The Reformulation-Optimization Software Engine
Leo Liberti, Sonia Cafieri, David Savourey

To cite this version:


HAL Id: hal-00979214
https://hal-enac.archives-ouvertes.fr/hal-00979214
Submitted on 23 Feb 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
The Reformulation-Optimization Software Engine

Leo Liberti\*, Sonia Cafieri\(^2\), and David Savourey\(^1\)

\(^1\) LIX, \`Ecole Polytechnique, Palaiseau, France, 
\{liberti,savourey\}@lix.polytechnique.fr
\(^2\) Dept. Mathématiques et Informatique, ENAC, 7 av. E. Belin, 31055 Toulouse, France, sonia.cafieri@enac.fr

Abstract. Most optimization software performs numerical computation, in the sense that the main interest is to find numerical values to assign to the decision variables, e.g. a solution to an optimization problem. In mathematical programming, however, a considerable amount of symbolic transformation is essential to solving difficult optimization problems, e.g. relaxation or decomposition techniques. This step is usually carried out by hand, involves human ingenuity, and often constitutes the “theoretical contribution” of some research papers. We describe a Reformulation-Optimization Software Engine (ROSE) for performing (automatic) symbolic computation on mathematical programming formulations.

Keywords: reformulation, MINLP.

1 Introduction

The aim of this paper is to describe a new optimization software called Reformulation-Optimization Software Engine (ROSE). Its main purpose is to allow the symbolic analysis and reformulation of Mathematical Programs (MP), although ROSE can also interface with numerical solvers. In practice, ROSE is used either as a pre-processor or is called iteratively within numerical solvers; it can be used either stand-alone or as an AMPL [1] solver. ROSE addresses MPs in the following very general form:

\[
\begin{align*}
\min & \quad f(x) \\
& \quad g^L \leq g(x) \leq g^U \\
& \quad x^L \leq x \leq x^U \\
& \quad \forall i \in \mathbb{Z} \quad x_i \in \mathbb{Z},
\end{align*}
\]

(1)

\* Supported by grants: ANR 07-JCJC-0151 “ARS”, Digiteo 2009-14D “RMNCCO”, Digiteo 2009-55D “ARM”. We acknowledge the contributions of Dr. C. D’Ambrosio (University of Bologna) and of Mr. P. Janes (Australian National University); to appear in an LNCS volume containing the Proceedings of the International Congress of Mathematical Software, 2010.

\** Corresponding author.
where \( x \) is a vector of \( n \) decision variables, \( x^L, x^U \in \mathbb{R}^n \), \( Z \subseteq \{1, \ldots, n\} \), \( g^L, g^U \in \mathbb{R}^m \), \( f : \mathbb{R}^n \to \mathbb{R} \) and \( g : \mathbb{R}^n \to \mathbb{R}^m \). MPs in the form (1) are known as Mixed-Integer Nonlinear Programs (MINLP). The restriction on \( f, g \) is that they should be representable as strings of a certain formal language (more details in Sect. 2 below).

Changing the formal description of optimization problems has an impact on the applicability and efficiency of the corresponding solution methods. Difficult problems are routinely decomposed, relaxed or transformed into simpler subproblems that we know how to solve efficiently, and which preserve some of the interesting mathematical properties of the original problem. Such transformations, called reformulations, almost always depend on the “mathematical structure” of the problem. Considering MP as a formal language, each formulation is a valid sentence in the language. A reformulation is a sequence of some basic symbolic transformations (such as add, modify or delete a variable, an objective or a constraint). In order to be useful, a reformulation must preserve some mathematical property: for example, all optima of the reformulation might be required to be also optima of the original formulation. Since the basic “atomic” reformulations (adding, modifying, deleting a formulation element) are in principle easy to conceive and implement, the absence of a generic software package for carrying out automatic reformulations in MP might come as a surprise. ROSE moves a few steps in this direction, providing a set of reformulators that can act on MP formulations. The roadmap for ROSE is to facilitate the implementation of a heuristically driven search for the best reformulation for a given solver.

