CSP2SAT4J: A Simple CSP to SAT translator

Daniel Le Berre and Inês Lynce

[1] CRIL-CNRS FRE 2499
Rue Jean Souvraz SP 18
62307 Lens Cedex FRANCE*
[2] IST/INESC-ID,
Technical University of Lisbon, Portugal

Abstract. SAT solvers can now handle very large SAT instances. As a consequence, many translations into SAT have been shown successful in recent years: Planning and Bounded Model Checking are two examples of applications in which SAT engines are reported to be as good as or even better than dedicated software. During the first CSP competition, SAT-based approaches were demonstrated competitive with the other CSP solvers on binary constraints. Constraints being provided in extension, it was a big advantage for techniques based on grounding predicates since only benchmarks that could be grounded in a reasonable space were available. As such, the comparison between SAT-based solvers (that need to ground predicates) and the approaches developed by the CSP community (that usually handle directly the constraints as expressed) was not fair. For the second edition of the competition, the constraints can now be given in intension, and global constraints such as *allDifferent* are available. The idea behind the submission of CSP2SAT4J is to show when SAT-based CSP solvers can still compete in some cases against "traditional" CSP solvers under those new conditions.

1 Introduction

The idea behind the submission of CSP2SAT4J is to show when SAT-based CSP solvers can still compete against "traditional" CSP solvers under those new conditions: on non-randomly generated binary constraints benchmarks, when the domain of the variables is not too big (a few hundred variables max) to allow grounding the predicates in reasonable time. The translation from CSP to SAT has been improved since last year submission: the solver can outperform another CSP solver (namely Abscon) on last year competition's benchmarks. The underlying SAT solver can handle cardinality constraints, which minimizes the number of constraints used in the translation. The evaluation of the constraints given in intention is done using a JavaScript to Java bytecode compiler, in order to keep the "Keep It Simple Stupid" approach of last year submission. The experimental results on the new benchmarks available in July 2006 do show the limit

* The first author has been supported in part by the IUT de Lens, the CNRS and the Region Nord/Pas-de-Calais under the TAC Programme and the COCOA project. The two authors were partly founded by the PAI-PESSOA project MUSICA.

of the approach when the size of the domains increases: for some categories of benchmarks in which the solver performed well last year (Queens and Knight for instance), the solver is unable to ground the bigger instances available in intention this year. Since the participation to the competition also implies submitting benchmarks, we included in the last section the description of our encoding of the *Social Golfer Problem*.

2 On the power of SAT solvers

Very recently, the translation from pseudo boolean into SAT was shown competitive and sometimes better than dedicated solvers during the PB05 evaluation [20]. Indeed, the MiniSAT+ solver shown surprising good performances on most of the benchmarks, and it can be considered as the "winner" of the evaluation. The MiniSAT solver, together with the SatELite preprocessor were the winners of the SAT competitions in the industrial and crafted categories. The solver MiniSAT+ has shown that using SAT engines as efficient generic problem-solving engines was a reality. For that reason, we decided to submit a SAT-based CSP solver to the first (and second) CSP competition.

The SATisfiability problem (SAT) gained interest from the industry a few years ago when SAT solvers where used to solve Bounded Model Checking problems [6] instead of BDDs. That interest pushed people to design solvers for those particular types of problems. One of the particularities of those benchmarks is their size: they are"huge" compared to the classical pathological pigeon hole or random k-SAT problems. As a consequence, the complexity of the algorithms and data structures becomes even more important. That observation was the origin of the design of the head-tail lazy data structure in SATO[26, 27], the Watched Literals and cheap VSIDS heuristic in Chaff [17, 28]. Grounding CSP problems into SAT does generate huge CNF, so it makes sense to use "industrial" SAT solvers for solving those formulas.

Another particularity of SAT instances coming from BMC, or more generally practical problems translated into SAT compared to the seminal 3-CNF instances, is the lack of real characterization of those SAT instances: while the theory around 3-CNF allows to build powerful heuristics-based SAT solvers for 3-SAT (TABLEAU, CSAT, POSIT, SATZ, CNFS, etc), non-chronological backtracking and learning looks like the best approach to tackle SAT-encoded problems (SATO,RELSAT,GRASP,CHAFF). Note that usually non-chronological backtracking and learning is useless if the heuristic is good enough, which explains why solvers using those techniques are outperformed on random 3-SAT instances by heuristics based solvers. Furthermore, it seems that the lack of "structure" in the problem makes the VSIDS heuristic ineffective. On the other hand, non-chronological backtracking can repair mistakes made by the heuristics by analyzing conflicts.

