linear function describing the inequalities in this subset. We end up with the following strengthened SDP relaxation of problem (1):

$$(\text{SDP}_{\mathcal{I}}) \qquad \begin{cases} \text{maximize} & \langle Q, X \rangle \\ \text{subject to} & A(X) \leq a \\ & B(X) = b \\ & A_{\mathcal{I}}(X) \geq -e \\ & X \succeq 0. \end{cases} \tag{9}$$

Note that the maximum is finite and attained if the problem is feasible. This follows from the fact the feasible set is included in the set of correlation matrices ($X \succeq 0$ and $\mathrm{diag}(X) = e$) which is well-known to be compact (see, e.g., [Malick 2007, Theorem 1] or more complete studies as [Laurent and Poljak 1995; Laurent and Poljak 1996]).

3.2. *BiqCrunch* adjustable semidefinite bounds

In this section, we define and sketch the main properties of the semidefinite bounds that are used by *BiqCrunch*. We refer to [Malick and Roupin 2013] for the background, the motivation and the theory behind the family of adjustable semidefinite bounds.

Recall that the nonnegative part X_+ of a symmetric matrix X is computable via an eigenvalue decomposition $X = V\,\mathrm{Diag}(\sigma)V^T$ (with the vector of eigenvalues $\sigma \in \mathbb{R}^n$, and an orthogonal matrix $V \in \mathbb{R}^{n\times n}$) by

$$X_+ = V\,\mathrm{Diag}(\max\{\sigma, 0\})V^T.$$

Note that X_+ is the orthogonal projection of X onto the set of positive semidefinite matrices [Higham 1988]. The nonpositive part X_- is defined similarly. We will need the following property:

$$\langle X, X_+\rangle = \|X_+\|_F^2\,. \tag{10}$$

Let $\mathcal{I}$ be a set of inequalities and define $\Omega := \mathbb{R}_+^{m_I} \times \mathbb{R}^{m_E+n+1} \times \mathbb{R}_+^{|\mathcal{I}|}$. For any dual variables $(\lambda, \mu, \nu) \in \Omega$, we define the positive semidefinite matrix

$$X_{\mathcal{I}}(\lambda, \mu, \nu) := [Q - A^*(\lambda) - B^*(\mu) + A^*_{\mathcal{I}}(\nu)]_+ \tag{11}$$

where A^*, B^*, and $A^*_{\mathcal{I}}$ are the adjoints of the linear operators A, B, and $A_{\mathcal{I}}$ representing the constraints. For example, for $\lambda \in \mathbb{R}^{m_I}$,

$$A^*(\lambda) = \sum_{i=1}^{m_I} \lambda_i U^T Q_i U.$$

Furthermore, for a parameter $\alpha > 0$, we introduce the function $F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu)$ defined for $(\lambda, \mu, \nu) \in \Omega$ by

$$F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu) := \frac{1}{2\alpha}\,\|X_{\mathcal{I}}(\lambda, \mu, \nu)\|_F^2 + a^T\lambda + b^T\mu + e^T\nu + \frac{\alpha}{2}(n+1)^2. \tag{12}$$

Up to a change of sign and a slight change of notation, $F^\alpha_{\mathcal{I}}$ corresponds to the function Θ in [Malick and Roupin 2013]. Using the current notation, Theorem 3 of [Malick and Roupin 2013] reads as follows.

THEOREM 3.1. *For a set of inequalities $\mathcal{I}$, a parameter $\alpha > 0$, and any $(\lambda, \mu, \nu) \in \Omega$, we have that $F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu)$ is an upper bound on the optimal value of the semidefinite relaxation* (9) *and therefore on the optimal value of the binary quadratic problem* (1)*.*

The question now is how to choose parameters to get these bounds $F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu)$ as tight as possible. For fixed α and $\mathcal{I}$, the tightest bounds can be obtained by minimizing $F^\alpha_{\mathcal{I}}$ over $(\lambda, \mu, \nu) \in \Omega$. The smoothness of $F^\alpha_{\mathcal{I}}$ is the key property that allows it to be efficiently minimized. Theorem 2 of [Malick and Roupin 2013] states that the function $F^\alpha_{\mathcal{I}}$ is convex and differentiable on Ω, and we have explicit expressions of its partial gradients. In particular, if $X = \frac{1}{\alpha}X_{\mathcal{I}}(\lambda, \mu, \nu)$, then we have

$$\begin{aligned}
\nabla_\lambda F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu) &= a - A(X)\,,\\
\nabla_\mu F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu) &= b - B(X)\,,\\
\nabla_\nu F^\alpha_{\mathcal{I}}(\lambda, \mu, \nu) &= e + A_{\mathcal{I}}(X)\,.
\end{aligned} \tag{13}$$

Thus, we can minimize $F_{\mathcal{I}}^{\alpha}$ using any first-order optimization algorithm that can handle nonnegativity constraints. We could also use the so-called second-order semismooth Newton method [Qi and Sun 1993] since it is possible to show that $\nabla F_{\mathcal{I}}^{\alpha}$ is semismooth using properties of the projection $[\cdot]_+$ (see, e.g., [Bolte et al. 2008; Hiriart-Urruty and Malick 2012]).

For its simplicity and robustness, *BiqCrunch* uses a quasi-Newton method (more specifically, a projected BFGS with Wolfe line-search), which lies between first-order and second-order methods. The properties guaranteeing convergence (see, e.g., [Bonnans et al. 2003, Theorem 4.9]) hold here, as $F_{\mathcal{I}}^{\alpha}$ is convex and differentiable with Lipschitz gradient.

LEMMA 3.2. *For given α and $\mathcal{I}$, the gradient $\nabla F_{\mathcal{I}}^{\alpha}$, as an operator from Ω to $\mathbb{R}^{m_I+m_E+n+1}$, is Lipschitz continuous with Lipschitz constant L/α, where L is a constant that depends on the norms of A, B and $A_{\mathcal{I}}$ and their adjoints.*

PROOF. We will use the standard inner product norms for each space with the associated operator norm (that is the largest singular value of the operator); all these norms are denoted $\|\cdot\|$. Consider (λ, μ, ν) and (λ', μ', ν') in Ω, and let $X = X_{\mathcal{I}}(\lambda, \mu, \nu)$ and $X' = X_{\mathcal{I}}(\lambda', \mu', \nu')$. Observe by equation (13) that we have

$$\nabla F_{\mathcal{I}}^{\alpha}(\lambda, \mu, \nu) - \nabla F_{\mathcal{I}}^{\alpha}(\lambda', \mu', \nu') = \frac{1}{\alpha}\big(A\,(X - X') + B\,(X - X') + A_{\mathcal{I}}\,(X - X')\,\big)$$

which yields, with the constant $N = \max\{\|A\|, \|B\|, \|A_{\mathcal{I}}\|\}$,

$$\|\nabla F_{\mathcal{I}}^{\alpha}(\lambda, \mu, \nu) - \nabla F_{\mathcal{I}}^{\alpha}(\lambda', \mu', \nu')\| \leq \frac{3N}{\alpha}\,\|X - X'\|\,. \tag{14}$$