ROSE consists of around 50K lines of GNU C++ code and is covered by the Common Public License (CPL). We are currently preparing its distribution through COIN-OR [2] and finalizing documentation and examples. At the moment the software can be obtained through http://www.lix.polytechnique.fr/~liberti/rose.tar.gz. This paper announces the first public distribution of ROSE, which provides symbolic (as opposed to numerical) methods for manipulating MPs. Currently, ROSE can perform basic and complex symbolic analysis and manipulation tasks on all formulation elements, including all expressions appearing in objective(s) and constraints in (1). These tasks have been put together in higher-level reformulation solvers, e.g. writing a (linear) convex relaxation of a MINLP automatically [3, 4]; writing a DAG representation of an AMPL-encoded MINLP [5, 4]; writing a cdd [6] or Porta [7] representation of an AMPL-encoded LP. A list of applications of ROSE is given in Sect. 5.

The rest of this paper is organized as follows. We review existing work in Sect. 1.1, give two motivating examples in Sect. 1.2, survey the theory that ROSE is built on in Sect. 2, explain ROSE’s architecture in Sect. 3, show how ROSE helps solving the motivating examples in Sect. 4 and discuss ROSE’s main applications in Sect. 5, which concludes the paper.

1.1 Existing work

Currently, optimization software focuses on solvers (implementations of solution algorithms), each of which includes the necessary layers of reformulation capa-
bilities. For example, all spatial Branch-and-Bound (sBB) MINLP solvers are able to construct a convex relaxation automatically [8, 4].

Solvers typically require their input in a non-quantified format: complex jagged arrays of variables and constraints must be transformed into flat lists thereof. This creates the need for “translators” that automatically convert quantified constraints to flat constraints. For example, \( \forall i \in \{1, 2, 3\} \sum_{j \neq i} x_i = 1 \) is converted to the flat form \( x_1 + x_2 = 1 \land x_1 + x_3 = 1 \land x_2 + x_3 = 1 \). Since different solvers read the flat form input according to different encodings, translators also include wrappers for most existing solvers, so that users can safely ignore the technicalities of the calling procedure. The two best known MP translators are AMPL and GAMS [9]: both optionally perform reformulations on the input MP before “flattening” it and passing it to the solver.

In general, the reformulation layers of existing solvers and translators cannot be accessed or modified by the user. Apart from ROSE we are aware of no user-accessible software for carrying out MP reformulations with such generality. Notwithstanding, at least two codes are available that perform symbolic analysis and reformulation to a certain extent. Dr. AMPL [10] is an analysis tool for MP formulations aimed to the automatic choice of an appropriate solver for the given formulation. The software described in [11] enriches the AMPL language with primitives for providing solvers with specific block-diagonal information about the problem.

1.2 Motivating examples

The need for a generic reformulation software layer is given by the mounting complexity of optimization software needed to solve ever more difficult problems.

Subproblems in sBB. In sBB, for example, a branching procedure constructs a search tree, each node of which represents a pair of reformulations \((Q, \bar{Q})\) of the original problem \(P\). The formulation \(Q\) is obtained from \(P\) by restricting the variable bounds; \(\bar{Q}\) is a relaxation of \(Q\) where each nonlinear term is replaced by linear lower and upper bounding functions. A possible sBB implementation might wish to solve \(Q\) using a MINLP heuristic and \(\bar{Q}\) using a MILP solver. In turn, the MINLP heuristic might alternate between solving a continuous Nonlinear Programming (NLP) reformulation and an auxiliary MILP reformulation of \(Q\), whereas the MILP solver is a standard BB algorithm which needs to call MILP heuristics and a Linear Programming (LP) solver such as the simplex algorithm. Testing new ideas in this complex calling chain often requires changing the reformulation algorithms, which is impossible as long as these are hard-coded into the solver.

