Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 103-143

The First Evaluation of Pseudo-Boolean Solvers (PB'05)

Vasco M. Manquinho vasco.manquinho@inesc-id.pt
IST/INESC-ID, Technical University of Lisbon
Rua Alves Redol n. 9 - sala 329
1000-029 Lisboa – Portugal

Olivier Roussel olivier.roussel@cril.univ-artois.fr
CRIL
rue de l'Université – SP 16
62307 Lens Cedex – France

Abstract

The first evaluation of pseudo-Boolean solvers was organized as a subtrack of the SAT 2005 competition. The first goal of this event is to take a snapshot of the current state of the art in the field of pseudo-Boolean constraints. The second goal is to stimulate the research efforts in this field and contribute to the creation of better technologies. This paper details the organization and the results of this event.

Keywords: *pseudo-Boolean, evaluation, benchmarks*

Submitted October 2005; revised December 2005; published March 2006

1. Introduction

The SAT competition[22] aims at promoting interesting techniques for the propositional satisfiability problem (SAT) as well as identifying challenging benchmarks. This competition organized since 2002 [34] by Daniel Le Berre and Laurent Simon has been very fruitful to the SAT community and has contributed to the wide use of SAT techniques in a number of applications. For its fourth edition, the SAT 2005 competition has introduced a few subtracks, one of them being devoted to pseudo-Boolean solvers.

This first evaluation of pseudo-Boolean solvers (PB'05)[25] inherits the same goals as the SAT competition. It aims at providing the community with a snapshot of the current state of the art in the field of pseudo-Boolean solvers through a comparison of their performances. The goal is both to identify successful techniques as well as encouraging researchers to develop new techniques. This first edition fulfills this desire since several well-known pseudo-Boolean solvers entered the evaluation as well as new solvers based on significantly different techniques.

Another goal of the evaluation is to identify a set of challenging benchmarks so as to stimulate the improvement of solvers. This first evaluation brought the opportunity to gather existing benchmarks under a common format as well as providing the community with a set of new instances.

This first event is called an evaluation rather than a competition because our goal is less to identify a possible best solver than to set up the conditions for assessing the quality of

©2006 Delft University of Technology and the authors.

the different techniques. Besides, since pseudo-Boolean solvers often solve an optimization problem, comparing two solvers is a significantly more difficult task than for the SAT competition. Therefore, our ambition was mainly to gather some experience in order to provide a better analysis of results the next time and try to setup the grounds for the evolution of the current evaluation into a competition.

We consider that the first evaluation was rather successful since 8 different solvers were submitted (with a number of different versions) by 16 authors and co-authors. New benchmarks were also submitted by 5 different contributers. This paper explains how the evaluation was organized and tries to identify what should be improved. It also presents the results of the different solvers.

Section 2 provides the reader with a definition of the pseudo-Boolean problem and introduces the key concepts to understand the sequel of the paper. The following section describes the input format that was adopted for the evaluation and section 4 addresses the issue of big integers that must be dealt by pseudo-Boolean solvers but not by SAT solvers. Section 5 presents the benchmarks used in the evaluation, as well as a proposed categorization of those benchmarks and section 6 gives a short description of the solvers that where submitted. The next sections present the experimental conditions used to run solvers and the experimental process used. Section 9 provides the reader with the final results of the evaluation. Finally, some perspective for the next evaluation are drawn and final conclusions are presented in section 11.

2. Definitions

In a propositional formula, a literal l_j denotes either a variable x_j or its complement $\bar{x}_j$. If a literal $l_j = x_j$ and x_j is assigned value 1 or $l_j = \bar{x}_j$ and x_j is assigned value 0, then the literal is said to be true. Otherwise, the literal is said to be false.

Formally, an instance of a Linear Pseudo-Boolean Optimization (PBO) problem can be defined as follows:

$$\begin{array}{ll} \text{minimize} & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} l_j \geq b_i, \\ & x_j \in \{0,1\}, a_{ij}, b_i \in {}_0^+, i \in \{1,\ldots,m\} \end{array} \tag{1}$$

where c_j is a non-negative integer cost associated with variable $x_j, 1 \leq j \leq n$ and a_{ij} denote the coefficients of the literals l_j in the set of m linear constraints. All pseudo-Boolean formulations can be rewritten such that all coefficients a_{ij} and right-hand side b_i be non-negative [8]. Moreover, equality constraints or other types of inequality constraints (such as *greater than*, *smaller than* or *smaller than or equal to*), can also be transformed in linear time into *greater than or equal to* constraints as defined in (1).

In a given constraint, if all a_{ij} coefficients have the same value k, then it is called a *cardinality constraint*, since it only requires that $\lceil b_i/k \rceil$ literals be true. A pseudo-Boolean constraint where any literal set to true is enough to satisfy the constraint, can be interpreted as a *propositional clause*. This occurs when the value of all a_{ij} coefficients are greater than or equal to b_i. Otherwise, if a constraint is neither a cardinality constraint or a propositional clause, then it is classified as a *general pseudo-Boolean constraint*.

If every constraint in a PBO instance P can be interpreted as a propositional clause then P is an instance of the *Binate Covering* (BCP) problem. If all constraints in P can be interpreted as propositional clauses with only positive literals, then P is an instance of the *Unate Covering* (UCP) problem. When all coefficients c_j of the cost function in P are equal to 0 (no cost function is present), then we say that P is an instance of the *Pseudo-Boolean Solving* (PBS) problem. If all constraints of a PBS instance can be interpreted as propositional clauses, then it is an instance of the *Propositional Satisfiability* (SAT) problem.

Notice that a linear pseudo-Boolean optimization problem can also be viewed as a special case of integer linear programming (ILP) problem. The ILP formulation for the constraints can be obtained if we replace literals $\bar{x}_j$ by $1 - x_j$.

3. Input Format

For this first evaluation, an input format that would be read by each solver had to be defined. This common format should have the following properties

- be already used by some existing solver
- be easily parsable by a solver
- be easily read by a human being
- have no ambiguity in its definition

In our opinion, the last three points are the reason of the success of the DIMACS format for CNF formulae used in SAT solvers. Among the different formats of pseudo-Boolean problems that were already available on the web, the OPB format[7] was certainly the one closest to our objectives. Unfortunately, it was soon discovered that there were a few ambiguities in its definition. Therefore, it was decided to use a strict variant of the OPB format to avoid any ambiguity and ease the parsing of files. This format can be described by a simple BNF grammar outlined below:

```
<unsigned_integer>::= <digit> | <digit><unsigned_integer>
<integer>::= <unsigned_integer> | "+" <unsigned_integer> | "-" <unsigned_integer>
<identifier>::= <letter_or_underscore>[<letters_or_digits_or_underscores>]
<relational_operator>::= ">=" | "="

<product>::= <integer> "*" <identifier> " "
<linearfunction>::= <product> | <product><linearfunction>
<constraint>::= <linearfunction> <relational_operator> <integer> ";"
<objective>::= "min:" <linearfunction> ";"

<comment>::= "*" <any_sequence_of_characters_other_than_EOL> <EOL>

<formula>::= <sequence_of_comments>
             <optional_objective>
             <sequence_of_comments_or_constraints>
```

The key points of our proposed format is that the objective function (if any) must be minimized, constraints must be 'greater than or equal to' or equality constraints, variables

names can be any legal identifier and numbers can be of any length. Another point is that it is easy to parse. To read the terms of a constraint, one only has to read in a loop an integer, then a star, then the variable name.

Some other points about this format are detailed below:

- A line starting with a '*' is a comment and can be ignored. Comment lines are allowed anywhere in the file.
- As a hint to perform memory allocation, the first line of the file will be a comment containing the word "#variable=" followed by a space and the number of variables in the file, then a space and the word "#constraint=" followed by a space and the number of constraints in the file. The space between the word and the number is mandatory to make parsing trivial. This information is only provided as a commodity for solvers which include a very limited parser. High quality provers are encouraged to ignore this information as it may not be accurate outside the evaluation environment (e.g. when a user creates a file by hand).
- Each non comment line must end with a semicolon ';'
- The first non comment line may be an objective function to minimize. It starts with the word "min:" followed by the linear function to minimize and terminated by a semicolon. No other objective function can be found after this first non comment line.
- A constraint is written on a single line and is terminated by a semicolon.
- An identifier represents the name of a Boolean variable (atom).
- Each identifier must be followed by a space (so that it can be read by a function which reads a word from the file)
- The negation of an atom A will not appear in the file (it will be translated to 1-A)
- An integer may contain an arbitrary number of digits. There must be no space between the sign of an integer and its digits.

A tiny example is given below:

```
* #variable= 5 #constraint= 3
*
* comments
*
min: 1*x2 -1*x3 ;
1*x1 +4*x2 -2*x5 >= 2;
-1 * x1 +4 * x2 -2 * x5 >= +3;
12345678901234567890*x4 +4*alpha3 >= 10;
```

To encourage the adoption of this strict variant of the OPB format, parsers for a few languages (C, C++ and Java) were made available.

To avoid any problem with the format of the benchmarks files, each instance was first normalized to adhere to this stricter format. Files which were obtained in other formats were systematically converted.

However, a few problems can still be attributed to this stricter format. Some solvers had problems with very long lines or with long variable identifiers, while other solvers wrongly assumed that a variable was called "x" followed by a number. Another problem which is not intrinsic to the format is the size of the integers. This is detailed in the next section.

4. The Big Integer issue

One problem that may occur when solving a linear pseudo-Boolean formula is integer overflow. Usually, programs use integers of size corresponding to the processor registers. On the usual 32 bits platforms, this means that the biggest positive integer is only 2,147,483,647 when using the int type (C/C++/Java). However, this limit can be easily reached. Using 64 bits integers gives a more comfortable limit but does not really solve the problem. In fact, it is easy and natural to get constraints with big integers. For example, the constraint $A = B + C$ where A, B, C are integers may be encoded as follows:

$$\sum_i 2^i.A_i = \sum_i 2^i.B_i + \sum_i 2^i.C_i$$

As soon as the size of A, B, C equals the size of the integers used by the solver, we get an integer overflow problem.

As far as pseudo-Boolean solvers are concerned, integer overflow can occur either during the input of the formula or during the resolution of the formula and will have different effects on the solver capabilities:

- during the input of the formula

  If a solver does not use big enough integers, it will fail to read some input files with large integers, or it will truncate some coefficients and give a random answer. This is a minor problem provided that this failure is either detected of at least documented in the solver manual.

- during the resolution of the formula

  This problem is more serious because it will break the correctness of the prover in ways that will be subtle to identify. Suppose that each constraint in the formula contains only small numbers (i.e. such that their sum fits into an integer). If the solver computes new constraints or simply new weights, it may from time to time overflow the limit of the integer it uses internally and give a wrong answer.

Such integer overflows were a concern for the evaluation because we expected to get wrong answers on some benchmarks triggering an integer overflow. This concern was justified since some solvers do provide wrong answers because of this issue.

On the other hand, integer overflows are easy to fix and are related to the implementation and not to the algorithm used by the solver. The use of a multiple precision integer library is enough to avoid this issue. However, numerical computations will be become slower with

the use of such library. Nevertheless, a sound solver is always preferable to a faster but unsound solver.

Our policy for this evaluation was the following

- to specify a format which does not hide this problem by specifying that integers may be of arbitrary size
- most of the benchmarks would use small integers to avoid integer overflows as much as possible
- any solver (subject or not to integer overflow) could be submitted
- to gather information about the integers internally used by each solver so as to explain possible failures

5. Set of benchmarks

During the build up to this first evaluation, an effort was made to gather the largest number of instances with pseudo-Boolean constraints. We were able to find several instances available on the web and others were submitted to the organizers. Additionally, a large number of integer linear programming problems in MPS format were converted to pseudo-Boolean optimization problems. In this section we present the benchmark set used in the evaluation. Moreover, we also describe the division of the benchmark set in several categories. Finally, we present the conversion process from MPS instances to pseudo-Boolean formulae.

5.1 Classification of Benchmarks

It is well-known that solvers using different strategies or techniques behave better than others depending on some characteristics of the benchmark instance. For example, pure SAT-based solvers are better in dealing with hard constrained instances while others are better dealing with information from the cost function [24]. Hence, we find it necessary to make a classification of the benchmarks in categories and analyze the results considering those sets of instances.

The two main sets of instances to consider are the optimization and the non-optimization instances. While in the first set the solver must find the optimum value for the cost function, in the latter it is sufficient to find a complete assignment that satisfies all problem constraints. However, there are several optimization instances that are unsatisfiable.

We should also consider that in some instances, variable coefficients are big integers and some solvers do not comply with it. Therefore, we divided the optimization set of instances into three categories (small, medium and big integers) depending on the value of the coefficients in the instance.