One of the consequence of using an "industrial" SAT solver to power a SAT-based CSP solver in the CSP competition is to have poor performances on randomly generated CSP benchmarks. While a randomly generated CSP benchmark

encoded into a SAT benchmark cannot be considered as a randomly generated SAT benchmark, we conjecture that the bad behavior of our approach on randomly generated CSP benchmarks is due to the lack of "structure" needed by conflict driven clause learning SAT solvers.

All the parts of a SAT solver received a huge interest from both an algorithmic and an implementation point of view so current SAT solvers are now heavily tuned and they should not be considered as prototype software but rather as production software. We use the SAT4J library[3], an open-source library of conflict driven clause learning (aka "industrial") SAT solvers in Java. The library is mature and competitive with state-of-the-art SAT solvers: it participated to the 2004 and 2005 SAT competitions in which it went in the second stage in the industrial category, and passed the qualification step of the SAT Race 2006, a competition of SAT solvers especially dedicated to industrial SAT benchmarks.

In the rest of the paper, we first describe how our SAT library follows the current trend to generalize SAT solvers to handle constraints more general than clauses. Then we explain the CSP to SAT encodings used in our solver and provide some experimental results.

3 From clauses to pseudo boolean constraints

Researchers are pushing the limit beyond SAT: Quantified Boolean Formulas (QBF) and Stochastic SATisfiability (SSAT) for instance are two extensions of SAT being studied recently. Another extension of SAT received some attention a decade ago: using pseudo boolean constraints (linear constraints with boolean variables) instead of plain clauses [3, 4]. Most of the solvers for those extensions to SAT are developed using techniques that were demonstrated powerful for SAT. Those solvers in the early 90s were based on DPLL[9, 8] while the solvers developed today are often related to Chaff-like solvers.

This is especially true for pseudo boolean solvers: Barth first developed a DPLL version of a pseudo boolean solver [4]. Walser [24] and later Prestwich [18, 19] developed local search or hybrid pseudo boolean solvers. Aloul et al [22] developed a version of Chaff handling pseudo boolean constraints instead of clauses as input, plus symmetry breaking predicates, with clause learning (same thing for the recent MiniSAT [13]). Dixon and Ginsberg[10] developed a pseudo boolean version of Relsat (PRS), which was the first pseudo boolean solver including true pseudo boolean learning. They developed a pseudo boolean version of Chaff (PBChaff[11]) in the same spirit while Chai and Kuehlmann [7] did extended all Chaff techniques (learning scheme and data structures) in the pseudo boolean solver Galena. Recent work from Dixon et al [12] describes a generic conflict driven constraint learning solver based on group theory while Thiffault et al [23] describe a conflict driven clause learning solver working with arbitrary boolean gates.

SAT4J uses some principles taken from both Chai and Kuehlmann and Dixon to allow some of its solvers to use cutting planes instead of resolution when using

[3] http://www.sat4j.org/

linear pseudo boolean constraints. As a result, those solvers can solve in a linear number of conflicts the pigeon hole problem when it is expressed by linear pseudo boolean constraints, which is not possible with a solver based only on resolution. But that power comes with a high price to pay in practice: on benchmarks with few pseudo boolean constraints, such approach is not as efficient as a solver applying resolution on pseudo boolean constraints.

As a consequence, many SAT solvers are currently using general constraints to express the problem in a more compact way than with pure clauses (e.g. for cardinality constraints), without using the full power of those constraints, but without additional running time either. This is how we setup our own SAT solver for the CSP competition.

4 From CSP to SAT

Our CSP to SAT translator uses two different encodings: direct encodings [25] and support encoding for binary clauses [14]. Note that we use a single cardinality constraint instead of using binary (so-called "at most") clauses to express that no more than one value can be chosen in a domain.