The Kissing Number Problem. Given positive integers \(D, \bar{N}\), the KNP [12] asks for the maximum number (between 1 and \(\bar{N}\)) of spheres of unit radius that can be arranged in \(\mathbb{R}^D\) around a unit sphere centered in the origin so that their
interiors are disjoint. The MP formulation [13] is:
\[
\max \sum_{i \leq \bar{N}} y_i \quad \begin{cases} 
\forall i \leq \bar{N} \quad \|x_i\|_2^2 = 4y_i \\
\forall i < j \leq \bar{N} \quad \|x_i - x_j\|_2^2 \geq 4y_i y_j \\
\forall i \leq \bar{N} \quad x_i \in \mathbb{R}^D, \quad y_i \in \{0, 1\}.
\end{cases}
\] (2)

Attempting to solve (2) directly with a MINLP solver such as BARON [8] or Couenne [4] results in the trivial solution with \(y = 0\) standing for incumbent (i.e. best optimum so far) for several days of computation as soon as \(D \geq 3\) and \(\bar{N} \geq 13\). We dispense with binary variables by transforming (2) into the corresponding decision problem: can \(N \leq \bar{N}\) spheres be arranged around the central one? The MP formulation is:
\[
\forall i \leq N \quad \|x_i\|_2^2 = 4 \quad \land \quad \forall i < j \leq N \quad \|x_i - x_j\|_2^2 \geq 4.
\] (3)

If (3) has a solution, then the instance \((D, N)\) is a YES one. Since both BARON and Couenne identify a feasible solution by calling a local NLP subsolver (e.g. SNOPT [14]), both are only as reliable as the subsolver. Computational experience shows that most local NLP solvers have difficulties in finding a local optimum of a heavily nonlinear MP if no feasible starting point is supplied. Again, days of computation will not yield any solution even for small instances. Inserting a tolerance to feasibility improves this situation:
\[
\max_{x, \alpha \in [0, 1]} \alpha \quad \text{s.t.} \quad \forall i \leq N \quad \|x_i\|_2^2 = 4 \quad \land \quad \forall i < j \leq N \quad \|x_i - x_j\|_2^2 \geq 4\alpha.
\] (4)

As shown in [12], (4) is computationally amenable to local NLP solution within a heuristic Global Optimization (GO) solver such as Variable Neighbourhood Search (VNS). Because of the large number of symmetric optima, however, sBB solvers are still far from finding any nontrivial solution. A study of the formulation group of (4) suggests adjoining the symmetry breaking constraints \(\forall i \leq N \setminus \{1\} \quad x_i - 1 \leq x_{i+1}\) to (4), yielding a reformulation for which sBB makes considerably more progress [15]. Identifying this reformulation chain, which leads to a more easily solvable MP, required considerable effort and resources. ROSE alleviates the situation by providing a uniform C++ interface to several reformulation needs. It is interesting to remark that other types of reformulations were recently instrumental in solving some high dimensional KNP instances [16].

2 Reformulations: formal definitions

We define MPs as valid sentences of a certain formal language. Instead of giving its syntax, i.e. the explicit grammar of this language (see the Appendix to [1] for an example), we describe the image of its semantic function, i.e. the data structure needed to encode a MP.

A parameter is a real number \(p\) (in its floating point computer representation). A decision variable is a symbol \(x_i\), indexed by some positive integer \(i\). Consider a finite set \(O\) of operators \(\{\oplus_1, \oplus_2, \ldots\}\) of given arities. An expression is defined recursively as follows:
1. parameters are expressions;
2. decision variables are expressions;
3. if \(e_1, \ldots, e_k\) are expressions and \(\oplus \in O\) has arity \(k\), then \(\oplus(e_1, \ldots, e_k)\) is an expression.