From [Hiriart-Urruty and Lemaréchal 2001, Eq. (3.1.6)], we have

$$\|X - X'\| \leq \|A^*(\lambda - \lambda') + B^*(\mu - \mu') + A_{\mathcal{I}}^*(\nu - \nu')\|\,.$$

Using the fact that $(u + v + w)^2 \leq 3(u^2 + v^2 + w^2)$ for all $u, v, w \in \mathbb{R}$, we have

$$\begin{aligned}\|X - X'\|^2 \;&\leq\; 3\big(\,\|A^*\|^2\,\|\lambda - \lambda'\|^2 + \|B^*\|^2\,\|\mu - \mu'\|^2 + \|A_{\mathcal{I}}^*\|^2\,\|\nu - \nu'\|^2\,\big)\\ &\leq\; 3N^2\,\|(\lambda, \mu, \nu) - (\lambda', \mu', \nu')\|^2\,,\end{aligned}$$

since the operators and their adjoints have the same norm. Combining this with equation (14), we end up with

$$\|\nabla F_{\mathcal{I}}^{\alpha}(\lambda, \mu, \nu) - \nabla F_{\mathcal{I}}^{\alpha}(\lambda', \mu', \nu')\| \leq \frac{3\sqrt{3}N^2}{\alpha}\,\|(\lambda, \mu, \nu) - (\lambda', \mu', \nu')\|\,,$$

which is the desired inequality with $L = 3\sqrt{3}N^2$. □

In addition to minimizing $F_{\mathcal{I}}^{\alpha}$ for fixed α and $\mathcal{I}$, we have two ways to get tighter bounds. Firstly, adding violated triangle inequalities to $\mathcal{I}$ enlarges the space Ω which allows us to further minimize $F_{\mathcal{I}}^{\alpha}$. Secondly, decreasing the parameter α also yields tighter bounds. Theorem 4 of [Malick and Roupin 2013] shows that α controls the tightness of the bound, in that smaller values of α give tighter bounds. In practice, special attention should be paid to decreasing α, since Lemma 3.2 relates α to the smoothness of $F_{\mathcal{I}}^{\alpha}$, indicating that when α is small, the gradient can have a sharp behavior, and therefore minimizing $F_{\mathcal{I}}^{\alpha}$ could become ill-conditioned.

The semidefinite bounding procedure presented in the next section efficiently combines these three levers (minimizing $F_{\mathcal{I}}^{\alpha}$, adding inequalities, and decreasing α). Its convergence analysis is studied in Section 4.4. Later in Section 5.1, practical advice is given on how to adjust key parameters to compute bounds efficiently (with a good ratio of tightness to computing time).

4. ALGORITHMIC DESCRIPTION OF *BIQCRUNCH*

BiqCrunch is a branch-and-bound algorithm, implemented using the branch-and-bound platform BOB [Le Cun et al. 1995] which automatically handles the management of subproblems. The BOB platform only requires the following functionalities to be implemented:

(1) a bounding procedure (producing an upper bound),
(2) a heuristic for generating a feasible solution (producing a lower bound),
(3) a method for generating subproblems (branching).

In this section, we provide details about how each of these are implemented in *BiqCrunch*. We consider the binary quadratic subproblem of the current node of the branch-and-bound tree (with a slight abuse of notation, we consider it to be problem (1) and use the same notation as before). At iteration k of the bounding procedure, the algorithm brackets the optimal value as

$$\beta_k \leq \quad \text{optimal value of the binary quadratic problem} \quad \leq F_k, \tag{15}$$

where β_k is the best lower bound given by the heuristics (described in Section 4.2), and F_k is the upper bound of the bounding procedure (described in Section 4.1). Using the fact that we know that the optimal value of problem (1) is integer, we have that

$$\text{if } F_k < \beta_k + 1, \text{ then we prune the node of the branch-and-bound tree,} \tag{16}$$

since all feasible solutions of the subproblem have an objective value no better than β_k. If this is not the case, we need to explore the branch-and-bound tree further. The different branching strategies that are available in *BiqCrunch* are described in Section 4.3. Finally, Section 4.4 provides a theoretical analysis of the convergence of our semidefinite bounding procedure.

4.1. Semidefinite bounding procedure

We first turn our attention to the bounding procedure used by *BiqCrunch* and its computational aspects. We start by emphasizing that no SDP problem is solved during the bounding procedure, which is a major difference compared to semidefinite-based procedures used by other software packages, such as *Biq Mac* or SCIP-SDP. Our bounding procedure can be very fast to run if the node is easy to prune, but is also able to provide tighter more expensive bounds if necessary.

The key numerical ingredient of the bounding procedure of *BiqCrunch* is the algorithm that minimizes the bounding function (12) for a given set of inequalities $\mathcal{I}$ and a given tightness parameter α. We use the projected quasi-Newton software L-BFGS-B [Zhu et al. 1997; Morales and Nocedal 2011] (with default parameters, except for `nitermax`, `minNiter`, and `maxNiter`; see Section 5.1). The quasi-Newton solver calls a subroutine that computes the value of the bounding function (12) and its gradient (13) at the current point (λ, μ, ν). This computation boils down to the computation of $X_{\mathcal{I}}(\lambda, \mu, \nu)$ as defined in equation (11), which, in turn, reduces to computing the positive eigenvalues and corresponding eigenvectors of the symmetric matrix $[Q - A^*(\lambda) - B^*(\mu) + A_I^*(\nu)]$ for which we use the routine `DSYEVR` of the package `MKL` (or `LAPACK`, if `MKL` is not available). Note that the eigendecomposition computed here is also used in the heuristic for computing feasible solutions (see Section 4.2).

