A Stochastic Non-CNF SAT Solver

Rafiq Muhammad and Peter J. Stuckey

NICTA Victoria Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Victoria 3010, Australia
[mrmu, pjs]@cs.mu.oz.au

Abstract. Stochastic local search techniques have been successful in solving propositional satisfiability (SAT) problems encoded in conjunctive normal form (CNF). Recently complete solvers have shown that there are advantages to tackling propositional satisfiability problems in a more expressive natural representation, since the conversion to CNF can lose problem structure and introduce significantly more variables to encode the problem. In this work we develop a non-CNF SAT solver based on stochastic local search techniques. Crucially the system must be able to represent how true a proposition is and how false it is, as opposed to the usual stochastic methods which represent simply truth or degree of falsity (penalty). Our preliminary experiments show that on certain benchmarks the non-CNF local search solver can outperform highly optimized CNF local search solvers as well as existing CNF and non-CNF complete solvers.

1 Introduction

Modern propositional satisfiability (SAT) solvers are usually designed to solve SAT formula encoded in conjunctive normal form (CNF). CNF solvers can be disadvantageous for problems which are more naturally encoded as arbitrary propositional formula. The conversion to CNF form may increase the size of the formula exponentially, or significantly reduce the strength of the formulation. The translation may introduce many new variables which increases the size of the raw valuation space through which the solver must search.

Recently, interest has arisen in designing non-clausal satisfiability algorithms. In 1993, Armando and Giunchiglia [1] introduced PTAUT, which was a generalization of Davis-Putnam-Logemann-Loveland (DPLL) algorithm [6], a complete CNF SAT algorithm, to work on non-CNF formula. The primary drawback of their implementation was the performance which was far below current implementations of CNF solvers based on the DPLL algorithm, because it failed to exploit clever techniques and efficient data structures used in manipulating CNF formulae, e.g.. [2, 3]. This work was improved by Giunchiglia and Sebastiani [4] who devised a non-CNF approach able to exploit all the present and future sophisticated technology of DPLL implementations. They converted the input formula to CNF, but ensured that the DPLL procedure would not backtrack on new variables introduced in CNF conversion hence maintaining the search space of the original non-CNF formulae.

More recently, Thiffault et al. [5] generalized the DPLL to work directly on non-CNF formulae. They argued that conversion to CNF is unnecessary and results in the drawback of losing structural information and increase in the search space. They implemented a complete non-CNF DPLL like solver that is capable of achieving an efficiency very similar to that of modern highly optimized CNF solvers using techniques very similar to these solvers. They exploited the additional structural information presented in non-CNF propositional formula to achieve significant gains in solving power, to the point where on various benchmarks their non-clausal solver outperforms the CNF solver it was based on.

Local search based SAT solvers can typically solve problems an order magnitude larger than those that can be handled by a complete solver. For CNF problems, GSAT [7] demonstrated that an incomplete SAT solver can solve many hard problems much more efficiently than traditional complete solvers based on DPLL. The success of GSAT gave birth to several variants based on stochastic local search techniques, see [8]. Several authors have attempted to generalize these techniques to non-CNF formula. Sebastiani [9] suggested how to modify GSAT to be applied to non-CNF formula but the idea was not implemented. Kautz et al. [10] introduced DAGSat, an improvement of WalkSAT in term of handling of variable with dependencies. DAGSat still required formula in CNF, but allowed handling of non-CNF formula without an increase in size. Later, Stachniak [11] introduced polWSAT an evolution of WalkSAT [12] to handle non-clausal formula, although it was restricted to formula using $\wedge$, $\vee$ and $\neg$ in negation normal form (where all negations appear on literals).