- **Small Integers (OPTSMALLINT)** : For all constraints, the sum of the coefficients is smaller than 2^{20} (20 bits). Benchmarks in this category should not cause any integer overflow.
- **Medium Integers (OPTMEDINT)**: For all constraints, no single coefficient is bigger than 2^{30} (30 bits). However, there is at least one constraint with a sum of

Table 1. Benchmark Categories

Categories	#benchs	Cl	Crd	PB	Cl&Crd	Cl&PB	Crd&PB	All
SAT/UNSAT	113	0	0	0	57	50	0	6
OPTSMALLINT	386	184	5	1	60	42	33	61
OPTMEDINT	191	0	28	57	5	1	61	39
OPTBIGINT	482	0	0	226	2	12	160	82
Total	1172	184	33	284	124	105	254	188

coefficients greater than 2^{20} (20 bits). Benchmarks in this category do not contain integers wider that the usual `int` variables. Solvers which do not learn constraints are probably safe but solver which learn new constraints may be faced with some integer overflows.

- **Big Integers (OPTBIGINT)**: There is at least one coefficient bigger than 2^{30} (30 bits). Benchmarks in this category will probably cause integer overflows.

It was observed that all non-optimization instances submitted to the evaluation have small integers. Hence, no distinction was made regarding the value of the coefficients in the non-optimization instances and one single category was created (**SAT/UNSAT**). Table 1 presents the number of benchmarks for each category. Also for each category, we present the number of instances considering the type of constraints. Instances with only a specific type of constraints appear first: propositional clauses (Cl), cardinality constraints (Crd) and general pseudo-Boolean constraints (PB). The following columns provide the number of instances that combine two types of constraints and the last column indicates the number of instances that contain all three types of constraints.

In Table 2 we present a short description of the several benchmark sets used in the evaluation. Observe that instances modelling different problem domains were used and in future evaluations, instances can also be categorized according to their domain of origin. However, for this evaluation, the number of instances from specific domains was considered to be too small. Hence, we chose to present a simpler categorization.

Figure 1 represents the distribution of the number of clauses, cardinality and pseudo-Boolean constraints for each instance. Each point of this 3D plot represents an instance file: the X coordinate is the number of clauses, the Y coordinate is the number of cardinality constraints and the Z coordinate is the number of pseudo-Boolean constraints contained in the file. The plot on figure 1 is in fact a subset of the global plot: only benchmarks where each coordinates are less than 1000 are displayed here. This zoom is representative of the global structure and shows interesting details. A 3D plot is generally poorly transcripted by a 2D projection and figure 1 is not an exception. However, the structure of the distribution which is clearly visible by interactively rotating the 3D graph is also perceptible on this snapshot. It appears clearly that our benchmark set is not uniformly distributed over the space of the number of clauses, cardinality and pseudo-Boolean constraints. We mainly have benchmarks with a majority of clauses or cardinality or pseudo-Boolean constraints (which appear along the axis on the plot). Some other benchmarks mostly contain only two

Table 2. Benchmark Descriptions

Category	#benchs	Short Problem Description
SAT/UNSAT	50	UCLID Benchmarks [20]
SAT/UNSAT	6	Progressive Party Problem [39]
SAT/UNSAT	57	FPGA switch-boxes [6]
OPTSMALLINT	40	Generated [40]
OPTSMALLINT	5	Basketball Scheduling Problem [39]
OPTSMALLINT	12	Radar Allocation Problem [39]
OPTSMALLINT	20	Converted from MPS to OPB format (submitted)
OPTSMALLINT	15	FPGA Routing [4]
OPTSMALLINT	17	Logic Synthesis [41]
OPTSMALLINT	156	Minimum-size Prime Implicant [32] DIMACS benchmarks [18]
OPTSMALLINT	10	Synthesis PTL-CMOS Circuits [43]
OPTSMALLINT	8	Travelling Tournament Problem [35]
OPTSMALLINT	7	Unknown Problem (submitted)
All Optimization Categories	181	Converted MIPLIB [1] MPS instances with $a = 13, b = 7$ (section 5.2)
All Optimization Categories	181	Converted MIPLIB [1] MPS instances with $a = 20, b = 10$ (section 5.2)
All Optimization Categories	184	Converted NetLib [2] MPS instances with $a = 13, b = 7$ (section 5.2)
All Optimization Categories	184	Converted NetLib [2] MPS instances with $a = 20, b = 10$ (section 5.2)
All Optimization Categories	223	Converted other [9, 27] MPS instances with $a = 13, b = 7$ (section 5.2)
All Optimization Categories	222	Converted other [9, 27] MPS instances with $a = 20, b = 10$ (section 5.2)

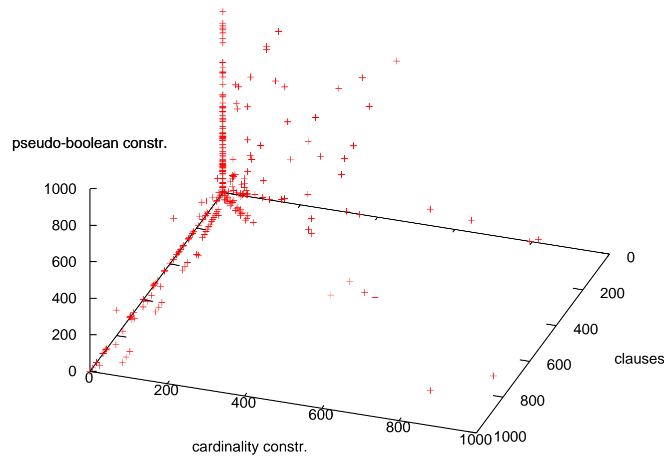

Figure 1. Distribution of the number of clauses, cardinality and pseudo-Boolean constraints in the benchmarks (zoomed)

Majority of clauses	
Category	#instances
SAT/UNSAT	58
OPTSMALLINT	249
Total	307

Minority of clauses	
Category	#instances
OPTSMALLINT	78
OPTMEDINT	161
OPTBIGINT	435
Total	674

Figure 2. Distribution of benchmarks with a majority/minority of clauses

kinds of constraints (with a fixed ratio). Very few benchmarks have all kinds of constraints with a balanced distribution.

This distribution suggests that an extra effort should be made next year to obtain a wider distribution of the benchmarks. Beyond the submission of new benchmarks, one possible solution would be to randomly modify some constraints in a benchmark file to generate a new file with another distribution.

Given the observed distribution of benchmarks, it might be interesting to consider another classification of benchmarks based on the ratio of the different kinds of constraints

Majority of cardinality constraints	
Category	#instances
OPTSMALLINT	28
OPTMEDINT	45
OPTBIGINT	35
Total	108

Minority of cardinality constraints	
Category	#instances
SAT/UNSAT	68
OPTSMALLINT	265
OPTMEDINT	70
OPTBIGINT	279
Total	682

Figure 3. Distribution of benchmarks with a majority/minority of cardinality constraints

Majority of pseudo-Boolean constraints	
Category	#instances
OPTSMALLINT	7
OPTMEDINT	68
OPTBIGINT	267
Total	342

Minority of pseudo-Boolean constraints	
Category	#instances
SAT/UNSAT	103
OPTSMALLINT	322
OPTMEDINT	56
OPTBIGINT	47
Total	528

Figure 4. Distribution of benchmarks with a majority/minority of pseudo-Boolean constraints

in the instance. We choose arbitrarily to define that there is a majority of a given kind of constraint when its represents more than 90% of all the constraints. Conversely, we define that there is a minority of a given kind of constraint when its represents less than 10% of all the constraints.

Obviously, this classification is neither exhaustive nor mutually exclusive. Some instances belong to none of these categories and instances with a majority of clauses necessarily belong to the set of instances with a minority of cardinality or pseudo-Boolean constraints. It is clearly noticeable from tables 2,3 and 4 that most instances with a majority of pseudo-Boolean constraints belong to the OPTBIGINT category (which some solvers did not support) and most instances with a majority of clauses belong to the OPTSMALLINT category.

5.2 The MPS instances

A number of integer linear programming problems are available on the web in MPS format. These problems did not seem too hard to translate to pseudo-Boolean constraints and were potentially an interesting source of industrial problems. Besides, we could notice that some of the pseudo-Boolean instances which were available at that time were already a translation from problems expressed in MPS format. As these problems could substantially increase the number of benchmarks used for the evaluation, we decided to translate these problems to pseudo-Boolean constraints.

The MPS input format was originally introduced by IBM to express linear and integer programs in a standard way. It is an old format, with fixed column width. Another peculiarity is that constraints are described in columns rather than lines, which means that it lists for each variable the constraints in which that variable appears instead of listing for each constraint the variables which appear in the constraint.

Variables and weights in the MPS format are real numbers. Variables can be constrained to be integers or even Booleans (0/1). A variable can be free (without bounds) or bounded by a upper or lower limit (or both). Constraints include the usual equality and inequality constraints.

The main problem for converting a problem in MPS format is that variables can have real values. To translate this to pseudo-Boolean constraints, we decompose a variable X into its binary representation as a fixed-point number: $X = \sum_{i=-a}^{b} 2^i.X_i$ In this decomposition, a represents the number of digits after the binary point and b represents the number of digits before the binary point. These two numbers are an important parameter of the translation. Greater numbers will provide more accuracy but will generate big integers in the translated

Table 3. The submitted solvers and their authors

solver	authors
bsolo	Vasco Manquinho and João Marques-Silva
galena	Donald Chai and Andreas Kuehlmann
minisat+	Niklas Eén and Niklas Sörensson
PBS4	Fadi Aloul and Bashar Al-Rawi
pb2sat+zchaff	Olivier Bailleux, Yacine Boufkhad, Olivier Roussel
Pueblo	Hossein Sheini and Karem Sakallah
sat4jpseudo	Daniel Le Berre, Mederic Baron, Anne Parrain, Olivier Roussel
vallst_0.9.258	Daniel Vallstrom

pseudo-Boolean constraints. Small a and b will generate smaller weights in the pseudo-Boolean constraint but will not be able to represent large values of X and will have a poor precision as well. Faced with this choice, it was decided to use two encodings: one with $a = 10$ and $b = 20$ and another one with $a = 7$ and $b = 13$. The name of the translated formulae indicates which value where used: they all follow the pattern `mps-v2-`b`-`a. The v2 simply indicates that this is the second version of the translator. Files named mps-v2-20-10-* therefore use real variables encoded on a total of 30 bits, while files named mps-v2-13-7-* use real variables encoded on a total of 20 bits.

Bounds on variables are used (when possible) to limit the size of their encoding. When a variable X is bound by $L \leq X \leq U$, it is rewritten as $X = L + X'$ where X' will be a positive variable that will be less or equal to $U - L$.

Once the encoding of each variable is chosen, each linear constraint is multiplied by the smallest coefficient so that all its weights become integer numbers. Clearly, this multiplication can create huge numbers. The biggest coefficient that was generated in an instance has 110 bits. These big integers are partly due to the number of digits of the coefficients in the original MPS file. It is questionable whether the number of digits after the decimal dot in the MPS files is really meaningful or is only an artefact caused by some output format.

384 instances mps-v2-20-10* and 385 instance mps-v2-13-7* where generated. Compared to the 403 instances which were submitted, the MPS instances are clearly over represented. Besides, some submitted instances are already a translation of some MPS files. If this was not considered as a major problem in this first evaluation, it is highly desirable to obtain a wider set of benchmarks for the next evaluation.

6. Description of the solvers

Eight solvers were submitted to the evaluation: *bsolo* [36, 24], *galena* [10], *minisat+* [30, 13], *PBS4* [5], *pb2sat+zchaff* [31, 29], *Pueblo* [15, 33], *sat4jpseudo* [21] and *vallst* [37, 38]. Table 3 presents the authors of each solvers. Table 4 reports their ability to deal with instances from the different categories (as defined in section 5.1). All solvers were presented to the evaluation as being able to solve PBS instances as well as PBO instances with small integers. However, only half the solvers have the ability to deal with big integers. None of the solvers uses local search.

Table 4. Ability of each solver to deal with different types of instances

solver	SAT	OPTSMALLINT	OPTMEDINT	OPTBIGINT
vallst_0.9.258	X	X		
galena	X	X	X	
PBS4	X	X	X	
Pueblo	X	X	X	
bsolo	X	X	X	X
minisat+	X	X	X	X
pb2sat+zchaff	X	X	X	X
sat4jpseudo	X	X	X	X

Most of the solvers are generalizations of SAT solvers adapted to deal with pseudo-Boolean constraints. In fact, *minisat+* and *pb2sat+zchaff* map the original pseudo-Boolean constraints into propositional clauses and use pure SAT solvers (*minisat* [13] and *zchaff* [42], respectively) to solve problem instances. Although these solvers were developed independently, they appear to share one common translation scheme to SAT based on some kind of BDD (Binary Decision Diagram). While *pb2sat+zchaff* uses only this encoding and often exceeds the memory limits, *minisat+* is more elaborated and implements two alternative encodings (based on sorting networks and binary adders respectively) which are used when the BDD encoding is too large.

