Exact Algorithms for MAX-SAT

Hantao Zhang [1,2] Haiou Shen [3]

Computer Science Department
The University of Iowa
Iowa City, IA 52242, USA

Felip Manyà [4]

Department of Computer Science
Universitat de Lleida
Jaume II, 69, 25001-Lleida, Spain

Abstract

The maximum satisfiability problem (MAX-SAT) is stated as follows: Given a Boolean formula in CNF, find a truth assignment that satisfies the maximum possible number of its clauses. MAX-SAT is MAX-SNP-complete and received much attention recently. One of the challenges posed by Alber, Gramm and Niedermeier in a recent survey paper asks: Can MAX-SAT be solved in less than 2^n "steps"? Here, n is the number of different variables in the formula and a step may take polynomial time of the input. We answered this challenge positively by showing that a popular algorithm based on branch-and-bound is bounded by $O(b2^n)$ in time, where b is the maximum number of occurrences of any variable in the input.

When the input formula is in 2-CNF, that is, each clause has at most two literals, MAX-SAT becomes MAX-2-SAT and the decision version of MAX-2-SAT is still NP-complete. The best bound of the known algorithms for MAX-2-SAT is $O(m2^{m/5})$, where m is the number of clauses. We propose an efficient decision algorithm for MAX-2-SAT whose time complexity is bound by $O(n2^n)$. This result is substantially better than the previously known results. Experimental results also show that our algorithm outperforms any algorithm we know on MAX-2-SAT.

[1] Partially supported by the National Science Foundation under Grant CCR-0098093.
[2] Email: hzhang@cs.uiowa.edu
[3] Email: hshen@cs.uiowa.edu
[4] Email: felip@eup.udl.es

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

In recent years, there has been considerable interest in the satisfiability problem (SAT) and the maximum satisfiability problem (MAX-SAT) of propositional logic. The input instance is a boolean formula in conjunctive normal form (CNF), and the problem is to find a truth assignment that satisfies all the clauses for SAT and the maximum number of clauses for MAX-SAT. The decision version of MAX-SAT is NP-complete, even if the clauses have at most two literals (so called the MAX-2-SAT problem). One of the major results in theoretical computer science in recent years is that if there is a polynomial time approximation scheme for MAX-SAT, then P = NP [5].

Because the MAX-SAT problem is fundamental to many practical problems in computer science [14] and electrical engineering [23], efficient methods that can solve a large set of instances of MAX-SAT are eagerly sought. One important application of MAX-2-SAT is that NP-complete graph problems such as Maximum Cut and Independent Set can be reduced to special instances of MAX-2-SAT [8,18]. Many of the proposed methods for MAX-SAT are based on approximation algorithms [9], some of them are based on branch-and-bound methods [14,7,6,16,15,13,20], and some of them are based on transforming MAX-SAT into SAT [23,2].

Regarding the problems for formulas in CNF, most authors consider bounds with respect to three parameters: the length L of the input formula (i.e., the number of literals in the input), the number m of its clauses, and the number n of variables occurring in it. The best currently known bounds for SAT are $O(2^{m/3.23})$ and $O(2^{L/9.7})$ [16]. Nothing better than trivial $O(m2^n)$ is known when n is concerned. However, for 3-SAT, we have algorithms of complexity $O(m1.481^n)$ [9]. For MAX-SAT, the best bounds are $O(L2^{m/2.36})$ and $O(L2^{L/6.89})$ [6]. For MAX-2-SAT, the best bounds have been improved from $O(m2^{m/3.44})$ [6], to $O(m2^{m/2.88})$ [20], and recently to $O(m2^{m/5})$ [13]. It was posed as an open problem in [19,1,13] to seek exact algorithms bounded by $O(L2^n)$ for MAX-SAT.