In this paper, we present a pure non-clausal solver based on stochastic local search techniques. It works directly on arbitrary non-clausal formulae including psuedo-Boolean constructs and other extended Boolean operators. The contributions of this paper are:

- We present a new way of expressing the scoring function for evaluating a valuation. We are not only able to define the "falsity" (or penalty) of a non solution, but also the "truthfulness" of an assignment, that is, how true it makes the result. This is required since we need to negate "truthfulness" to obtain "falsity". This complex scoring is necessary since we don't restrict ourselves to negation normal forms. It provides more accurate heuristic information to assist the local search process.
- We tested our solver on non-CNF benchmarks and demonstrate experimentally that on some benchmarks, our incomplete non-clausal solver can outperform current incomplete clausal solvers as well as current complete non-clausal solvers.

2 A Stochastic Non-Clausal Solver

Let ϕ be a propositional formula, our aim is to decide if ϕ is satisfiable. The propositional formula is represented as a Boolean DAG where each internal node

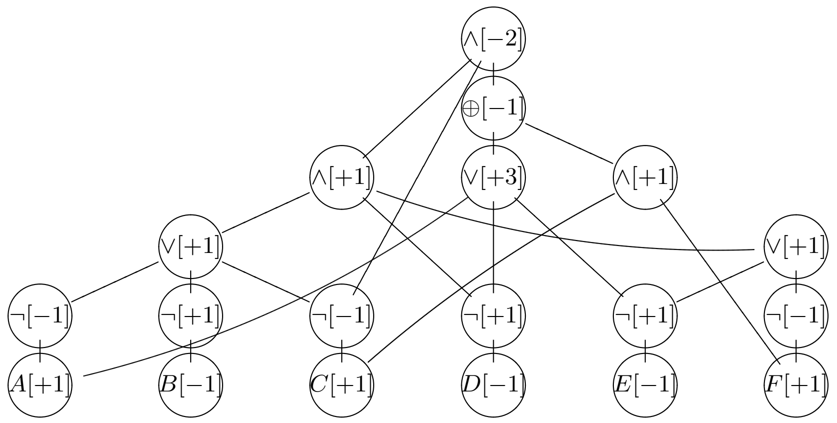

Fig. 1. The propositional formula $\phi = (((\neg A \vee \neg B \vee \neg C) \wedge \neg D \wedge (\neg E \vee \neg F)) \wedge \neg C \wedge ((\neg D \vee A \vee \neg E) \oplus (C \wedge F)))$ with scores for the valuation $\{A, \neg B, C, \neg D, \neg E, F\}$ bracketed.

represents an (extended) Boolean operator and its children are sub-trees representing its operands. For example, Figure 1 represents the Boolean formula ϕ as a DAG.

The aim of our algorithm is to find values for the variables such that will result in the satisfiability of the propositional formula. For each assignment, a score is computed for each node in the DAG to represent the state of satisfiability of the corresponding propositional (sub-)formula.

2.1 Score

The notion of score plays a key role in determine the "distance" from the current valuation to a satisfying one. We allow our scoring function to take positive and negative values in order to enable our algorithm to express the "truthfulness" as well as the "falsity" of the state of the truth assignment. All Boolean variables (leaf nodes) have a score of either 1 (*true*) or (-1) *false*. For each internal node we calculate the score as defined below. The calculations are such that: if the Boolean DAG were a tree then a score of $+n$ means that n variables must change their value for the corresponding (sub-)formula to become *false*, while if the score is $-n$ then n variables must change their value for the corresponding (sub-)formula to become *true*. Of course in practice the DAG is never a tree.

We will determine the score s_0 for a node ϕ_0, in terms of the scores $s_1, \ldots s_n$ of its n children $\phi_1, \ldots \phi_n$.

NOT($\neg$) $\phi_0 = \neg\phi_1$

Not is the negation of the truth value of a variable, therefore the score for a negated node, s_0, is the negation of the score of its child: $s_0 = -(s_1)$.

