

Expressiveness and Utility of Generic Representation Schemes

David W. Etherington
CIRL
1269 University of Oregon
Eugene OR 97404
USA

<http://www.cirl.uoregon.edu/ether/>

Andrew J. Parkes
CIRL
1269 University of Oregon
Eugene OR 97404
USA

<http://www.cirl.uoregon.edu/parkes/>

ABSTRACT

We describe how problems can be solved by “compiling” them to a general-purpose representation scheme such as SAT or Pseudo-Boolean (PB). We discuss why PB is a so much more expressive and better language than SAT for problems that involve any arithmetic reasoning. We describe PARIS, a program that solves decision problems expressed in PB, that is based on an adaptation of the best SAT solvers. We also describe OPARIS, an extension of PARIS to solve optimization as well as decision problems. For OPARIS we give a high-level description of the algorithms and heuristics used, and some experimental results on real-world problem instances obtained from ISI.

1. INTRODUCTION

A standard approach to solving decision/optimization problems is to express them in a general purpose language for which specialized solvers are available. In operation research (OR), it is standard to use integer programming (IP) formulations: variables are integer-valued, and constraints are simple linear inequalities over these constraints. In artificial intelligence (AI), it has been more common to use the logical formalism of propositional satisfiability (SAT), in which variables are Boolean (true or false) and constraints are clauses—disjunction of literals, where a literal is a variable or its negation: for example, $\neg x \vee y \vee \neg z$.

Both formulations, IP and SAT, are NP-complete and so are suitable for encoding problems in NP. Furthermore there are many well-developed solvers for both. Generally speaking IP seems to be best for problems with a significant arithmetic component—meaning that someone solving them might do a lot of counting and arithmetic reasoning. SAT seems better for cases where logical reasoning is more natural “if x then either y or z is true, and hence ...”. However, there are many problems that seem to be a mix of both arithmetic and logical reasoning. Such problems present two difficulties:

- **IP solvers are bad at logic.** SAT solvers are good at logical problems because they do a lot of logical inference internally, and find logical consequences of the constraints that can be used to prune out unnecessary search. Some OR solvers have some aspects of inference (those doing branch-and-cut) but generally inference is not emphasized.
- **SAT solvers are bad at arithmetic.** In fact, it is inherent in the nature of the representation that SAT solvers are spec-

tacularly bad at arithmetic. The best example is that of the propositional pigeonhole principle (PHP); the PHP simply says that it is not possible to put $n + 1$ pigeons into n holes without any pigeons sharing a hole. This is obvious by simple arithmetic, but, no matter how clever their heuristics, SAT solvers provably take exponential time to prove the PHP unsolvable.

Our goal has been to develop a solver that is good at both arithmetic and logic and so is suitable for some “intermediate problems”. For example, some arithmetic/logic problems of interest to ANTS arise in SNAP [?]. The arithmetic aspects arise from the resource bounds—counting naturally occurs to see whether the tasks can fit within the resource limits. The logical aspects arise from dependencies between tasks—for example, pilots need to perform some missions in order to obtain qualifications permitting them to do other missions.

To enable these investigations, we selected the representational language of “pseudo-Boolean” (PB) constraints, which is a simple intermediate between IP and SAT. PB has:

- Boolean variables—this enables the logical reasoning
- linear inequalities as constraints—this enables the arithmetic reasoning

Hence, for example, typical PB constraints include “cardinality constraints”:

$$\sum_{i=1}^N l_i \geq K \tag{1}$$

where the literals l_i are all 0 or 1.¹

Note that such a constraint can be converted to a set of SAT clauses, but we can end up with an exponential number of clauses. Hence, PB is exponentially more concise than SAT. This is a significant advantage in itself. (Actually, if we allow the introduction of new variables, then we can convert it to a polynomial number of clauses, but in practice the extra variables confuse solvers, leading to poor performance.) We also allow “full-PB” constraints,

$$\sum_{i=1}^N w_i l_i \geq K \tag{2}$$

¹A literal is either a variable x or its complement \bar{x} , defined to be $1 - x$.

in which we are allowed a weight w_i for each literal in the sum. This is particularly useful when encoding optimization problems—e.g., the weights can correspond to costs of tasks associated with the literals.

In the OR world, this representation is already well-known as “0/1 programming.” We use the term pseudo-Boolean to emphasize that our approach to solving problems is based on Boolean solvers (SAT solvers) rather than standard OR methods.