The semidefinite bounding procedure of *BiqCrunch* is described in Algorithm 1. Given a set of inequalities $\mathcal{I}_{k-1}$ and tightness parameter α_k, *BiqCrunch* runs a quasi-Newton algorithm on $F_{\mathcal{I}_{k-1}}^{\alpha_k}$: it is warm-started from the previous $(\lambda_{k-1}, \mu_{k-1}, \nu_{k-1})$ and

ALGORITHM 1: Semidefinite bounding algorithm of *BiqCrunch*

Data: `alpha0` > 0; `tol0` > 0; $0 <$ `scaleAlpha`, `scaleTol` < 1.
Initialize parameters: $k \leftarrow 1$, $\beta_1 \leftarrow -\infty$, $\varepsilon_1 \leftarrow$ `tol0`, $\alpha_1 \leftarrow$ `alpha0`.
Initialize variables: $\mathcal{I}_0 \leftarrow \emptyset$, $\lambda_0 \leftarrow 0 \in \mathbb{R}^{m_E+n+1}$, $\mu_0 \leftarrow 0 \in \mathbb{R}^{m_I}$, $F_0 \leftarrow +\infty$.

while $F_k \geq \beta_k + 1$ **do**

Minimize the function $F^{\alpha_k}_{\mathcal{I}_{k-1}}$ using a quasi-Newton method:
Starting from $(\lambda_{k-1}, \mu_{k-1}, \nu_{k-1})$, compute $(\lambda_k, \mu_k, \hat{\nu}_k)$ such that (17) holds.

if `withCuts` **then**
Run inequality update subroutine to get $\mathcal{I}_k$ (and associated multipliers ν_k).
end

Update the upper bound: $F_k \leftarrow F^{\alpha_k}_{\mathcal{I}_{k-1}}(\lambda_k, \mu_k, \hat{\nu}_k) = F^{\alpha_k}_{\mathcal{I}_k}(\lambda_k, \mu_k, \nu_k)$.
Update the lower bound: run the heuristic in Algorithm 3 to get β, and update $\beta_k \leftarrow \max\{\beta_{k-1}, \beta\}$

if $\mathrm{Card}(\mathcal{I}_k - \mathcal{I}_{k-1}) \leq$ `minCuts` **or** α_k has not changed for `maxNAiter` iterations **then**
$\alpha_{k+1} \leftarrow \max\{$`minAlpha`, `scaleAlpha` $\cdot\, \alpha_k\}$, $\quad \varepsilon_{k+1} \leftarrow \max\{$`minTol`, `scaleTol` $\cdot\, \varepsilon_k\}$
else
$\alpha_{k+1} \leftarrow \alpha_k$, $\quad \varepsilon_{k+1} \leftarrow \varepsilon_k$
end
$k \leftarrow k + 1$;

end
Run the heuristic in Algorithm 3.

it computes a solution $(\lambda_k, \mu_k, \hat{\nu}_k)$ of ℓ_∞-tolerance ε_k:

$$\max\left\{ \left\| \left[a - A\left(X_k\right)\right]_- \right\|_\infty,\ \left\| b - B\left(X_k\right) \right\|_\infty,\ \left\| \left[e + A_{\mathcal{I}}\left(X_k\right)\right]_- \right\|_\infty \right\} < \varepsilon_k, \tag{17}$$

where $X_k = \frac{1}{\alpha_k} X_{\mathcal{I}_{k-1}}(\lambda_k, \mu_k, \hat{\nu}_k)$. We stop the bounding procedure when the value of the bound is less than $\beta_k + 1$; in practice, we also stop it when it is likely that a bound less than $\beta_k + 1$ is not attainable within a reasonable amount of time. It is important to note that the bounding procedure may be stopped anytime and will return a valid upper-bound for problem (1) (by Theorem 3.1).

The remainder of the bounding procedure consists of updating parameters. Algorithm 1 interlaces the decrease of α_k and ε_k with the management of the set of inequalities $\mathcal{I}_k$ (by Algorithm 2 that we describe below). The idea is to reduce the tightness parameter α_k when we can no longer make good progress by adding inequalities. We reduce α_k and ε_k when the number of violated triangle inequalities is lower than the threshold `minCuts`, or when they have not been reduced for `maxNAiter` iterations.

Having a lot of enforced inequalities is both good and bad: the more inequalities the better the bound, but on the other hand it increases the number of dual variables that must be optimized over in the quasi-Newton method. *BiqCrunch* updates the set of enforced inequalities by Algorithm 2. First, it gets rid of ε_k-inactive triangle inequalities for X_k (i.e., the indices i such that $(\hat{\nu}_k)_i$ is zero and $\mathcal{A}_i(X_k) + 1 > \varepsilon_k$). Second, it adds a predefined number of the most violated inequalities to improve the bound as quickly as possible. Once the set $\mathcal{I}_k$ is updated, Algorithm 2 generates ν_k such that

$$X_k = \frac{1}{\alpha_k} X_{\mathcal{I}_k}(\lambda_k, \mu_k, \nu_k)$$

and

$$F_k = F^{\alpha_k}_{\mathcal{I}_k}(\lambda_k, \mu_k, \nu_k) = \frac{\alpha_k}{2} \left\| X_k \right\|^2 + a^T \lambda_k + b^T \mu_k + e^T \nu_k + \frac{\alpha_k}{2}(n+1)^2. \tag{18}$$

ALGORITHM 2: Inequality update subroutine of the bounding procedure

Data: (at iteration k of the bounding procedure) $\mathcal{I}_{k-1}$, $\hat{\nu}_k$, X_k, ε_k and X_{k-1}
Remove the triangle inequalities that are not ε_k-active:

$$\mathcal{I}_{k-1}^{-} \leftarrow \{ i \in \mathcal{I}_{k-1} : (\hat{\nu}_k)_i = 0 \text{ and } \mathcal{A}_i(X_k) + 1 > \varepsilon_k \} .$$

Add the most-violated triangle inequalities:

Let $i_1, \ldots, i_\ell$ be the indices $i \notin \mathcal{I}_{k-1}$ such that $\mathcal{A}_i(X_k) + 1 \leq \texttt{gapCuts} < 0$, ordered such that $\mathcal{A}_{i_1}(X_k) \leq \cdots \leq \mathcal{A}_{i_\ell}(X_k)$. Let

$$\mathcal{I}_{k-1}^{+} \leftarrow \{ i_1, \ldots, i_K \} , \quad \text{where } K = \min\{\ell, \texttt{cuts}\}.$$

Update the set of inequalities: $\mathcal{I}_k \leftarrow \left(\mathcal{I}_{k-1} \setminus \mathcal{I}_{k-1}^{-} \right) \cup \mathcal{I}_{k-1}^{+}$.
Initialize the multipliers for added inequalities to zero:

$$\text{for each } i \in \mathcal{I}_k, \qquad (\nu_k)_i \leftarrow \begin{cases} (\hat{\nu}_k)_i & \text{if } i \in \mathcal{I}_{k-1}, \\ 0 & \text{if } i \notin \mathcal{I}_{k-1}. \end{cases}$$

4.2. Heuristics: options and generic semidefinite heuristic

BiqCrunch uses heuristics for generating feasible solutions for problem (1). The best feasible solution found provides the lower bound β_k in (15). This lower bound is used to prune parts of the branch-and-bound search tree according to the rule (16).

BiqCrunch allows the use of three types of heuristics:

(1) root-node heuristic (called once before starting the branch-and-bound method),
(2) bound heuristic (called each iteration of the bounding procedure),
(3) node heuristic (called at the end of bounding procedure).

These three type of heuristics can all be the same or be completely different; they can also depend on the type of problem that is being solved. We include several specific heuristics in the *BiqCrunch* release for Max-Cut, Max-k-cluster, and Maximum-Independent-Set. Users can also specify their own heuristics for their problems of interest as explained in Section 5.2. By default, the generic *BiqCrunch* solver uses an empty heuristic for the root-node heuristic and a semidefinite heuristic for both the bound heuristic and the node heuristic.

