J Glob Optim (2008) 41:61–73
DOI 10.1007/s10898-007-9166-9

An efficient solver for weighted Max-SAT

Teresa Alsinet · Felip Manyà · Jordi Planes

Received: 3 July 2006 / Accepted: 18 May 2007 / Published online: 29 June 2007
© Springer Science+Business Media LLC 2007

Abstract We present a new branch and bound algorithm for weighted Max-SAT, called `Lazy` which incorporates original data structures and inference rules, as well as a lower bound of better quality. We provide experimental evidence that our solver is very competitive and outperforms some of the best performing Max-SAT and weighted Max-SAT solvers on a wide range of instances.

Keywords Max-SAT · Weighted Max-SAT · Branch and bound · Lower bound · Heuristics · Data structures

1 Introduction

In recent years we have seen an increasing interest in propositional satisfiability (SAT) that has led to the development of fast and sophisticated complete SAT solvers like Chaff [29], SATO [27], and Satz [15], which are based on the well-known Davis–Putnam–Logemann–Loveland (DPLL) procedure [6]. Such algorithms determine whether there is a truth assignment that satisfies the input CNF formula.

A Conjunctive Normal Form (CNF) formula is a conjunction of clauses, each clause being a disjunction of literals, and each literal being a propositional variable or its negation. A truth assignment is a mapping that assigns to each propositional variable either the value 0 (for false) or the value 1 (for true), and it satisfies a clause if it satisfies at least one of its literals and satisfies a CNF formula if it satisfies all its clauses.

T. Alsinet · J. Planes
Computer Science Department, Universitat de Lleida, Jaume II, 69, 25001 Lleida, Spain
e-mail: tracy@diei.udl.es

J. Planes
e-mail: jplanes@diei.udl.es

F. Manyà (✉)
Artificial Intelligence Research Institute (IIIA, CSIC), Campus UAB, 08193 Bellaterra, Spain
e-mail: felip@iiia.csic.es

Unfortunately, SAT algorithms are not able to solve the optimization version of SAT: Max-SAT. Given a CNF formula ϕ, Max-SAT consists of finding a truth assignment that minimizes the number of unsatisfied clauses in ϕ. A more general and related problem is weighted Max-SAT. In this case, a positive integer, called weight, is associated with each clause and the problem consists of finding a truth assignment that minimizes the sum of weights of unsatisfied clauses. A weight represents the significance of the clause or an induced penalty if it is violated.

To the best of our knowledge, there are only four exact algorithms for weighted Max-SAT that are variants of the DPLL procedure. The first was developed by Wallace and Freuder [25] (WF), the second was developed by Borchers and Furman [4] (BF), the third, which is based on BF, was developed by Alsinet et al. [2] (AMP), and the fourth was developed by Xing and Zhang [26] (XZ). All of them are depth-first branch and bound algorithms. The first was implemented in Lisp, while the rest were implemented in C and are publicly available. A weighted Max-SAT solver that encodes the input instance as a weighted constraint network and solves that network with a state-of-the-art weighted Max-CSP solver was developed by Givry et al. [8] (toolbar). There are other exact algorithms for weighted Max-SAT, but based on mathematical programming techniques [5,13,14]. There are also two exact DPLL-based algorithms specialized to solve Max-2-SAT (they do not solve weighted Max-2-SAT): one is due to Zhang et al. [28] (ZSM), and the other to Alber et al. [1] (AGN).

An alternative to exact methods is provided by heuristic and approximation algorithms; they cannot guarantee that the solution found is optimal, but can solve instances that are beyond the reach of the existing exact Max-SAT algorithms. Furthermore, they can be used to compute good quality upper bounds in Max-SAT branch and bound solvers [4,26]. One of the first heuristic algorithms for Max-SAT is the steepest ascent, mildest descent approach by Hansen and Jaumard [10]. Other relevant heuristic algorithms are the reactive search approach by Battiti and Protasi [3], and the Greedy Randomized Adaptive Search Procedures (GRASP) approach to Max-SAT by Resende et al. [20,21]. In the SAT community, variants of the local search algorithms GSAT [23] and WalkSAT [22] have been used to solve Max-SAT; see [24] for a survey. Approximations algorithms based on semidefinite programming are described in Ref. [7,9].

In this paper we first present a new branch and bound algorithm for weighted Max-SAT which incorporates original data structures and inference rules, as well as a lower bound of better quality. We then report on an experimental investigation we have conducted in order to evaluate our solver on (weighted) Max-SAT instances. The results obtained provide experimental evidence that our solver is very competitive and outperforms some of the best performing (weighted) Max-SAT solvers on a wide range of instances.

Our new solver, which we call `Lazy`, differs from previous solvers in the data structures used to represent and manipulate CNF formulas, in the preprocessing simplification techniques applied, in the lower bound computation method, and in the variable selection heuristic (which is static in our solver).

2 Preliminaries

A weighted clause is a pair (C_i, w_i), where C_i is a disjunction of literals and w_i, its weight, is a positive integer. A weighted CNF formula is a conjunction of weighted clauses. In the following when we say clause we refer to a weighted clause, and when we say formula we refer to a weighted CNF formula.