The encoding used depends on the way the constraints are expressed:

extension (conflict) In that case, it is straightforward to use a direct encoding since each tuple is translated into a clause.

extension (support) If the constraint is binary, then we use the binary support encoding, else the direct encoding, by generating all conflicting tuples.

intension The constraint is grounded by generating all the possible input values and checking if it satisfies or not the constraint. If the constraint is binary, then the binary support encoding is used, else the direct encoding is used. A better option might be to approximate the number of allowed or forbidden tuples and to select the encoding accordingly. A more sophisticated and efficient way to generate the tuples to be considered is also a possible way of improvement.

The main drawback of our method is the way we handle n-ary constraints: for a constraint of arity 8 with domains of size 10, 10^8 tuples need to be generated. It is currently impossible to simply enumerate all those tuples in a reasonable time.

We also implemented the generalized support encoding [5] for n-ary constraints. However, the cost for generating that encoding is much higher than the direct encoding. We are aware of another CSP to SAT encoding that we have not experimented because it relies on a very specific way to describe the problem in terms of disjunction of forbidden values [19].

4.1 Common encoding

The translator takes the new XML representation (XML CSP 2.0 format) of the problem as input and outputs a set of constraints (mixing clauses and cardinality constraints) to feed our extended SAT solver.

Variables For each variable $v_i \in V$, and each domain value $d_j \in Domain(v_i)$, a propositional variable $p_{i,j}$ is created.

Domains For each variable $v_i \in V$, a cardinality constraint denotes that only one value from the domain can be chosen: $\sum_x p_{i,x} = 1$. In practice, that constraint is expressed in the solver using the clause $\vee_x p_{i,x}$ and the cardinality constraint $\sum_x p_{i,x} \leq 1$

4.2 Direct encoding [25]

Forbidden tuples (nogoods) Each tuple representing a forbidden combination of values is represented by a propositional clause composed by the negation of the propositional variables representing those values. So the length of the generated clause is the arity of the constraint. Note that in case of binary constraints, binary clauses will be generated.

Allowed tuples (supports) When a relation is represented by allowed tuples, we deduce all the forbidden tuples and translate them into clauses as described above.

The main drawback of that translation is the translation of the allowed tuples. It can take a lot of time to generate them when the arity of the constraint increases.

4.3 Support encoding for binary constraints [14]

Forbidden tuples (nogoods) Each tuple representing a forbidden combination of values is represented by a propositional clause composed by the negation of the propositional variables representing those values.

Allowed tuples (supports) For binary constraints, we create a clause $\neg a \vee b_1 \vee \ldots \vee b_k$ for each variable a that appears in tuples $(a, b_1), (a, b_2), \ldots (a, b_k)$

4.4 From intension to extension

Our solver grounds predicates in intension into tuples, in order to apply the above translation. Compared to the first competition, the cost of grounding the predicate is added to the CSP solver, and it might not be possible to ground some of the problems in reasonable time or space. The biggest issue for dealing with constraints in intention in our SAT-based approach is to evaluate the expressions. Since our aim is simply to evaluate them for a complete assignment of the variables, and since we want to keep as low as possible the portion of our code dedicated to CSP solving, we decided to have both a pragmatic and extensible approach to do it. Indeed, it can be expected that the next versions of the input format will see more and more built-in functions. Furthermore, the new version of the Java virtual machine will ship with a Java Script interpreter called Rhino[4]. So we decided to interpret the predicates defined in the input file as a javascript expression. This can be easily achieved by defining in

[4] http://www.mozilla.org/rhino/

JavaScript the built-in functions and load them before evaluating the expression. That framework also has the good property to allow to compile directly the JavaScript expression into Java bytecode, so the cost of evaluating the expression is reduced. The main drawback of that approach in our opinion is that the Rhino framework is 700Kb big while the SAT4J library is only 400Kb big. Adding a dependency to such a package makes our CSP solver temporarily more than 1MB big on current JVM).