The generic semidefinite heuristic of *BiqCrunch* is presented in Algorithm 3. It is based on the celebrated Goemans-Williamson heuristic for Max-Cut [Goemans and Williamson 1995] using randomly generated hyperplans and a factorization of the optimal semidefinite solution computed by the bounding procedure, called $\hat{X}$ here. From a factorization $\hat{X} = WW^T$ with $W \in \mathbb{R}^{(p+1)\times m}$ and a random unit vector $v \in \mathbb{R}^m$, a $\{0,1\}$-vector z is generated from the sign of the inner-product of v with the i^{th} row of W. Then the feasibility of this $\{0,1\}$-vector z for problem (1) is tested. The best lower bound is updated if z is both feasible and improves the objective value. Note that, contrary to [Goemans and Williamson 1995], we do not need to compute a Cholesky factorization of $\hat{X}$, since a factorization is already available from the bounding procedure (which computes an eigendecomposition of the matrix, see Section 4.1). Since this process is computationally inexpensive, this is repeated for several random v and different z.

At the end of the semidefinite heuristic, we also add a simple "1-opt" local-search. This local-search routine returns a solution that is locally optimal, in the sense that changing any variable from zero to one, or from one to zero, does not result in a better feasible solution. Note that for some problems (Max-k-Cluster for example), this local-

search does not make sense since it cannot produce feasible points; in this case, a parameter `local_search` allows us to disable it.

ALGORITHM 3: Generic semidefinite heuristic for finding feasible solutions

Data: a positive semidefinite matrix $X = WW^T$, a vector $\hat{z} \in \{0,1\}^n$ and its objective value β
for *many iterations* **do**
  Generate a random unit vector v
  for $i = 1, \ldots, n$ **do**
    if z_i *is a fixed variable* **then**
      $z_i \leftarrow$ fixed value of z_i
    else
      $z_i \leftarrow \begin{cases} 0, & \text{if } v^T \mathrm{row}_i(W) < 0 \\ 1, & \text{otherwise} \end{cases}$
    end
  end
  Test of improvement:
  if z *is feasible for problem* (1) and $z^T S_0 z + s_0^T z > \beta$ **then**
    $\hat{z} \leftarrow z$ and $\beta \leftarrow z^T S_0 z + s_0^T z$
  end
end
if $\hat{z}$ *is feasible for problem* (1) **then**
  while $\hat{z}$ *is not locally optimal* **do**
    $\hat{z} \leftarrow$ a strictly better local solution
  end
end

4.3. Branching strategies

BiqCrunch provides three branching strategies, each of which can be selected by changing the value of the parameter `branchingStrategy` in the input parameter file. The branching rule uses the optimal semidefinite solution given by the semidefinite bounding procedure, as follows. First we extract the last column $\hat{x}$ of $\hat{X}$ and define $\hat{z} = \frac{1}{2}(\hat{x} + e)$. Then we choose a variable z_i to branch on, using one of the following three strategies:

(1) least-fractional: a variable z_i for which $\hat{z}_i$ is furthest from $\frac{1}{2}$ is selected;
(2) most-fractional: a variable z_i for which $\hat{z}_i$ is closest to $\frac{1}{2}$ is selected;
(3) closest-to-one: a variable z_i for which $\hat{z}_i$ is closest to 1 is selected.

The most-fractional branching strategy is used as the default in *BiqCrunch*.

Branching on variable z_i creates two new subproblems, one where z_i is fixed to 0 and the other where z_i is fixed to 1. These subproblems correspond to nodes in the branch-and-bound search tree. When branching occurs, two nodes are created and added to this search tree. The BOB branch-and-bound platform [Le Cun et al. 1995] automatically selects the subproblem having the weakest bound to be the next subproblem to branch on; in the case of a tie, BOB selects the subproblem that is lower in the search tree (i.e., having more variables fixed); if the subproblem is already near the bottom of the search tree (i.e., where all variables are fixed), BOB switches to a depth-first-search traversal of that subtree.

4.4. Convergence of the semidefinite bounding procedure

In this section, we study the theoretical convergence of Algorithm 1, the semidefinite bounding procedure of *BiqCrunch*. We assume that it runs an infinite number of iterations: more precisely, we set $\texttt{maxNIter} = +\infty$, $\texttt{minAlpha} = 0$, $\texttt{minTol} = 0$, and we suppose that β is small enough that the loop will not stop. In this case, the two tightness parameters vanish ($\alpha_k \to 0$ and $\varepsilon_k \to 0$). In addition, since the number of sets of inequalities is finite, there exists a set of inequalities $\mathcal{I}$ that is visited an infinite number of times. With this setting, the following theorem shows that the bounds F_k converge (Property *(i)*) and that we know the limit (Property *(ii)*), under a technical boundedness assumption. This result is related to Theorem 4 of [Malick and Roupin 2013] (which is a theoretical convergence of ideal bounds under a strict feasibility assumption) and Theorem 1 of [Krislock et al. 2014] (which is a similar result for the specific version of *BiqCrunch* for Max-Cut).

THEOREM 4.1. *Let $(X_k, \lambda_k, \mu_k, \nu_k, \mathcal{I}_k, F_k)_k$ be the sequence of iterates generated by the bounding algorithm. Let $\mathcal{I}$ be a set of inequalities such that there exist infinitely many $\mathcal{I}_k = \mathcal{I}$. Assume that the feasible set of the binary quadratic optimization problem* (1) *in nonempty, so that there exists an optimal solution to* (1)*. Then the following properties hold:*

(i) The sequence of the bounds $(F_k)_k$ converges to $\bar{F}$, which is a bound for the optimal value of (9)*, and a subsequence of the primal iterates $(X_k)_k$ converges to $\bar{X}$, which is feasible for* (9)*.*

(ii) If moreover the sequence of dual variables $(\lambda_k, \mu_k, \nu_k)_k$ is bounded, then $\bar{F}$ is the optimal value of (9)*, and $\bar{X}$ is an optimal solution of* (9)*.*

PROOF. The assumption that the feasible set of the initial problem (1) is nonempty yields that the feasible sets of its reformulations (6) and (7) and its relaxations (8) and (9) are nonempty as well. For this proof, we denote by K the (infinite) set of indexes such that $\mathcal{I}_k = \mathcal{I}$ for $k \in K$.

