Modelling 0-1 Problems in CLP($\mathcal{PB}$)

Peter Barth   Alexander Bockmayr
Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany
{barth|bockmayr}@mpi-sb.mpg.de

Abstract

Many practical problems involve constraints in 0-1 variables. We apply the constraint logic programming language CLP($\mathcal{PB}$) to model and reason about 0-1 problems. Given a set of possibly non-linear 0-1 constraints, the solver of CLP($\mathcal{PB}$) computes an equivalent set of extended clauses. By exploiting the metaprogramming facilities of the logic programming environment, we are able to deal with arbitrary logical conditions between the constraints, in particular with disjunction and implication. At the end, the simplified constraint set is given to an underlying 0-1 constraint solver, which can be either a constraint programming or a mathematical programming system.

1 Introduction

Constraint programming has become a promising new technology for solving complex combinatorial problems [JM94]. Similarly to what happened earlier in mathematical programming, most research in constraint programming so far has been concentrating on the development of more powerful and efficient constraint solvers. Practical problem solving, however, usually involves three stages [Sha93, Wil93a, Wil93b]:

1. Building a model of the problem.

2. Solving the model.

3. Understanding and analysing the solution.

In this paper, we are interested in the first step of this process, the modelling part. From a practical point of view, model building seems to be even more important than model solving. A practitioner is usually not interested in developing a new solver. He would like to concentrate on building the model and then use an existing solver to find the solution.

Many practical problems involve 0-1 variables [Wil93a]. 0-1 problems are modelled by *pseudo-Boolean constraints*, which are equations or inequalities between multilinear integer polynomials in 0-1 variables [Boc95]. A constraint logic programming language CLP($\mathcal{PB}$) for logic programming with pseudo-Boolean constraints was introduced in [Boc93]. A prototype implementation of CLP($\mathcal{PB}$) has been developed in [Bar94]. Pseudo-Boolean constraints generalise Boolean constraints. At the same time, they are a restricted form of finite domain constraints, where all domains are equal to the two-element set $\{0,1\}$. In operations research, pseudo-Boolean constraints correspond to non-linear 0-1 programming problems [HJM93].

Our aim in this paper is to show how the constraint solver of CLP($\mathcal{PB}$) can be applied to model and reason about 0-1 problems. We do not discuss the underlying constraint solving techniques, which have been described elsewhere (for an in-depth treatment, we refer to the monograph [Bar96]).

Given a set of possibly non-linear pseudo-Boolean constraints, our solver computes an equivalent simplified set of *extended clauses* of the form

$$L_1 + \cdots + L_k \geq d,$$

saying that at least d out of k literals L_i have to be true, where a literal is either a 0-1 variable X or its negation $\tilde{}X = 1 - X$.

By exploiting the metaprogramming facilities of the logic programming environment, we can deal with arbitrary logical conditions between the constraints, in particular with disjunction and implication. At the end, the simplified constraint set is given to an underlying 0-1 constraint solver, which can be either a constraint programming or a mathematical programming system. The prototype system of CLP($\mathcal{PB}$) is not intended to be used for really solving large 0-1-problems. This should be better done with the 0-1 optimisation software OPBDP [Bar95].

2 Simplifying pseudo-Boolean constraints

First, we show how a set of 0-1 constraints can be simplified by the constraint solver of CLP($\mathcal{PB}$). The main steps of the simplification procedure in CLP($\mathcal{PB}$) are the following:

- Linearisation of non-linear pseudo-Boolean constraints.
- Transformation of linear pseudo-Boolean constraints into an equivalent set of extended clauses.
- Derivation of stronger extended clauses including a check of consistency.

2.1 Linearisation

Many practical applications involve non-linear pseudo-Boolean constraints [Boc93]. For example, to model the interaction of n objects $ob_1, ob_2, \ldots, ob_n$, each of which can be chosen or not, it is natural to use a quadratic pseudo-Boolean function of the form

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} X_i X_j.$$

Here a_{ij} measures the interaction between ob_i and ob_j and the 0-1 decision variable X_i indicates whether ob_i is chosen or not. Most existing solvers cannot deal with non-linear pseudo-Boolean constraints. Therefore, a common problem is to *linearise* non-linear constraints, i.e. to find a system of linear constraints with the same set of 0-1 solutions. For a given set of non-linear constraints, there exist many different linearisations. Computing good linearisations is a hard problem [HJM93].

Using CLP($\mathcal{PB}$), we can compute compact linearisations of non-linear pseudo-Boolean constraints that do not introduce additional variables.