4.5 The allDifferent global constraint

One of the new features of the second version of the CSP input format is the ability to express global constraints. For the second competition, only the *allDifferent* constraint is available. That constraint has some nice properties and is very useful to eliminate values in domains. A translation into SAT of the *allDifferent* constraint preserving some of those properties was proposed in [16]. However, we decided to use a simpler approach: for each *allDifferent* constraint we simply add the binary clauses ensuring that no couple of variables share a common value: it is a sort of local direct encoding of the constraint, since the forbidden tuples of the global constraint can be easily expressed by binary constraints ($allDiff(x_1, x_2, \ldots, x_n) \equiv \wedge_{i<j} x_i \neq x_j$). In some sense, we are not taking advantage that way of the constraint being global.

Note that a specific data structure proposed by Lawrence Ryan [21] is used in our SAT solver to handle binary clauses because the implementation of the *allDifferent* constraint is likely to produce many of them.

5 A few experimental results

We present here some experimental results comparing our SAT-based CSP solver against Abscon, one of the strongest CSP solvers that participated in the first CSP competition. Note that Abscon and our own solver are to the best of our knowledge the only CSP solvers freely available for research purpose that are compatible with the first and second CSP competition input format. Note also that the two solvers are written in Java.

All the results were obtained on a cluster of Bi-Xeon 2.6 GHz with 2GB of memory (1GB per processor) running Linux, using Java 1.5.0_06 for 32 bits architecture. The timeout was 20 mn per benchmark.

5.1 First CSP competition benchmarks (extension)

These results were obtained on January 2006 on the set of benchmarks used for the first CSP competition, plus some additional random benchmarks. The version of SAT4J used was 1.5_01. We used a developer version of Abscon. The benchmarks were given in extension. The first part of the table (*All* column) summarizes the results of the two solvers on all the benchmark (number of problems solved) classified into binary and n-ary ones. Abscon is far better than SAT4J

overall, and especially on n-ary satisfiable benchmarks. The second column restricts the results to the benchmarks that were not randomly generated. In that case, SAT4J is slightly better than Abscon on binary benchmarks.

	All		Non-random	
	SAT4J	Abscon	SAT4J	Abscon
non binary constraints				
	(186 benchmarks)		(150 benchmarks)	
UNSAT	27	**28**	27	**28**
SAT	61	**125**	48	**108**
binary constraints				
	(2031 benchmarks)		(1041 benchmarks)	
UNSAT	842	**995**	**400**	396
SAT	760	**827**	**560**	536

These results simply show that provided that grounding the problem is possible, a SAT-based approach is competitive with Abscon for binary benchmarks non-randomly generated.

5.2 Benchmarks in XML 2.0 format

These results were obtained in July 2006. The version of SAT4J used was a CVS snapshot tagged OBJECTWEB 1.0.90 (the one submitted to the CSP competition) and the version of Abscon was 105.

	SAT4J	Abscon
non binary constraints (978 benchmarks)		
UNSAT	69	**78**
SAT	273	**453**
binary constraints (2673 benchmarks)		
UNSAT	613	**1053**
SAT	861	**1285**

Unfortunately, we do not have the details of random/non-random benchmarks. However, a few remarks can help reading these results:

Queens/Knights During the first CSP competition, the direct encoding gave poor results on those benchmarks (none of them solved). Using the support encoding allowed SAT4J to solve all them quickly (in less than 2 minutes overall). The benchmarks proposed this year are much bigger: queensKnights-50 has for instance a domain size of 2500. As a consequence, enumerating 2500^2 tuples just makes the SAT-based approach hopeless on those bigger benchmarks.

Fapp There are 40 series of 11 benchmarks for the FAPP benchmarks (binary benchmarks). For the first CSP competition, only the first two series (the smaller ones) were submitted because the other ones were too big to be expressed in extension. Our approach is only able to solve the benchmarks of the first series and a few from the second series, and runs out of memory on the other ones. On the other hand, Abscon is able to solve almost all of them. Those particular series of benchmarks represents 1/6 of the total number of benchmarks, while there are more than 25 different sets of benchmarks: the difference in number of problems solved in the table should be considered at the light of that fact. We expect to have closer results between SAT4J and Abscon during the second CSP competition because the number of problems will be close for each kind of benchmarks.