The weighted Max-SAT problem for a formula ϕ is the problem of finding an assignment of values to propositional variables that minimizes the sum of weights of unsatisfied clauses (or equivalently, that maximizes the sum of weights of satisfied clauses). When all the clauses of a formula ϕ have weight 1, we usually omit the weights. The Max-SAT problem is the weighted Max-SAT problem restricted to formulas whose clauses have weight 1, and is defined as the problem of finding an assignment of values to propositional variables that satisfies as many clauses as possible (i.e., minimizes the number of unsatisfied clauses). When all the clauses have at most k literals per clause, (weighted) Max-SAT is called (weighted) Max-k-SAT.

Finally, we introduce the mixed integer programming (MIP) formulation of weighted Max-SAT defined in Ref. [20]. That formulation is used in the experimental investigation to compare our solver with a MIP solver.

Let $\phi = (C_1, w_1) \wedge \cdots \wedge (C_m, w_m)$ be a weighted Max-SAT instance over the the propositional variables $p_1, \ldots, p_n$. Let $y_j = 1$ if variable p_j is true and $y_j = 0$, otherwise. Furthermore, the continuous variable $z_i = 1$ if clause C_i is satisfied and $z_i = 0$, otherwise. The MIP formulation of the weighted Max-SAT instance ϕ is:

$$\max F(y, z) = \sum_{i=1}^{m} w_i z_i$$

subject to

$$\begin{gathered} \textstyle\sum_{j \in I_i^+} y_j + \sum_{j \in I_i^-} (1 - y_j) \geq z_i \quad i = 1, \ldots, m, \\ y_i \in \{0, 1\} \quad i = 1, \ldots, n, \\ z_i \in \{0, 1\} \quad i = 1, \ldots, m, \end{gathered}$$

where I_i^+ denotes the set of unnegated variables in clause c_i and I_i^- is the set of negated variables in C_j.

3 A basic solver for weighted Max-SAT

The space of all possible assignments for a propositional formula ϕ can be represented as a search tree, where internal nodes represent partial assignments and leaf nodes represent complete assignments. A basic branch and bound algorithm for weighted Max-SAT explores the search tree in a depth-first manner. At every node, the algorithm compares the sum of the weights of the clauses unsatisfied by the best complete assignment found so far—called upper bound (UB)—with the sum of the weights of the clauses unsatisfied by the current partial assignment ($unsat$) plus an underestimation of the sum of the weights of the clauses that will become unsatisfied if we extend the current partial assignment into a complete assignment ($underestimation$). The sum $unsat + underestimation$ is called lower bound (LB). Obviously, if UB $\leq$ LB, a better assignment cannot be found from this point in search. In that case, the algorithm prunes the subtree below the current node and backtracks to a higher level in the search tree. If UB $>$ LB, it extends the current partial assignment by instantiating one more variable; which leads to the creation of two branches from the current branch: the left branch corresponds to instantiating the new variable to false, and the right branch corresponds to instantiating the new variable to true. In that case, the formula associated with the left (right) branch is obtained from the formula of the current node by deleting all the clauses containing the literal $\neg p$ (p) and removing all the occurrences of the literal p ($\neg p$);

```
Input: max-sat(φ, UB) : A CNF formula with weights φ and an upper bound
   UB
1: if φ = ∅ or φ only contains empty clauses then
2:    return sum-weights-empty-clauses(φ)
3: end if
4: if  LB(φ) ≥ UB then
5:    return ∞
6: end if
7: p ← select-variable(φ)
8: UB ← min(UB, max-sat(φ_¬p, UB))
9: return min(UB, max-sat(φ_p, UB))
Output: The minimal sum of weights associated with unsatisfied clauses of φ
```

Fig. 1 A basic branch and bound algorithm for weighted Max-SAT

i.e., the algorithm applies the one-literal rule [16]. The solution to weighted Max-SAT is the value that UB takes after exploring the entire search tree.

Figure 1 shows the pseudo-code of a basic branch and bound algorithm for weighted Max-SAT. We use the following notation:

- *sum-weights-empty-clauses*(ϕ) is a function that returns the sum of weights associated with the empty clauses of ϕ. Empty clauses are unsatisfied by any truth assignment.
- LB(ϕ) is a lower bound for ϕ.
- UB is an upper bound of the sum of weights of unsatisfied clauses in an optimal solution. We assume that the initial value is ∞.
- `select-variable`(ϕ) is a function that returns a variable of ϕ through a heuristic procedure.
- ϕ_p ($\phi_{\neg p}$) is the formula obtained by applying the one-literal rule to ϕ using the literal p ($\neg p$).

State-of-the-art weighted Max-SAT solvers implement such a basic algorithm augmented with preprocessing techniques, the computation of an initial upper bound by a local search algorithm, clever variable selection heuristics, powerful inference techniques, lower bounds of good quality, and suitable data structures.

4 Lazy: a new weighted Max-SAT solver

`Lazy` implements the previous basic branch and bound algorithm augmented with a number of improvements that are described below: preprocessing techniques, inference methods, lower bound computation, variable selection heuristics, and data structures.

4.1 Preprocessing

Before starting to explore the search tree, `Lazy` obtains an upper bound on the sum of the weights of unsatisfied clauses in an optimal solution using a variant of the local search procedure GSAT [23]. This technique was first used by Borchers and Furman in their solver BF [4] and helps accelerate the search for an optimal solution.