Since an algorithm which enumerates all the 2^n assignments and then counts the number of true clauses in each assignment will take exactly time $O(L2^n)$, we assume that the challenge posed in [19,1,13] would ask for an algorithm better than $O(L2^n)$. In this paper, we present a simple algorithm based on branch-and-bound whose time complexity is only $O(b2^n)$, where b is the maximum number of occurrences of any variable in the input. Typically, $b \simeq L/n$. In particular, for MAX-2-SAT, $b \leq 2n$ and the bound becomes $O(n2^n)$. When $m = 4n^2$, the bound is $O(\sqrt{m}1.414^{\sqrt{m}})$, which is substantially better than the best known bound $O(m2^{m/5})$ [13].

Our branch-and-bound algorithm works in a similar way as the well-known Davis-Putnam-Logemann-Loveland (DPLL) procedure [10,11], which is a depth-first search procedure. To the best of our knowledge, there are only three implementations of exact algorithms for MAX-SAT that are variants of the

DPLL method. One is due to Wallace and Freuder (implemented in Lisp) [22], one is due to Borchers and Furman [7] (implemented in C and publicly available) and the last is due to Alsinet, Manyà and Planes [3] (a substantial improvement over Borchers and Furman's implementation). However, no attempts were made to analyze the complexities of the algorithms used in these implementations [22,7,3].

In this paper, we will present a rigorous analysis of exact algorithms for MAX-SAT and MAX-2-SAT. This kind of analysis, missing in [22,7,3], was used in [6,20,13] but the results presented there are not as strong as ours. It involves the design of data structures for clauses and the implementations of various operations on clauses. We also present experimental results to show that our algorithms are faster than other known algorithms not only in theory but also in practice.

2 Preliminary

We assume the reader is familiar with the basic concepts of the SAT problems, such as variable, literal, clause, CNF, assignment, and satisfiability. For every literal x, we use variable(x) to denote the variable appearing in x. That is, variable(x) $= x$ for positive literal x and variable($\overline{x}$) $= x$ for negative literal $\overline{x}$. If y is a literal, we use $\overline{y}$ to denote x if $y = \overline{x}$. A literal x in a set S of clauses is said to be *pure* in S if its complement does not appear in S. A partial (complete) assignment can be represented by a set of literals (or unit clauses) in which each variable appears at most (exactly) once and each literal is meant to be true in the assignment. If a variable x does not appear in a partial assignment A, then we say literals x and $\overline{x}$ are *unassigned* in A.

Given a set S of clauses and a literal x, we use $S[x]$ to denote the set of clauses obtained from S by removing the clauses containing x from S and removing $\overline{x}$ from the clauses in S. Given a set of literals $A = \{x_1, x_2, ..., x_k\}$, $S[A] = S[x_1][x_2]\cdots[x_n]$. Given a variable x, let $\#(x, S)$ be the number of clauses containing x (either positively or negatively).

3 A Branch-and-Bound Algorithm for MAX-SAT

Before presenting an algorithm for MAX-SAT, we make the following assumption. We regard the input clauses S_0 as a multiset. When some literals are assigned a truth value, this multiset is simplified by removing the assigned literals. If a unit clause is generated in this process, we remove this unit clause from the multiset and store this information in one variable, $u(x)$, associated with each literal x. That is, $u(x)$ records the numbers of unit clauses x generated during the search. If there are no unit clauses in the input, these variables are initialized to zero.

Our new algorithm for MAX-SAT is illustrated in Fig. 1. The variable min_false_clauses in the function branch_bound_max_sat and the recursive pro-