Example 2.1 Given the non-linear constraint

```
?- X*Y + X*Z + Y*Z #>= 1.
```

the solver of CLP($\mathcal{PB}$) computes the solved form

```
    X + Y + Z #>= 2.
```

saying that at least two of the three 0-1 variables `X,Y,` and `Z` have to be true. If we ask

```
?- X*Y + X*Z + Y*Z #>= 2.
```

we get the answer

```
    X = 1, Y = 1, Z = 1.
```

In a naive approach, we would introduce new variables `XY, XZ, YZ` for the products `X*Y, X*Z, Y*Z` and add linear constraints saying that `XY = X*Y, XZ = X*Z, YZ = Y*Z`. For `X*Y + X*Z + Y*Z #>= 1`, this would yield the much more complex linearisation:

```
    XY + XZ + YZ  #>= 1,
    XY + ~X + ~Y  #>= 1,
    2*~XY + X + Y #>= 2,
    XZ + ~X + ~Z  #>= 1,
    2*~XZ + X + Z #>= 2,
    YZ + ~Y + ~Z  #>= 1,
    2*~YZ + Y + Z #>= 2.
```

Here are two more complicated examples taken from [BM84]:

Example 2.2 Given the non-linear constraint

```
?- 8*X1*X2 + 5*X1*X5 + 5*X1*X6 + 4*X2*X3*X4 + X3*X5 + X4*X6 #<= 12.
```

our solver computes the linearised form

```
    ~X1 + ~X2 + ~X5 #>= 1,
    ~X1 + ~X2 + ~X6 #>= 1.
```

For the non-linear constraint

```
?- 6*X1*X2*X3*X4 +4*X1*X5 -3*X3*X4*X6 +2*X1*X2*X4*X7 +2*X3*X4*X8 #<= 7.
```

we obtain

```
    ~X1 + ~X2 + ~X3 + ~X4 + ~X5 + X6            #>= 1,
    ~X1 + ~X2 + ~X3 + ~X4 + ~X5 +      ~X7      #>= 1,
    ~X1 + ~X2 + ~X3 + ~X4 + ~X5 +           ~X8 #>= 1,
    ~X1 + ~X2 + ~X3 + ~X4 +       X6 + ~X7      #>= 1,
    ~X1 + ~X2 + ~X3 + ~X4 +       X6 +      ~X8 #>= 1.
```

2.2 Transforming and strengthening linear constraints

Linear pseudo-Boolean constraints are much simpler than non-linear constraints. But, they are still quite complicated. For example, the entailment problem for two linear pseudo-Boolean constraints is still NP-hard. In CLP($\mathcal{PB}$), a linear pseudo-Boolean constraint is therefore further simplified to an equivalent set of extended clauses. Extended clauses are linear pseudo-Boolean inequalities of the form

$$L_1 + \cdots + L_k \geq d,$$

with positive or negative literals L_i and a right-hand side $d \in \{0, \ldots, k\}$. Clauses in propositional logic

$$L_1 \vee \cdots \vee L_k$$

correspond to extended clauses with right-hand side $d = 1$.

Example 2.3 Given the linear pseudo-Boolean inequality [Wil93a, p. 216]

```
?- 3*X1 + 3*X2 -2*X3 + 2*X4 +2*X5 #<= 4.
```

the solver computes the solved form

```
    ~X1 + ~X2 +      ~X4 + ~X5 #>= 2,
    ~X1 + ~X2 + X3       + ~X5 #>= 2,
    ~X1 + ~X2 + X3 + ~X4       #>= 2.
```

which, in this case, corresponds to three facets of the convex hull of the set of feasible 0-1 solutions of the original inequality. This means that we do not only get an equivalent set of extended clauses, but also strong inequalities with respect to the linear programming relaxation of the problem (for a general discussion of the polyhedral properties of the solved form see [Bar96, Sect. 7.7])

For some inequalities, the full transformation may be computationally prohibitive because a huge number of extended clauses may be needed. However, for inequalities that are equivalent to a small number of extended clauses, performing the full transformation can be very useful. On the one hand, it helps the modeller to better understand his constraints, due to the natural "d out of n" interpretation of extended clauses. On the other hand, it may considerably improve the performance of the solver once the model building phase has been completed.

Example 2.4 We consider the MIPLIB benchmark `p0033` [NW88, Table 2.1, p. 465]. This is a 0-1 minimisation problem with 15 constraints and 33 variables. Among the 15 constraints, 13 are each equivalent to a small number of extended clauses. For example, the inequality

```
?- 300*X2 + 200*X18 + 400*X19 + 200*X20 + 400*X21 #>= 500.
```

is simplified by our solver to

```
    X21 + X20 + X19 + X18 + X2 #>= 2,
    X21 + X19 + X2 #>= 1.
```

The inequality

```
?- 300*X1 + 285*X6 + 265*X9 + 190*X14 + 200*X26 + 400*X27 #>= 900.
```

is equivalent to four extended clauses

```
X27 + X26 + X9 + X6 + X1 #>= 3,
X27 + X14 + X6 + X1 #>= 2,
X27 + X14 + X9 + X1 #>= 2,
X27 + X26 + X14 #>= 1.
```

Only two constraints in the problem are really complicated. Each of these requires several thousands of extended clauses.