Besides, `Lazy` simplifies the input formula by applying a novel resolution refinement: It replaces every pair of binary clauses $(p_1 \vee p_2, w_1)$ and $(\neg p_1 \vee p_2, w_2)$ with the clauses $(p_2, \min(w_1, w_2))$, $(p_1 \vee p_2, w_1 - \min(w_1, w_2))$, and $(\neg p_1 \vee p_2, w_2 - \min(w_1, w_2))$. That resolution refinement for unweighed Max-SAT is defined as follows: every pair of binary clauses $p_1 \vee p_2$ and $\neg p_1 \vee p_2$ can be replaced with the unit clause p_2. The advantage of that

preprocessing is that new unit clauses are generated. As we will show in the experimental investigation, it gives rise to substantial performance improvements.

4.2 Inference

When branching is done, branch and bound algorithms for Max-SAT apply the one-literal rule (simplifying with the branching literal) instead of applying unit propagation (i.e., the repeated application of the one-literal rule until a saturation state is reached) as in most SAT solvers. If unit propagation is applied at each node, the algorithm can return a non-optimal solution. For example, if we apply unit propagation to the unweighted clauses $p \wedge \neg q \wedge (\neg p \vee q) \wedge \neg p$ using the unit clause $\neg p$, we derive one empty clause while if we use the unit clause p, we derive two empty clauses. However, when the difference between the lower bound and the upper bound is one, unit propagation can be safely applied, because otherwise by fixing to false any literal of any unit clause we reach the upper bound. This technique, which `Lazy` incorporates, was developed by Borchers and Furman [4].

Lazy also incorporates the weighted complementary unit-clause (CUC) rule: The sum of the weights of unsatisfied clauses in an optimal solution of a formula $\phi \wedge (p, w_1) \wedge (\neg p, w_2)$ that contains two complementary unit clauses $((p; w_1)$ and $(\neg p; w_2))$ is equal to $\min(w_1, w_2)$ plus the sum of the weights of unsatisfied clauses in an optimal solution of the formula $\phi \wedge (p, w_1 - \min(w_1, w_2)) \wedge (\neg p, w_2 - \min(w_1, w_2))$.

The unweighted CUC rule was defined in Ref. [18]: The number of unsatisfied clauses in an optimal solution of a formula $\phi \wedge p \wedge \neg p$ is equal to 1 plus the number of unsatisfied clauses in an optimal solution of ϕ.

4.3 Lower bound computation

Wallace and Freuder [25] defined a lower bound computation method for Max-SAT that can be generalized to weighted Max-SAT as follows:

$$\texttt{LB}(\phi) = unsat(\phi) + \sum_{p \text{ occurs in } \phi} min(ic(p), ic(\neg p)), \quad (1)$$

where ϕ is the formula associated with the current partial assignment, $unsat(\phi)$ is the sum of the weights of the clauses unsatisfied by the current partial assignment, and $ic(p)$ $(ic(\neg p))$—inconsistency count of p $(\neg p)$—is the sum of the weights of the clauses that will become unsatisfied if the current partial assignment is extended by fixing p to true (false). Note that $ic(p)$ $(ic(\neg p))$ coincides with the sum of the weights of unit clauses of ϕ that contain $\neg p$ (p).

The lower bound of `Lazy`, LB_{Lazy}, is of better quality and can be understood as the weighted version of the lower bound of Wallace and Freuder (LB) extended with a new rule that we call star rule. We define the star rule for unweighted Max-SAT as follows: If a formula contains a clause of the form $l_1 \vee \cdots \vee l_k$, where $l_1, \ldots, l_k$ are literals, and k unit clauses of the form $\neg l_1, \ldots, \neg l_k$, then the lower bound can be incremented by one. In the weighted case, we only consider clauses of length two,[1] and define the rule as follows: If a formula contains a binary clause of the form $(l_1 \vee l_2, w_1)$ and two unit clauses of the form $(\neg l_1, w_2)$ and $(\neg l_2, w_3)$, then the lower bound can be incremented by $w = \min(w_1, w_2, w_3)$ and those clauses have to be replaced with $(l_1 \vee l_2, w_1 - w)$, $(\neg l_1, w_2 - w)$, and $(\neg l_2, w_3 - w)$.

[1] For longer clauses the star rule did not lead to performance improvements in our experimental investigation.

```
1: LB_Lazy(φ) := 0
2: for every clause (l1 ∨ l2, w1) ∈ φ do
3:   if (¬l1, w2) ∈ φ and (¬l2, w3) ∈ φ then
4:     w := min(w1, w2, w3)
5:     LB_Lazy(φ) := LB_Lazy(φ) + w
6:     φ := φ − {(l1 ∨ l2, w1), (¬l1, w2), (¬l2, w3)} ∪
7:            {(l1 ∨ l2, w1 − w), (¬l1, w2 − w), (¬l2, w3 − w)}
8:   end if
9: end for
10: w' := LB_Lazy(φ)
11: LB_Lazy(φ) := LB(φ) + w'
```

Fig. 2 Pseudo-code of the lower bound computation method of `Lazy`

The pseudo-code of $\mathrm{LB}_{\mathrm{Lazy}}$, for an input formula ϕ, is shown in Fig. 2. Note that after applying our variant of the star rule, `Lazy` applies the lower bound of Wallace and Freuder (LB) to the resulting formula (line 11 of the pseudo-code).