All other solvers manipulate directly pseudo-Boolean constraints and also use SAT-based techniques, namely conflict analysis and constraint learning. However, while *vallst*, *PBS4* and *bsolo* use clause learning, *galena* [10] learns cardinality constraints, *Pueblo* [33] has a mixed learning scheme and learns both clauses and pseudo-Boolean constraints and *sat4jpseudo* learns pseudo-Boolean constraints.

Besides being SAT-based, another common feature is that all solvers in the evaluation also use lazy data structures to manipulate propositional clauses. However, only *Pueblo* and *sat4jpseudo* use lazy data structures to manipulate other types of constraints.

From an implementation point of view, *Pueblo*, *sat4jpseudo* and obviously *minisat+* are all based on the *minisat* framework.

The approach used by most solvers for solving PBO is the linear search on the value of the cost function, first proposed in [8]. In fact, the only two exceptions in using this method are *pb2sat+zchaff* and *bsolo*. While *pb2sat+zchaff* uses binary search on the value of the cost function, *bsolo* is a SAT-based branch and bound algorithm that uses lower bound estimation procedures, namely the maximum independent set of constraints [12] and linear programming relaxations (LPR) [23], to bound the search. Moreover, when using LPR, it also generates cutting planes [14] from the information provided by the LPR solution.

7. Experimental Conditions

7.1 Available resources

The solvers were run on a cluster of 32 computers kindly provided by Michal Kouril and the LINC Lab, Department of ECECS, University of Cincinnati. Each node of this cluster

is a bi processor Pentium III cadenced at 450MHz with 1GB RAM. The operating system was a Red Hat Enterprise Linux WS release 3 running linux kernel version 2.6.8.1 (SMP).

Each solver was allowed to run for 1200 seconds of CPU time and could use up to 900 MB of RAM.

7.2 Output requirements

Solvers were required to give their answer by two means: a message displayed on the standard output and a specific exit code for each possible answer. A solver was also allowed to output any data as long as each line started by "c " to define a comment line.

Solvers were asked to give one out of four possible answers:

- *"s SATISFIABLE" with exit code 10:* the solver has found a solution but either there is no function to optimize or it cannot prove that this solution gives the least value of the objective function.
- *"s OPTIMUM FOUND" with exit code 20:* the solver has found a model and it can prove that no other solution will give a value of the objective function strictly less than the one obtained with this model. Let v be the value of the objective obtained with the valuation output by the solver. Giving this result is a commitment that the formula extended with the constraint $objective < v$ is unsatisfiable.
- *"s UNSATISFIABLE" with exit code 30:* the solver can prove that the formula has no solution.
- *"s UNKNOWN" with exit code 0:* the solver is unable to tell anything about the formula

Invalid output was considered as an UNKNOWN answer. Whenever the solver answered "OPTIMUM FOUND" or "SATISFIABLE", it was required to output the best solution it had found on a line starting with "v " (as Value line). This solution had to define the value of each variable to avoid any ambiguity on the value of the objective function.

8. Experimentations

There has been two campaigns of experimentations during the evaluation. The first phase which took place from April to June 2005 allowed us to run several versions of the solvers and to detect the first bugs in the solvers implementation. Unfortunately, the bug correctives that were submitted by the authors still had some bugs. Therefore, the results that were presented to the SAT conference in June 2005 were not completely satisfying since every solver had at least a problem (but some of these problems were simple mistakes in the output of the solver).

To obtain more reliable results, it was decided to run the solvers one last time in September 2005. Authors were allowed to fix the bugs as well as to improve their solver. During this second phase, only one version of the solvers was allowed to run.

There were a few differences between the two phases. A new version of the program `runsolver` which controls the execution of a solver was used during the second phase. This second version fixed some problems encountered during the first phase. Unfortunately, as

will be seen in the next section, it introduces some time penalty which was not anticipated. Fortunately, as the main criterion to evaluate the solvers was not time but the number of instances it was able to solve in a fixed period of time, this had insignificant influence (if any) on the results. Another difference is that the solvers exit code was mostly ignored in the second phase and only the message output by the solver was considered (to avoid the strange case where a solver message output and its exit code did not match). At last, the classification of the answers of the solvers was slightly changed.

Each time a solver was run, some information was recorded about the host, the solver and the instance (including checksums). The solver output messages were also recorded (up to a global limit of 1 MB) as well as the information collected by the `runsolver` program. All these form the execution trace of the solver and is available on the evaluation web site.

Each solution provided by a solver was verified by an independent program which checked that each constraint was satisfied and computed the value of the objective function given by this solution

8.1 The `runsolver` program

Each solver was run under the control of another program called `runsolver`. The task of the `runsolver` program is to ensure that the solver will not take too much resources (especially time and memory) as well as gathering some data about the running solver (CPU time, exit code,...).

Two different versions of the `runsolver` program were used during the evaluation: one during the first phase, and a much improved version during the second phase of the evaluation.

The first version of `runsolver` starts by enforcing some resource limits, then launches the solver with its arguments and waits until the solver completes its execution. Every ten seconds, it fetches some data about the system (average load) as well as informations on the solver process such as the current memory consumption and CPU time elapsed so far (obtained from the `/proc/*/stat*` files). On completion of the solver process, `runsolver` prints the child exit code, as well as the CPU time used by the process. `runsolver` enforces limits on the CPU time, the memory usage and the stack size through the `setrlimit` system call. When a solver exceeds these limits, it is killed by the system (through signal SIGKILL) and is not given any chance to output a partial result. To give solvers a chance to output a partial result when the time limit is exceeded, the `runsolver` program sends a `SIGTERM` signal to the solver before it reaches the system limit. Afterwards, the solver has two seconds to output the best result it got so far and after this delay, it is killed by `runsolver`.

All in all, the first version of `runsolver` is merely an integration of the `ulimit` and `time` system commands with just a few improvements. This version has mainly two weaknesses: it does not support correctly multi-processes solvers and cannot send a SIGTERM when the solver exceeds the memory limit.

The problem with multi-processes solvers is that a child process CPU time is reported to its parent process (through the `wait()` system call) only when the child process exits. This means that the first version of `runsolver` which only watched the parent process CPU time could not notice that the child exceeded the time limit until the child exited (when it

was actually too late). This is what happens when a shell script is used to run the solver since there are 2 processes. For this reason, we observed that a solver which used a shell script to control the search actually used up to 4800 seconds (instead of the normal limit of 1200 seconds).

Another point is that if the first version of `runsolver` could easily anticipate the system CPU time limit (remember it gathered data about the solver every ten seconds), it was absolutely unable to anticipate a memory exhaustion because the program may request some memory from the system at any time and at any rate. The only way to anticipate the violation of the memory limit (and send a SIGTERM to the solver) is to intercept the system calls. This is what the second version of `runsolver` does. This was inspired by two programs: `strace` [3] which prints the system calls performed by a program and `s4g` [28] which is a generic sandbox for programs run on a grid.

The second version of `runsolver` intercepts the system calls by running the solver in a trace mode. In this mode, the solver will be suspended by the kernel each time it enters or exits a system call and the `runsolver` process will be notified by a `SIGTRAP` signal. The controlling process can examine the system call and intercept its parameters and result. One big advantage is that this technique does not require any privilege since it is accessible to any process through the `ptrace()` call. Another advantage is that it works on any kind of binary (statically or dynamically linked) without any modification.

The system calls of interest in our case are `clone` and `exit` to track the creation and deletion of processes or threads, and the `brk, mmap, munmap, mremap` system calls to track memory allocation. We also intercept `open, execve` and the sockets system calls to check if the solver respects the evaluation policy. In this version, we only log the file and network accesses but the next version of `runsolver` will actively control these system calls and stop the process as in `s4g` as soon as the solver violates the policy (such as trying to connect to a remote host).

The `runsolver` program easily maintains a list of the processes created by the solver and adds the CPU time of all its child processes to decide if the solver must be stopped by a `SIGTERM`. Tracking the memory usage of the solver is a bit more difficult because there are a number of system calls to allocate memory with subtle interactions. The current version of the program maintains a upper bound of the memory used by the solver and its children and, when this bound exceeds the memory limit, it fetches the actual memory usage of the processes in the `/proc/*/stat*` files. When the memory used by the process and all its children is over the imposed limit, it sends a `SIGTERM` to the solver and all its children.

Stopping a solver when it uses too much memory is more difficult than when the time limit is exceeded. In fact, we impose two limits: a soft limit which sends a `SIGTERM` to the solver and a hard limit which will immediately kill the solver. The hard limit was set as the soft limit plus 50 MB. For these reasons, a solver should not allocate too much memory in a single call to avoid bumping into the hard limit immediately. Besides, when it is sent a `SIGTERM`, a solver should be very careful about its memory usage to avoid reaching the hard limit while it outputs its results (which might be difficult in some languages such as Java).

Intercepting system calls has necessarily a side effect: it slows down the solver. However, the solver is only stopped when it performs system calls and, as it should not happen that often in a pseudo-Boolean or SAT solver, we could expect only slightly different perfor-

Table 5. Difference of time measured by the `time` command and the `runsolver` program with or without interception of system calls (all times in seconds). Host has a single hyperthreaded Pentium 4 at 2.8GHz with 1GB RAM and kernel 2.4.20smp (RedHat 9)

solver	`time`			`runsolver v1`			`runsolver v2`		
	user	system	total	user	system	total	user	system	total
Pueblo	7.277	0.24	7.517	7.257	0.16	7.417	8.982	3.12	12.102
vallst_0.9.258	16.85	1.205	18.055	16.825	1.189	18.014	25.92	18.09	44.01
sat4jpseudo	3.73	0.1	3.83	3.8	0.1	3.9	6.5	0.17	6.67
	9.165	0.245	9.41	10.525	0.4	10.925	12.3	0.3	12.6

mances. Table 5 compares the time measured by the `time` command with the time measured by the two versions of the `runsolver` program for a few different solvers on bench mps-v2-20-10/MIPLIB/miplib/normalized-mps-v2-20-10-p0040.opb. The Pueblo solver is a classic mono-process program. The *vallst* solver uses a script to run another solver in a loop. Therefore, it uses several processes. The *sat4jpseudo* solver is written in Java and therefore uses multiple threads. These experiments were run on a Pentium 4 (HT) at 2.8GHz with 1GB RAM and kernel 2.4.20smp (RedHat 9). The reported time is the average of 4 runs.

The times measured for solver *sat4jpseudo* (written in Java and run by java -server -Xms650M -Xmx650M) are extremely different from one run to another. We reported the average of the two fastest runs on the first line and the average of the two slowest runs on the second line. We have no explanation yet why these times are so different.

We can check that there is no real difference between the time command and the first version of `runsolver`. However, the impact of the second version of `runsolver` is quite noticeable, from a rough 50% (which could be considered as acceptable) to more than 240% for *vallst* (which is clearly not acceptable).

To add some confusion, it appears that the time penalty induced by the interception of signals is different form one version of the kernel to another and from a single CPU host to a multi processor host.

Therefore, we are still missing the best way to time a solver and impose accurate restrictions on the resources it uses. The first version of `runsolver` gives accurate timing but does not enforce correct limits. It can be fooled by multi-processes solvers. The second version of `runsolver` has a good support for multi-processes but has a time penalty which is too high in some cases.

Hopefully, the time penalty induced by the second version of `runsolver` had an insignificant impact on the number of instances solved by each solver. As can be checked on the evaluation web site, there are very few differences between the number of unsatisfiable formulae or optimums found during the first phase and the second phase. However, the `runsolver` program must clearly be improved for the next evaluation.

9. Results analysis

In this section we present the experimental results of the evaluation for the different solvers. For each solver we present the information described in Table 6. Results are first presented

Table 6. Classification of the possible outcomes of a solver

Answer	**Description**
UNSAT.	solver proved unsatisfiability ("s UNSATISFIABLE" was output)
OPT.	solver found an optimum solution ("s OPTIMUM FOUND" was output), the provided implicant satisfies each constraint and no solver gave a better solution
SAT.	solver found a solution ("s SATISFIABLE" was output) and the provided implicant satisfies each constraint
SAT (timeout)	solver exceeded the time limit but was able to find a solution ("s SATISFIABLE" was output) and the provided implicant satisfies each constraint
SAT (out of mem.)	solver exceeded the memory limit but was able to find a solution ("s SATISFIABLE" was output) and the provided implicant satisfies each constraint
UNKNOWN	solver could not decide ("s UNKNOWN" was output)
UNKNOWN (timeout)	solver exceeded the time limit and gave no answer. These runs are considered to give result UNKNOWN
UNKNOWN (out of mem.)	solver exceeded the memory limit and gave no answer. These runs are considered to give result UNKNOWN.
UNKNOWN (exit code)	solver did not output a solution line and terminated with an unexpected exit code (different of 0, 10, 20 and 30). These runs are considered to give result UNKNOWN.
Sig. Caught	solver was terminated by a signal (SIGSEGV for example) and did not output a solution line
NO CERT.	solver answered SATISFIABLE but either did not provide a certificate (the "v " line) or either gave a truncated certificate (which does not end in a new line)
WRONG CERT.	solver gave an implicant but it appears that it does not satisfy every constraint
WRONG OPT.	solver found an optimum solution ("s OPTIMUM FOUND" was output) but there exists an implicant which gives a better value to the objective function
WRONG UNSAT.	solver proved unsatisfiability but was wrong ("s UNSATISFIABLE" was output)

for each of the categories defined in section 5.1. Moreover, in this section we also present an analysis of partial solutions from the different solvers. Finally, we discuss bugs detected in solvers.