Fig. 1. A branch-and-bound algorithm for MAX-SAT.

```
function branch_bound_max_sat ( S: clause set ) return integer
    // initiation
    for each literal y do u(x) := 0; end for
    min_false_clauses := MAX_INT;
    bb_max_sat(S, 0, ∅);
    return min_false_clauses;
end function

procedure bb_max_sat ( S: clause set, k: integer, A: assignment )
    if |A| = n then // n is the number of variables.
        if (k < min_false_clauses) then
            print_model(A);
            min_false_clauses := k;
        end if
    else
        pick an unassigned literal x in S;
        // decide if we want to set literal x to false
        if (is_not_pure(x) ∧ u(x) + k < min_false_clauses) then
            S' := record_unit_clauses(S[x̄]);
            bb_max_sat(S', k + u(x), A ∪ {x̄});
            undo_record_unit_clauses(S[x̄]);
        end if
        // decide if we want to set literal x to true
        if (u(x̄) + k < min_false_clauses) then
            S' := record_unit_clauses(S[x]);
            bb_max_sat(S', k + u(x̄), A ∪ {x});
            undo_record_unit_clauses(S[x]);
        end if
    end if
end procedure

function record_unit_clauses ( S: clause set ) return clause set
    for each unit clause y ∈ S do S := S − {y}; u(y) := u(y) + 1; end for;
    return S;
end function

procedure undo_record_unit_clauses ( S: clause set )
    for each unit clause y ∈ S do S := S − {y}; u(y) := u(y) − 1; end for;
end procedure
```

cedure bb_max_sat is a global variable and is initialized with a maximum integer. In practice, this variable can be initialized with the minimum number of false clauses found by a local search procedure [7]. After the execution of bb_max_sat, min_false_clauses records the minimum number of false clauses under any assignment.

The parameter A in bb_max_sat(S, k, A) records a set of literals as a partial assignment (making every literal in A true). Note that A can be omitted if there is no need to print out an assignment whenever a better assignment is found. However, A is useful in the analysis of this algorithm.

The algorithm presented in Fig. 1 is certainly not the most efficient as many optimization techniques such as the pure-literal lookahead rule [21] or the *dominating unit-clause rule* [20] can be used. We made our best effort to present it as simple as possible so that it is easy to analyze.

Theorem 3.1 *Suppose min_false_clauses is initialized with a maximum integer. Then after the execution of* bb_max_sat$(S_0, 0, \emptyset)$*, min_false_clauses records the minimum number of false clauses in S_0 under any assignment.*

Proof. Let us consider the following pre-conditions of bb_max_sat(S, k, A).

- A is a partial assignment for variables appearing S_0.
- Let the multiset $S_0[A]$ be divided into empty clauses E, unit clauses U and non-unit clauses N. Then

(i) $S = N$;
(ii) $u(y)$ is the number of unit clauses y in U for any unassigned literal y under A;
(iii) $k = |E|$, the number of false clauses in S_0 under A.

At first, these conditions are true for the first call bb_max_sat$(S_0, 0, \emptyset)$, assuming neither unit clauses nor empty clauses are in S_0. Suppose these conditions are true for bb_max_sat(S, k, A). If we want to assign the literal x to true, then this action creates $u(\overline{x})$ empty clauses. If $u(\overline{x}) + k \geq$ min_false_clauses, then the current A will not lead to a better assignment. Otherwise, we will try to extend A further by assigning x to true and compute $S' = S[x]$. The procedure record_unit_clauses will update $u(y)$ for the newly created unit clauses for each literal y and remove these unit clauses from S'. It is easy to see that the pre-conditions for the call bb_max_sat$(S, k + u(\overline{x}), A \cup \{x\})$ are all true. Similarly, the pre-conditions for the call bb_max_sat$(S, k + u(x), A \cup \{\overline{x}\})$ are all true when we assign literal x to false. By an inductive reasoning, the pre-conditions are true for any multiset S. Finally when $|A|$ becomes empty, A is a complete assignment for S_0: If $k <$ min_false_clauses, then A is a better assignment and we update min_false_clauses by k accordingly.

The search conducted by bb_max_sat is an exhaustive one because every assignment is tried except the cases when we know the number of false clauses under that assignment exceeds min_false_clauses. This justifies the correctness of bb_max_sat(S, k, A). □

Besides the correctness of the algorithm, we show that the time complexity of the algorithm is bounded by $O(b2^n)$, where b is the maximum number of occurrences of any variable. For any variable x, recall that $\#(x, S)$ is the number of occurrences of x (both positively and negatively) in S.