It is worth to mention that we took the name star rule from Ref. [18]. The star rule of Niedermeier and Rossmanith is not used to compute lower bounds. It simply states that the minimum number of unsatisfied clauses of the formula $\neg l_1 \wedge \cdots \wedge \neg l_k \wedge (l_1 \vee \cdots \vee l_k) \wedge (l_1 \vee \cdots \vee l_k)$ is 1.

4.4 Variable selection heuristic

The variable selection heuristic of `Lazy` is static, and computed before applying the preprocessing. It uses heuristic `MOMS` [19] adapted to weighted Max-SAT. `Lazy` orders the variables by the sum of the weights associated with the clauses of minimum size in which the variable appears. Variables are instantiated following that ordering.

4.5 Data structures

Existing Max-SAT and weighted Max-SAT solvers use adjacency lists to represent formulas, and their variable selection heuristics are typically dynamic. `Lazy` uses a static variable selection heuristic that allows us to implement extremely efficient data structures for representing and manipulating formulas. In this section, we first describe the data structures of a simpler solver. Based on that description, we then describe the data structures of `Lazy`.

If we would like to define data structures for the basic solver described in Sect. 3, incorporating the weighted version of the lower bound of Wallace and Freuder, we should take into account that the solver is only interested in knowing when a clause has become unit or empty. Thus, given a clause with four variables, it is not necessary to perform any operation in that clause until three of the variables have been instantiated; i.e., the evaluation of a clause with k variables can be delayed until $k - 1$ variables have been instantiated.

In that case, and using a static variable selection heuristic, we could define the following data structures: Clauses are ordered lists of literals (literals are ordered following the order used to instantiate variables) and there is a pointer to the penultimate literal and to the last literal of the clause. When a variable p is fixed to true (false), the clauses whose penultimate literal is $\neg p$ (p) are evaluated. If there is an instantiated literal in the clause which is true, the clause becomes satisfied; otherwise, a unit clause with the same weight, whose only literal is the last literal of the clause, is derived. This approach has two advantages: the cost of maintaining that data structure when the solvers backtracks is constant (we do not have to undo pointers like in adjacency lists) and, at each step, we evaluate a minimum number of

clauses (we do not evaluate all the clauses that contain the variable we are instantiating, we only evaluate the clauses in which the penultimate literal contains that variable). In addition, we also maintain an array that contains, for each literal, the sum of the weights of the unit clauses in which that literal appears. This array is used to derive empty clauses and to compute lower bounds. This array is the only data structure that the algorithm maintains when it backtracks.

`Lazy`, as considers the star rule in the computation of the lower bound, has pointers, besides to the last and penultimate literals, to the second from last literal of each clause. This way, it can maintain an array of binary clauses in order to compute the new lower bound efficiently.

5 Experimental results

We conducted an experimental investigation in order to compare the performance of `Lazy` with the following state-of-the-art solvers:

- `BF` [4]: It is a branch and bound weighted Max-SAT solver which uses MOMS as dynamic variable selection heuristic, and it does not consider any underestimation in the lower bound. Formulas are represented using adjacency lists. It was developed by Borchers and Furman in 1999.
- `AMP` [2]: It is a branch and bound Max-SAT solver based on `BF` that incorporates the lower bound of Wallace & Freuder, and uses the Jeroslow-Wang rule [12][2] as dynamic variable selection heuristic. It was developed by Alsinet, Manyà and Planes in 2003.
- `XZ` [26]: It is a branch and bound weighted Max-SAT solver developed by Xing and Zhang in 2004. We used the second release of this solver which is known as MaxSolver.
- `Toolbar` [8]: It is a weighted Max-SAT solver that encodes the input instance as a weighted constraint network and solves that network with a state-of-the-art weighted Max-CSP solver with a sophisticated and good performing lower bound. It was developed by Givry, Larrosa, Meseguer, and Schiex in 2003.
- `AGN` [1]: It is a branch and bound Max-2-SAT solver; the weighted version is not available. It was developed by Alber, Gramm, and Niedermeier in 1998.
- `ZSM` [28]: It is a branch and bound Max-2-SAT solver; the weighted version is not available. It was developed by Zhang, Shen, and Manyà in 2003.
- `CBC`: It is an open source branch and cut MIP solver developed within the COIN-OR project.[3] We used version 1.1.0.

As benchmarks we used randomly generated (weighted) Max-2-SAT and (weighted) Max-3-SAT instances, as well as all the weighted Max-SAT instances solved in the Max-SAT Evaluation 2006 [4] and the modified instances of the class *jnh* from the 2nd DIMACS implementation challenge [11] used in [20,21]. Unweighted random Max-2-SAT (Max-3-SAT) instances were generated using the method described in [17]. Weighted random Max-2-SAT (Max-3-SAT) instances were generated as unweighted random Max-2-SAT (Max-3-SAT) instances except that each clause was given a random integer weight uniformly distributed

[2] Given a formula ϕ, for each literal l of ϕ the following function is defined: $J(l) = {}_{l \in C \in \phi} 2^{-|C|}$, where $|C|$ is the length of clause C. It selects a variable p of ϕ among those that maximize $J(p) + J(\neg p)$.