Out of Memory happened in 633 cases on binary benchmarks, and 204 cases on n-ary benchmarks, i.e. respectively in 24% and 21% of the total number of benchmarks! It happened starting at domino-2000, fapp-02, knights-50, queens-knights-50 and js-taillard-15 for binary benchmarks. For n-ary benchmarks, it happened mostly on pseudo boolean benchmarks translated into CSP and on traveling salesman problems, golomb ruler, all interval series and mknap. It happened even on some problems given in extension in n-ary benchmarks, because we need to generate forbidden tuples when supports tuples are given.

6 The social golfer problem

The social golfer problem is derived from a question posted to `sci.op-research` in May 1998:

The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion.

In other words, this problem can be described more explicitly by enumerating four constraints which must be satisfied:

- The golf club has 32 members.
- Each member plays golf once a week.
- Golfers always play in groups of 4.
- No golfer plays in the same group as any other golfer twice.

Since 1998, this problem has become a famous combinatorial problem. It is problem number 10 in CSPLib (http://www.csplib.org/). A solution is said to be optimal when maximum socialisation is achieved, i.e. when one golfer plays with as many other golfers as possible. Clearly, since a golfer plays with three new golfers each week, the schedule cannot exceed 10 weeks. This follows from the fact that each golfer plays with three other golfers each week. Since there is a total of 31 other golfers, this means that a golfer runs out of opponents

after 31/3 weeks. For some years, it was not known if a 10 week (and therefore optimal) solution for 32 golfers exists. In 2004, Aguado found a solution using design-theoretic techniques [1].

Even though the social golfer problem was described for 32 golfers playing in groups of 4, it can be easily generalized. An instance to the problem is characterized by a triple w-p-g, where w is the number of weeks, p is the number of players per group and g is the number of groups. The original question therefore is to find a solution to the w-4-8 problem, with w being the maximum, i.e. to find a solution to 10-4-8 (or prove that none exists).

The social golfer problem is related with other well-known combinatorial problems. Indeed, this problem is a generalisation of the problem of constructing a round-robin tournament schedule, the main difference being that in the social golfer problem the number of players in a group may be greater than two. Also, the social golfer problem of finding a 7 week schedule for 5 groups of 3 players (7-3-5) is the same as Kirkman's Schoolgirl Problem, where the main goal is to arrange fifteen schoolgirls in rows of three so that each schoolgirl walks in the same row with every other schoolgirl exactly once a week.

The encoding used is the one proposed by Walser available on CSPLIB that can be summarized as follows:

- 0-1 variables $Golfer_{i,j,k} = 1$ indicate that golfer i plays in group j in week k.
- 0-1 variables $Meet_{i,j,k} = 1$ indicate that golfers i and j meet in week k (thus are in the same group).
- constraints relating the above variables: $Golfer_{i,k,l} + Golfer_{j,k,l} - Meet_{i,j,l} \leq 1$.
- golfers play in exactly one group per week: $\sum_j Golfer_{i,j,k} = 1$.
- each pair of golfer plays only once: $\sum_k Meet_{i,j,k} = 1$.
- each group has exactly p golfers: $\sum_i Golfer_{i,j,k} = p$.

Note that while the domain of the variables is small (boolean), some constraints have a big arity (the number of golfers $p * g$) which makes the SAT-based approach inefficient (enumerating 2^{32} tuples for a 8-4-8 problem for instance is out of reach for our solver).

The problems 8-4-8, 9-4-8 and 10-4-8 that we have submitted are quite challenging. It would be a good news if some competitors were able to solve them. Some recent work on a SAT encoding with symmetry breaking predicates can be found in [15]. [2] has proposed to dynamically break symmetries in the social golfers problem. This new approach is often able to outperform the traditional approaches, although at the cost of eliminating some solutions. Hence, the proposed method is incomplete.

7 Conclusion

We presented our new SAT-based CSP solver as submitted to the second CSP competition. We presented some experimental results showing that a SAT-based

approach for CSP is quite competitive provided that the problems are not randomly generated and contain binary constraints with "reasonable size" domain to make the grounding of the predicates possible. We also presented our encoding of the *Social Golfer Problem* for which we provided a generator and 10 samples benchmarks for the competition. We believe that one solution to cope with the predicates given in intension that cannot be grounded in reasonable time or space is to manage them as a new kind of constraint in the SAT solver. Two manipulations are needed in the case of a conflict driven constraint learning solver:

value propagation the constraint should be able to cope with partial assignment (of boolean variables provided by the SAT solver) and to detect which values in the domains need to be assigned/forbiden as a result of a domain assignment (thus leading to unit propagation in the SAT solver).

reason computation Conflict analysis is an important part of the SAT solver. It relies on computing for each propagated assignment a reason for that assignment in the form of a set of literals (the set of falsified literals in a clause). The biggest issue in our opinion will be to make sure that such a reason can be computed in a predicate given in intention and to see how a possible solution relates with CSP backjumping and nogood learning.

