Pseudo-Boolean and Finite Domain Constraint Programming: A Case Study

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

Abstract

Pseudo-Boolean constraints are a special form of finite domain constraints where all variables are defined over the domain $\{0,1\}$. To solve pseudo-Boolean constraints, specialized constraint solving algorithms have been developed. In this paper, we compare finite domain and pseudo-Boolean constraint techniques on a classical application of finite domain constraint programming, the warehouse location problem. Although the finite domain model is very natural and theoretically has a much smaller search space, the 0-1 model with specialized constraint solving techniques turns out to be more efficient.

1 Introduction

Constraint programming has become a promising new technology for solving complex decision problems. It combines new programming paradigms from computer science, like constraint logic programming and concurrent constraint programming, with efficient constraint solving techniques from mathematics, artificial intelligence, and operations research. Constraint programming allows the user both to build and to solve a model in the same framework:

- Expressive programming language constructs make it possible to formulate the problem in a natural and declarative way.
- Powerful constraint handling techniques supporting these language constructs provide for efficiency in problem solving.

Compared to other approaches, constraint programming offers incremental development, flexibility for modifications and extensions, and interactivity, without sacrificing efficiency. Constraint technology has been applied successfully to a large variety of practical problems arising in industry, for example in production planning, scheduling, and resource allocation.

Solving combinatorial problems is one of the most important application areas of constraint programming. Here, the constraints are formulated over finite domains, i.e. the variables take their values in finite sets of non-negative integer numbers. Many practical problems can be modeled naturally using such constraints (see for example [12, 1]). A special case of finite domain constraints are *pseudo-Boolean constraints* [6], where all variables are defined over the domain $\{0,1\}$. This corresponds to 0-1 integer programming in operations research. Finite domain constraints are usually solved with local consistency techniques and constraint propagation from artificial intelligence. However, in the 0-1 case, local consistency techniques

tend to be rather weak. For example, if the solution set $S \subseteq \{0,1\}^n$ of a pseudo-Boolean constraint set is full-dimensional, then the domains of the variables cannot be reduced. Even if the domains could be reduced, consistency techniques that work locally are often not able to detect this [4].

To overcome these problems and in order to infer more information from a given constraint set, we have developed a new approach for pseudo-Boolean constraint solving. It uses techniques from mathematical programming and is based on the idea of generating strong valid inequalities within a branch-and-cut framework [7, 8].

Often a given practical problem can be modeled in different ways. We may use either a

- *finite domain model*, where every variable takes its value in some finite set of natural numbers, or a
- *0-1 model*, where all variables are defined over the domain $\{0,1\}$.

In this paper, we compare these two approaches on a particular example. After a brief introduction to our polyhedral branch-and-cut solver, we focus on a classical application of finite domain constraint programming, the warehouse location problem. We consider both finite domain and 0-1 models and compare different solution strategies. Although the finite domain model is very natural and theoretically has a much smaller search space, the 0-1 model with branch-and-cut turns out to be more efficient.

2 Branch-and-Cut for pseudo-Boolean constraint solving

Pseudo-Boolean constraints are equations or inequalities between multivariate real polynomials in 0-1 variables. In this paper, we consider only linear *pseudo-Boolean constraints*

$$a_1x_1 \quad + \quad \ldots \quad + \quad a_nx_n \quad \geq \beta \quad \Leftrightarrow a^Tx \geq \beta,$$

with $a = (a_1, \ldots, a_n)^T \in \mathbb{R}^n, \beta \in \mathbb{R}$ and 0-1 variables $x_1, \ldots, x_n \in \{0,1\}$.[1] A constraint set C then has the form

$$\begin{array}{cccccccc} a_{11}x_1 & + & \ldots & + & a_{1n}x_n & \geq b_1 & & \\ \vdots & & & & & \vdots & \Leftrightarrow & Ax \geq b \\ a_{m1}x_1 & + & \ldots & + & a_{mn}x_n & \geq b_m & & \end{array}$$

with a matrix $A \in \mathbb{R}^{m \times n}$, a column vector $b \in \mathbb{R}^m$, and a vector of 0-1 variables $x \in \{0,1\}^n$.

A solver in constraint programming has to handle pseudo-Boolean constraints incrementally and provide answers to the following questions:

Consistency: Is there a *feasible* solution to the current constraint set C?

Entailment: Is a new constraint implied by the current constraint set C?