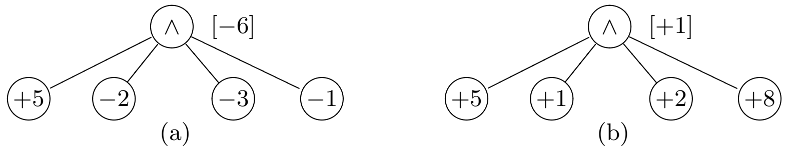

Fig. 2. AND score (a) where the node is *false* and (b) where it is *true*

AND(∧) $\phi_0 = \phi_1 \wedge \cdots \wedge \phi_n$

An AND node (with n children) can only be *true* if all its children are *true*, therefore, if the node is currently *false*, the score of the node is the sum of negative children (Fig. 2(a)) and if the node is *true*, the score of the node is the score of the child with minimum value (Fig. 2(b)).

$$s_0 = \begin{cases} \sum\{s_i | 1 \leq i \leq n, s_i < 0\} & \exists 1 \leq i \leq n \ s_i < 0 \quad (false) \\ \min\{s_i | 1 \leq i \leq n\} & \text{otherwise} \quad (true) \end{cases}$$

For example, in order to change the node in Fig. 2(a) to be *true*, we have to change the nodes with [−2], [−3] and [−1] to have a positive score, hence the score of the parent is [−6]. On the other hand, the score of the node in Fig. 2(b) is [+1] because the minimum change to turn the node to *false*, is by turning the truth value of the child with score [+1] to *false*.

OR(∨) $\phi_0 = \phi_1 \vee \cdots \vee \phi_n$

An OR node becomes *true* when any one of its children is *true*, and *false* if all of its children are *false*, therefore, if the node is *false*, the score of the node is the score of the child with maximum value and if the node is *true*, the score of the node is the sum of positive children.

$$s_0 = \begin{cases} \max\{s_i | 1 \leq i \leq n\} & \forall 1 \leq i \leq n \ s_i < 0 \quad (false) \\ \sum\{s_i | 1 \leq i \leq n, s_i > 0\} & \text{otherwise} \quad (true) \end{cases}$$

XOR(⊕) $\phi_0 = \phi_1 \oplus \cdots \oplus \phi_n$

An XOR ϕ_0 is *true* if the parity of the total number of *true* children is odd. Simply flipping the truth value of any one of the children of ϕ_0 will change the parity from odd to even and vice versa. Hence the XOR node that is *false* has a score equal to the negative of the smallest absolute value of any child score, while if its true, the score is the smallest absolute value of the child score.

$$s_0 = \begin{cases} -\min\{|s_i| \ | 1 \leq i \leq n\} & |\{s_i | 1 \leq i \leq n, s_i > 0\}| \bmod 2 = 0 \quad (false) \\ +\min\{|s_i| \ | 1 \leq i \leq n\} & \text{otherwise} \quad (true) \end{cases}$$

For example, the XOR node in Fig. 3(a) is *false* because 2 of the children are *true* (an even number). In order to make the node to be *true*, we have to flip the truth value of one of the children. The least change required is to change the [+2] child to be *false*, hence the score is [−2]. Fig. 3(b) shows the score of an XOR node that is *true*.

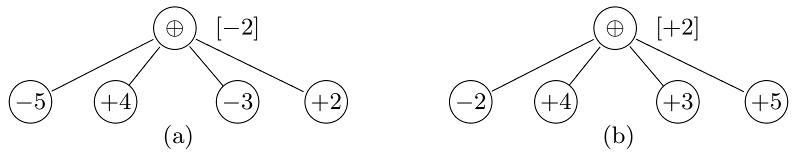

Fig. 3. XOR score (a) where the node is *false* and (b) where it is *true*

IFF($\Leftrightarrow$) $\phi_0 = \phi_1 \Leftrightarrow \cdots \Leftrightarrow \phi_n$

Equivalent (IFF) is the opposite of XOR. An IFF node ϕ_0 is *true* if the parity of the total number of *true* children is even. The resulting score function is then:

$$s_0 = \begin{cases} -\min\{|s_i| \; |1 \le i \le n\} & |\{s_i | 1 \le i \le n, s_i > 0\}| \bmod 2 = 1 \quad (false) \\ +\min\{|s_i| \; |1 \le i \le n\} & \text{otherwise} \quad (true) \end{cases}$$