[3] The COIN-OR (Computational Infrastructure for Operations Research) project is available at `http://www.coin-or.org/`

[4] `http://www.iiia.csic.es/~maxsat06/`

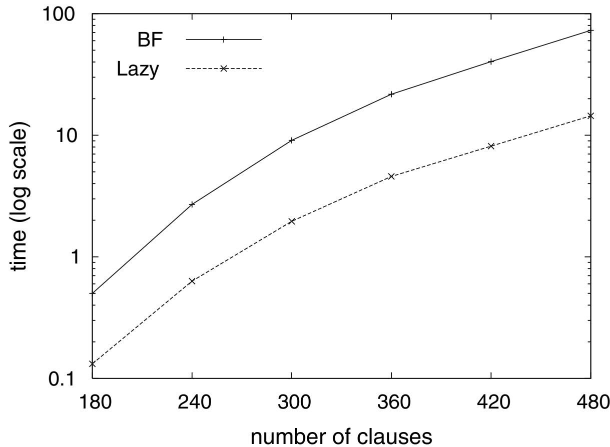

Fig. 3 Experimental results for 30-variable Max-3-SAT instances. Mean time (in seconds)

between one and ten. This generation method was used, among others, by Refs. [4,26]. The experiments were performed on a 2 GHz Pentium IV with 512 Mb of RAM under Linux.

In our first experiment, we evaluated the relevance of defining lazy data structures to get substantial performance improvements. To this end, we compared `BF` and `Lazy` using a simple variable selection heuristic: variables are instantiated in lexicographical order. Moreover, we removed all the improvements we introduced into `Lazy` and replaced $\mathrm{LB}_{\mathrm{Lazy}}$ with the lower bound of `BF`. This way, we have that `BF` and `Lazy` traverse the same search tree. Fig. 3 shows the results obtained when solving sets of randomly generated Max-3-SAT instances with 30 variables and a different number of clauses. We generated sets for 180, 240, 300, 360, 420, and 480 clauses, where each set had 500 instances. We observe that this modified version of `Lazy` is about five times faster than BF when both solvers traverse the same search tree.

In our second experiment, we generated sets of random Max-2-SAT instances with 50 variables and a different number of clauses. Each set had 500 instances. The results of solving such instances with CBC, BF, XZ, AMP, Toolbar, ZSM, AGN, and `Lazy` are shown in Fig. 4. Along the horizontal axis is the number of clauses, and along the vertical axis is the mean time (in seconds) needed to solve an instance of a set. Notice that we use a log scale to represent run-time. Observe that `Lazy` outperforms the rest of solvers in almost all the instances, even ZSM and AGN that are specifically designed to solve Max-2-SAT instances. Toolbar is very competitive and outperforms `Lazy` on large clauses/variables ratios. For XZ, we consider only formulas with less than 1,000 clauses because the available version of XZ does not deal with bigger formulas.

In our third experiment, we generated sets of random Max-3-SAT instances with 50 variables and a different number of clauses. Each set had 300 instances. The results of solving such instances with CBC, BF, AMP, Toolbar, ZX, and `Lazy` are shown in Fig. 5. We observe that `Lazy` outperforms the rest of solvers. The second best performing solver is Toolbar. CBC, BF, AMP, and XZ are much worse than the rest of solvers.

In our fourth experiment, we generated sets of random weighted Max-2-SAT and weighted Max-3-SAT instances with 50 variables and a different number of clauses. Each set had 500 instances. The results of solving such instances with CBC, BF, AMP, Toolbar, XZ, and `Lazy` are shown in Figs. 6 and 7. We observe that `Lazy` is the best performing solver for weighted

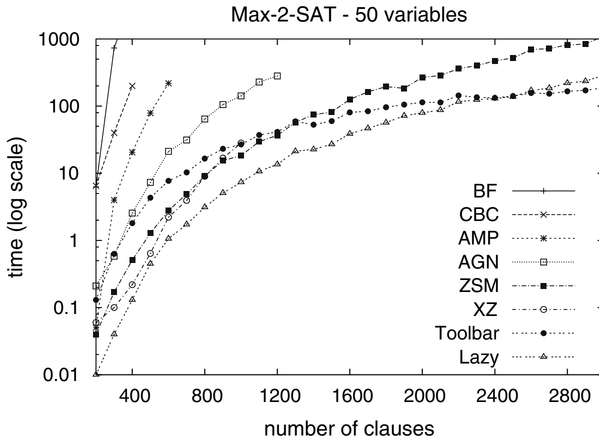

Fig. 4 Experimental results for 50-variable Max-2-SAT instances. Mean time (in seconds)

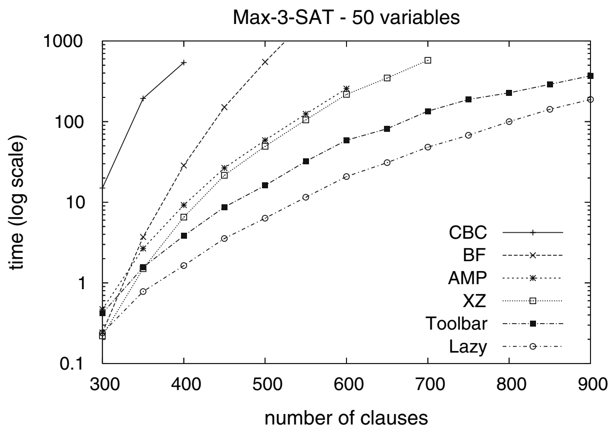

Fig. 5 Experimental results for 50-variable Max-3-SAT instances. Mean time (in seconds)

Max-2-SAT and weighted Max-3-SAT. Toolbar is competitive in both cases, while CBC, BF, and AMP are not competitive. It is worth mentioning that weighted Max-CSP has been intensively studied in the constraint programming community during the last decade, while DPLL-based solvers for weighted Max-SAT have only recently been investigated in the SAT community.