Optimization: What is the optimal value of an objective function $f(x) = c^Tx$ subject to the current constraint set and what is an *optimal* solution?

Our approach for solving these problems is based on two main ideas [8]:

[1] $.^T$ denotes transposition.

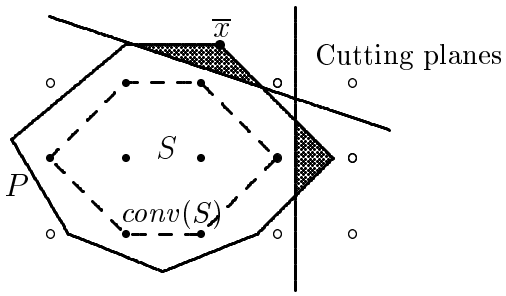

Figure 1: Constraint solving by cutting plane generation

- Lazy strengthening of the constraint set.
- Incremental approximation of an ideal solved form.

Given two constraint sets $C : Ax \geq b$ and $C' : A'x \geq b'$ with the same set of 0-1 solutions $S \subset \{0,1\}^n$, we say that C is *stronger* than C' (w.r.t. $\mathbb{R}^n$) iff

$$P \stackrel{\text{def}}{=} \{x \in \mathbb{R}^n \mid Ax \geq b\} \subseteq P' \stackrel{\text{def}}{=} \{x \in \mathbb{R}^n \mid A'x \geq b'\}.$$

Geometrically, this means that the *polyhedron* P in n-dimensional real space defined by C is contained in the polyhedron P' defined by C'. Our *ideal solved form* is the set of facet-defining inequalities for the convex hull $conv(S)$ of S.

2.1 Cutting planes

We approximate the ideal solved form by computing *cutting planes*. These are new constraints $a^T x \geq \beta$ that cut off a vertex $\overline{x}$ of the current polyhedron P (see Fig. 1). The result is a smaller polyhedron P', which gives a better approximation of the 0-1 solution set S. Our aim is not to compute the full convex hull of S, which would be very difficult from the computational point of view. The idea is rather to strengthen the constraint set in a lazy manner. In the case of a pseudo-Boolean optimization problem

$$\min\{c^T x \mid Ax \geq b, x \in \{0,1\}^n\}$$

we stop the approximation as soon as an optimal 0-1 solution is found or the current polyhedron becomes empty. Consistency and entailment problems can be reduced to optimization problems:

- $Ax \geq b$ entails $c^T x \geq \delta$ iff

$$\min\{c^T x \mid Ax \geq b, x \in \{0,1\}^n\} \geq \delta.$$

- $Ax \geq b$ is consistent iff

$$\min\{x_0 \mid x_0 d + Ax \geq b,\ (x_0, x) \in \{0,1\}^{n+1}\} = 0,$$

  where x_0 is a new variable and $d \in \mathbb{R}^m, d_i = b_i + \sum_{j=1}^{n} \max(0, -a_{ij})$, for $i = 1, \ldots, m$.

The variable x_0 is introduced to guarantee feasibility of the constraint set $x_0 d + Ax \geq b$. However, such an additional variable is not really needed. Since our solver detects infeasibility, we could simply optimize an arbitrary objective function subject to the original constraint set $Ax \geq b$.

2.2 Branch-and-Cut

From a theoretical point of view, cutting planes alone would be sufficient to solve the basic problems of pseudo-Boolean constraint solving [7]. In practice, however, they converge very slowly. Therefore, for efficiency reasons, cutting plane generation is usually combined with *branch-and-bound*. In order to solve, for example, a maximization problem

$$\max\{f(x) \mid x \in S\}$$

we split the set of feasible solutions S in two subsets $S = S_0 \cup S_1$. Next, we compute upper bounds u_0 resp. u_1 for the two subproblems

$$\max\{f(x) \mid x \in S_0\} \text{ and } \max\{f(x) \mid x \in S_1\}.$$

If some upper bound $u_k, k = 0, 1$, is smaller than some known lower bound on the maximal objective function value, then we can discard S_k from further consideration.