Tables 7,8,9 and 10 present the results for each instance category. The overall results for all instances are presented in Table 11. However, we note that solvers cannot be compared solely using the results from Table 11 since they were not run on the same number of benchmarks.

Table 7 contains the results for instances with no optimization function. For these instances, *Pueblo* have the best results for both UNSAT and SAT instances. *PBS4* also performed very well in UNSAT instances, being able to solve as many as *Pueblo*, but using less time (see Figure 6). Both *galena* and *bsolo* have the lowest performances (specially in

Table 7. Results of the second phase for category "no optimization function" (SAT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	113	36	0	8	0	0	69	0	0	0	0	0	0	0	0
galena	113	36	0	7	0	0	70	0	0	0	0	0	0	0	0
minisat+	113	43	0	0	0	0	0	35	0	0	0	*35*	0	0	0
PBS4	113	61	0	28	0	0	0	24	0	0	0	0	0	0	0
Pueblo	113	61	0	42	0	0	0	10	0	0	0	0	0	0	0
sat4jpseudo	113	52	0	17	0	0	0	44	0	0	0	0	0	0	0
vallst_0.9.258	113	38	0	29	0	0	0	46	0	0	0	0	0	0	0
pb2sat+zchaff	113	42	0	36	0	0	0	35	0	0	0	0	0	0	0

Table 8. Results of the second phase for category "optimization, small integers" (OPTSMALLINT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	386	10	159	159	21	0	31	0	6	0	0	0	0	0	0
galena	386	9	98	135	0	0	140	0	0	0	0	0	0	0	*4*
minisat+	386	10	176	0	0	0	0	78	1	1	0	*120*	0	0	0
PBS4	386	10	133	0	0	0	0	243	0	0	0	0	0	0	0
Pueblo	386	10	160	182	0	0	0	33	0	0	0	0	*1*	0	0
sat4jpseudo	386	10	120	0	225	0	1	29	0	0	0	*1*	0	0	0
vallst_0.9.258	386	10	131	4	0	0	0	231	0	0	0	0	*3*	0	*7*
pb2sat+zchaff	386	10	136	0	148	10	0	33	7	42	0	0	0	0	0

SAT instances). *minisat+* did not provide the necessary certificates for SAT instances (see section 9.2).

For the optimization instances with small integers, results are presented in Table 8. All solvers (with the exception of *galena*) were able to solve the same number of UNSAT instances. In Figure 8 we can notice that *minisat+* has the best performance for these instances. This result is only natural since in 8 of the 10 instances all constraints are propositional clauses.

In the category of small integers, *minisat+* is the solver with more instances for which it was able to prove optimality. *Pueblo* and *bsolo* are also very effective to find the optimum solution. In fact, *Pueblo*, *bsolo* and *sat4jpseudo* were able to find solutions (either optimum or approximations) for almost 90% of instances. Observe that *sat4jpseudo* was able to prove optimality to a smaller number of instances, probably due its mechanism to deal with the cost function. Nevertheless, it was the solver that was able to output more certificates in this category. In Figure 7, we can also see that the curve of *pb2sat+zchaff* is able to pass *PBS4* and *vallst_0.9.258* as the time limit increases. Hence, this hints that for higher time limits, *pb2sat+zchaff* would improve on its results.

Table 9. Results of the second phase for category "optimization, medium integers" (OPTMEDINT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	191	0	28	78	4	0	50	5	26	0	0	0	0	0	0
galena	191	4	5	15	0	0	109	0	0	0	51	0	0	0	*7*
minisat+	191	0	24	0	0	0	0	92	7	1	0	*67*	0	0	0
PBS4	191	0	33	0	0	0	0	158	0	0	0	0	0	0	0
Pueblo	191	0	34	74	0	0	48	35	0	0	0	0	0	0	0
sat4jpseudo	191	2	19	0	107	0	1	62	0	0	0	0	0	0	0
pb2sat+zchaff	191	0	14	0	14	2	0	56	31	71	3	0	0	0	0

Table 10. Results of the second phase for category "optimization, big integers" (OPTBIGINT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	482	90	9	81	1	1	241	5	54	0	0	0	0	0	0
minisat+	482	103	26	0	0	0	0	239	41	9	0	*64*	0	0	0
sat4jpseudo	482	85	3	12	157	0	4	218	0	0	1	*2*	0	0	0
pb2sat+zchaff	482	8	11	0	11	0	0	69	119	259	5	0	0	0	0

In the category with medium integers (Table 9), *Pueblo* and *PBS4* were able to solve and prove optimality to more instances. In Figure 9 we can also note that *Pueblo* takes less time to prove optimality than *PBS4*. As in the small integer category, *sat4jpseudo* is the solver able to provide more certificates, but unable to prove optimality for the vast majority of instances inside the time limit. Because very few instances in the category with medium integers could be proved unsatisfiable, no meaningful graph could be drawn for the UNSAT answers in that category.

In the category with big integers, *minisat+* was the solver able to find more optimum values to problem instances. For these instances, *bsolo* does not use cuts since the linear programming package used with the solver has precision problems for these instances. Nevertheless, it was able to provide certificates (as well as proving unsatisfiability) to a large number of instances. *pb2sat+zchaff* was unable to find solutions for most instances and only proves unsatisfiability to a small number of instances. Considering Figure 11, it seems that *pb2sat+zchaff* finds it difficult to convert pseudo-Boolean and cardinality constraints to propositional clauses. Notice that for those instances that is able to solve, it does not take much time.

The overall results of the evaluation are presented in Table 11. Remember that the solvers were not run on the same number of instances since some of them had no support for some category. These results show that *minisat+* was able to prove unsatisfiability and optimality to a larger number of instances than other solvers. Hence, the approach of converting pseudo-Boolean formulations to propositional clauses seems to be competitive

Table 11. Results of the second phase for all categories

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	1172	136	196	326	26	1	391	10	86	0	0	0	0	0	0
galena	690	49	103	157	0	0	319	0	0	0	51	0	0	0	*11*
minisat+	1172	156	226	0	0	0	0	444	49	11	0	*286*	0	0	0
PBS4	690	71	166	28	0	0	0	425	0	0	0	0	0	0	0
Pueblo	690	71	194	298	0	0	48	78	0	0	0	0	*1*	0	0
sat4jpseudo	1172	149	142	29	489	0	6	353	0	0	1	*3*	0	0	0
vallst_0.9.258	499	48	131	33	0	0	0	277	0	0	0	0	*3*	0	*7*
pb2sat+zchaff	1172	60	161	36	173	12	0	193	157	372	8	0	0	0	0

Table 12. Results of the second phase for categories "no optimization function" (SAT) and "optimization, small integers" (OPTSMALLINT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	499	46	159	167	21	0	100	0	6	0	0	0	0	0	0
galena	499	45	98	142	0	0	210	0	0	0	0	0	0	0	*4*
minisat+	499	53	176	0	0	0	0	113	1	1	0	*155*	0	0	0
PBS4	499	71	133	28	0	0	0	267	0	0	0	0	0	0	0
Pueblo	499	71	160	224	0	0	0	43	0	0	0	0	*1*	0	0
sat4jpseudo	499	62	120	17	225	0	1	73	0	0	0	*1*	0	0	0
vallst_0.9.258	499	48	131	33	0	0	0	277	0	0	0	0	*3*	0	*7*
pb2sat+zchaff	499	52	136	36	148	10	0	68	7	42	0	0	0	0	0

with pure pseudo-Boolean solvers. *bsolo* and *Pueblo* are also able to prove optimality to a large number of instances due to their specific techniques. *bsolo* is more effective in instances with small integers since the linear programming relaxations and cut generation are more effective. The learning mechanisms and lighter data structures from *Pueblo* were also able to provide good results in several instances, in particular for the SAT category. Observe from Figure 18 that *sat4jpseudo* finds it very hard to prove optimality. However, there are several instances for which no other solver was able to find a partial solution. Therefore, *sat4jpseudo* was a valuable contribution to this evaluation. *PBS4* also has some good results, specially in the UNSAT instances. *pb2sat+zchaff* also has sound overall results. Finally, *vallst_0.9.258* and *galena* have a more erratic behavior. Moreover, for some instances these solvers provided wrong answers. This issue is discussed in section 9.2.

Table 12 presents the overall results for the non-optimization instances and optimization instances with small integers. These are the categories which all solvers presented in the evaluation are able to solve, since the author of *vallst_0.9.258* declared that the correctness of his solver was not guaranteed for the remaining categories of instances. Figure 12 shows than *minisat+* and then *Pueblo* and *bsolo* are the best solvers to give a positive answer (satisfiability or optimality) in these two categories. In contrast, *PBS4* and *Pueblo* are the

Table 13. Results of the second phase for all categories, except "optimization, big integers" (OPTBIGINT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	690	46	187	245	25	0	150	5	32	0	0	0	0	0	0
galena	690	49	103	157	0	0	319	0	0	0	51	0	0	0	*11*
minisat+	690	53	200	0	0	0	0	205	8	2	0	*222*	0	0	0
PBS4	690	71	166	28	0	0	0	425	0	0	0	0	0	0	0
Pueblo	690	71	194	298	0	0	48	78	0	0	0	0	*1*	0	0
sat4jpseudo	690	64	139	17	332	0	2	135	0	0	0	*1*	0	0	0
pb2sat+zchaff	690	52	150	36	162	12	0	124	38	113	3	0	0	0	0

Table 14. Results of the second phase for all categories, including "optimization, big integers" (OPTBIGINT)

Solver Name	#benchs	unsat.	opt.	satisfiable			unknown				sig.	no	wrong		
					TO	MO		TO	MO	Exit	caught	cert.	cert.	opt.	unsat.
bsolo	1172	136	196	326	26	1	391	10	86	0	0	0	0	0	0
minisat+	1172	156	226	0	0	0	0	444	49	11	0	*286*	0	0	0
sat4jpseudo	1172	149	142	29	489	0	6	353	0	0	1	*3*	0	0	0
pb2sat+zchaff	1172	60	161	36	173	12	0	193	157	372	8	0	0	0	0

best solvers to prove unsatisfiability (see Figure 13) in these categories. When the kind of answer is not distinguished (see Figure 14), *Pueblo* and *minisat+* appear as the best solvers.

Table 13 presents the overall results considering all benchmark instances, except optimization instances of the big integer category. In the call for solvers for the evaluation, we did not require that all solvers be able to deal with big integers. This option was taken in order to be able to gather the largest number of solvers. *vallst_0.9.258* does not appear in this table since it did not enter the OPMEDINT category. It can be seen in Figures 15, 16 and 17 that *minisat+*, *Pueblo* and *bsolo* are the leading solvers for proving optimality. *PBS4* and *Pueblo* have the best results on unsatisfiable instances. All in all, in these combined categories, *Pueblo* has the best results closely followed by *minisat+*.

Table 14 presents the overall results considering all benchmark instances, including optimization instances of the big integer category. Only solvers which have support for the OPTBIGINT category are shown. In comparison with Table 13, the major changes when considering instances with big integers is that *bsolo*, *minisat+* and *sat4jpseudo* are able to provide more UNSAT answers. The number of SAT answers of *bsolo* and *sat4jpseudo* also increases significantly if we consider big integer instances. Figures 18, 19 and 20 show that *minisat+* is overall the leading solver out of the solvers with support for big integers. *bsolo* also appears as a strong solver. *sat4jpseudo* has very good results on UNSAT instances.