In our fifth experiment, we compared solvers CBC, BF, AMP, Toolbar, and Lazy on the instances from the Max-SAT Evaluation 2006 and the modified instances of the class *jnh*. This way, we evaluated the solvers on more structured instances. The results obtained are shown in Table 1, where we give, for each set of instances and for each solver, the mean time of the instances that were solved within 1,800 s, as well as the total number of solved instances (in brackets).[5] We observe that `Lazy` outperforms the rest of solvers on most of the sets of instances, providing empirical evidence that it also has a good performance profile on more structured instances.

[5] The set of instances not solved by any solver is not shown.

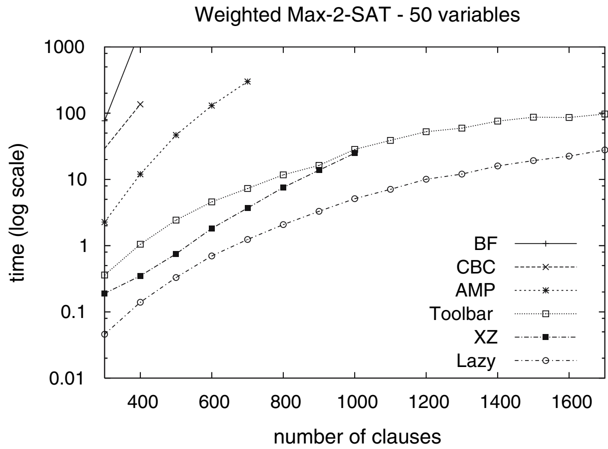

Fig. 6 Experimental results for 50-variable weighted Max-2-SAT instances. Mean time (in seconds)

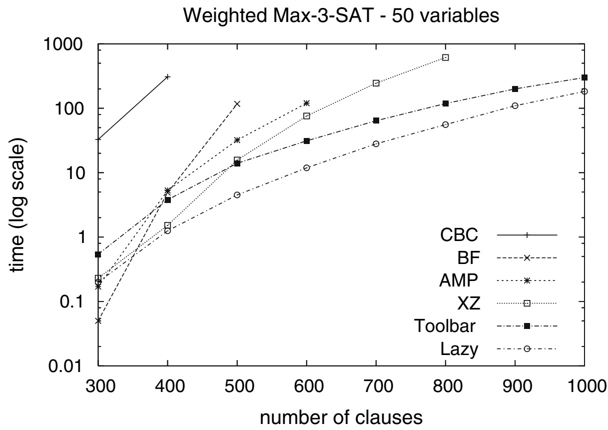

Fig. 7 Experimental results for 50-variable weighted Max-3-SAT instances. Mean time (in seconds)

In our sixth experiment, we analyzed the impact of the improvements we have incorporated into `Lazy` on sets of random weighted Max-2-SAT instances with 50 variables and a different number of clauses. Each set had 500 instances. The results obtained are shown in Figs. 8 and 9. Figure 8 shows the mean time needed to solve an instance while Fig. 9 shows the mean number of backtracks. *Basic* refers to the basic solver explained in Sect. 3 augmented with the weighted version of the lower bound of Wallace and Freuder and the computation of the initial upper bound with GSAT; *Lower bound* refers to *Basic* augmented with the lower bound LB_{Lazy}; *Resolution refinement* refers to *Basic* augmented with the preprocessing based on resolution; *Unit propagation* refers to *Basic* augmented with the safe application of unit propagation explained in Sect. 4.2; and *CUC* refers to *Basic* augmented with the weighted CUC rule.

From the results obtained we draw the following conclusions that help understand the good performance profile of `Lazy` :

Table 1 Experimental results for benchmarks in the MAX-SAT Evaluation 2006 for weighted instance