Let \(E\) be the set of all such expressions. We remark that each expression \(e(p, x) \in E\) involving parameters \(p = (p_1, \ldots, p_l)\) and decision variables \(x = (x_1, \ldots, x_n)\) corresponds to a function \(f_e(p, x)\), which associates to \(x\) the evaluation of \(e(p, \cdot)\) at \(x\). An objective function is a pair \((d, e) \in \{-1, 1\} \times E\) where \(d\) is the optimization direction: \((-1, e(p, x))\) corresponds to \(\min f_e(p, x)\) and \((1, e(p, x))\) to \(\max f_e(p, x)\). A constraint is a triplet \((g^L_k, e, g^U_k) \in \mathbb{R} \times E \times \mathbb{R}\) encoding the double inequality \(g^L_k \leq f_e(p, x) \leq g^U_k\). A range constraint is a triplet \((x^L_i, x_i, x^U_i) \in \mathbb{R} \times E \times \mathbb{R}\) encoding the restriction \(x^L_i \leq x_i \leq x^U_i\). An integrality constraint is a positive integer \(i\) which encodes the restriction \(x_i \in \mathbb{Z}\). A mathematical program is a 7-tuple \((p, x, E, O, C, B, Z)\) such that for all \(e \in E\), \(e\) depends on no further parameters (resp. decision variables) than \(p\) (resp. \(x\)). \(O\) is a set of \(s\) objective functions \((d, e)\) with \(e \in E\). \(C\) is a set of \(m\) constraints \((g^L_k, e, g^U_k)\) with \(e \in E\). \(B\) is a set of \(n\) range constraints, and \(Z \subseteq \{1, \ldots, n\}\) is a set of integrality constraints. An element of any component set in the 7-tuple is also called an entity of the MP. Semidefinite and multilevel programming can be dealt with by letting constant and/or variables symbols range over sets of matrices or other mathematical programs.

2.1 Flat and structured MPs

MPs can be given either in structured form (i.e. by using quantifiers over indices) or flat form. Flat MPs are those corresponding to the definition of Sect. 2. We now define structured MPs.

Given a sequence \(I = \{I_i \subseteq \mathbb{N} \mid i \leq \alpha\}\) of finite subsets of integers and a multi-index \(i = (i_1, \ldots, i_\alpha)\) where \(i_\beta \in I_\beta\) for all \(\beta \leq \alpha\), a structured parameter \(p\) is a jagged array of (scalar) parameter symbols \(p_i\) (with \(i \in I\)) with an assigned (scalar) value \(p_i\). A structured decision variable \(x\) is defined similarly for scalar variable symbols \(x_i\). Structured expressions, resting on an operator set \(O'\) enriched with the quantifier operators \(\sum, \Pi\), are defined recursively similarly to flat expressions, but with parameters and variables replaced by their structured versions. A structured constraint is a triplet \((g^L_k, f_e(p_i, x_j), g^U_k)\) where all multi-indices \(i, j\) are universally quantified over some subsets of \(I\). Structured range and integrality constraints are defined similarly. A MP defined over structured entities is a structured MP. Given a structured MP \(P\) with multi-indices \(i_1, \ldots, i_\gamma\), ranging over set families \(I = \{I_1, \ldots, I_\gamma\}\), and the jagged array of values \(p\) to be assigned to all parameter symbols, a translator (such as AMPL or GAMS) is able to write a flat MP \(P\) corresponding to the triplet \((P, I, p)\). In general, an operator \(\oplus \in O'\) acts on structured entities in a componentwise fashion. Different operator semantics can be defined by simply adding new operators to \(O'\).
2.2 Flat reformulations

Reformulations may occur either at the flat or structured level. Because of a technical limitation of AMPL (i.e. the AMPL API only allows user access to the flat, rather than structured, MP), ROSE only performs flat reformulations; we therefore only define these. Structured reformulations would essentially require hooking reformulation primitives at the AMPL grammar parsing level.

Let \( \mathbb{M}P_F \) be the class of all flat MPs; for \( P \in \mathbb{M}P_F \) we denote the feasible region of \( P \) by \( \mathcal{F}(P) \), the set of local optima of \( P \) by \( \mathcal{L}(P) \) and the set of global optima of \( P \) by \( \mathcal{G}(P) \).