OR solvers are usually based on solving “linear relaxations” of problems; that is, the requirement that variables be integer-valued is dropped, giving a linear programming (LP) problem, for which polytime solvers are available. However, such solvers are still fairly slow, and so the overall emphasis is that of a search with substantial reasoning at each node, and so the expectation that the nodes studied per second will be small. In comparison the methods underlying SAT solvers (to be discussed later) tend to involve very lightweight reasoning; the reasoning captures less information, but this is potentially compensated by a much higher search speed—more nodes per second. Also, since the OR methods are so reliant on the LP relaxation it is essential that the encoding be “LP-tight”, that is, the LP relaxation should be fairly informative. Unfortunately, for SAT problems, the LP relaxation is useless: this is reflected in the fact that SAT solvers are needed at all—SAT is a subset of IP, and if OR solvers performed well on this subset then specialized SAT solvers would be redundant. Hence, for problems with a large number of logical constraints, we should expect that LP-methods will be ineffective, and that logical methods will be needed. Even for standard OR problems it can take a significant effort by a domain/OR expert, to find an encoding that is LP-tight, making automatic production of useful encodings much harder. One of our hopes is to reduce the reliance on LP-tight encodings by using logic-based solvers, and so make it easier to produce automatic encodings.

In relatively unstructured problems (as from a heterogeneous environment, with few clean rules), it may well be easier to live with a PB encoding rather than expend significant manual effort looking for a clever (but fragile) IP encoding.

Hence, our overall aim has been to take AI search methods developed for SAT and adapt them to the better PB representation.

The resulting solver for decision problems is called PARIS (for some relevant background see [5]). We also give a description of OPARIS, a version of PARIS designed to handle optimization problems, which also incorporates significantly better heuristics.

The specific context of use is to take problems from the AT-TEND encoder that represent (a subset of) a SNAP problem. [?]

2. BASIC PARIS

PARIS is basically a direct reimplementaion (by Heidi Dixon) of zCHAFF ([12] the best SAT solver at the time) but with the addition of support for cardinality and full-PB clauses. The paper [5] actually discusses an obsolete version of PARIS that was based on RELSAT [2] rather than CHAFF, however the high-level ideas are the same.

The two most important features to get right in a SAT-based solver are fast propagation and the ability to learn constraints when backtracking. We discuss each in turn.

2.1 Propagation

Unit propagation is the process of finding all cases in which the combination of a clause and the current partial assignment P force a new literal; for example if we have the clause $x \vee y$ and $\bar{x} \in P$ then we can deduce y and extend P .

It is critical that unit propagation be efficient. Clearly, it is possible to propagate by simply walking every clause, however, there

are better methods (now common in SAT solvers):

- **counting based methods:** for each clause, maintain a count of the number of valued, and the number of true, literals. A clause can only lead to propagation (or contradiction) if it has zero true literals and at most one unvalued literal. Hence there is no need to walk the clause until the counting tests are met. This is better than constantly having to walk the clause, but the cost of maintaining the counts is still high.
- **watched literal methods:** developed for SAT solvers [15] and now the generally preferred approach, this relies on the observation that if two of the literals in a clause are unvalued then the clause cannot cause propagation, independently of the value of any of the other literals. Hence for every clause just two literals of the clause are marked to be watched. The clause is not inspected until the search inspects one of these two literals—in particular, changes to other literals in the clause do not require any action. This saves many cycles. If the search does cause one of the watched literals to be changed, then the clause might have to be walked, and new literals selected for watching, but in practice, this method is still a significant win.

It is straightforward to extend the watched literals method to handle cardinality constraints. For a constraint of the form:

$$\sum_{i=1}^N l_i \geq K \quad (3)$$

propagation cannot happen if $(K+1)$ or more literals are unvalued, and so we simply watch $K+1$ literals within the constraint.

However, for full PB constraints we need to take account of all the weights of the literals and so we use a counting-based method. It is useful to talk about the deficit of a clause as being the value of the LHS - the RHS, that is $\sum_i w_i - K$ for eq. 2. Then we maintain:

- **poss** the highest potential value of the deficit. Note that $poss \geq 0$ is necessary for the clause to be satisfiable. The value of *poss* can be used to detect when propagations will happen, or when the clause forces a backtrack.
- **curr** the lowest possible value of the deficit. *curr* can only increase as literals are valued. If $curr \geq 0$ then the clause is satisfied.