In the pseudo-Boolean case, branching is normally done by fixing some variable x_j to the values 0 and 1. This leads to a binary *search tree*. Using upper and lower bounds, one tries to prune this tree as much as possible. Upper bounds are obtained by solving the *linear relaxation*

$$\max\{c^T x \mid \tilde{A}x \geq \tilde{b}, 0 \leq x \leq 1, x \in \mathbb{R}^n\}$$

of a pseudo-Boolean subproblem

$$\max\{c^T x \mid \tilde{A}x \geq \tilde{b}, x \in \{0,1\}^n\}.$$

Upper bounds are *local*, i.e. they are only valid for the current subproblem and its sons. Lower bounds are obtained by finding feasible solutions. They are *global*, i.e. valid for the original problem. Using upper and lower bounds together in order to prune the search tree is characteristic for a branch-and-bound approach in the sense of mathematical programming. In finite domain constraint programming, branch-and-bound usually involves only lower bounds. This is one main source of inefficiency in finite domain constraint solvers.

Branch-and-bound combined with cutting plane generation is called *branch-and-cut*. The following diagram (Fig. 2) gives an overview of our branch-and-cut procedure. The algorithm can be divided into two parts: the enumerative part and the bounding part.

In the *enumerative part*, the list of active nodes is initialized with the original problem and the greatest lower bound `glb` is set to $-\infty$. If the list of active branch-and-cut nodes is empty, `glb` is the optimal value and the algorithm stops. Otherwise, we select a problem from the list and enter the bounding phase. The bounding phase is leaved if no more cuts shall be generated, if the problem is infeasible, if the optimal solution of the relaxation is smaller than the greatest lower bound, or if the optimal solution of the relaxation is integral. If we do not generate cutting planes and cannot discard the problem, the node is expanded, i.e. two new subproblems are generated by fixing some variable to the values 0 resp. 1. The new nodes are inserted into the list of active nodes. In case that we could improve the global lower bound `glb`, all nodes in the list whose local upper bound `lub` is smaller than `glb` can be removed.

In the *bounding part*, first the linear relaxation of the current subproblem is solved. If the optimal solution is fractional, we may apply a heuristics to find a better lower bound. For the correctness of the algorithm, however, this is not necessary. Otherwise, lower bounds are obtained if the optimal solution of the relaxation is integral. Next, we have to decide whether