2.1 Definition

A flat reformulation is a relation \( \rightsquigarrow \) on \( \mathbb{M}P_F \) such that there exists a formula \( \psi \) with two free variables for which

\[
\forall P, Q \in \mathbb{M}P_F (P \rightsquigarrow Q \Rightarrow \psi(P, Q)).
\]

The invariance scope of \( \rightsquigarrow \) is the class \( S(\rightsquigarrow) \) of all \( \psi \) for which (5) holds.

We distinguish three remarkable types of reformulations.

1. **Exact reformulations**, denoted by \( \equiv \): \( S(\equiv) \) contains the formula “there is a function \( \varphi : \mathcal{F}(Q) \to \mathcal{F}(P) \) such that \( \varphi|_{\mathcal{L}(Q)} \) is onto \( \mathcal{L}(P) \) and \( \varphi|_{\mathcal{G}(Q)} \) is onto \( \mathcal{G}(P) \)”; 

2. **Narrowings**, denoted by \( \triangleright \): \( S(\triangleright) \) contains the formula “there is a function \( \varphi : \mathcal{F}(Q) \to \mathcal{F}(P) \) such that \( \varphi(\mathcal{G}(Q)) \subseteq \mathcal{G}(P) \)”;

3. **Relaxations**, denoted by \( \succeq \): \( S(\succeq) \) contains the formula “\( \mathcal{F}(Q) \supseteq \mathcal{F}(P) \) and \( \mathcal{O}(Q) = \{(-1, e')\} \) and \( \mathcal{O}(P) = \{(-1, e)\} \) and, for all \( x \in \mathcal{F}(P) \), \( f_{e'}(p, x) \leq f_e(p, x) \)”.

2.2 Theorem ([17])

The relations \( \equiv, \triangleright, \succeq \) are all transitive. Furthermore, \( \equiv \subseteq \triangleright \) and \( \equiv \subseteq \succeq \).

Thus, if \( P \equiv Q_1 \triangleright Q_2 \) then \( P \triangleright Q_2 \); if \( P \equiv Q_1 \succeq Q_2 \) then \( P \succeq Q_2 \). This allows the construction of reformulation chains with invariant properties. We only consider reformulations corresponding to computable relations. A taxonomy of useful flat reformulations is given in [18].

2.3 Example

The PRODBinCont exact reformulation [18] replaces every product \( xy \) where \( x \in \{0, 1\} \) and \( y \in [y_L, y_U] \) with an added variable \( w \), which is constrained by the natural extension of Fortet’s inequalities [19]: \( w \leq y^U x \), \( w \geq y^L x \), \( w \leq y - (1 - x)y^U \), \( w \geq y - (1 - x)y^L \).

3 ROSE architecture

ROSE consists of a simple modular architecture based on two main classes (Problem and Solver) and a separate library (Ev3) for storing and manipulating expressions in E. The overall architecture is depicted graphically in Fig. 1. More
detailed information about ROSE's and Ev3's architecture, capabilities and Application Programming Interface (API) can be found in [18], Sect. 5.2-5.3.

The Problem class contains lists of Variable, Objective and Constraint structures. Structures of Variable type include information about decision variables such as index, current and optimal value, and range and integrality constraints; Objectives include information about objective functions such as index, current and optimal value, corresponding expression and optimization direction; Constraints include information about constraints such as index, current and optimal value, corresponding expression and bound restrictions. The Problem class also stores information about problem cardinalities, feasibility of a current solution with respect to the constraints, a reference to a previous Problem object in a reformulation chain, and other useful information. It has methods for accessing data, adding or deleting decision variables, objective functions and constraints, evaluating expressions appearing in objectives or constraints, parse a given input file (a description of an MP) into its data structures, and so on.