Indexing structures are maintained so that we can quickly find all the clauses that contain a literal and adjust the counts.

2.2 Clause Learning

Learning during the reasoning process can make it possible for the solver to avoid repeating unproductive work. The basic idea is to take the clauses involved in a contradiction and combine them to produce a new clause. In particular, the search will discover a backtrack when it finds a constraint that forces some variable to have a value opposite to that already assigned to it. The variable is called the conflict variable, the clauses that force it to have opposite values are called conflict clauses. In SAT, the two conflict clauses are resolved together to produce a new learned clause, and the backtrack corresponds to performing a sequence of such resolutions. (Note that while most SAT solvers do not explicitly do resolution, they do other reasoning that has the same net effect as a set of resolutions.)

The basic idea in PARIS is to make the inference within the constraint learning become explicit, and to replace the inference by resolution with inference by a reasoning system appropriate to PB, specifically that of “cutting planes” (CP). CP reasoning has two simple components:

- taking linear combinations of inequalities:
e.g., from $2x + y \geq 2$ and $x + y \geq 1$, derive $3x + 2y \geq 3$
- rounding: e.g., from $2x + 2y \geq 3$ we have $x + y \geq 3/2$, but because variables are 0/1 we must also have $x + y \geq 2$.

However, there are still some choices about what should be learned, and the user can select among these. Options are:

- **learn cardinality:** learn the best cardinality constraint. If the conflict clauses, or the learned clause, is full PB then some clauses are weakened until they become cardinality constraints (the weakening is done in such a fashion as to preserve the learning needed to drive the backtracking). This learning method has the disadvantage that the weakening may discard some chance to learn something important. However, it has the advantage that the coefficients are always one, and so the constraints remain relatively simple.
- **learn PB:** allow the inference to produce a full PB formula. This is not always possible, as sometimes the learned clause might be satisfied: when the conflict variable actually has a consistent real value (e.g., $1/2$), since the learned clause does not contain the conflict variable, it cannot care whether the value is real or Boolean; the learned clause must be satisfiable.
- **learn “best”** infer multiple clauses (such as both cardinality and full-PB), and pick the one that gives the largest back-jump.

Initial expectations were that full PB or “best” learning would be the best performing, but, somewhat surprisingly the cardinality learning generally out-performs them. It is not yet understood why, and might be just a side-effect of other heuristics. Certainly, there is significant potential for improvement.

Note that the SAT solvers are based on the DPLL [4, 3] algorithm, and that this algorithm was extended to PB by Barth, in the solver “opbdp” [1]. Both OPARIS and opbdp handle the same input problems. The essential difference between them is that OPARIS incorporates clause learning; without this the latest methods of SAT solvers cannot be used, but more importantly, it is the learning of PB clauses and not just CNF (or no learning at all) that can really take advantage of the PB representation. For example, even though opbdp works with the PB representation it will still take exponential time on the PHP. In contrast, PARIS/OPARIS take polynomial time.

3. OPTIMIZATION WITH PARIS: OPARIS

The point of OPARIS is to extend PARIS to handle optimization problems where, in addition to the constraints, we are given an objective function, Q , a linear sum of literals representing the quality of the solutions:

$$Q = \sum_i w_i l_i \quad (4)$$

The intention is to maximize Q . That is, to solve problems expressed in the form:

$$\begin{array}{ll} \max & Q \\ \text{such that} & \\ & \Gamma \end{array} \quad (5)$$

where Γ is a set of PB constraints.

Note that if we pick a target quality, q , then demanding that it is achieved is simply equivalent to adding the constraint $Q \geq q$, which is itself a PB constraint, so we can solve $\Gamma \wedge (Q \geq q)$ using PARIS.

Hence OPARIS works by simply using PARIS in order to solve a sequence of problems in which solution quality is forced to increase.