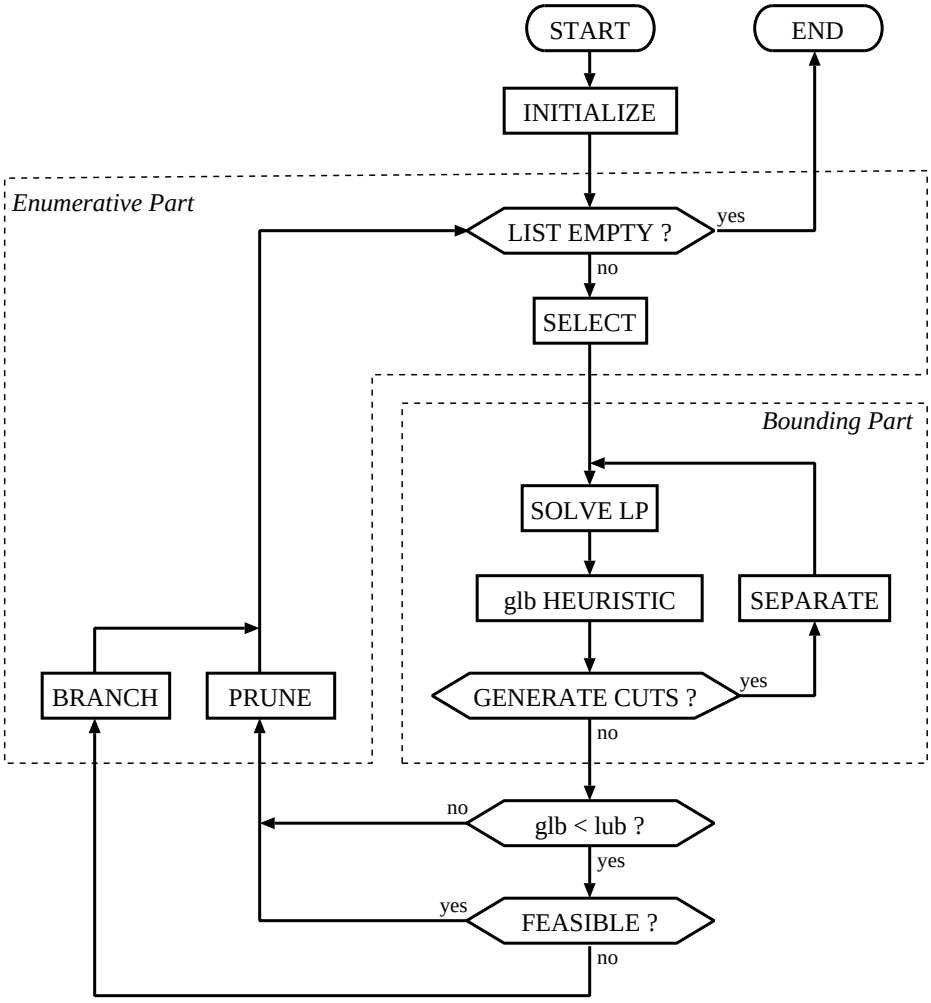

Figure 2: Branch-and-cut procedure

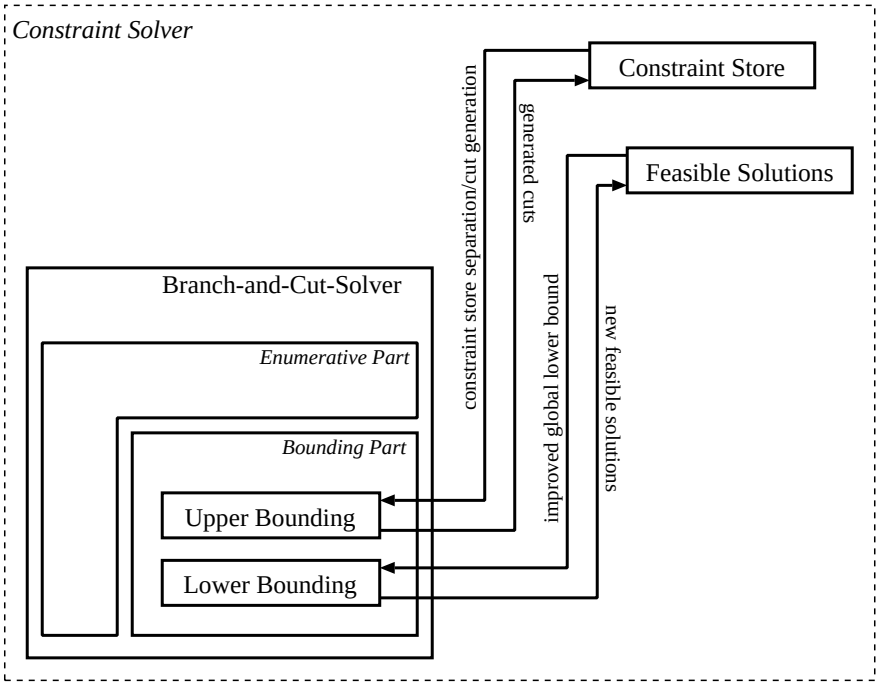

Figure 3: Pseudo-Boolean constraint solver based on branch-and-cut

we want to generate cutting planes. If this is the case, we enter the separation phase to find strong valid inequalities that cut off some part of the linear relaxation and thus improve the current upper bound. In our solver, we use lift-and-project cutting planes [3]. After cutting planes have been generated, they are added to the current subproblem and the relaxation is reoptimized.

2.3 Incrementality

This branch-and-cut approach fits perfectly well the demands of constraint programming. Although we heavily use techniques from mathematical programming, we do not just integrate an operations research solver into a constraint programming system. A standard operations research solver would give us only a feasible or optimal solution to one given constraint problem. What we do here, is much more (see Fig. 3). Whenever we solve one of our basic problems (consistency, entailment, optimization), we generate new constraints, i.e. cutting planes, and possibly new feasible solutions, which make subsequent problems easier to solve. Cutting planes tighten the linear relaxation of the pseudo-Boolean constraint set. By applying linear programming, we obtain better upper bounds that allow us to prune more often the search tree. Feasible solutions improve the global lower bound. Using cutting planes and feasible solutions together, we constantly improve our knowledge of the overall problem, and thus get an inherently incremental approach for constraint solving.