Set Name	#Instances	BF	AMP	XZ	CBC	Toolbar	Lazy
Auction (paths)	30	416.5(9)	244.3(9)	926.6(6)	**146.4(15)**	95.4(9)	261.3(15)
Auction (regions)	30	1.5(1)	0.8(1)	(0)	299.9(26)	**237.7(30)**	5.8(14)
Auction (schedulling)	30	10.7(11)	5.8(11)	(0)	544.2(8)	454.3(15)	**217.3(24)**
Max-Clique (brock)	12	(0)	(0)	(0)	717.4(1)	**475.7(3)**	589.4(2)
Max-Clique (c-fat)	7	(0)	(0)	(0)	(0)	**304.0(7)**	(0)
Max-Clique (hamming)	6	0.3(2)	0.1(2)	0.1(1)	(0)	39.2(4)	**27.5(4)**
Max-Clique (johnson)	4	51.7(3)	**28.4(3)**	0.2(2)	(0)	184.6(3)	458.9(3)
Max-Clique (keller)	2	(0)	(0)	(0)	(0)	**291.9(1)**	623.9(1)
Max-Clique (MANN a)	4	3.5(1)	2.1(1)	0.7(1)	(0)	**0.6(1)**	19.4(1)
Max-Clique (p hat)	12	(0)	(0)	(0)	(0)	**535.8(2)**	(0)
Max-Clique (sanr)	4	(0)	(0)	(0)	(0)	(0)	**1,561.0(1)**
Weighted Max-Cut (brock)	12	(0)	998.2(2)	1,157.0(5)	577.2(1)	260.2(12)	**222.0(12)**
Weighted Max-Cut (c-fat)	7	32.8(4)	196.6(5)	331.2(7)	123.0(6)	**225.3(7)**	289.8(7)
Weighted Max-Cut (hamming)	6	(0)	301.0(1)	**1,457.1(5)**	(0)	352.6(2)	523.8(2)
Weighted Max-Cut (johnson)	4	92.4(1)	264.5(2)	0.54 (1)	43.9(1)	20.3(2)	**15.0(2)**
Weighted Max-Cut (keller)	2	(0)	293.0(1)	(0)	(0)	294.1(2)	**233.8(2)**
Weighted Max-Cut (p hat)	12	528.4(3)	146.1(8)	9.7(8)	90.1(6)	287.2(12)	**139.5(12)**
Weighted Max-Cut (san)	11	(0)	742.8(2)	1,238.3(6)	952.5(2)	543.1(9)	**496.4(10)**
Weighted Max-Cut (sanr)	4	(0)	289.5(1)	54.2(1)	(0)	497.8(4)	**321.7(4)**
Weighted Max-Cut (random)	40	(0)	(0)	(0)	(0)	626.7(11)	**800.4(15)**
Weighted Max-Cut (spinglass)	5	(0)	(0)	611.5(3)	**302.0(3)**	2.1 (2)	3.1(2)
Max-One	45	1,042.2(1)	1,217.1(3)	117.8(4)	**112.2(39)**	276.3(4)	273.2(11)
Quasigroup Completion	25	2.3(10)	1.1(10)	(0)	**150.5(18)**	83.3(5)	1,385.9(2)
Ramsey	48	3.6(31)	1.9(31)	**46.5(35)**	(0)	49.0(29)	51.9(28)
Weighted CSP (DENSE LOOSE)	40	98.1(39)	**57.1(39)**	54.5(16)	378.6(29)	409.7(20)	638.1(2)
Weighted CSP (DENSE TIGHT)	60	(0)	(0)	1,619.4(10)	(0)	**501.4(23)**	(0)
Weighted CSP (SPARSE LOOSE)	40	57.9(40)	**32.7(40)**	2.1(11)	237.0(34)	393.4(24)	(0)
Weighted CSP (spot)	42	153.3(4)	73.6(4)	(0)	133.9(8)	**53.1(12)**	113.3(5)
JNH	44	**0.2(44)**	**0.2(44)**	0.4(42)	315.5(41)	6.1(44)	625.2(27)

- The safe application of unit propagation explained in Sect. 4.2 does not seem to be useful when a good quality lower bound is applied.
- The resolution refinement that we applied as preprocessing leads to significant improvements on both time and backtracks.
- The CUC rule accelerates the computation of the lower bound because it reduces the number of variables that the lower bound computation method must consider at each node of the search tree.
- The lower bound of `Lazy` gives rise to important performance improvements compared with the weighted version of the lower bound of Wallace and Freuder.

We believe that our results could be further improved by adapting the lazy data structures defined in the paper to deal with dynamic variable selection heuristics, as well as by applying more inference rules at each node of the search tree.

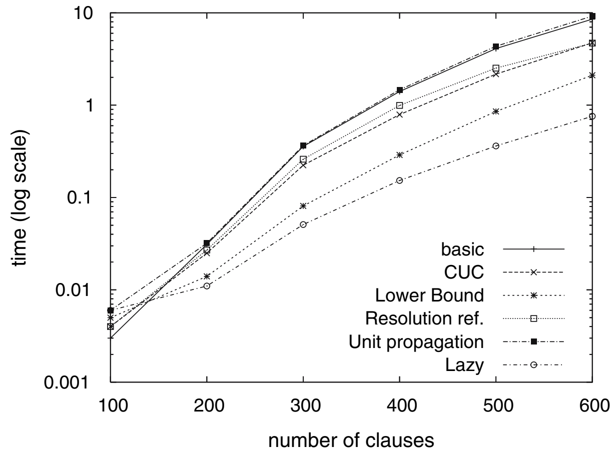

Fig. 8 Experimental results for 50-variable weighted Max-2-SAT instances. Mean time (in seconds)

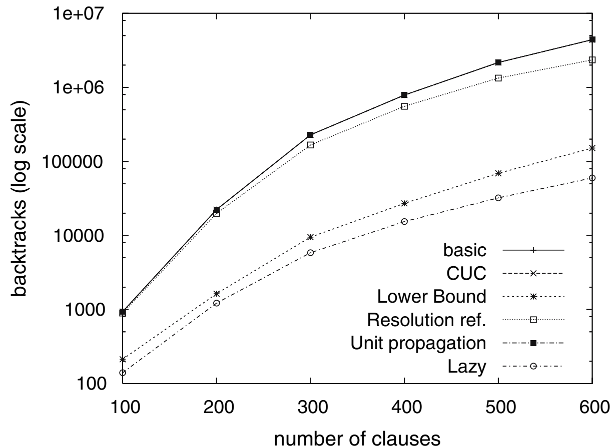

Fig. 9 Experimental results for 50-variable weighted Max-2-SAT instances. Mean number of backtracks

Acknowledgements Research partially supported by projects TIN2006-15662-C02-02 and TIN2004-07933-C03-03 funded by the *Ministerio de Ciencia y Tecnología. The second author was supported by a grant* Ramón y Cajal.