After the evaluation, many instances are still unresolved. For most of the optimization benchmarks, the optimum value of the cost function is unknown and for those where a

Table 15. Overall benchmark results by category

Categories	#benchs	UNSAT	SAT	OPT.	Unknown
SAT/UNSAT	113	61	42	0	10
OPTSMALLINT	386	10	150	202	24
OPTMEDINT	191	6	106	41	38
OPTBIGINT	482	113	157	26	186
Total	1172	190	455	269	258

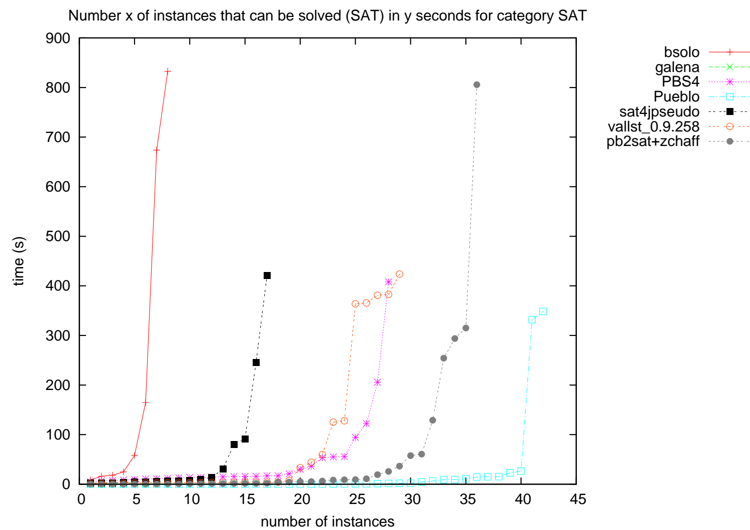

Figure 5. Number x of instances that can be solved (SAT answers only) in y seconds for category SAT

partial solution is known, there is no idea of how far the optimum value is. In Table 15 we present the number of instances for each category already solved, as well as the number of unresolved instances.

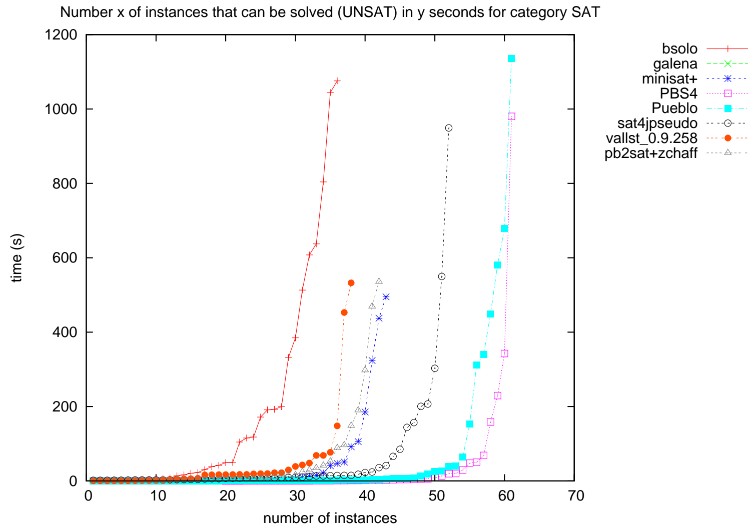

Figure 6. Number x of instances that can be solved (UNSAT answers only) in y seconds for category SAT

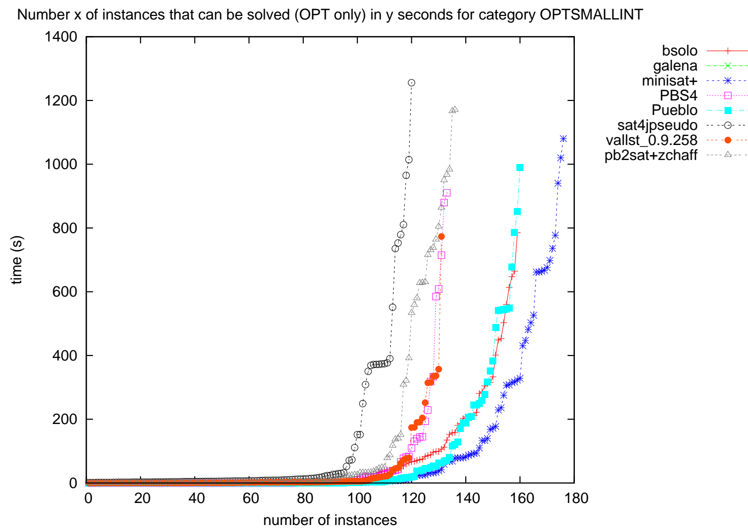

Figure 7. Number x of instances that can be solved (OPT answers only) in y seconds for category OPTSMALLINT

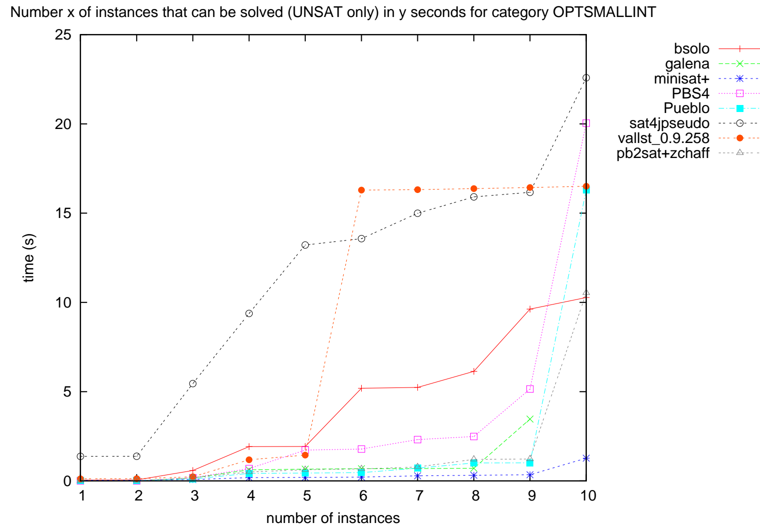

Figure 8. Number x of instances that can be solved (UNSAT answers only) in y seconds for category OPTSMALLINT

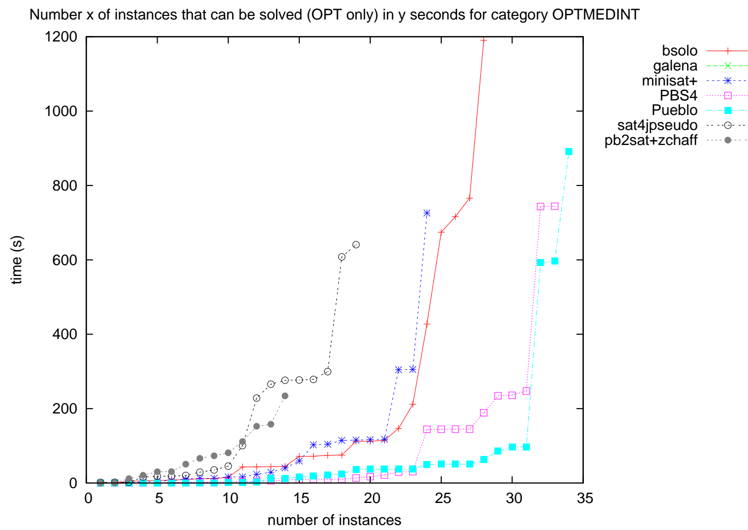

Figure 9. Number x of instances that can be solved (OPT answers only) in y seconds for category OPTMEDINT

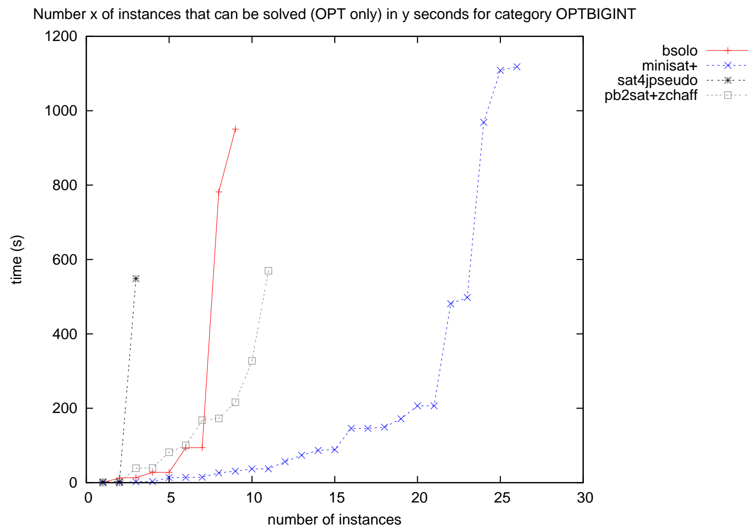

Figure 10. Number x of instances that can be solved (OPT answers only) in y sec. for category OPTBIGINT

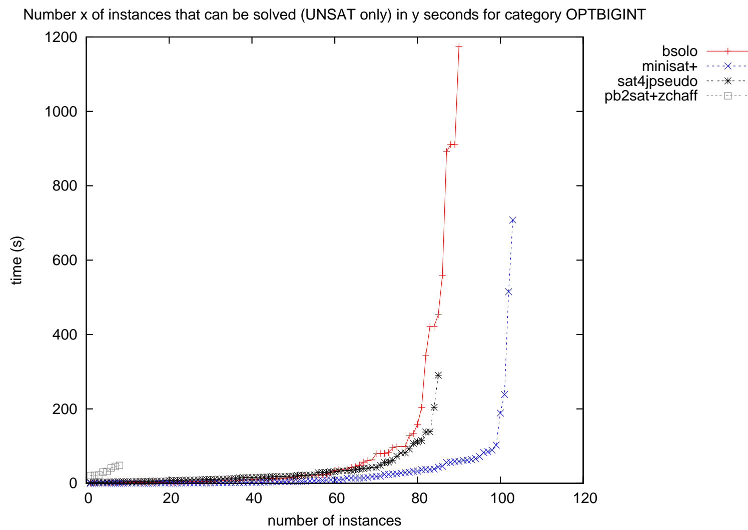

Figure 11. Number x of instances that can be solved (UNSAT answers only) in y sec. for category OPTBIGINT

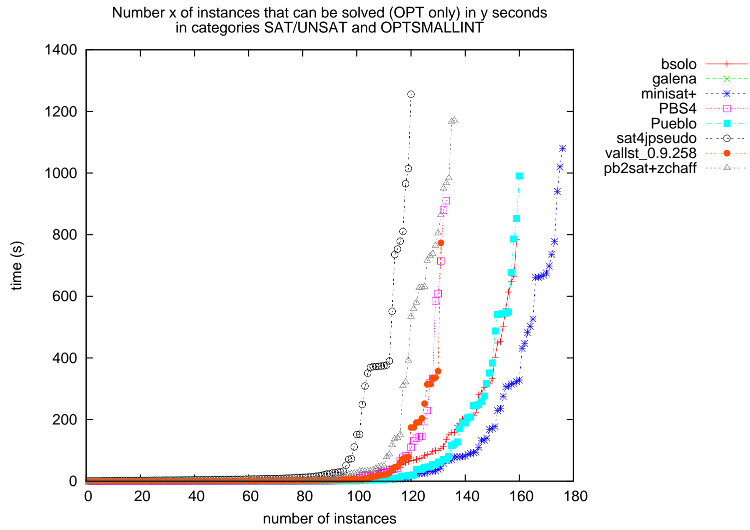

Figure 12. Number x of instances that can be solved (OPT answers only) in y seconds for categories SAT/UNSAT and OPTSMALLINT

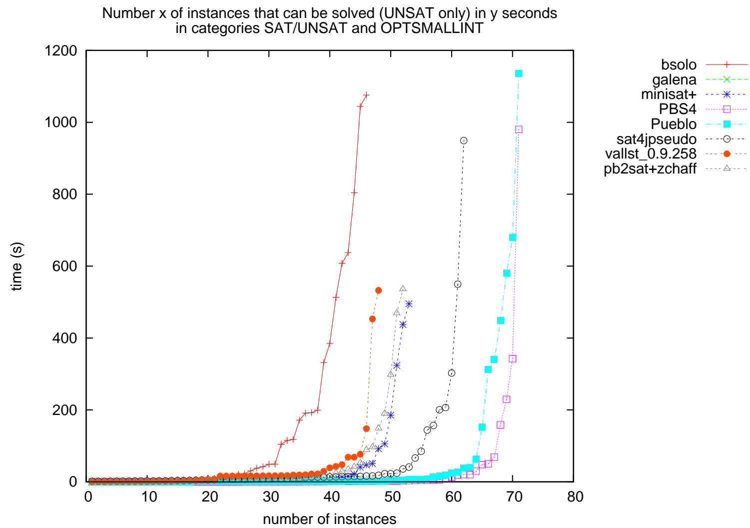

Figure 13. Number x of instances that can be solved (UNSAT answers only) in y seconds for categories SAT/UNSAT and OPTSMALLINT

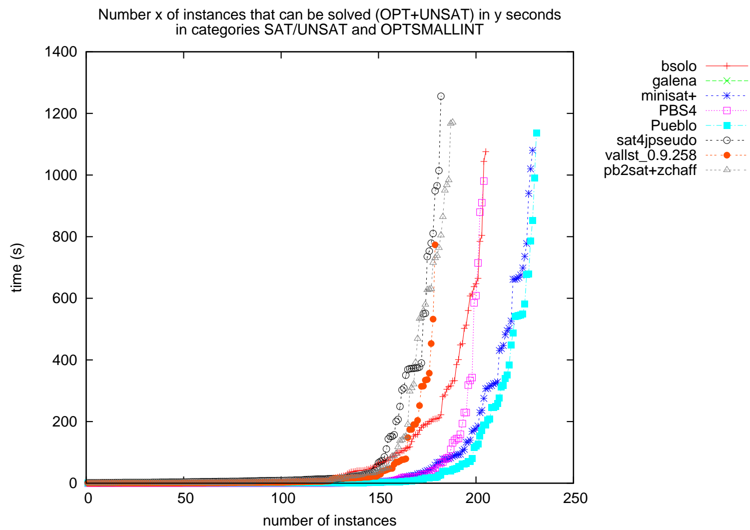

Figure 14. Number x of instances that can be solved (OPT+UNSAT answers) in y seconds for categories SAT/UNSAT and OPTSMALLINT