Example 2.1 We show how our solver approximates the ideal solved form if a sequence of basic problems is solved. Suppose that, at the beginning, our constraint store contains one

pseudo-Boolean constraint, namely

$$C_0 = \{3x_1 + 2x_2 + x_3 + x_4 + x_5 \geq 5\}.$$

We check feasibility by the method given in Sect. 2.1. The consistency check indicates that the constraint is feasible with solution $(1, 1, 1, 1, 1)$. This test can be done without cutting planes, since already the relaxation of the problem results in the above integral solution. Next we set up an optimization problem and solve $min\{3x_1 + x_2 + x_3 + x_4 | x \in C_0\}$. The optimal solution is $(0, 1, 1, 1, 1)$ with optimal value 3. Here, one cutting plane is generated and added to the constraint store, which becomes

$$C_1 = \{3x_1 + 2x_2 + x_3 + x_4 + x_5 \geq 5, 2x_1 + x_2 + x_3 + x_4 \geq 3\}.$$

In a final step, we ask if $x_1 + x_2 + x_4 \geq 1$ is entailed by the current constraint store. The constraint solver confirms this. Another cutting plane is generated and added to the constraint store, which now contains

$$C_2 = \{3x_1 + 2x_2 + x_3 + x_4 + x_5 \geq 5, \quad 2x_1 + x_2 + x_3 + x_4 \geq 3, \quad x_1 + x_2 \geq 1\}.$$

For this small example, we can also give the ideal solved form, i.e. the set of (non-trivial) facet-defining inequalities of the convex hull of all 0-1 solutions,

$$\begin{gathered} C_I = \{\ x_1 + x_2 \geq 1, \quad x_1 + x_3 \geq 1, \quad x_1 + x_4 \geq 1, \quad x_1 + x_5 \geq 1, \\ 2x_1 + x_2 + x_3 + x_4 \geq 3, \quad 2x_1 + x_2 + x_3 + x_5 \geq 3, \quad 2x_1 + x_2 + x_4 + x_5 \geq 3, \\ 3x_1 + 2x_2 + x_3 + x_4 + x_5 \geq 5\ \}. \end{gathered}$$

We see that our solver has found a nice approximation of the ideal solved form. We have computed two facet-defining inequalities.

One might think that combining cutting plane generation with branching destroys this incrementality. When a cutting plane is first generated, it is only locally valid at the current node of the branch-and-cut tree, and at the sons of this node. Variables have been fixed to the values 0 or 1 according to the path that leads from the root of the tree to this node. However, for lift-and-project cutting planes, we can easily compute coefficients for these variables in order to obtain a cutting plane that is globally valid at all nodes of the tree. This *cut lifting* is crucial for the efficiency of the branch-and-cut procedure. It means that the work that is done locally at one node can be used globally at all other nodes.

3 Warehouse location revisited

We now focus on a classical application of finite domain constraint programming, the (uncapacitated) warehouse location problem [13, 12, 2]. We will present three formulations of the problem, one finite domain and two 0-1 models, and compare their efficiency.

3.1 Problem Description

Given a number of customers and possible warehouse locations, the problem is to decide which warehouses should be opened and which customers they should supply such that the overall costs are minimized. Two kinds of costs have to be considered, the fixed costs of building and maintaining the warehouse, and the variable costs for the transportation between a warehouse and a customer. We denote by

- m the number of warehouse locations
- n the number of customers
- f_i the fixed cost of warehouse i
- v_{ij} the variable cost for warehouse i and customer j

According to [13], this problem is NP-complete.

3.2 Finite domain model

We first recall the standard finite domain model of the warehouse location problem (see [13, 12] for more details). Two classes of finite domain variables are used:

- For each warehouse location $i = 1, \ldots, m$, a 0-1 variable W_i with

$$W_i = \begin{cases} 1: & \text{warehouse } i \text{ is open} \\ 0: & \text{warehouse } i \text{ is closed} \end{cases}$$

- For each customer $j = 1, \ldots, n$, two finite domain variables
  - $G_j \in \{1, \ldots, m\}$ for the number of the warehouse that supplies customer j, and
  - $VC_j \in \{v_{1j}, \ldots, v_{ij}, \ldots, v_{mj}\}$ for the corresponding variable cost.