IMPLIES($\Rightarrow$) $\phi_0 = \phi_1 \Rightarrow \phi_2$

Implication is a simple binary operator. An implication formula is *false* if and only if the left operand is *true* and right operand is *false*. If the implication node is *false*, then flipping either operand will turn the implication node to be *true*. If the implication node is *true*, the cost of turning the node to *false* is the sum of the cost of turning the left operand to *true* and the cost of turning the right operand to *false*.

$$s_0 = \begin{cases} \max\{-s_1, +s_2\} & s_1 > 0 \wedge s_2 < 0 \quad (false) \\ \max\{s_2, 0\} - \min\{s_1, 0\} & \text{otherwise} \quad (true) \end{cases}$$

ATMOST($\le k$) $\phi_0 = (\phi_1 + \cdots + \phi_n \le k)$

An ATMOST formula ϕ_0 with parameter k ($\le k$) is true if at most k of its n inputs are *true*. If the node is *false*, and currently $k' > k$ children are *true*, we need to make *false* $k' - k$ more variables, so the score is negative of the minimum sum required to do this. Similarly, if the node is *true*, and currently $k' \le k$ children are *true*, we need to turn $k - k' + 1$ children *true* to make it *false*.

$$s_0 = \begin{cases} -\sum \min_{k'-k} T & T = \{s_i \mid 1 \le i \le n, s_i > 0\}, \\ & k' = |T|, k' > k \quad (false) \\ -\sum \max_{k-k'+1}\{s_i \mid 1 \le i \le n\} - T & T = \{s_i \mid 1 \le i \le n, s_i > 0\}, \\ & k' = |T|, k' \le k \quad (true) \end{cases}$$

where $\min_l S$ returns the minimal l elements of S, that is a subset M of S of cardinality l such that $\forall x \in S - M.\forall y \in M.x \ge y$. Similarly $\max_l$ returns the maximal l elements of S.

For example, in Fig. 4(a), to change the node to be *true*, we need to change two children to be *false*. The cheapest way of doing this is with the [+1] and [+2] children. $T = \{+6, +4, +1, +2\}$, $k' = 4$ and $\min_2 T = \{+1, +2\}$. Hence the resulting score is [−3]. In Fig. 4(b), to change the node to be *false*, we would need to make at least two more children *true*. $T = \{+6\}$, $k' = 1$ and $\max_2\{-4, -3, -2\} = \{-2, -3\}$. Hence the score is [+5].

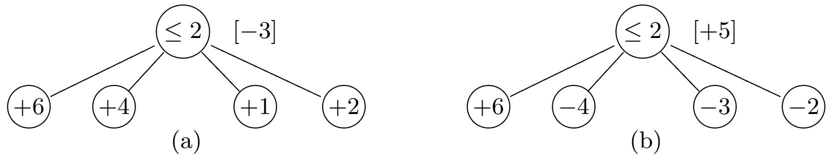

Fig. 4. ATMOST 2 score (a) where the node is *false* and (b) where it is *true*

ATLEAST($\geq k$) $\phi_0 = (\phi_1 + \cdots + \phi_n \geq k)$

The ATLEAST operator is similar to the ATMOST operator. The resulting scoring function is thus a mirrored form.

$$s_0 = \begin{cases} \sum \max_{k-k'}\{s_i \mid 1 \leq i \leq n\} - T & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\ & k' = |T|, k' < k \qquad (false) \\ \sum \min_{k'-k+1} T & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\ & k' = |T|, k' \geq k \qquad (true) \end{cases}$$

COUNT($\equiv k$) $\phi_0 = (\phi_1 + \cdots + \phi_n = k)$

The COUNT operator is simply a combination of the ATMOST and ATLEAST operators, but it ends up with a slightly different form of scoring function, after simplifying. If the node is *false* to make it *true* we need to flip truth values to bring the count either up or down to k. To make it *false* from *true* we only have to flip one truth value.