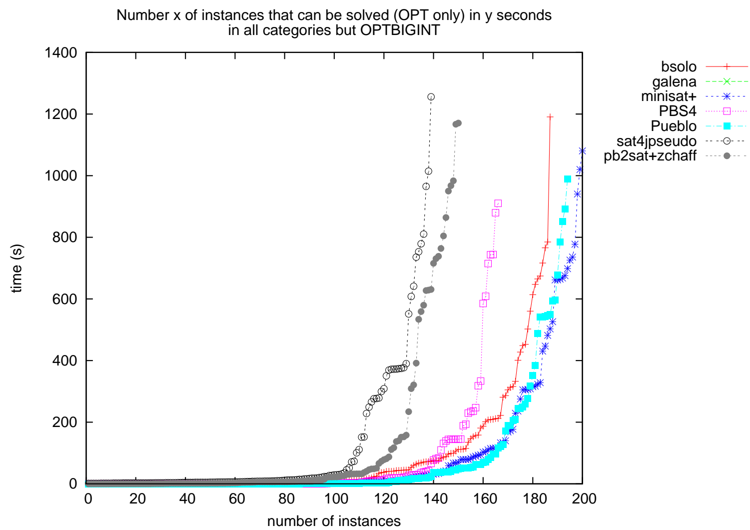

Figure 15. Number x of instances that can be solved (OPT answers only) in y seconds for all categories but OPTBIGINT

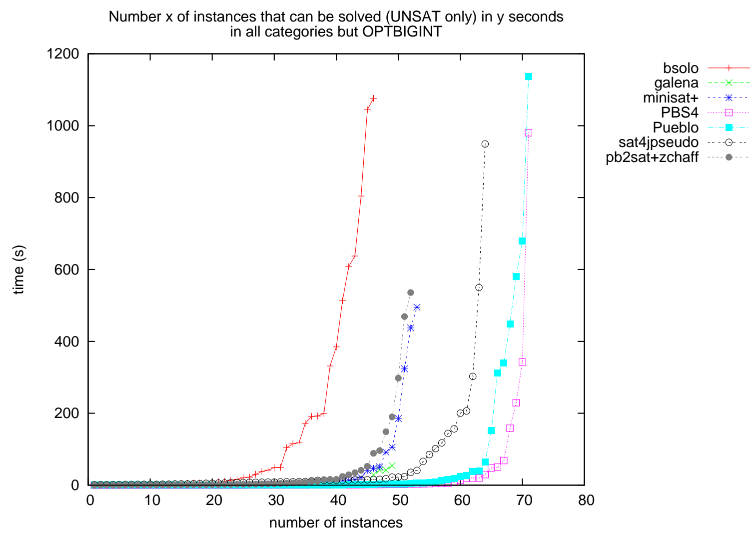

Figure 16. Number x of instances that can be solved (UNSAT answers only) in y seconds for all categories but OPTBIGINT

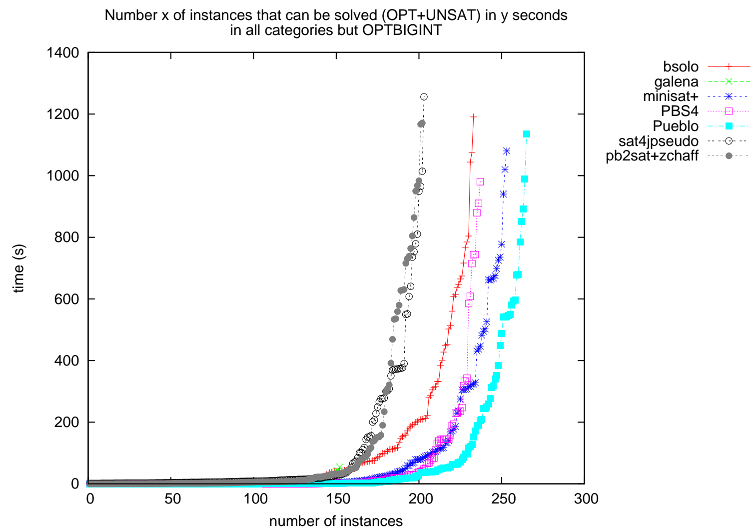

Figure 17. Number x of instances that can be solved (OPT+UNSAT answers) in y seconds for all categories but OPTBIGINT

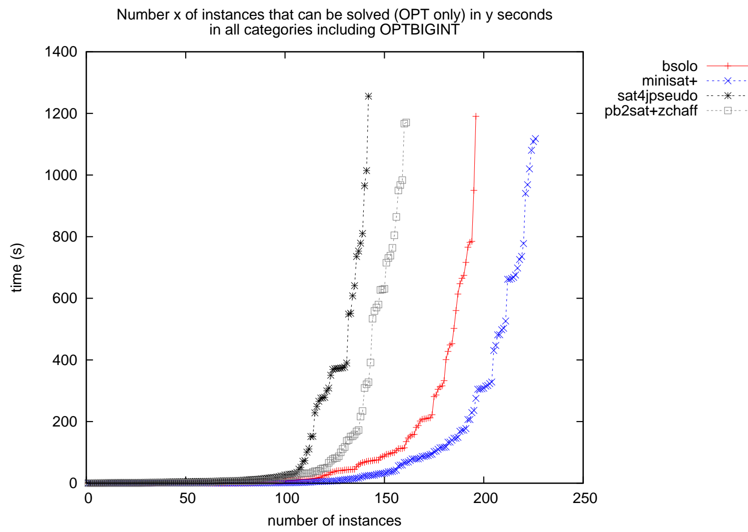

Figure 18. Number x of instances that can be solved (OPT answers only) in y seconds for all categories

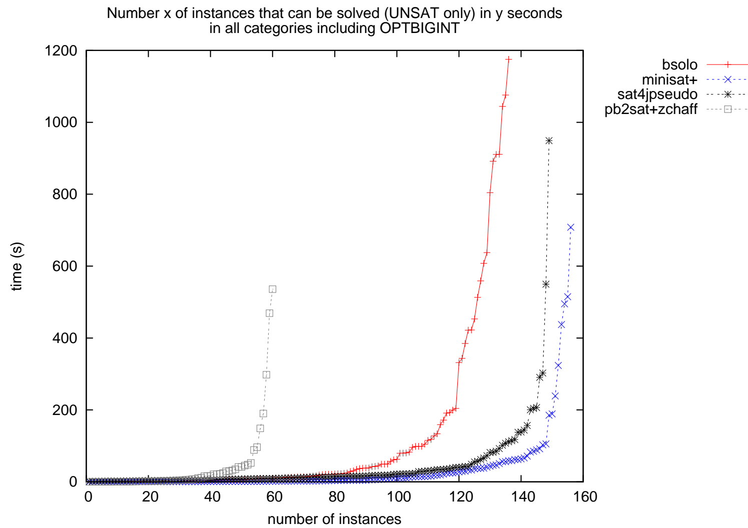

Figure 19. Number x of instances that can be solved (UNSAT answers only) in y seconds for all categories

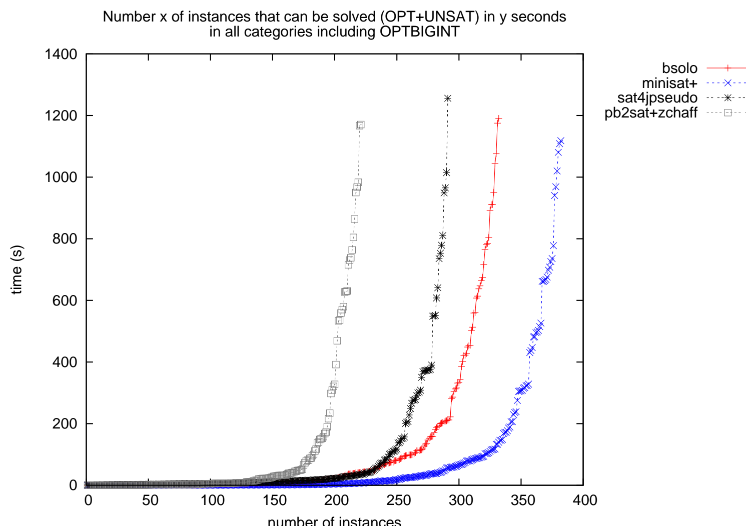

Figure 20. Number x of instances that can be solved (OPT+UNSAT answers) in y seconds for all categories

Figures 21 to 24 present the solvers performances on another classification of the benchmarks based on the majority or minority of a kind of constraints. Recall from section 5.1 that we defined that there is a majority of a given kind of constraint when its represents more than 90% of all the constraints. Conversely, we defined that there is a minority of a given kind of constraint when its represents less than 10% of all the constraints. This classification is nor exhaustive, nor mutually exclusive. However, it can help us determine if some solvers are more suitable to a given kind of constraints that other and therefore presents some interest.

Graphs concerning the influence of cardinality constraints are not presented because there are too few points to draw any conclusion.

It can be seen on Figure 21 that the proportion of clauses does not seem to influence the relative ranking of solvers concerning the search of an optimum solution. *vallst_0.9.258* is an exception but this may be caused by the limited number of categories it supported.

Figure 22 shows that *pb2sat+zchaff* is more successful in proving optimality than *bsolo* when there is a majority of pseudo-Boolean constraints. Conversely, *bsolo* is more successful than *pb2sat+ zchaff* when there is a minority of pseudo-Boolean constraints. An explanation is certainly that instances with a majority of pseudo-Boolean constraints mostly contain big integers and *bsolo* does not use cuts in this case since the linear programming package used with the solver has precision problems for these instances.

The difference that can be observed on Figure 23 between unsatisfiable instances with a majority or minority of clauses is certainly caused by the number of unsatisfiable instances in category OPTBIGINT and is a simple artefact. At last, the predominance of *PBS4* and *Pueblo* on unsatisfiable instances with a minority of pseudo-Boolean constraints is consistent with the results of Figure 16 for unsatisfiable instances without big integers. As instances

Table 16. Number of Best Results for each solver

Solver	SAT/UNSAT	OPTSMALLINT	OPTMEDINT	OPTBIGINT	Total
bsolo	44	271	62	136	513
galena	43	112	10	0	165
minisat+	43	186	24	129	382
PBS4	89	143	33	0	265
Pueblo	103	207	70	0	380
sat4jpseudo	69	154	64	208	495
vallst_0.9.258	67	141	34	0	242
pb2sat+zchaff	78	146	15	19	258

with a majority of pseudo-Boolean constraints mostly belong to the OPTBIGINT category, it is hardly surprising that solvers with support for big integers be successful in this category.

To sum up, beyond the differences that can be explained by some implementation point or by our imperfect benchmarks set, it does not seem to be obvious differences of behavior that could be linked to the different ratios of a kind of constraint.

9.1 Evaluating partial solutions

In optimization instances, finding the optimum value is the main goal. However, for many instances that cannot be achieved inside a given time limit. As shown in Table 15, for most optimization instances, only partial solutions were obtained. Nevertheless, we need to evaluate how good these partial solutions are.

In Table 16 we can see how many *best* solutions were provided by each solver. We consider as *best* solutions the UNSAT solutions, the SAT solutions for the non-optimization category and when the optimum value is found. Moreover, we should also consider as *best*, solutions to optimization instances for which no other solver was able to find a better solution.

Clearly, *bsolo* and *sat4jpseudo* are the solvers able to provide the largest number of *best* solutions. However, remember that solvers were not run in the same number of benchmark instances. Moreover, note that *minisat+* was unable to output certificates for its partial solution answers. Hence, we did not consider *minisat+* partial solutions, since we were unable to verify its correctness. Nevertheless, *Pueblo* has the best results in SAT and optimization instances with medium integers, while *bsolo* is clearly the solver with more *best* solutions in optimization instances with small integers. *sat4jpseudo* has best results in big integer category, since in several instances it was the only solver able to output a certificate.

We can also observe a different perspective by considering a solver contribution to the evaluation. We say that a solver provides a contribution to the evaluation if it is the only solver to give a *best* answer to an instance. In Table 17 we present the results for each solver in each category.

These clearly show that *bsolo* (in the small integer category) and *sat4jpseudo* (in the big integer category) are the main contributers. For solvers with 0 values, this indicates that there is not a particular instance for which only that solver is able to give a best solution.