Let us start the proof of Property *(i)* by noting that, from (17), the diagonal entries of X_k for all k are bounded by ε_1:

$$\|e - \operatorname{diag}(X_k)\|_\infty \leq \|b - B(X_k)\|_\infty < \varepsilon_k \leq \varepsilon_1 \quad \text{for all } k. \tag{19}$$

Since $X_k \succeq 0$, the determinant of its submatrix with indices $\{i, j\}$ is nonnegative:

$$\det \begin{pmatrix} (X_k)_{ii} & (X_k)_{ij} \\ (X_k)_{ij} & (X_k)_{jj} \end{pmatrix} = (X_k)_{ii}(X_k)_{jj} - (X_k)_{ij}^2 \geq 0.$$

By (19), the diagonal entries $(X_k)_{ii}$ and $(X_k)_{jj}$ lie between $[1-\varepsilon_1, 1+\varepsilon_1]$, and therefore we have $(X_k)_{ij}^2 \leq (1+\varepsilon_1)^2$. The norm of X_k is thus bounded:

$$\|X_k\|_F^2 = \sum (X_k)_{ij}^2 \leq (1+\varepsilon_1)^2(n+1)^2 \quad \text{for all } k. \tag{20}$$

The boundedness of the subsequence $(X_k)_{k\in K}$ implies that we can further extract a converging subsequence; we denote its limit by $\bar{X}$. The closedness of the set of positive semidefinite matrices yields that $\bar{X} \succeq 0$. Notice also that (17) implies $B\left(\bar{X}\right) - b = 0$, $[A\left(\bar{X}\right) - a]_- = 0$, and $[e + A_{\mathcal{I}}\left(\bar{X}\right)]_- = 0$, since $\varepsilon_k \to 0$. Thus we have that the limit matrix $\bar{X}$ is feasible for (9).

Let us now turn to the other part of Property *(i)*, that $(F_k)_k$ converges to a bound of (9). Fix k; we are going to show first that F_{k+1} cannot be significantly larger than F_k. Start by observing in Algorithm 2 that we have $F_{k+1} = F_{\mathcal{I}_k}^{\alpha_{k+1}}(\lambda_{k+1}, \mu_{k+1}, \hat{\nu}_{k+1})$,

by definition of ν_k and $\mathcal{I}_k$. This implies that $F_{k+1} \leq F_{\mathcal{I}_k}^{\alpha_{k+1}}(\lambda_k, \mu_k, \nu_k)$, since the quasi-Newton algorithm with Wolfe line-search can only decrease the objective value (see [Bonnans et al. 2003, Chap.1]). Using the definition (12) of the bounds, we then write

$$\begin{aligned} F_{k+1} &\leq \frac{1}{\alpha_{k+1}} \left\| X_{\mathcal{I}_k}(\lambda_k, \mu_k, \nu_k) \right\|^2 /2 + a^T \lambda_k + b^T \mu_k + e^T \nu_k + \alpha_{k+1}(n+1)^2/2 \\ &= F_{\mathcal{I}_k}^{\alpha_k}(\lambda_k, \mu_k, \nu_k) + \left(\frac{1}{\alpha_{k+1}} - \frac{1}{\alpha_k} \right) \left\| X_{\mathcal{I}_k}(\lambda_k, \mu_k, \nu_k) \right\|^2 /2 + (\alpha_{k+1} - \alpha_k)(n+1)^2/2 \\ &= F_k + \left(\frac{\alpha_k^2}{\alpha_{k+1}} - \alpha_k \right) \left\| X_k \right\|^2 /2 + (\alpha_{k+1} - \alpha_k)(n+1)^2/2 \end{aligned}$$

If $\alpha_{k+1} = \alpha_k$, this inequality is simply $F_{k+1} \leq F_k$. If $\alpha_{k+1} = \texttt{scaleAlpha} \cdot \alpha_k$, this reads

$$F_{k+1} \leq F_k + \frac{\alpha_k}{2} \frac{(1 - \texttt{scaleAlpha})}{\texttt{scaleAlpha}} \left(\left\| X_k \right\|^2 - \texttt{scaleAlpha}(n+1)^2 \right).$$

In both cases, this yields, using again (20),

$$F_{k+1} \leq F_k + C\alpha_k \quad \text{with } C = \frac{1}{2} \frac{(1 - \texttt{scaleAlpha})}{\texttt{scaleAlpha}} (n+1)^2 \big((1+\varepsilon_1)^2 - \texttt{scaleAlpha} \big) > 0. \tag{21}$$

This bound on the growth of F_k enables us to argue, as follows, that the sequence converges. Let us repeat the above bounding for $\ell > k$: let $k_1, \ldots, k_p$ be the p indices $k \leq k_i < k + \ell$ such that $\alpha_{k_i+1} = \texttt{scaleAlpha} \cdot \alpha_{k_i}$; from repeated application of inequality (21), and using the fact that $F_{k+1} \leq F_k$ when $\alpha_{k+1} = \alpha_k$, we obtain

$$\begin{aligned} F_{k+\ell} &\leq F_k + C \left(\alpha_{k_1} + \alpha_{k_2} + \cdots + \alpha_{k_p} \right) \\ &= F_k + C \left(\alpha_k + \texttt{scaleAlpha} \cdot \alpha_k + \cdots + \texttt{scaleAlpha}^{p-1} \cdot \alpha_k \right) \\ &\leq F_k + C \left(\frac{1}{1 - \texttt{scaleAlpha}} \right) \alpha_k. \end{aligned}$$

Taking $\ell \to +\infty$ and then $k \to +\infty$ above, we get $\limsup_{k \to +\infty} F_k \leq \liminf_{k \to +\infty} F_k$, hence the sequence $(F_k)_k$ converges; let us call its limit $\bar{F}$.

Recall now that Theorem 3.1 implies that F_k, for all $k \in K$, is an upper bound for (9) (since $\mathcal{I}_k = \mathcal{I}$ for $k \in K$). Since $\bar{F}$ is obviously also the limit of the subsequence $(F_k)_{k \in K}$, $\bar{F}$ is an upper bound for (9) as well. Thus we have Property *(i)*:

$$\langle Q, \bar{X} \rangle \leq \text{the optimal value of (9)} \leq \bar{F}. \tag{22}$$

We prove now Property *(ii)*. We start by observing that, for a given k, we have by (10)

$$\langle Q - A^*(\lambda_k) - B^*(\mu_k) + A_{\mathcal{I}}^*(\nu_k), X_k \rangle = \alpha_k \left\| X_k \right\|^2$$

which in turn yields

$$\begin{aligned} \langle Q, X_k \rangle &= \alpha_k \left\| X_k \right\|^2 + \langle A^*(\lambda_k), X_k \rangle + \langle B^*(\mu_k), X_k \rangle - \langle A_{\mathcal{I}}^*(\nu_k), X_k \rangle \\ &= \alpha_k \left\| X_k \right\|^2 + \lambda_k^T A(X_k) + \mu_k^T B(X_k) - \nu_k^T A_{\mathcal{I}}(X_k). \end{aligned}$$

Combining this equation with (18), we get

$$F_k - \langle Q, X_k \rangle = \frac{\alpha_k}{2} ((n+1)^2 - \left\| X_k \right\|^2) + \lambda_k^T (a - A(X_k)) + \mu_k^T (b - B(X_k)) + \nu_k^T (e + A_{\mathcal{I}}(X_k)). \tag{23}$$