The Solver class is a virtual class whose implementations are either numerical solvers or reformulators; the latter are recognizable because their names are prefixed by R- (e.g. Rprodbincont). All Solver objects maintain: a pointer to the Problem object being solved, numerical information about current and optimal points, information about linear and nonlinear cuts and a few other items mainly used by numerical solvers. Reformulators are allowed to change the Problem they reference; problems can be duplicated before being changed by a reformulator by using the special Rcopy reformulator. Basic reformulation steps for adding or deleting problem entities are implemented in Problem; modification of expressions occurs via interfacing with the Ev3 expression library (Sect. 3.3). Many
methods in Problems and Solvers can be configured by user-defined parameters that passed to Problem and Solver objects via a unique object of the class ParameterBlob.

3.1 MP input

ROSE can read an MP via either its own intuitive flat MP format (see [3] p. 238) or via interfacing with the AMPL interpreter [1]. Each MP entity is assigned two integer scalar indices: a unique entity ID (which is preserved across reformulations) and a local index (which is an ordinal running from 1 to the number of entities of a given type within a Problem object). Methods are provided for switching from ID to local indices.

3.2 MP output

Since the AMPL API does not offer primitives for modifying the current MP, the only possibility for ROSE is to output its reformulations to a flat MP written to an AMPL formatted file. Users can then instruct AMPL to read this file. This situation is far from optimal, as it requires hard disk access, but there is no way around it — according to the AMPL authors, it is unlikely that AMPL will ever have an API which is sufficiently flexible as to allow modification of the internal data. Developers can also choose to have individual reformulators write their output to whatever syntax they wish, bypassing the default output.

3.3 Expression tree library

Following the recursive definition of expressions given in Sect. 2, an expression \( e \in E \) is encoded in a tree data structure \( T_e = (V_e, A_e) \): leaf nodes of \( V_e \) are labelled by parameters in \( p \) and by decision variables in \( x \), and intermediate nodes are labelled by operators in \( \oplus \in O \). A \( k \)-ary operator node has \( k \) subnodes in its star. An arc \((u, v)\) is in \( A_e \) if \( v \) is a subnode of \( u \).

An Expression is synonym to a Pointer<BasicExpression> template class. The Pointer<T> class is used to perform automatic memory management (i.e. automatic deallocation) from a node of type T. A BasicExpression inherits from Operand and from Tree<BasicExpression>. The Operand class simply includes information concerning a particular node (whether leaf or nonleaf, operator label, variable index, parameter value, and so on). The Tree<T> class includes a list of nodes of type Pointer<T>, and is used to represent a list of subnodes of a given node; it has methods for accessing and editing nodes. This complex architecture for expression trees makes it easy to edit, move or copy entire subtrees recursively, but floating point evaluation of the expression is slow. To circumvent this problem, expressions are also encoded in much simpler C-style tree structures (called FastEvalTrees) without any memory management in order to speed up evaluation. Their activation and use is completely transparent to the user.
The Ev3 library capabilities include simplification of expressions, reduction to (partial) normal form, identification of subexpressions of certain structures, conditional editing of subexpressions, recognition and separation of linear and nonlinear parts in a given expression, symbolic differentiation and many others. Since they act on trees, most methods are recursive, and consist of two functions: the “recursion start” and the “recursion step”. In the case of Example 2.3, the \texttt{ProdBinCont} reformulator is implemented according to the pseudocode below.

\begin{verbatim}
ProdBinCont(Expression e) {
    ProdBinContRecursive(e.root);
}
ProdBinContRecursive(Expression e) {
    if (!e.leaf) {
        for(v in e.subnodes) {
            ProdBinContRecursive(v);
        }
    }
    if (e.structure == x*y && x.binary && !y.binary) {
        AddVariable(w);
        ReplaceBy(e,w);
        AdjoinConstraint(Fortet's extension inequalities);
    }
}
\end{verbatim}