PROCEDURE 3.1 (OPARIS). *Given a PB maximization problem, compute a solution with maximal quality:*

```

1   $q \leftarrow 0$ 
2   $\Gamma_W \leftarrow \Gamma$                                      # Constraint Set
3   $M_B \leftarrow \text{none found}$                              # Best Model
4  while (1)
5      $\Gamma_W \leftarrow \Gamma_W \wedge (Q \geq q)$ 
6      $M_W \leftarrow \text{PARIS}(\Gamma_W)$ 
7     if (  $M_W$  is failure )
8         then
9             return  $M_B$ 
10        else                                           #  $M_W$  is the new best
11             $M_B \leftarrow M_W$ 
12             $q \leftarrow \text{quality}(M_W)$ 
13            if (  $q$  is already maximal ) return  $M_B$ 
14             $q \leftarrow q + 1$ 

```

The constraint is tightened on line 5, unless (on line 13) we already know that the objective is maximally satisfied (all the literals in Q were already true), and no improvement is possible.

Note that the calls to PARIS have the side effect of adding new derived clauses to Γ_W during the search. However, the logic is monotonic—adding a constraint never invalidates a derived clause. Hence, trivially, there is no need to delete learned clauses when we tighten. This means that some of the work used to derive early solutions can be used to prune the search on subsequent calls to PARIS.

In practice, the call to PARIS is given a time-limit, and if reached then the returned M_B is reported as the best found, rather than the optimal.

4. HEURISTICS EXPLOITED

In moving from PARIS to OPARIS we also extended, and improved the implementation of, many of the standard heuristics. These include the familiar SAT technique:

- **rapid restarts:** the search is repeatedly restarted at the root node. It has long been known that restarts can significantly improve the performance of depth-first search [10]. More recently, restarts have very effectively been combined into algorithms with learning [9].

and the usual methods for selecting branch variables:

- **VSIDS:** from ZCHAFF [12]
- **“Current-Top-Clause” (CTC):** from Berkmin [8]

Both VSIDS and CTC try to keep the search focused within the region defined by previously learned clauses, rather than jumping around the search space too much. This seems to enhance the effectiveness of the learning.

Somewhat more unusual are methods to keep a history of good values, and methods to efficiently handle pure literals, described below.

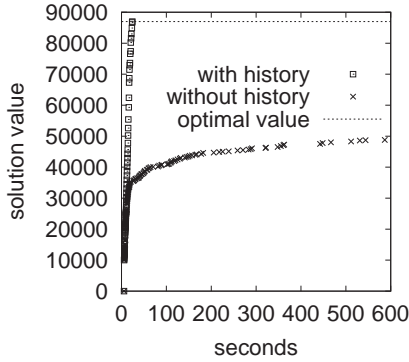


Figure 1: Performance with/without the History mechanism

4.1 History

A well-known technique in the search community is that of keeping a “history”, that is, storing the previous choices within the search. This has been particularly well exploited for SAT by the algorithm “UnitWalk” [11]. The “history” value of a variable is stored and is only changed when propagation forces it to the opposite value.

In particular, if a variable is unset during search, for example by a restart, then the history value can be reused. It is reasonable (though unconfirmed) to expect that this functions by

- allowing the storage of “goods”—if a lot of search effort has gone into finding a solution to sub-problem, then it would be retained, unless forced to change.
- keeping the search focused, by giving some impetus to return to previous regions, and so retaining the relevance of the learned information.

Since both SAT and PB have Boolean variables, it was straightforward to implement this within OPARIS. However, despite the simplicity, it made a large difference. For example, Figure 1 shows performance on an instance from ISI, and shows that without the history the search can easily get stuck at a very sub-optimal solution quality.

4.2 Structure-Dependent Heuristics

This section describes some heuristics that were motivated by the particular structure of ISI instances. In particular, there are variables s_i that are true iff if task i is to be performed. Thus many critical constraints have the form:

$$s_i \rightarrow \text{constraint} \quad (6)$$

The objective function is of the form:

$$\sum_i v_i s_i \quad (7)$$

where v_i is the value of task i .

4.2.1 Pure Literals

A pure literal is a literal that occurs only positively in the (unsatisfied) constraints. In this case, it cannot affect satisfiability to simply enforce the pure literal. It is rare that a literal is pure in the input theory, but literals can become pure at nodes within the search process (all clauses with the bad sign being satisfied because of some other literal), and could be enforced at those node (and all their children). This is a well-known technique in SAT, but has usually been omitted from solvers, because the complexity of testing for pure literals outweighs the advantages from enforcing them.