References

1. Alber, J., Gramm, J., Niedermeier, R.: Faster exact algorithms for hard problems: a parameterized point of view. In: Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics. LNCS, pp. 168–185. Springer, Berlin (1998)
2. Alsinet, T., Manyà, F., Planes, J.: Improved branch and bound algorithms for Max-SAT. In: Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing. S. Margherita Ligure – Portofino, Italy (2003).
3. Battiti, R., Protasi, M.: Reactive search, a history-sensitive heuristic for MAX-SAT. ACM J. Exp. Algorithms, **2**, Art. 2(1997)
4. Borchers, B., Furman, J.: A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. J. Combi. Optim **2**, 299–306 (1999)
5. Cheriyan, J., Cunningham, W., Tunçel, L., Wang, Y.: A linear programming and rounding approach to MAX-2-SAT. In: Johnson D., Trick M. (eds.), Cliques, Coloring and Satisfiability, Vol. **26** of DIMACS, pp. 395–414. American Mathematical Society, Providence, USA (1996)

6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**, 394–397 (1962)
7. Feige, U., Goemans, M.: Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT. In: Proceedings of the 3rd Israel Symposium on Theory of Computing and Systems, pp. 182–189. Tel Aviv, Israel (1995)
8. de Givry, S., Larrosa, J., Meseguer, P., Schiex, T.: Solving Max-SAT as weighted CSP. In: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP-2003, Kinsale, Ireland, LNCS 2833, pp. 363–376. Springer, Berlin (2003)
9. Goemans, M., Williamson, D.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. J. Assoc. Comput. Mach **42**(6), 1115–1145 (1995)
10. Hansen, P., Jaumard, B.: Algorithms for the maximum satisfiability problem. Computing **44**, 279–303 (1990)
11. Johnson, D., Trick, M. (eds): Cliques, coloring, and Satisfiability: second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, Providence, USA (1996)
12. Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1**, 167–187 (1990)
13. Joy, S., Mitchell, J., Borchers, B.: A branch and cut algorithm for MAX-SAT and weighted MAX-SAT. In: Proceedings of the DIMACS Workshop on Satisfiability: theory and Applications, Rutgers University, NJ, USA (1996)
14. de Klerk, E., Warners, J.P.: Semidefinite programming approaches for MAX-2-SAT and MAX-3-SAT: computational perspectives. Technical report, Delft, The Netherlands (1998)
15. Li, C.M., Anbulagan, A.: Look-ahead versus look-back for satisfiability problems. In: Proceedings of the 3rd International Conference on Principles of Constraint Programming, CP'97, Linz, Austria, LNCS 1330, pp. 341–355. Springer, Berlin (1997)
16. Loveland, D.W.: Automated Theorem Proving. A Logical Basis, volume 6 of Fundamental Studies in Computer Science. North-Holland, Amsterdam (1978)
17. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distributions of SAT problems. In: Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92, San Jose, CA, USA, pp. 459–465. AAAI Press (1992)
18. Niedermeier, R., Rossmanith, P.: New upper bounds for maximum satisfiability. J. Algorithms **36**, 63–88 (2000)
19. Pretolani, D.: Efficiency and stability of hypergraph SAT algorithms. In: Proceedings of the DIMACS Challenge II Workshop. Rutgers University, NJ, USA (1993)
20. Resende, M., Pitsoulis, L., Pardalos, P.: Approximate solutions of weighted MAX-SAT problems using GRASP. In: Du, D.-Z., Gu, J., Pardalos, P. (eds) Satisfiability Problem: theory and aplications, vol. 35 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 393–405. American Mathematical Society, Providence, USA (1997)
21. Resende, M., Pitsoulis, L., Pardalos, P.: FORTRAN subroutines for computing approximate solutions of weighted MAX-SAT problems using GRASP. Discrete Appl. Math. **100**(1.2), 95–113 (2000)
22. Selman, B., Kautz, H., Cohen, B.: Noise Strategies for Improving Local Search. In: Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94, Seattle, WA, USA, pp. 337–343. AAAI Press (1994)
23. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92, San Jose/CA, USA, pp. 440–446. AAAI Press (1992)
24. Stützle, T., Hoos, H., Roli, A.: A review of the literature on local search algorithms for MAX-SAT. Technical Report AIDA-01-02, FG Intellektik, FB Informatik, TU Darmstadt, Germany (2001)
25. Wallace, R., Freuder, E.: Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In: Johnson, D., Trick, M. (eds.) Cliques, Coloring and Satisfiability, vol. **26**, pp. 587–615. American Mathematical Society. Providence, USA (1996)
26. Xing, Z., Zhang, W.: Efficient strategies for (weighted) maximum satisfiability. In: Proceedings of CP-2004, pp. 690–705. Toronto, Canada (2004)
27. Zhang, H.: SATO: an efficient propositional prover. In: Proceedings of the Conference on Automated Deduction (CADE-97), pp. 272–275 (1997)
28. Zhang, H., Shen, H., Manyà, F.: Exact algorithms for MAX-SAT. Electron. Notes Theor. Comput. Sci. **86**(1) 190–203 (2003)
29. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the International Conference on Computer Aided Design, ICCAD-2001, San Jose/CA, USA, pp. 279–285 (2001)