The objective function that has to be minimized can now be stated as

$$VC_1 + \ldots + VC_n + f_1 W_1 + \ldots + f_m W_m.$$

Next we express the relation between the variables VC_j and G_j, for $j = 1, \ldots, n$. We use the symbolic constraint `element(I, L, El)`, which states that `El` is the `I`th element of the list `L`.

```
element(G_1, [v_11,...,v_m1], VC_1).
:
element(G_n, [v_1n,...,v_mn], VC_n).
```

It remains to formulate the constraints between the warehouses and the customers, i.e. whenever a warehouse is closed, it cannot supply any customer. Here, we use some conditional propagation of the form

```
if W_i #= 0 then outof(Gs,i).
```

for $i = 1, \ldots, m$, which removes the value i from the domain of the variables $G_j, j = 1, \ldots, n$, as soon as W_i becomes 0. The size of the search space in this model is $2^m \cdot m^n$.

3.3 Weak 0-1 model

We now proceed to our second model, which uses only 0-1 variables. Instead of the finite domain variables G_i for the warehouse supplying customer i, we now use $m \cdot n$ 0-1 variables G_{ij} with

$$G_{ij} = \begin{cases} 1: & \text{warehouse } i \text{ supplies customer } j \\ 0: & \text{else} \end{cases}$$

The size of the search space is now $2^m \cdot 2^{mn}$, which is much larger than in the previous finite domain model.

Using these variables, we get the following *weak 0-1 model* for the warehouse location problem, which was used in [13] to compare finite domain and 0-1 integer programming:

$$\begin{aligned} \min \sum_{i=1}^{m}\sum_{j=1}^{n} v_{ij}G_{ij} + \sum_{i=1}^{m} f_i W_i \qquad & \text{subject to} \\ \sum_{i=1}^{m} G_{ij} = 1, \qquad & \text{for } j = 1, \ldots, n \\ \sum_{j=1}^{n} G_{ij} \leq n \cdot W_i, \qquad & \text{for } i = 1, \ldots, m \\ G_{ij} \in \{0,1\}, W_i \in \{0,1\}, \qquad & \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n. \end{aligned} \tag{1}$$

The first set of constraints models the fact that each customer is supplied by exactly one warehouse. The second set of constraints expresses that if a customer is supplied by some warehouse then this warehouse must be open.

3.4 Computational experience with the weak 0-1 model

We compared the two models on 12 instances from the OR-Library [5]. Each of these examples has 50 customers. The number of warehouses ranges from 16 to 50. All floating point numbers in the data were truncated in order to get integer numbers. In Tab. 1, the problem characteristics are summarized. The column *Opt* gives the optimal cost of the problem and the column *LP-Opt* the cost of an optimal solution to the linear relaxation of the weak 0-1 model (1).

The finite domain model was solved with CHIP V.4.1.0 [10, 2], the 0-1 model with our polyhedral branch-and-cut solver YAZOO [9], where we compared two solution strategies:

- Branch-and-Bound with linear programming relaxations (B&B)
- Branch-and-Cut with lift-and-project cutting planes (B&C).

The underlying linear programming package used by YAZOO was CPLEX 4.0.4. All experiments were done on a SPARC 10/31. The running times are given in seconds. Tab. 2 summarizes the results.

Although the finite domain model is very natural and theoretically has a much smaller search space, the 0-1 model with branch-and-cut turns out to be more efficient. In particular this holds for the larger problem instances, which we could not solve to optimality without cutting plane generation.

Problem	Warehouses	Customers	0-1 Variables	LP-opt	Opt
cap41	16	50	816	844787	932597
cap42	16	50	816	849149	977779
cap43	16	50	816	853415	1010619
cap44	16	50	816	859443	1034956
cap81	25	50	1275	659319	796626
cap82	25	50	1275	663994	854682
cap83	25	50	1275	668480	893758
cap84	25	50	1275	674711	928918
cap111	50	50	2550	631398	793416
cap112	50	50	2550	636298	851472
cap113	50	50	2550	641198	893053
cap114	50	50	2550	648402	928918

Table 1: Problem characteristics

	Yazoo B&B		Yazoo B&C			Chip
Problem	Nodes	Time	Nodes	Cuts	Time	Time
cap41	3803	116	10	69	6	33
cap42	6154	232	16	151	14	57
cap43	6502	241	68	982	229	422
cap44	3090	116	60	1046	271	383
cap81	*	*	28	209	34	> 20000
cap82	*	*	72	725	255	> 20000
cap83	*	*	112	2000	1179	> 20000
cap84	*	*	96	1514	853	> 20000
cap111	*	*	117	1676	2425	> 30000
cap112	*	*	396	3694	10459	> 30000
cap113	*	*	398	4736	18195	> 30000
cap114	*	*	336	3256	5449	> 30000