Notice that the three inner products in the above equation can be bounded with (17) as follows:

$$|\lambda_k^T(a - A(X_k))| \leq \|\lambda_k\|\,\varepsilon_k, \quad |\mu_k^T(b - B(X_k))| \leq \|\mu_k\|\,\varepsilon_k, \text{ and } |\nu_k^T(e + A_{\mathcal{I}}(X_k))| \leq \|\nu_k\|\,\varepsilon_k.$$

We use now the additional assumption that the sequence $(\lambda_k, \mu_k, \nu_k)_k$ is bounded and we conclude that the three terms vanish when $k \to +\infty$. Recall (20) which implies that the term $\frac{\alpha_k}{2}((n+1)^2 - \|X_k\|^2)$ also goes to zero when $k \to +\infty$. Therefore, we can pass to the limit in (23) when $k \to +\infty$ with $k \in K$ and we get $\bar{F} = \langle Q, \bar{X} \rangle$. Therefore, by equation (22), $\bar{F}$ is the optimal value of (9) and $\bar{X}$ is optimal. □

Property *(ii)* of this theorem says that, under a boundedness assumption, the bounding procedure eventually solves the SDP relaxation (9) as F_k approximates the optimal value and X_k approximates an optimal solution. Thus this result theoretically supports what we observe in practice: once a "good" set of inequalities is "identified," the algorithm solves the corresponding SDP relaxation. However, the bounding procedure is not meant to be just another SDP solver: it combines fast initial iterations (α_k large for small k) and the ability to gain more and more tightness (α_k small for large k). The bounding procedure is therefore primarily designed to compute efficient bounds inside a branch-and-bound routine; solving the SDP relaxation to optimality is not necessary.

It turns out that the bounding procedure has good observed convergence and returns high-quality bounds within a reasonable amount of time. For the numerical illustrations, we refer to experiments with specialized versions of *BiqCrunch* for Max-Cut in [Krislock et al. 2014] and for Max-k-Cluster in [Krislock et al. 2016].

5. IMPROVING THE PERFORMANCE OF *BIQCRUNCH*

As described in the previous section, *BiqCrunch* can theoretically solve any binary quadratic problem. In practice we can improve the performance of *BiqCrunch* for specific problems by:

(1) adjusting the parameters of *BiqCrunch*,
(2) providing specific heuristics to produce better feasible solutions,
(3) strengthening the problem formulation to obtain better upper bounds.

In the *BiqCrunch* package, we have provided different versions of *BiqCrunch*, each of which has been adapted to solve specific problems with tailored heuristics and parameter files. In the rest of this section we discuss each of the above three items.

5.1. *BiqCrunch* parameters

The parameters of *BiqCrunch* are listed in Table I. These parameters are specified in a `biq_crunch.param` file that must be provided when running the solver. *BiqCrunch* provides parameter files with the default parameters, as well as parameter files that have been adjusted for Max-Cut, Max-k-Cluster, and Max-Independent-Set.

For most problems, these parameters do not need to be modified. Nevertheless, some of them are crucial to the performance of *BiqCrunch* for specific instances. The most important parameter is `alpha0` which determines the initial value of α_k in Algorithm 1. For problems that do not require a semidefinite approach to obtain good bounds (for instance when linear programming relaxations are known to be efficient), `alpha0` could be set to a higher value to reduce the computing time when evaluating each node. For more difficult problems (when weak relaxations are not efficient), `alpha0` should be set to a lower value to have tighter initial bounds when evaluating each node.

The `gapCuts` and `cuts` parameters are also important since they can be adjusted to find the right trade-off between adding too many or too few cuts. Typically, we want to avoid adding many cuts that are only violated by a small amount. By default

Table I. *BiqCrunch* main parameters

parameter	definition / role	default value
alpha0	initial value of α	1e-1
scaleAlpha	scaling value of α	0.5
minAlpha	minimum value of α	5e-5
tol0	initial value of tolerance ε for L-BFGS-B	1e-1
scaleTol	scaling value of tolerance ε for L-BFGS-B	0.95
minTol	minimum value of the tolerance ε for L-BFGS-B	1e-2
nitermax	maximum number of iterations per call of L-BFGS-B	2000
minNiter	minimum number of L-BFGS-B calls	12
maxNiter	maximum number of L-BFGS-B calls	100
maxNAiter	maximum number of L-BFGS-B calls with fixed α	50
withCuts	use the triangle inequalities	1
gapCuts	minimum violation of added cuts (inequalities)	-5e-2
cuts	maximum number of cuts to add per iteration	500
minCuts	minimum number of violated cuts to reduce α and ε	50
scaling	pre-scale the constraints	1
heur_1	use the root-node heuristic	1
heur_2	use the bound heuristic	1
heur_3	use the node heuristic	1
seed	random number generator seed	2016
local_search	use the local search	1
branchingStrategy	0: Branch on least-fractional variable 1: Branch on most-fractional variable 2: Branch on variable that is closest to one	1
root	just evaluate root node (no branch-and-bound)	0
time_limit	limit on computing time (in seconds)	0 (*i.e.*, no time limit)
soln_value_provided	user is providing a known feasible solution value	0
soln_value	the value of a known feasible solution	0

BiqCrunch only adds at most cuts $= 500$ triangle inequalities each iteration that each have a violation of at most gapCuts $= -0.05$.

We recommend to users who are looking for better performance to adjust the three key parameters (alpha0, gapCuts, and cuts) in the following way: set the "root" parameter to 1 and use the verbose command-line option ("-v 1"), then do tests with different instances of your problem and inspect the output files. A useful rule of thumb is that if the values of the parameters nitermax or cuts are reached when evaluating the root node then the three key parameters should be adjusted accordingly.

5.2. Problem-specific heuristics

The generic heuristic (described in Section 4.2) can be substituted with heuristics tailored for specific problems. In the *BiqCrunch* directory, there are several "problems/<PROBLEM>" folders for different optimization problems, and a "problems/user" directory where users can create their own heuristics. For a new heuristic to be called by *BiqCrunch*, one just has to create a directory in the problems/ directory that contains their heur.c file; upon compiling *BiqCrunch*, a biqcrunch executable will be created in the location of the heur.c file. An example heur.c is given in the problems/user directory.

5.3. Strengthening bounds with additional constraints

BiqCrunch does not perform any reformulation or preprocessing of the input problem. The user has complete control over the formulation of their problem. This allows users to try different formulations of the same problem, such as adding redundant constraints to strengthen the semidefinite relaxation and obtain tighter bounds.

Adding linear or quadratic constraints that are redundant for the binary quadratic problem (1) does not change its set of optimal solutions, nor its optimal value, but may

result in tighter bounds. This is because, with each additional constraint the space of dual multipliers Ω increases, resulting in possibly smaller upper bounds of problem (1). In this section, we discuss a set of strengthening constraints that we recommend adding to the formulation of a problem to be solved by *BiqCrunch*.