$$s_0 = \begin{cases} \sum \max_{k-k'}\{s_i \mid 1 \leq i \leq n\} - T & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\ & k' = |T|, k' < k \qquad (false) \\ -\sum \min_{k'-k} T & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\ & k' = |T|, k' > k \qquad (false) \\ \min\{|s_i| | 1 \leq i \leq n\} & \text{otherwise} \qquad (true) \end{cases}$$

Other Operators There are also other operators defined in DIMACS non-clausal style [13] such as NAND, NOR, and XNOR. These operators are the negation of operators we have already defined. We can generate their scoring functions by simply negating the corresponding versions: NAND = − AND, NOR = − OR, and XNOR = − IFF.

2.2 Searching for a Solution

All local search based solvers work in essentially the same way. A candidate valuation for the variables is determined, and then the search looks for a "neighboring" valuation which is better. In SAT solvers the usual definition of neighboring valuations, is those obtained by flipping the value of one Boolean variable. A valuation is considered better if it satisfies more clauses, or satisfies a greater sum of weighted clauses. In the non-clausal solver the situation is the same. We have a current valuation and its score for the overall formula. We look at neighboring valuations which improve the score, attempting to drive the score to be positive (and hence satisfying the root propositional formula).

We consider a search strategy similar to the approach used by WalkSAT [12] and its extension to non-clausal solvers [11]. WalkSAT works by first selecting an unsatisfied clause, and then selecting a variable in that clause for flipping. A CNF formula is simply a root AND node above many OR nodes relating to literals (variables or NOT variables sub trees). How do we generalize the WalkSAT approach to an arbitrary formula? We consider it in this way. The WalkSAT approach begins at the root and considers which child nodes could improve the root score (a *false* child has a score of -1, and changing it will improve the score of the AND parent) and randomly selects one. It then does the same for the OR node (although in this case all children can improve the score). Hence the generalization is clear. At node n we randomly select a child node, for which flipping the truth value would move the nodes score towards the opposite sign it has now. We continue this process until we reach a variable node. This is the variable we then flip in the search process. Note that the children that can improve a node are implicitly readable from the definition of the scoring function. We flip the variables truth value and with probability p accept the change if it is downhill (improves the score at the root), and with probability $1 - p$ accept the move whether it is downhill, uphill or flat. For our experiments we found a value of $p = 0.9$ was best and use this value throughout our experiments.

Examining the DAG shown in Figure 1. At the root the score of $[-2]$ is the sum of the negative children of $[-1]$, so we randomly select one of them, say the XOR node. Its score is as a result of both children being positive so we randomly select one, say the OR node on the right. Its score is positive because of all three children so we randomly choose one say $\neg E$. This is positive because of its child E. So this is our variable for flipping.

3 Preliminary Experimental Results

We implemented our stochastic non-clausal SAT solver in C++ using the technology of one way constraints or invariants [14]. We compare our stochastic non-CNF solver (SNCNFS) with WalkSAT [12] (an incomplete CNF solver), MiniSat [15] (a complete CNF solver), and NoClause [5] (a complete non-CNF solver) on several test suites. Since our solver is incomplete, we only considered formula that is satisfiable. For the incomplete solvers (WalkSAT and SNCNFS) we used a maximum flips of 250,000 and repeated each test 100 times. The initial starting valuations were uniformly randomly generated. We used the Random strategy of WalkSAT most similar to the search strategy defined above for SNCNFS. Since flips in WalkSAT are substantially faster than those in SNCNFS, whenever SNCNFS solved a larger percentage of problems we increased the maximum flips for WalkSAT to 10,000,000. The benchmarks where this occurred are marked as †. For the complete solvers, we aborted if no solution is found after running for