We hope to be able to compare our solver against numerous other CSP solvers in the future: it would be nice if the solvers that participate in the CSP competition could be freely available for research purpose after the competition.

References

1. Alexandro Aguado. A 10 days solution o the social golfer problem., 2004. Manuscript.
2. Francisco Azevedo. An Attempt to Dynamically Break Symmetries in the Social Golfers Problem. In Francisco Azevedo, editor, *11th Annual ERCIM Workshop on Constraint Programming (CSCLP'2006)*, pages 101–115, 2006.
3. Peter Barth. Linear 0-1 inequalities and extended clauses. Technical Report MPI-I-94-216, Saarbrcken, Germany, 1994.
4. Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Saarbrcken, 1995.
5. Christian Bessière, Emmanuel Hebrard, and Toby Walsh. Local consistencies in sat. In *Proceedings of the Sixth International Conference on Theory and Application of Satisfiability Testing (SAT'2003)*, volume 2919 of *LNCS*, pages 299–314. Springer, May 2003.
6. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of bdds. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.
7. Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *ACM/IEEE Design Automation Conference (DAC'03)*, pages 830–835, Anaheim, CA, 2003.
8. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. In *Communications of the Association for Computing Machinery 5*, pages 394–397, 1962.

9. M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM, 7*, pages 201–215, 1960.
10. Heidi E. Dixon and Matthew L. Ginsberg. Combining satisfiability techniques from AI and OR. In *The Knowledge Engineering Review 15*, page 53, 2000.
11. Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 635–640, 2002.
12. Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing boolean satisfiability I: Background and survey of existing work. In *Journal of Artificial Intelligence Research 21*, 2004.
13. Niklas Een and Niklas Sorensson. An extensible sat solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing*, LNCS 2919, pages 502–518, 2003.
14. Ian Gent. Arc consistency in sat. In *Proceedings of ECAI'2002*, pages 121–125, 2002.
15. Ian P. Gent and Ines Lynce. A sat encoding for the social golfer problem. In *Proceedings of the IJCAI'05 workshop on Modeling and Solving Problems with Constraints*, July 2005.
16. Ian P. Gent and Peter Nightingale. A new encoding of alldifferent into SAT. In *3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (CP2004)*, pages 95–110, 2004.
17. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.
18. S. Prestwich. Randomised backtracking for linear pseudo-boolean constraint problems. In *Proceedings of Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'2002)*, pages 7–20, 2002.
19. S. Prestwich. Incomplete dynamic backtracking for linear pseudo-boolean problems: Hybrid optimization techniques. *Annals of Operations Research*, 130(1-4):57–73, August 2004.
20. Olivier Roussel and Vasco Manquinho. The first pseudo boolean solver evaluation. http://www.cril.univ-artois.fr/PB05/.
21. Lawrence Ryan. Efficient algorithms for clause learning sat solvers. Master's thesis, SFU, February 2004. Available at http://www.cs.sfu.ca/∼mitchell/papers/ryan-thesis.ps.
22. Fadi A. Aloul Arathi Ramani Igor L. Markov Karem A. Sakallah. Symmetry-breaking for pseudo-boolean formulas. In *International Workshop on Symmetry on Constraint Satisfaction Problems (SymCon)*, pages 1–12, County Cork, Ireland, 2003.
23. Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving Non Clausal Formulas with DPLL Search. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'2004)*, Toronto, Canada, September 2004.
24. J. P. Walser. Solving Linear Pseudo-Boolean Constraint Problems with Local Search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 269–274, 1997.
25. Toby Walsh. Sat vs csp. In Springer, editor, *Proceedings of CP'2000*, pages 441–456, 2000.

26. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.
27. Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.
28. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, November 2001.