3 Reasoning with user-defined constraints

After explaining how the constraint solver of CLP($\mathcal{PB}$) can be used to simplify a given constraint set, we now combine constraint solving with the surrounding logic programming system. We use logic programming for meta-programming with pseudo-Boolean constraints.

In a first step, we associate with a linear pseudo-Boolean inequality `L #>= R` a 0-1 variable `B` such that `B` logically implies `L #>= R`:

```
pb_switch_constraint(L #>= R, B) :-
    pb_lowerbound(L - R, M),
    L - M * ~B #>= R.
```

Here, `pb_lowerbound` computes a lower bound `M` for `L - R`, for example by summing up the negative coefficients. Similarly, we can impose the condition that `B` should be logically equivalent to `L #>= R`:

```
pb_equiv_constraint(L #>= R, B) :-
    pb_switch_constraint(L #>= R, B),
    pb_switch_constraint(L #< R, ~B).
```

After having defined such indicator variables [Wil93a] we can now express logical conditions on the constraints [MLM94]. For example, we can define for pseudo-Boolean constraints `C1,C2` the disjunction

```
pb_disj(C1,C2) :-
    pb_switch_constraint(C1,B1),
    pb_switch_constraint(C2,B2),
    B1 + B2 #>= 1.
```

or the implication

```
pb_impl(C1,C2) :-
    pb_equiv_constraint(C1,B1),
    pb_switch_constraint(C2,B2),
    B2 #>= B1.
```

Using our solver, we can eliminate such logical conditions and get fully automatically an equivalent constraint set that uses only conjunction of constraints.

Example 3.1 We consider a non-trivial example from [MW89]. The problem is to model the condition:

> If 3 or more of products (1 to 5) are made, or less than 4 of products (3 to 6, 8, 9) are made then at least 2 of products (7 to 9) must be made unless none of products (5 to 7) are made.

If we represent the decision to make product i by a 0-1 variable `Pi` this condition can be expressed as follows:

```
pb_condition(P1,P2,P3,P4,P5,P6,P7,P8,P9) :-
    pb_equiv_constraint(P1+P2+P3+P4+P5 #>= 3, A),
    pb_equiv_constraint(P3+P4+P5+P6+P8+P9 #< 4, B),
    pb_equiv_constraint(P5+P6+P7 #>= 1, C),
    pb_equiv_constraint(P7+P8+P9 #>= 2, D),
    (A + B) * C #<= 2*D.
```

Note that $(A + B) * C \leq 2 * D$ is the pseudo-Boolean equivalent of $(A \vee B) \wedge C \rightarrow D$. Given the query

```
?- pb_condition(P1,P2,P3,P4,P5,P6,P7,P8,P9).
```

the solver of CLP($\mathcal{PB}$) computes fully automatically an equivalent conjunction of extended clauses:

```
P7 + P9 + ~P3 + ~P4 + ~P5 #>= 1,
P7 + P8 + ~P2 + ~P3 + ~P5 #>= 1,
P7 + P9 + ~P2 + ~P3 + ~P5 #>= 1,
P3 + P4 + P7 + P9 + ~P5 #>= 1,
P3 + P4 + P7 + P8 + ~P5 #>= 1,
P4 + P6 + P7 + P9 + ~P5 #>= 1,
P3 + P6 + P7 + P9 + ~P5 #>= 1,
P4 + P6 + P7 + P8 + ~P5 #>= 1,
P3 + P6 + P7 + P8 + ~P5 #>= 1,
P8 + P9 + ~P5 #>= 1,
P7 + P8 + ~P1 + ~P4 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P4 + ~P5 #>= 1,
P7 + P8 + ~P1 + ~P3 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P3 + ~P5 #>= 1,
P7 + P8 + ~P3 + ~P4 + ~P5 #>= 1,
P8 + P9 + ~P7 #>= 1,
P7 + P8 + ~P1 + ~P2 + ~P4 + ~P6 #>= 1,
P7 + P9 + ~P1 + ~P2 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P1 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P1 + ~P2 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P2 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P9 + ~P2 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P1 + ~P2 + ~P3 + ~P6 #>= 1,
P7 + P9 + ~P1 + ~P2 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P2 + ~P3 + ~P6 #>= 1,
P4 + P5 + P7 + P9 + ~P6 #>= 1,
P3 + P5 + P7 + P9 + ~P6 #>= 1,
P3 + P4 + P7 + P9 + ~P6 #>= 1,
P4 + P5 + P7 + P8 + ~P6 #>= 1,
P3 + P5 + P7 + P8 + ~P6 #>= 1,
P3 + P4 + P7 + P8 + ~P6 #>= 1,
P7 + P8 + ~P2 + ~P4 + ~P5 #>= 1,
P7 + P9 + ~P2 + ~P4 + ~P5 #>= 1,
P8 + P9 + ~P6 #>= 1.
```

A tighter representation is possible using compact extended clauses [Bar96]. But, we only wanted to show how logical conditions between constraints can be naturally expressed in the logic programming environment. Using CLP($\mathcal{PB}$), they can be eliminated fully automatically and an equivalent set of extended clauses is obtained, which can be given to a standard solver.