Table 1. Comparative results for solvers on hard random formula

	CNF					Non-CNF				
		WALKSAT			**MiniSat**	**SNCNFS**				**NoClause**
R	# Clause	% Succ	Mean Flips	Mean Time(s)	Mean Time(s)	# Nodes	% Succ	Mean Flips	Mean Time(s)	Mean Time
1.6	8327	100	153	0.0009	0.0600	8743	100	434	0.2020	2.994
1.8	9333	100	219	0.0014	0.4100	9801	100	2448	1.0364	22.121
2.0	10388	100	700	0.0040	29.0000	10837	100	3256	1.7596	71.352
2.2	11494	100	1975	0.0116	—	11791	100	5282	3.2681	—
2.4	12508	100	9519	0.0406	—	12749	100	19596	13.4205	—
2.6	13491	100	56517	0.3773	—	13907	92	64783	51.0984	—
2.8†	14545	7	4543650	70.3112	—	14942	38	42775	37.3086	—
3.0†	15577	0	—	—	—	15904	0	—	—	—

— Max flips or max time exceeded.

two hours. In the tables, where solutions are found, we give the averages over the successful runs only.

3.1 Hard Random Non-CNF Formulas

Randomly generated formulae provide a good test bed for evaluating the performance of satisfiability algorithms. We used the random formula generator of Navarro and Voronkov [16] which is based on a fixed shape model. The generator is capable of generating formulae with different level of difficulty in non-CNF as well as CNF format hence making it an ideal choice to test our stochastic non-CNF solver. The difficulty and the satisfiability of the output formulae is controlled by r the *ratio of formulae-to-variables*. Small r produces formulae that are satisfiable and under-constrained, while large r results in formulae that are unsatisfiable and over-constrained. The hardest problems appear in the transition region where there are just enough constraints to make the problem potentially unsatisfiable, but not too many to make it easy for a solver to determine. We generated 200-variable random formulae of shape $\langle 3, 3, 2\rangle$ with r increasing by 0.2 within the range from 1.6 to 3.0.

The comparative results are shown in Table 1. For these problems the size of the non-CNF formula is not substantially small than the CNF form, so there is no advantage to the non-CNF solver in size. Clearly the the execution time of SNCNFS are much longer than WalkSAT for the same number of flips, illustrating the highly optimized implementation of WalkSAT. But SNCNFS is capable of solving harder problems than WalkSAT, illustrating there is an advantage in treating the formula in the non-CNF form. For this class of problems the complete solvers are unable to tackle the more difficult cases.

Table 2. Comparison of results on the CNF and Non-CNF encoding of MDP problem

	CNF						Non-CNF				
	WalkSAT					Minisat	SNCNFS				
Problem	# Vars	# Cls	% Succ	Mean Flips	Mean Time(s)	Time	# Vars	# Nodes	% Succ	Mean Flips	Mean Time(s)
par8-1†	350	1149	2	298872	0.149	0.02	8	441	100	3227	0.1782
par16-1†	1015	3310	0	—	—	0.1	16	1649	90	57471	7.0971
par32-1†	3176	10277	0	—	—	—	32	6369	—	—	—
par8-1-c	64	254	100	8516	0.0054	0.01	8	36	100	3123	0.0572
par16-1-c†	317	1264	0	—	—	0.06	16	66	100	32284	1.1499
par32-1-c†	1315	5254	0	—	—	—	32	131	—	—	—

The trailing 'c' in the problem is the compressed version of the instance.
— Max flips or max time exceeded.