Suppose problem (1) has a linear constraint $s^T z = a$. For instance, the $\sum_{i=1}^{n} z_i = k$ constraint in the Max-k-Cluster problem is an example of such a linear constraint. The *product constraints* are the valid quadratic constraints generated from $s^T z = a$:

$$z_i s^T z - z_i a = 0, \qquad i = 1, \ldots, n.$$

Introducing quadratic constraints by multiplication is a well-known technique; see [Sherali and Adams 1990] for the general approach and [Lovász and Schrijver 1991] for the semidefinite case. It was shown [Faye and Roupin 2007] that adding any number of redundant quadratic constraints results in semidefinite bounds that are never better than the one obtained by adding these product constraints (see also [Helmberg et al. 2000] for an early study of this question). These product constraints therefore form an optimal set of redundant quadratic constraints.

In practice, adding these constraints to the formulation of problem (1) often significantly improves the tightness of the bounds computed by *BiqCrunch* and reduces the overall computing time. As an illustration, we consider solving a problem with $n = 20$ binary variables, a random quadratic objective function, and the cardinality constraint $\sum_{i=1}^{n} z_i = 10$. First we solve the problem without the product constraints.

```
$ tools/lp2bc.py randprob.lp > randprob.bc
$ problems/generic/biqcrunch randprob.bc biq_crunch.param
Output file: randprob.bc.output
Input file:  randprob.bc
Parameter file: biq_crunch.param
Node 0 Feasible solution 30
Node 1 Feasible solution 95
Node 1 Feasible solution 108
Node 2 Feasible solution 109
Nodes = 27
Root node bound = 113.54
Maximum value = 109
Solution = { 2 3 4 5 8 11 12 13 14 19 }
CPU time = 0.2050 s
```

Next we solve the same problem after having added the product constraints.

```
$ tools/lp2bc.py randprob_prod.lp > randprob_prod.bc
$ problems/generic/biqcrunch randprob_prod.bc biq_crunch.param
Output file: randprob_prod.bc.output
Input file:  randprob_prod.bc
Parameter file: biq_crunch.param
Node 0 Feasible solution 30
Node 1 Feasible solution 109
Nodes = 1
Root node bound = 109.96
Maximum value = 109
Solution = { 2 3 4 5 8 11 12 13 14 19 }
CPU time = 0.0169 s
```

We notice that the root node bound is much tighter with the product constraints. In this case, the root node bound was tight enough to be able to solve the problem without

branching. On the other hand, without the product constraints, 27 nodes of the branch-and-bound search tree are visited before solving the problem. Including such product constraints often significantly improves the performance of *BiqCrunch*.

6. CONCLUSION

In this paper, we have introduced *BiqCrunch*, an exact solver for general binary quadratic problems. The main feature of *BiqCrunch* is its ability to dynamically set the tightness of its bounding procedure (node by node), using adjustable semidefinite bounds. The bounding procedure automatically adjusts from cheap/poor bounds to expensive/good bounds as needed.

Since *BiqCrunch* uses high-quality bounds, the number of nodes visited throughout the branch-and-bound process is relatively small. Thus, *BiqCrunch* can perform well on problems which are difficult to solve by methods based on linear bounds. *BiqCrunch* complements other exact methods by expanding on the types of problems that we can now efficiently solve. *BiqCrunch* also complements heuristic methods by providing tight bounds that give an accurate measure of the suboptimality of the solutions generated by such methods. *BiqCrunch* can also benefit from high-quality heuristic solutions since having such solutions can further reduce the number branch-and-bound nodes visited.

The source code for *BiqCrunch* is now publicly available. We hope it is a valuable resource to those interested in solving binary quadratic problems. With feedback from the community, we look forward to continuing to improve the code and expanding the range of problems that can be efficiently solved by *BiqCrunch*. In particular, we aim at exploiting structural properties of the problems and reducing the cost induced by the eigenvalue decompositions computed during the bounding procedure.

ACKNOWLEDGMENTS

The authors are deeply indebted to three anonymous referees whose knowledgable comments and feedback led to major improvements to both this paper and the *BiqCrunch* code.

REFERENCES

T. Achterberg. 2009. SCIP: Solving constraint integer programs. *Mathematical Programming Computation* 1, 1 (2009), 1–41.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

M. Anjos and J.B. Lasserre. 2012. *Handbook of semidefinite, conic and polynomial optimization*. Springer.

M. Anjos and A. Vannelli. 2008. Computing Globally Optimal Solutions for Single-Row Layout Problems Using Semidefinite Programming and Cutting Planes. *INFORMS Journal on Computing* 20, 4 (2008), 611–617.

M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin. 2012. LP and SDP branch-and-cut algorithms for the minimum graph bisection problem: a computational comparison. *Mathematical Programming Computation* 4, 3 (2012), 275–306.

F. Barahona, M. Jünger, and G. Reinelt. 1989. Experiments in quadratic 0–1 programming. *Mathematical Programming* 44, 1 (1989), 127–137.

A. Billionnet and S. Elloumi. 2007. Using a Mixed Integer Quadratic Programming Solver for the Unconstrained Quadratic 0-1 Problem. *Mathematical Programming* 109 (2007), 55–68. Issue 1. 10.1007/s10107-005-0637-9.

A. Billionnet, S. Elloumi, and M.-C. Plateau. 2009. Improving the performance of standard solvers for quadratic 0-1 programs by a tight convex reformulation: The QCR method. *Discrete Applied Mathematics* 157, 6 (2009), 1185–1197.

J. Bolte, A. Daniilidis, and A.S. Lewis. 2008. Tame functions are semismooth. *Math. Program.* 117, 1 (2008), 5–19.

P. Bonami, L. Biegler, A. Conn, G. Cornuéjols, I. Grossmann, C. Laird, J. Lee, A. Lodi, F. Margot, and N. Sawaya. 2008. An algorithmic framework for convex mixed integer nonlinear programs. *Discrete Optimization* 5, 2 (2008), 186–204.

J.F. Bonnans, J.Ch. Gilbert, C. Lemaréchal, and C. Sagastizábal. 2003. *Numerical Optimization*. Springer Verlag.

S. Burer and A. Letchford. 2012. Non-convex mixed-integer nonlinear programming: A survey. *Surveys in Operations Research and Management Science* 17, 2 (2012), 97 – 106.

M. Bussieck, S. Vigerske, J. Cochran, L. Cox, P. Keskinocak, J. Kharoufeh, and J. Smith. 2010. *MINLP Solver Software*. John Wiley, Inc. Updated Feb 21, 2012.

C. D'Ambrosio and A. Lodi. 2011. Mixed integer nonlinear programming tools: a practical overview. *4OR* 9, 4 (2011), 329–349.