Another convenient modelling operator is *cardinality* [vHD93], which states an upper and lower bound for the number of constraints in a list that must hold.

```
pb_card(Lower,L,Upper) :-
    pb_equiv_sum(L,S),
    S #>= Lower, S #<= Upper.

pb_equiv_sum([],0).
pb_equiv_sum([C|Cs],S+BV) :-
    pb_equiv_constraint(C,BV),
    pb_equiv_sum(Cs,S).
```

Hence, a disjunction of constraints can be expressed by

```
pb_card(1,[C1,...,CN],N).
```

and a conjunction by

```
pb_card(N,[C1,...,CN],N).
```

Parts of a problem can be semantically investigated using the interactivity of the logic programming environment together with the powerful constraint solving capabilities offered by constraint logic programming.

Example 3.2 Consider the constraint list

```
L = [A + B #>= 1, A + C #>= 1, A + D #>= 1,
     B + C #>= 1, B + D #>= 1, C + D #>= 1]
```

The query

```
?- pb_card(0, L, 2).
```

yields the answer `no`. If we ask

```
?- pb_card(3, L, 6).
```

we get

```
    A + B + C + D #>= 1.
```

Note that in all these examples various features of the constraint solver have to be used: transformation, simplification, elimination of the auxiliary 0-1 variables introduced by `pb_switch_constraint`, and a consistency check.

4 Conclusion

We have applied the constraint logic programming language CLP($\mathcal{PB}$) to model and reason about 0-1 problems. In the first part, we have shown how the constraint solver of CLP($\mathcal{PB}$) can be used to simplify constraints in 0-1 variables. In the second part, we have combined constraint solving with logic programming. In particular, we were able to eliminate fully automatically logical conditions between pseudo-Boolean constraints. Simplified constraint sets can be given to a mathematical programming or to a constraint programming system. Our favorite tool is the 0-1 optimisation software OPBDP [Bar95].

References

[Bar94] P. Barth. *Short Guide to CLP($\mathcal{PB}$).* Max-Planck-Institut für Informatik, 1994. System available: ftp://www.mpi-sb.mpg.de/pub/tools/CLPPB/clppb.html.

[Bar95] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Saarbrücken, January 1995. System available: http://www.mpi-sb.mpg.de/~barth/opbdp/opbdp.html.

[Bar96] P. Barth. *Logic-based 0-1 constraint programming.* Operations Research/Computer Science Interfaces Series. Kluwer, 1996.

[BM84] E. Balas and J. B. Mazzola. Nonlinear 0-1 programming: II. Dominance relations and algorithms. *Mathematical Programming*, 30:22–45, 1984.

[Boc93] A. Bockmayr. Logic programming with pseudo-Boolean constraints. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming. Selected Research*, chapter 18, pages 327 – 350. MIT Press, 1993.

[Boc95] A. Bockmayr. Solving pseudo-Boolean constraints. In *Constraint Programming: Basics and Trends*, pages 22 – 38. Springer, LNCS 910, 1995.

[HJM93] P. Hansen, B. Jaumard, and V. Mathon. Constrained nonlinear 0-1 programming. *ORSA Journal on Computing*, 5(2):97 – 119, 1993.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503 – 581, 1994.

[MLM94] G. Mitra, C. Lucas, and S. Moody. Tools for reformulating logical forms into zero-one mixed integer programs. *Europ. J. Oper. Res.*, 72:262 – 276, 1994.

[MW89] K. I. M. McKinnon and H. P. Williams. Constructing integer programming models by the predicate calculus. *Annals of Operations Research*, 21:227–246, 1989.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley, 1988.

[Sha93] R. Sharda. *Linear & discrete optimization and modeling software.* UNICOM, 1993.

[vHD93] P. van Hentenryck and Y. Deville. The cardinality operator: a new logical connective for constrain logic programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming. Selected Research*, chapter 20, pages 383–403. MIT Press, 1993.

[Wil93a] H. P. Williams. *Model building in mathematical programming.* John Wiley, third revised edition, 1993.

[Wil93b] H. P. Williams. *Model solving in mathematical programming.* John Wiley, 1993.