4 How ROSE helps solving the motivating examples

Subproblems in sBB. ROSE can construct a convex relaxation of (1) automatically from its Smith reformulation [20], which isolates all the nonlinearities of the problem in constraints with a simple structure; these are then replaced by appropriate convex relaxations [3]. The ROSE \texttt{Rsmith} reformulator (tasked with constructing the Smith reformulation) calls a recursive Ev3 procedure which looks for subtrees of an \texttt{Expression} \( e \) having a certain “shape” in order to replace them with an added variable \( w \); the constraint \( w = e \) is then added to the formulation. The shape of an expression is defined as an expression \texttt{schema}, i.e. an expression tree search pattern whose variable nodes are labelled by a wildcard “?” with the meaning of “any variable”. Thus, for example, the tree ? ← x → ? represents a generic product of two variables, and it matches every tree \( x_i ← x → x_j \) (for any \( i, j \)).
The pseudocode above shows the essential functionality of the \texttt{Rsmith} reformulator and the corresponding Ev3 recursive auxiliary function. ROSE has several relaxation reformulators (e.g. \texttt{Rconvexifier}, \texttt{RQuarticConvex}) which are chained to the \texttt{Rsmith} reformulator as per Thm. 2.2.

\textbf{KNP.} Solving the KNP formulation (2) requires several reformulations:

1. derive a restriction of (2) to certain neighbourhoods (in order to solve the KNP using heuristics);
2. convert the optimization problem to the corresponding decision problem for a given objective function value (3);
3. relax some constraints by means of a multiplicative tolerance;
4. adjoin an objective that maximizes the tolerance (4);
5. derive a symmetry-free narrowing and a convex relaxation of (4) (in order to solve via sBB).

The two heuristics tested on (2) are VNS and the MINLP Feasibility Pump (FPMINLP) [21]. Both rely on certain subproblems of (2) obtained by adjoining appropriate constraints: VNS requires Local Branching type constraint [22], whilst FPMINLP requires a specific outer approximation. The reformulations listed in 2.-4. above can be implemented using the basic reformulations encoded in ROSE. The symmetry-free narrowing in 5. is obtained automatically using a chain of software packages (i.e. AMPL, ROSE, \texttt{nauty} [23], GAP [24]) held together via Unix scripts. In particular, ROSE is used to analyze an AMPL flat MP and produce its Directed Acyclic Graph (DAG) encoding [4, 5], which is then fed into \texttt{nauty} in order to derive its symmetry group. ROSE can also obtain a convex relaxation of (4) based on the ideas given in [20, 3, 4]. Computational results for sBB on (4) are reported in [15].

It may be noted that the above examples were chosen arbitrarily by a large set of ROSE applications (see Sect. 5). We believe that the first example shows how ROSE can be useful \textit{per se}, whereas the second demonstrates ROSE's ability to interface with other tools in order to perform complex reformulating tasks.
5 ROSE’s existing applications

ROSE’s current role is to help automatize flat MP reformulations which would be too long to perform by hand, but which are necessary to implement and test research ideas. ROSE was and is instrumental to several past and current research projects: in some cases it is key to their success, in other cases it allows research teams to quickly weed out bad ideas; it is sometimes influential to other software, in that ideas found in ROSE are re-implemented (for practical reasons) in other codes.

- Fundamentals of reformulation theory [17, 18], where ROSE served as a proof of concept (successful).
- Experiments on spherical cuts for Binary Linear Programs (BLPs) [25] (successful).
- An investigation of the convex relaxation of quadrilinear terms [26, 27], where ROSE was used both to produce the convex relaxation and to automatically write input data to other software packages (e.g. cdd [6]) (successful).
- The FPMINLP heuristic [21], where ROSE is used both to analyze MINLPs (e.g. to find convex constraints), and to reformulate them, i.e. to build the Feasibility Pump subproblems (successful).
- The RECIPe MINLP heuristic [31] was implemented independently of ROSE with what was essentially ROSE code (influential).
- The conception of the COUENNE [4] solver code that builds the convex relaxation was heavily influenced by ideas implemented in ROSE (influential).
- Reduced RLT-based relaxations for polynomial programs (current work, unpublished).
- A general-purpose MINLP Tabu Search heuristic based on tabu spheres (unsuccessful, unpublished).
- A general-purpose MINLP feasibility heuristic based on branching with no bounding until a feasible solution is found (unsuccessful, unpublished).

References