However, in the ISI instances, turning off a switch means deciding not to perform a task, and so all the variables corresponding to the decision of how to perform that task become irrelevant. However, in the encodings, these irrelevant variables do not disappear, instead they become purely negative, and so the desirable thing to do is to simply set them all to false. Unfortunately, if this is not done, there is a danger that the branch heuristics might consider them for branching. In some cases, the branch heuristics might carefully decide, one-by-one, which of the thousands of pure literals to branch on first. If this is allowed to happen, then even the trivial solution, in which no tasks are enabled, and all constraints ought to be trivially satisfied, can take many seconds to compute. To stop this happening we enable a limited form of pure literal reasoning.

In a preprocessing stage we build “pure literal propagation” rules of the form:

$$x \Rightarrow y \quad (8)$$

with the meaning that, if x is enforced then y becomes a pure literal, and so we can enforce y . It is important to note that this is a rule, and not a logical implication—it should not be reversed or used within the learning.

The preprocessing will typically only be done with x taken from a small set. For the ISI instances it is generally only worthwhile to look for rules following from a switch being turned off, that is:

$$\neg s \Rightarrow y \quad (9)$$

These rules can be used:

- to prevent a pure literal from being selected for branching,
- to enforce pure literal rules when no other branch variables are available.

These are often essential for preventing the heuristics from having to make many pointless decisions.

4.2.2 Branching on switch variables

Given the important role played by the switch variable, it can be advantageous to encourage the search to branch on them. Hence, methods are supplied to give an extra weight to selected groups of variables.

5. RESULTS

Firstly, we remark that it makes little sense to compare with SAT solvers because of the difficulty of converting to sensible sized SAT representation. In the ISI instances, the weights in the objective function are quite large and so make a SAT equivalent very large and unwieldy.

Instead, the main practical alternative to OPARIS is the local search algorithm WSAT(OIP) [14], which works with the same representation and does iterative repair. Although it has the disadvantage of being an incomplete solver (it can never prove a solution is optimal) it can often be highly effective.

The relative performance of OPARIS and WSAT(OIP) on instances from ATTEND depends a lot on whether or not the instances include the addition of extensions needed to handle pilot qualifications [?]. For example, Figure 2 gives performance on an instance without the extensions. The optimal value is 19200 and OPARIS both finds the optimal and proves it optimal before WSAT(OIP) even finds it. However, in this case, in the earlier stages of the search the performance of WSAT(OIP) is better than OPARIS. This seems to be fairly typical behavior—in the easy early stages nonsystematic

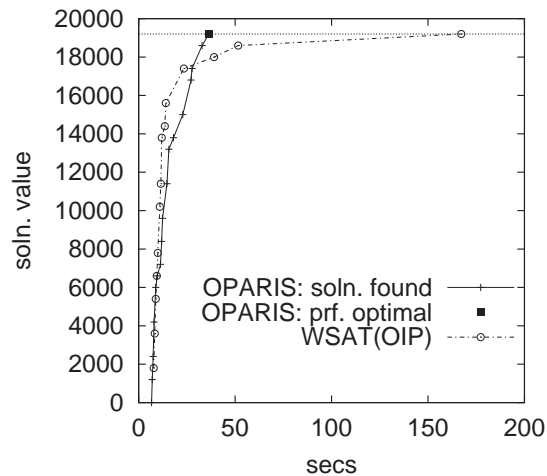


Figure 2: OPARIS vs. WSAT without the pilot qualifications extension.

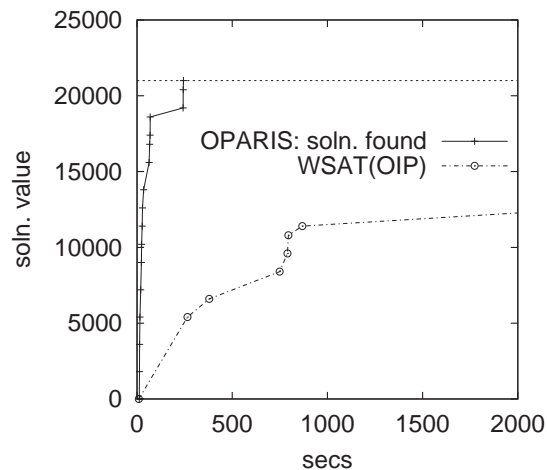


Figure 3: OPARIS vs. WSAT(OIP) with the pilot qualifications extension.

search is faster, but as the problem gets harder the systematic search does better.