* out of memory

Table 2: Computational results for the warehouse problem (weak 0-1 model)

3.5 Strong 0-1 model

The previous 0-1 model can be greatly improved by replacing each of the constraints

$$\sum_{j=1}^{n} G_{ij} \leq n \cdot W_i \tag{2}$$

for $i = 1, \ldots, m$, with n individual constraints

$$\begin{aligned} G_{i1} &\leq W_i, \\ &\vdots \\ G_{in} &\leq W_i. \end{aligned} \tag{3}$$

This reformulation is well-known in operations research [14, 11]. Summing up the constraints in (3) yields the constraint (2), which shows that (3) implies (2). Concerning 0-1 solutions, the two formulations are equivalent. However, if we compare their linear relaxations, then (3) is much stronger. For example, (2) admits the solution

$$G_{i1} = \frac{1}{2}, G_{i2} = \ldots = G_{in} = \frac{1}{2(n-1)}, W_i = \frac{1}{n},$$

which is excluded by (3).

Using this reformulation technique, we get our *strong 0-1 model* of the warehouse location problem:

$$\begin{aligned} \min \sum_{i=1}^{m}\sum_{j=1}^{n} v_{ij}G_{ij} + \sum_{i=1}^{m} f_i W_i \qquad & \text{subject to} \\ \sum_{i=1}^{m} G_{ij} = 1, \qquad & \text{for } j = 1, \ldots, n \\ G_{ij} \leq W_i, \qquad & \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n \\ G_{ij} \in \{0,1\}, W_i \in \{0,1\}, \qquad & \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n. \end{aligned} \tag{4}$$

3.6 Computational experience with the strong 0-1 model

We solved the same test problems as before using the strong 0-1 model. By the reformulation, many vertices of the underlying relaxation polyhedron become integral, so that all our test examples can be solved to optimality very quickly by pure linear programming, without branching or cutting plane generation (see Tab. 3).

4 Conclusion

Given a concrete practical problem, the following fundamental questions have to be answered:

- How should the problem be modeled ?
- How should the model be solved ?

In this paper, we have compared finite domain and 0-1 models with different solution strategies on a classical application of finite domain constraint programming, the warehouse location problem. Although the finite domain model is very natural and theoretically has a much smaller search space, the 0-1 model with specialized constraint solving techniques turns out to be more efficient.

	YAZOO Branch&Cut		
Problem	Nodes	Cuts	Time
cap41	0	0	1
cap42	0	0	1
cap43	0	0	1
cap44	0	0	1
cap81	0	0	2
cap82	0	0	2
cap83	0	0	2
cap84	0	0	2
cap111	0	0	10
cap112	0	0	10
cap113	0	0	10
cap114	0	0	10

Table 3: Computational results for the warehouse problem (strong 0-1 model)

References

[1] *Practical Application of Constraint Technology, PACT'96, London*, 1996. Proceedings.

[2] A. Aggoun, N. Beldiceanu, and M. Dincbas. *CHIP Primer*. COSYTEC SA, June 1993. Version 4.0.

[3] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295 – 324, 1993.

[4] P. Barth and A. Bockmayr. Finite domain and cutting plane techniques in CLP($\mathcal{PB}$). In L. Sterling, editor, *Logic Programming. 12th International Conference, ICLP'95, Kanagawa, Japan*, pages 133 – 147. MIT Press, 1995.

[5] J. E. Beasley. OR-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069 – 1072, 1990. http://mscmga.ms.ic.ac.uk/info.html.

[6] 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.

[7] A. Bockmayr. Cutting planes in constraint logic programming. Technical Report MPI-I-94-207, Max-Planck-Institut für Informatik, Saarbrücken, February 1994.

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

[9] A. Bockmayr and T. Kasper. Implementing a polyhedral branch-and-cut solver for pseudo-Boolean constraint programming. In preparation, 1996.

[10] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Fifth Generation Computer Systems, Tokyo, 1988*. Springer, 1988.

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

[12] P. van Hentenryck. *Constraint satisfaction in logic programming.* MIT Press, 1989.

[13] P. van Hentenryck and J.-P. Carillon. Generality versus specifity: an experience with AI and OR techniques. In *American Association for Aritificial Intelligence, 7th National Conference AAAI'88, St. Paul*, pages 660 – 664, 1988.

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