3.2 Minimal Disagreement Parity Problem

The Minimal Disagreement Parity (MDP) Problem [17] is a well-known class of hard satisfiability problem. The advantage of a non-CNF encoding for these problems is that we can maintain the encoding of XOR and use the $ATMOST$ gate to determine the correctness of the parity function, hence reducing the problem size significantly. We compare the non-CNF encoding versus the standard CNF encoding. Table 2 shows the results of SNCNFS compared to WalkSAT and MiniSat (we cannot apply NoClause since it does not support the ATMOST gate). Although we are unable to beat Minisat, SNCNFS can solve 16 bit instances which are beyond the capability of WalkSAT.[1]

4 Conclusions and Future Work

We have introduced an incomplete non-clausal solver based on stochastic local search. This is the first work we are aware of which uses both negative and positive scores for evaluating the degree of "truthfulness" or "falsity" of a propositional formula. Our experiments demonstrate that on certain benchmarks, our stochastic local search non-clausal solver can out perform existing incomplete CNF solver as well as complete CNF and non-CNF solvers. The results of our preliminary experiments are very promising.

It would be interesting to find more complex non-clausal benchmarks to experiment on, almost all SAT benchmarks are presently in CNF. The advantage of a non-clausal solver should be more evident on large difficult benchmarks that appear in non-clausal form. There remains a great deal of scope to explore different strategies for selecting a neighborhood move. We could take into account the magnitude of the children scores in weighting the random choice of child to

[1] Systematic methods are known to work well on this class of problem. WalkSAT using the Novelty strategy can solve the compressed version of 16 bits instance, but not the uncompressed version [18].

take. We could also extend the approach to generate a candidate set of variables to flip by keeping more than one child that can change the truth value, and then picking the variable whose flipping leads to the best overall score for the root node. We could add penalties to nodes, and learn penalties when we find ourselves in local minima (as in DLM [20]). Finally, it would be worthwhile reimplementing the algorithm in Comet [19] which provides efficient and built in evaluation of invariants.

References

1. Armando, A., Giunchiglia, E.: Embedding complex decision procedures inside an interactive theorem prover. Ann. Math. Artif. Intell. **8**(3-4) (1993) 475–502
2. Crawford, J., Auton, L.: Experimental results on the crossover point in random 3-SAT. Artif. Intell. **81**(1-2) (1996) 31–57
3. Zhang, H., Stickel, M.: Implementing the Davis-Putnam method. J. Autom. Reasoning **24**(1/2) (2000) 277–296
4. Giunchiglia, E., Sebastiani, R.: Applying the Davis-Putnam procedure to non-clausal formulas. In: AI*IA. (1999) 84–94
5. Thiffault, C., Bacchus, F., Walsh, T.: Solving non-clausal formulas with DPLL search. In: CP. (2004) 663–678
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (1962) 394–397
7. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: AAAI. (1992) 440–446
8. McAllester, D.A., Selman, B., Kautz, H.A.: Evidence for invariants in local search. In: AAAI/IAAI. (1997) 321–326
9. Sebastiani, R.: Applying GSAT to non-clausal formulas (research note). J. Artif. Intell. Res. (JAIR) **1** (1994) 309–314
10. Kautz, H., Selman, B., McAllester, D.: Exploiting variable dependency in local search. In Abstracts of the Poster Session of IJCAI-97 (1997)
11. Stachniak, Z.: Going non-clausal. In 5th International Symposium on Theory and Applications of Satisfiability Testing (2002)
12. Selman, B., Kautz, H., Cohen, B.: Noise strategies for improving local search. In: AAAI. (1994) 337–343
13. Bacchus, F., Walsh, T.: A non-CNF DIMACS style. Available from http://www.satcompetition.org/2005/ (2005)
14. Van Hentenryck, P., Michel, L.: Localizer. Constraints **5** (2000) 41–82
15. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. (2003) 502–518
16. Navarro, J.A., Voronkov, A.: Generation of hard non-clausal random satisfiability problems. In: AAAI. (2005) 436–442
17. Crawford, J., Kearns, M., Schapire, R.: The minimal disagreement parity problem as a hard satisfiability problem. unpublished manuscript (1995)
18. Hoos, H.H., Stützle, T.: Systematic vs. local search for sat. In: KI. (1999) 289–293
19. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
20. Wah, B.W., Shang, Y.: A discrete lagrangian-based global-search method for solving satisfiability problems. J. of Global Optimization **12** (1998)