Figure 3 shows performance on an instance with the extensions. The extensions allow pilot qualifications to be achieved, and now it turns out all tasks can be scheduled leading to an optimal value of 21000. In this case, WSAT(OIP) does not even do well during the early stages. We think this is because the pilot quals extension correspond roughly to a planning problem, with long chains of reasoning, and it is known that WSAT does poorly with long implication chains. In contrast, the propagation-based search of OPARIS, handles long implication chains naturally.

6. DISCUSSION AND CONCLUSIONS

SAT solvers perform well on problems that do not require a significant amount of arithmetic reasoning, but can lose out exponentially badly on problems, such as PHP, in which even the simplest arithmetic can be of help. To remedy this we have taken the underlying methods of SAT solvers, and modified them for the Pseudo-Boolean representation: In particular, clause learning is modified to allow the learning of PB clauses and so gain the power of some arithmetic reasoning.

There are many possible avenues for future work, but of these

two stand out: making the search algorithm adaptive to the time bound; and “lifting” the representation so as to make it more concise.

6.1 Using the time bound

Usually, on realistic problems, OPARIS is not going to be able to prove optimality, and so must be given an explicit time bound (returning the best solution found on timing out). It would be standard for such an anytime algorithm with a time bound (a contract algorithm in this case) to reason based on the time bound and so select or adjust the heuristics so as to find better solutions. This is currently not done for two reasons. Firstly, the heuristics are relatively new and, though effective, they are poorly understood. This makes it hard to obtain the necessary predictions of their effects on “performance profiles”. Secondly, some preliminary experiments suggested that any exploitable effects were small; it tended to be that if a heuristic worked well, then it worked well over all time-bounds, and so should just be selected uniformly. However, further work on this topic is warranted.

6.2 Lifting

Even though the PB representation is significantly more concise than SAT, the problems can still get quite large, and be in danger of exhausting available memory.

A significant part of the reason for this is that many constraints are most naturally expressed using a lifted form, that is, with universal quantifiers over domains, e.g.:

$$\forall i, j, k. \quad p(i, j) \vee \neg r(j, k) \quad (10)$$

but the quantifiers must be expanded, “ground out”, before using OPARIS or WSAT(OIP).

This problem is already familiar from work in SAT, and a natural solution is to modify the solvers so as to work directly with the lifted form. Originally it was believed that this would be too inefficient, however, work at CIRL on QPROP ([13, 7] and surveyed in [6]) showed that use of the lifted form can be efficient. In fact, surprisingly, there are theoretical reasons why it might even be more efficient than using the ground form.

A reasonable goal would be a language that mixes QPROP and PB, “Quantified Pseudo-Boolean”, QPB, and so for example allows constraints such as

$$\forall i, j, k. \quad 2p(i, j) + r(j, k) + q(r, k) \geq 2 \quad (11)$$

This is especially reasonable as it is essentially (a subset of) the language of OR modeling languages such as AMPL², and so there is already evidence as to the utility of the representation itself.

7. ACKNOWLEDGMENTS

This was sponsored in part by grants from the Defense Advanced Research Projects Agency (DARPA), under contract number F30602-00-2-0534. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory, or the U.S. Government.

²<http://www.ampl.com>

8. REFERENCES

- [1] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-003, Max Planck Institute, 1995.
- [2] R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, Providence, RI, 1997.
- [3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Journal of the ACM*, 5:394–397, 1962.
- [4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [5] H. E. Dixon and M. 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.
<http://www.cirl.uoregon.edu/dixon/>.
- [6] H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Likely near-term advances in SAT solvers. In *Workshop on Microprocessor Test and Verification (MTV'02)*, June 2002.
<http://citeseer.nj.nec.com/dixon02likely.html>.
- [7] M. L. Ginsberg and A. J. Parkes. Satisfiability algorithms and finite quantification. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*, pages 290–701. Morgan Kaufmann, 2000.
- [8] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
- [9] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998. <http://citeseer.nj.nec.com/gomes98boosting.html>.
- [10] W. D. Harvey. *Nonsystematic Backtracking Search*. PhD thesis, Stanford University, 1995.
- [11] E. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, 2001. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg.
- [12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, 2001.
- [13] A. J. Parkes. *Lifted Search Engines for Satisfiability*. PhD thesis, University of Oregon, June 1999. Available from <http://www.cirl.uoregon.edu/parkes>.
- [14] 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.
- [15] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.