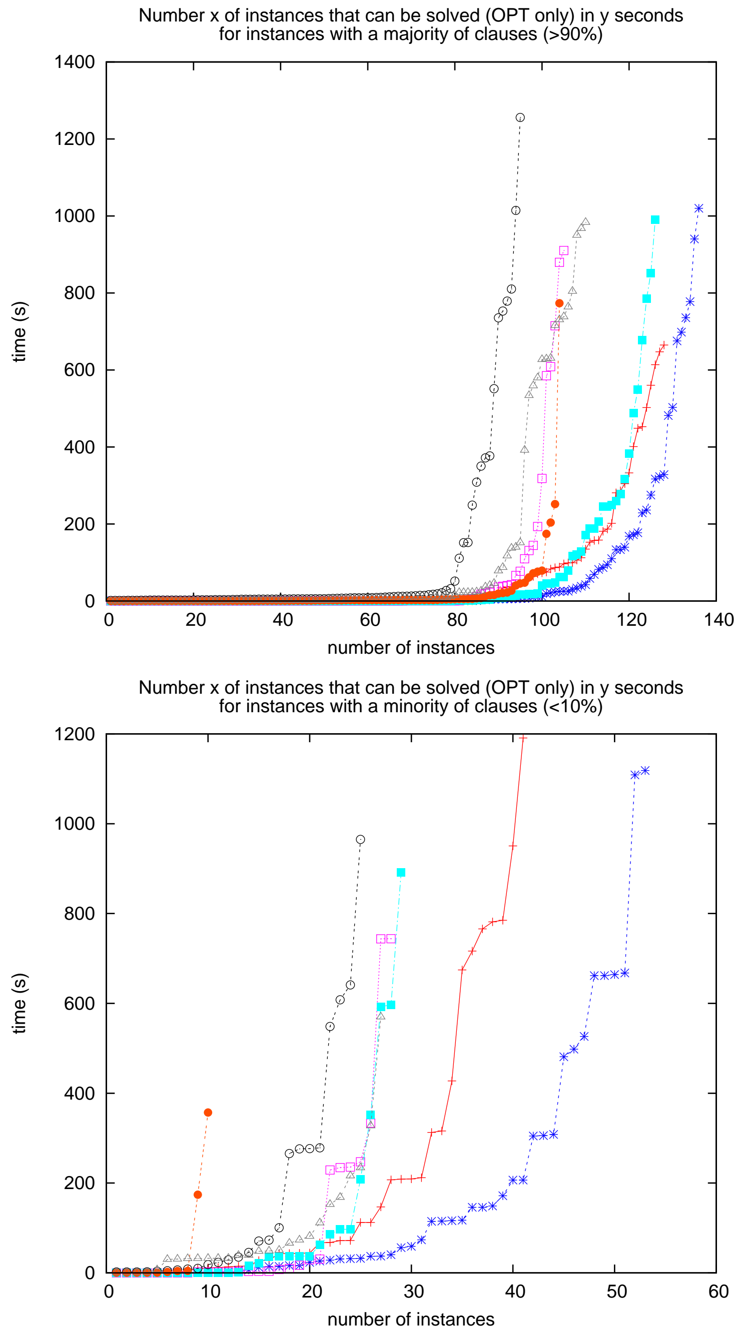

Figure 21. Number x of instances that can be solved (OPT answers only) in y seconds for instances with a majority or minority of clauses

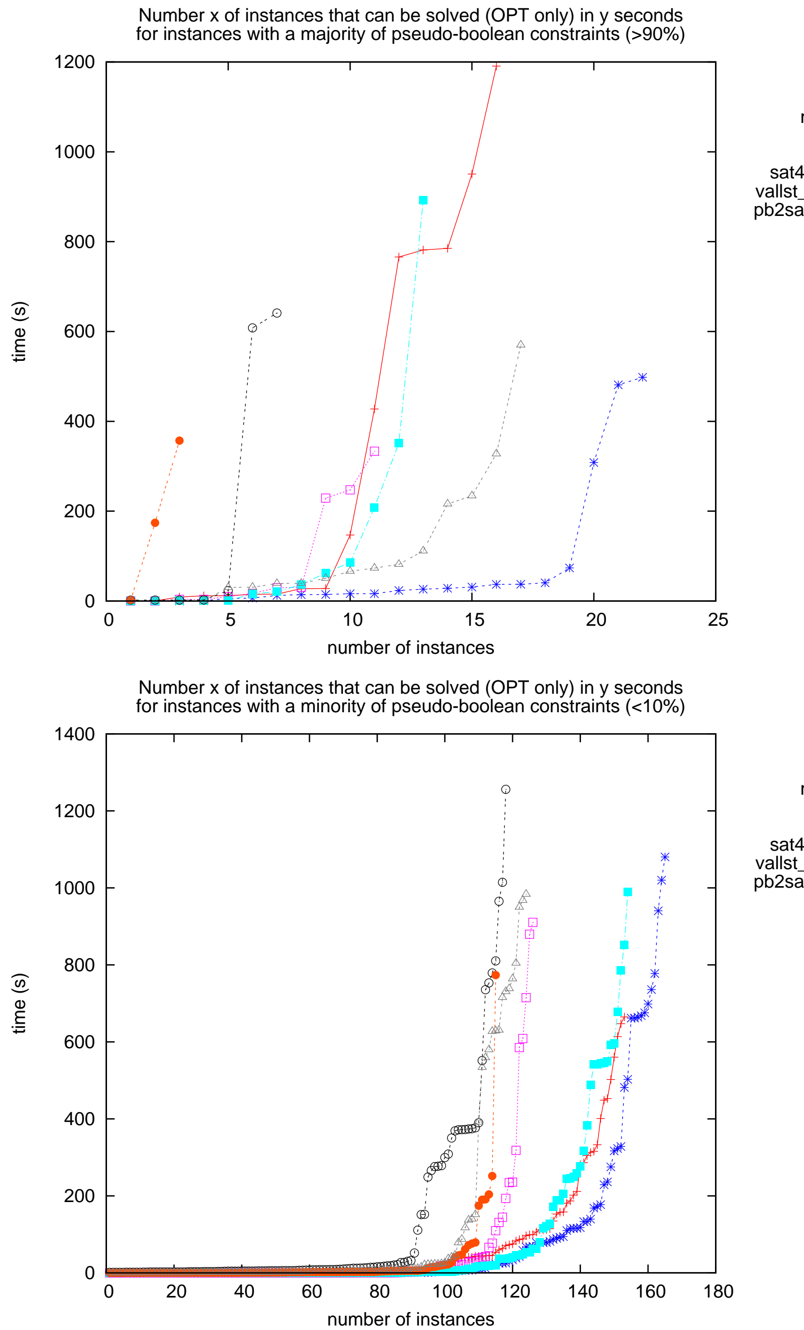

Figure 22. Number x of instances that can be solved (OPT answers only) in y seconds for instances with a majority or minority of pseudo-Boolean constraints

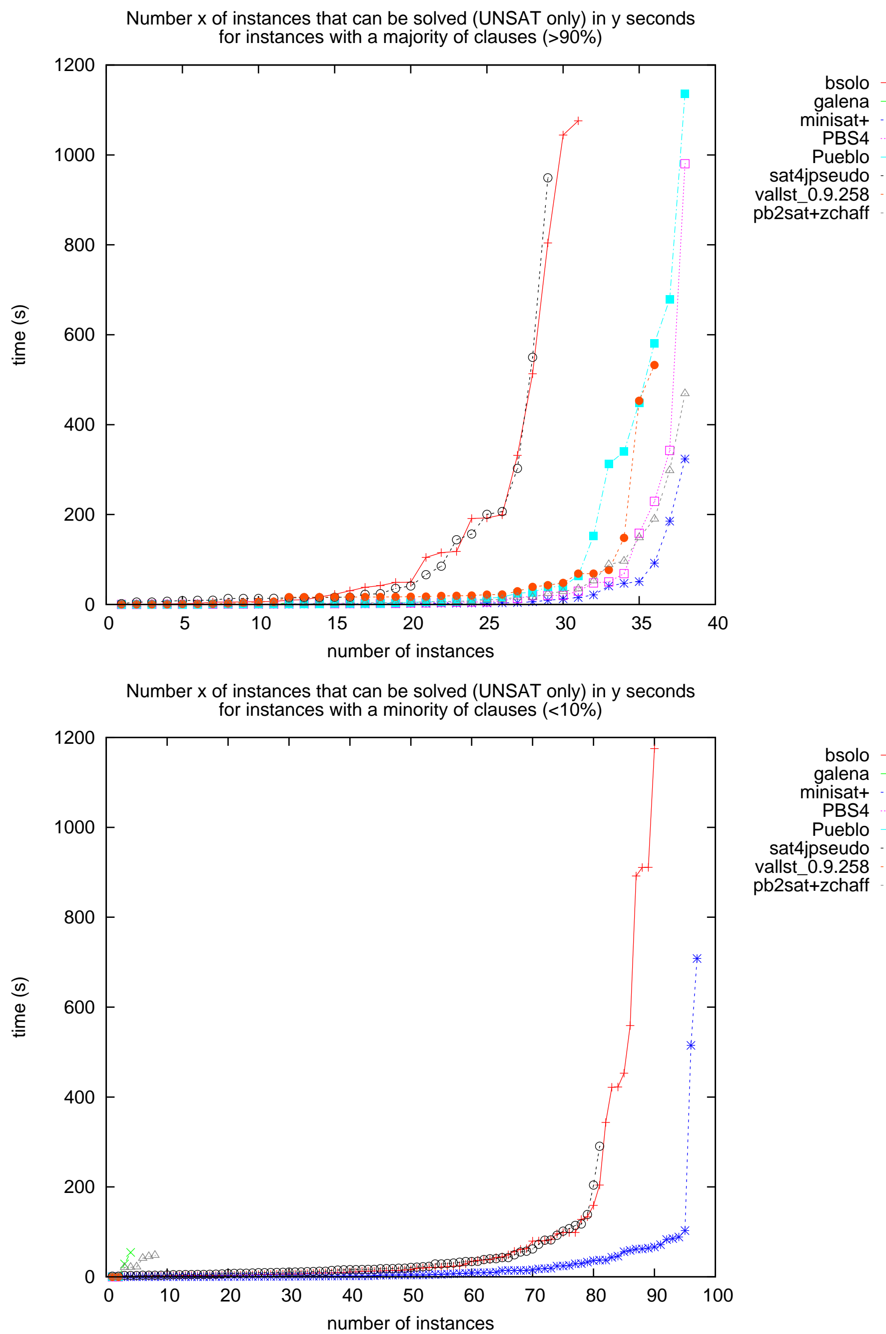

Figure 23. Number x of instances that can be solved (UNSAT answers only) in y seconds for instances with a majority or minority of clauses

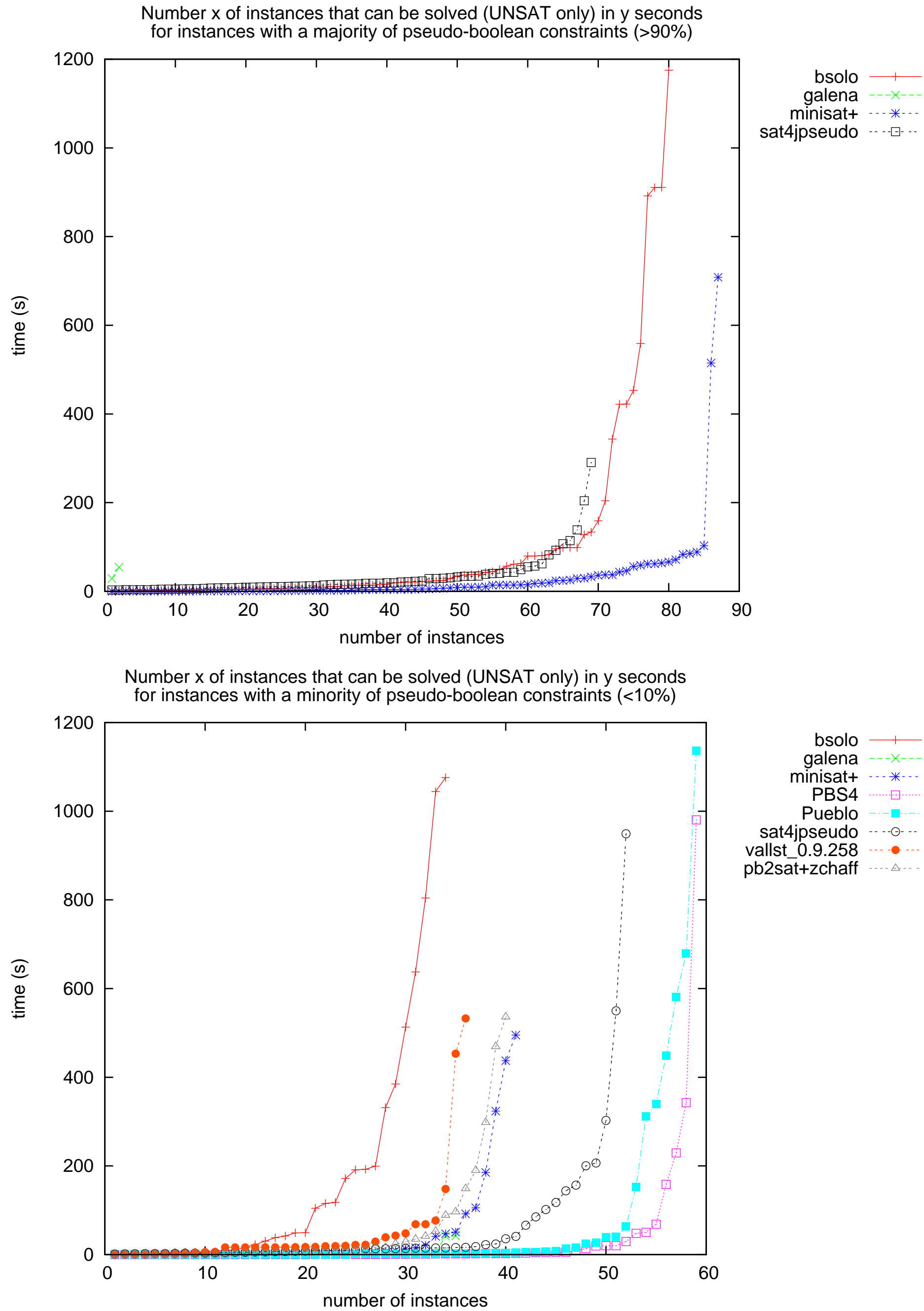

Figure 24. Number x of instances that can be solved (UNSAT answers only) in y seconds for instances with a majority or minority of pseudo-Boolean constraints