M. Deza and M. Laurent. 1997. *Geometry of Cuts and Metrics*. Algorithms and Combinatorics, Vol. 15. Springer, Berlin.

A. Engau, M. F. Anjos, and A. Vannelli. 2012. On handling cutting planes in interior-point methods for solving semi-definite relaxations of binary quadratic optimization problems. *Optimization Methods and Software* (2012), 1–21.

A. Faye and F. Roupin. 2007. Partial Lagrangian relaxation for general quadratic programming. *4OR: A Quarterly Journal of Operations Research* 5 (2007), 75–88.

L. Galli and A. Letchford. 2014. A compact variant of the QCR method for quadratically constrained quadratic 01 programs. *Optimization Letters* 8, 4 (2014), 1213–1224.

Tristan Gally, Marc E. Pfetsch, and Stefan Ulbrich. 2016. *A Framework for Solving Mixed-Integer Semidefinite Programs*. Technical Report. Technische Universität Darmstadt.

M.X. Goemans and D. Williamson. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM* 42, 6 (Nov. 1995), 1115–1145.

C. Helmberg and F. Rendl. 1998. Solving quadratic (0,1)-problems by semidefinite programs and planes. *Mathematical Programming* 82, 3 (1998), 291–315.

C. Helmberg, F. Rendl, and R. Weismantel. 2000. A Semidefinite Programming Approach to the Quadratic Knapsack Problem. *Journal of Combinatorial Optimization* 4, 2 (2000), 197–215.

N. Higham. 1988. Computing a nearest symmetric positive semidefinite matrix. *Linear Algebra Appl.* 103 (1988), 103–118.

J.-B. Hiriart-Urruty and C. Lemaréchal. 2001. *Fundamentals of Convex Analysis*. Springer Verlag, Heidelberg.

J.-B. Hiriart-Urruty and J. Malick. 2012. A Fresh Variational-Analysis Look at the Positive Semidefinite Matrices World. *Journal of Optimization Theory and Applications* 153, 3 (2012), 551–577.

L.D. Iasemidis, P. Pardalos, J.C. Sackellares, and D.-S. Shiau. 2001. Quadratic Binary Programming and Dynamical System Approach to Determine the Predictability of Epileptic Seizures. *Journal of Combinatorial Optimization* 5 (2001), 9–26.

A. Joulin, F. Bach, and J. Ponce. 2010. Discriminative clustering for image co-segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. 1943–1950.

N. Krislock, J. Malick, and F. Roupin. 2014. Improved semidefinite bounding procedure for solving Max-Cut problems to optimality. *Mathematical Programming* 143, 1-2 (2014), 61–86.

N. Krislock, J. Malick, and F. Roupin. 2016. Computational results of a semidefinite branch-and-bound algorithm for k-cluster. *Computers and Operations Research* 66 (2016), 153–159.

M. Laurent and S. Poljak. 1995. On a positive semidefinite relaxation of the cut polytope. *Linear Algebra Appl.* 223224 (1995), 439 – 461. DOI:http://dx.doi.org/10.1016/0024-3795(95)00271-R

M. Laurent and S. Poljak. 1996. On the Facial Structure of the Set of Correlation Matrices. *SIAM J. Matrix Anal. Appl.* 17, 3 (1996), 530–547.

B. Le Cun, C. Roucairol, and The Pnn Team. 1995. *BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms*. Technical Report. Laboratoire Prism.

C. Lemaréchal and F. Oustry. 1999. *Semidefinite relaxations and Lagrangian duality with application to combinatorial optimization*. Rapport de Recherche 3710. INRIA.

F. Liers, M. Jünger, G. Reinelt, and G. Rinaldi. 2005. *Computing Exact Ground States of Hard Ising Spin Glass Problems by Branch-and-Cut*. Wiley-VCH Verlag GmbH & Co. KGaA, 47–69.

L. Lovász and A. Schrijver. 1991. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal on Optimization* 1, 2 (1991), 166–190.

J. Malick. 2007. Spherical constraint in Boolean quadratic programming. *Journal of Global Optimization* 39, 4 (2007).

J. Malick and F. Roupin. 2013. On the bridge between combinatorial optimization and nonlinear optimization: a family of semidefinite bounds for 01 quadratic problems leading to quasi-Newton methods. *Mathematical Programming* 140, 1 (2013), 99–124.

Sonja Mars. 2013. *Mixed-Integer Semidefinite Programming with an Application to Truss Topology Design*. Ph.D. Dissertation. Technische Universitat Darmstadt, Germany.

J. Morales and J. Nocedal. 2011. Remark on "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization". *ACM Trans. Math. Softw.* 38, 1 (2011), 1–4.

S. Poljak, F. Rendl, and H. Wolkowicz. 1995. A recipe for semidefinite relaxation for (0,1)-quadratic programming. *Journal of Global Optimization* 7 (1995), 51–73.

L.Q. Qi and J. Sun. 1993. A nonsmooth version of Newton's method. *Mathematical Progamming* 58, 3 (1993), 353–367.

F. Rendl, G. Rinaldi, and A. Wiegele. 2010. Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming* 121 (2010), 307–335.

F. Roupin. 2004. From Linear to Semidefinite Programming: An Algorithm to Obtain Semidefinite Relaxations for Bivalent Quadratic Problems. *Journal of Combinatorial Optimization* 8, 4 (2004), 469–493.

N. V. Sahinidis. 2013. *BARON 12.1.0: Global Optimization of Mixed-Integer Nonlinear Programs,* User's Manual.

R. Saigal, L. Vandenberghe, and H. Wolkowicz. 2000. *Handbook of Semidefinite Programming*. Kluwer.

H. Sherali and W. Adams. 1990. A Hierarchy of Relaxations between the Continuous and Convex Hull Representations for Zero-One Programming Problems. *SIAM Journal on Discrete Mathematics* 3, 3 (1990), 411–430. http://link.aip.org/link/?SJD/3/411/1

N.Z. Shor. 1987. Quadratic optimization problems. *Sovie Journal of Computer and Systems Sciences* 25 (1987), 1–11.

M. Yamashita, K. Fujisawa, M. Fukuda, K. Kobayashi, K. Nakata, and M. Nakata. 2012. Latest Developments in the SDPA Family for Solving Large-Scale SDPs. In *Handbook on Semidefinite, Conic and Polynomial Optimization*, Miguel F. Anjos and Jean B. Lasserre (Eds.). International Series in Operations Research & Management Science, Vol. 166. Springer US, 687–713.

Q. Zhao, S. Karisch, F. Rendl, and H. Wolkowicz. 1998. Semidefinite Programming Relaxations for the Quadratic Assignment Problem. *Journal of Combinatorial Optimization* 2, 1 (1998), 71–109.

C. Zhu, R. Byrd, P. Lu, and J. Nocedal. 1997. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.* 23, 4 (Dec. 1997), 550–560.