Table 17. Solver contributions for the best solutions in the evaluation

Solver	SAT/UNSAT	OPTSMALLINT	OPTMEDINT	OPTBIGINT	Total
bsolo	0	112	32	37	181
galena	0	1	3	0	4
minisat+	0	20	1	12	33
PBS4	0	0	0	0	0
Pueblo	4	29	27	0	60
sat4jpseudo	0	19	41	120	180
vallst_0.9.258	0	0	0	0	0
pb2sat+zchaff	0	0	1	0	1

Note that this does not mean that these solvers are not good. In fact, some solvers with all 0 in Table 17 have very good results in a large number of instances. What it means is that solvers with more contributions have specific features that enable them to be unique for several benchmark instances.

9.2 Bugs in solvers

Fixing bugs is often a long and tedious task and each programmer knows that a bug can remain hidden for a long time. The last version of some solvers were unfortunately still buggy. We report here what we know about these problems.

On one single instance, solver *Pueblo* outputs a solution which did not satisfy all constraints. According to its author, that problem was caused by an objective function that contains terms with both positive and negative coefficients, which was not correctly handled by the solver. The problem has been fixed since then.

Because of a missing flush in its signal handler, *minisat+* never outputs a complete solution on the "v " line and therefore, all its SAT answer are counted as NO CERTIFICATE.

On 3 instances classified as NO CERTIFICATE, solver *sat4jPseudo* required more than two seconds to output a solution when it received a SIGTERM and was therefore killed before it could provide a complete solution.

The wrong UNSAT answers of *vallst_0.9.258* seem to be caused by an error in the script (a 'grepres' command is called but not found).

10. Next Evaluation

In this section we present suggestions for the next evaluation of pseudo-Boolean solvers. We start by proposing an input format simpler to parse and afterwards we discuss several issues to be dealt with in the next evaluation of pseudo-Boolean solvers.

10.1 Input Format

In this evaluation, the input format for the solvers was a restriction of the general OPB format, as described in section 3. However, several problems were detected in some solvers, namely related with the variable identifiers.

One approach is to modify the input format to a CNF-like format commonly used in SAT. In this case, each text-based variable identifier must be mapped into a numerical identifier. Each line in the file would represent a constraint starting with pairs (coefficient,variable) and ending with the constraint sign and the value of the right-hand side. In pratice, we do not think it is a good idea to assume that the sign in all constraints is $\geq$, as in (1), since some solvers might introduce strategies that take advantage of some types of constraints, namely equality constraints. Hence, constraint

$$2 * x + 3 * y + 1 * z \geq 3 \tag{2}$$

could be written as

$$2\ 1\ 3\ 2\ 1\ 3 \geq 3 \tag{3}$$

where variables x, y and z are respectively mapped into numerical identifiers 1, 2 and 3.

The major drawback from this CNF-like format is that some perspective is lost to the human eye when observing the contents of an instance file. We think it is better to keep a simple way to identify what are the coefficients and variables in the constraint. Therefore, we propose to use a fixed letter x as a prefix in the variable identifiers. Hence, the constraint in (2) could be written as

$$2\ x1\ 3\ x2\ 1\ x3 \geq 3 \tag{4}$$

10.2 Evaluation Procedures

In this first evaluation, categories of benchmark instances were defined only after the first phase of submission of solvers. Initially, submitters only had to declare if a solver was able to solve PBS and/or PBO instances. However, we noticed from the first phase results that some solvers were unable to deal with medium or big integers. Hence, categories depending on the type of coefficients were defined.

For the next evaluation, categories must be defined before the submission process and submitters must declare which categories the solver is able to tackle without providing wrong answers. Moreover, if a solver provides a wrong answer, it should be excluded from that category.

Besides having a categorization of benchmarks depending on the type of coefficients, we should also have other categories depending on different features of instances, namely the type of constraints. For example, Binate and Unate Covering formulations (PBO instances with only propositional clauses in constraints) are special cases of PBO with several real world applications, in particular in Circuit Aided Design [11, 16, 17, 19, 26].

The number of non-zero variable coefficients in the cost function of PBO instances is also a feature to consider. Note that if the cost function has a very small number of non-zero variable coefficients, the problem might be easier to solve by solvers with more constraint manipulation techniques. On the other hand, if the number of non-zero variable coefficients in the cost function is large, it might be easier for solvers with better techniques to use the information from the cost function. However, this conjecture is yet to be verified.

Having benchmark categories depending on the domain from which instances were generated is also an option. However, a significant number of instances from a given domain must be gathered in order to categorize instances using this criteria. Additionally, a more representative set of benchmarks must be considered in order to avoid a polarization of the

evaluation process, since in this first evaluation most instances were converted from MPS instances.

Finally, we should note that the use of different categorization criteria may allow us to look at the experimental results from different perspectives, since these category proposals are not mutually exclusive. Overall, analyzing experimental results using different views will hopefully allow us to understand better the relation between the techniques used by the solvers and the benchmark features.

11. Conclusion

The first evaluation of pseudo-Boolean solvers, has allowed the gathering of pseudo-Boolean solvers using very different techniques in a single experimental evaluation. Comparing the pseudo-Boolean solvers results is a challenging task because there are a few more parameters than for a SAT solver and also because half of the submitted solvers were not able to deal with all the benchmarks categories. It would be hazardous to try to summarize the solvers performances in a single ranking.

In spite of the variety of techniques used by the solvers, most optimization instances remain an open challenge for future events. Nevertheless, people are encouraged to submit new benchmarks from different problem domains, in order to diversify the benchmark set.

We believe that this first evaluation has contributed to the evaluation of different techniques and the improvement of existing solvers. Moreover, we also expect that it will contribute to the development of future solvers.

We would like to insist on the fact that the evaluation of pseudo-Boolean solvers has two goals which cannot be dissociated. The first one of course is to identify the most successful techniques. The second one – and probably the most important – is to encourage researchers to submit innovating algorithms. Such an event would be a terrible waste of time if the community only remembered the solver with the best results and only tried to marginally improve its performances. Major improvements necessarily come from new visions of the problem and the pseudo-Boolean evaluation must encourage the development of new techniques.

Acknowledgements

The authors are greatly indebted to Daniel Le Berre and Laurent Simon for providing the scripts of the SAT competition, to the LINC Lab, Department of ECECS, University of Cincinnati for providing the computing resources and to Robert Montjoy for his assistance. Special thanks to Michal Kouril for his great help and his continuous interest in this evaluation.

This work was supported in part by l'IUT de Lens, l'Université d'Artois and by "l'Action Integrée Luso-Française de la Conférence des Présidents d'Université (CPU)".

References

[1] Miplib - mixed integer problem library.
http://miplib.zib.de/.

[2] Netlib repository.
ftp://ftp.netlib.org/index.html.

[3] W. Akkerman. The strace homepage.
http://www.liacs.nl/~wichert/strace.

[4] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *Proceedings of the International Conference on Computer Aided Design*, pages 450–457, November 2002.

[5] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A Backtrack Search Pseudo-Boolean Solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 346–353, Cincinnati, Ohio, 2002.

[6] F. Aloul, A. Ramani, I. Markov, and K. A. Sakallah. Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry. *IEEE Transactions on Computer Aided Design*, **22**(9):1117–1137, 2003.

[7] P. Barth. OPBDP.
http://www.mpi-sb.mpg.de/units/ag2/software/opbdp.

[8] P. Barth. A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science, 1995.

[9] J. Burkardt. Linear programming datasets.
http://www.csit.fsu.edu/~burkardt/datasets/mps/mps.html.

[10] D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. In *Proceedings of the Design Automation Conference*, pages 830–835, 2003.

[11] O. Coudert. Two-Level Logic Minimization, An Overview. *Integration, The VLSI Journal*, vol. **17**(2):677–691, October 1993.

[12] O. Coudert and J. C. Madre. New Ideas for Solving Covering Problems. In *Proceedings of the Design Automation Conference*, pages 641–646, June 1995.

[13] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume **2919** of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[14] R. Gomory. An Algorithm for the Mixed-Integer Problem. Technical Report RM-2597, Rand Corporation, 1960.

[15] H. Sheini and K. Sakallah. Pueblo: A Hybrid Pseudo-Boolean Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006. This issue.

[16] G. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Pub., 1996.

[17] S. Jeong and F. Somenzi. A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations. In *Proceedings of the International Conference on Computer-Aided Design*, 1992.

[18] D. S. Johnson and M. A. Trick. Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1994.

[19] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Pub., 1997.

[20] S. Lahiri and S. Seshia. Uclid generated benchmarks. http://www.cs.cmu.edu/~uclid/.

[21] D. Le Berre. SAT4J: A satisfiability library for Java, 2005. http://www.sat4j.org.

[22] D. Le Berre and L. Simon. SAT Competitions, 2005. http://www.satcompetition.org.

[23] S. Liao and S. Devadas. Solving Covering Problems Using LPR-Based Lower Bounds. In *Proceedings of the Design Automation Conference*, pages 117–120, June 1997.

[24] V. Manquinho and J. P. Marques-Silva. Effective Lower Bounding Techniques for Pseudo-Boolean Optimization. In *Proceedings of the Design and Test in Europe Conference*, pages 660–665, March 2005.

[25] V. Manquinho and O. Roussel. The Pseudo Boolean Evaluation 2005, 2005. http://www.cril.univ-artois.fr/PB05.

[26] D. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[27] H. Mittelmann. Benchmark of free LP solvers. http://plato.asu.edu/ftp/lpfree.html.

[28] T. Morlier. S4G: a Sandbox for Grids. http://s4g.gforge.inria.fr.

[29] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver . In *39th Design Automation Conference*, pages 530–535, June 2001.

[30] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006. This issue.

[31] O. Bailleux, Y. Boufkhad and O. Roussel. A Translation of Pseudo Boolean Constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006. This issue.

[32] C. Pizzuti. Computing Prime Implicants by Integer Programming. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, pages 332–336, November 1996.

[33] H. Sheini and K. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT Solver. In *Proceedings of the Design and Test in Europe Conference*, pages 684–685, March 2005.

[34] L. Simon, D. Le Berre, and E. Hirsch. The SAT 2002 Competition. *Annals of Mathematics and Artificial Intelligence*, **43**, Issue 1 - 4:307–342, 2005.

[35] M. Trick. Traveling tournament problem. http://mat.tepper.cmu.edu/TOURN/.

[36] V. Manquinho and J. Marques-Silva. On Using Cutting Planes in Pseudo-Boolean Optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006. This issue.

[37] D. Vallstrom. Vallst, 2005. http://vallst.satcompetition.org/papers/SAT2005/vallst.pdf.

[38] D. Vallstrom. Vallst, 2005. http://vallst.satcompetition.org.

[39] J. Walser. 0-1 Integer Programming Benchmarks. http://www.ps.uni-sb.de/~walser/benchmarks/benchmarks.html.

[40] K. Xu. Pseudo-Boolean (0-1 Integer Programming) Benchmarks with Hidden Optimum Solutions. http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/pb-benchmarks.htm.

[41] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide. Microelectronics Center of North Carolina, January 1991.

[42] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, November 2001.

[43] Z. Zhu. Synthesis for Mixed PTL/CMOS Circuit. http://www-unix.ecs.umass.edu/~zzhu/.