Theorem 3.2 *If it takes $O(b)$ to* pick an unassigned literal x in S, *then the time complexity of branch_bound_max_sat(S_0) is $O(b2^n)$, where n is the number of variables in S_0 and $b = \max_x \#(x, S_0)$, the maximum number of occurrences of any variable in S_0.*

Proof. The number of calls to bb_max_sat is bounded by 2^n because the tree representing the relation between recursive calls of bb_max_sat is a binary tree (each internal node has at most two children) and the height of the tree is n.

We need to show that computing $S[x]$ and identifying new unit clauses in $S[x]$ can be done in $O(\#(x, S))$. To achieve this, we can use the data structure suggested by Freeman [12] as follows: For each clause $c \in S_0$, let count(c) be the number of unassigned literals in c and flag(c) be true if and only if one of the literals of c is assigned true. For each variable v, let pos(v) be the list of clauses in S_0 in which v appears positively and neg(v) be the list of clauses in which v appears negatively. To compute $S[x]$ from S, if x is positive, then for every clause c in pos(x), we assign true to flag(c) and for every clause c in neg(x) such that flag(c) is not true and count$(c) > 1$, we decrease count(c) by one. If count$(c) = 1$ after decreasing, we have obtained a new unit clause. The case when x is negative is similar. So the total cost for computing $S[x]$ and identifying new unit clauses is $O(\#(x, S_0))$.

In other words, record_unit_clauses and undo_record_unit_clauses can be done $O(\#(x, S_0))$. Since $\#(x, S_0)) \geq b$ and there are at most 2^n nodes, the total cost is bounded by $O(b2^n)$. □

Note that popular literal selection heuristics such as MOMS [12] and Jeroslow-Wang (JW) [17] take more than $O(b)$. MOMS is used by Borchers and Furman [7] and JW is used by Alsinet et al. [3] for MAX-SAT. In the next section, we will present an efficient decision algorithm for MAX-2-SAT in which it takes a constant time to select literals.

4 An Efficient Decision Algorithm for MAX-2-SAT

The decision version of MAX-SAT takes the following form:

Instance: A formula S in CNF and a nonnegative integer g.

Question: Is there a truth assignment for the variables in S such that at most g clauses in S are false under this assignment?

It is well-known that if the decision version of MAX-SAT can be solved in time T, then the optimization version of MAX-SAT can be solved in time $lg(m)T$, where m is the number of input clauses. For MAX-2-SAT, $m \leq 4n^2$ if no duplicate clauses are allowed.

Before presenting the algorithm for MAX-2-SAT, we define the following data structure for binary clauses. We assume that the n propositional variables are named (and ordered in the obvious way) from 1 to n. For each variable i, we define the following two sets:

$$B_0(i) = \{y \mid (\bar{i} \vee y) \in S, i < \text{variable}(y)\}$$
$$B_1(i) = \{y \mid (i \vee y) \in S, i < \text{variable}(y)\}$$

Intuitively, $B_0(i)$ is an economic representation of $\text{neg}(i)$ and $B_1(i)$ is an economic representation of $\text{pos}(i)$. The decision algorithm for MAX-2-SAT is illustrated in Fig. 2.

Theorem 4.1 *Suppose S_0 is a set of binary clauses. Then max_2_sat2(S_0, n, g_0) returns true if and only if there exists an assignment under which at most g_0 clauses in S are false.*

Proof. The proof is analogous to that of Theorem 3.1. The pre-conditions considered here for dec_max_2_sat(i, g, A) are the following.

- A is a partial assignment for variables 1 to $i-1$.
- g is equal to g_0 minus the number of false clauses in S under A, where g_0 is the parameter in the decision problem and the first call to dec_max_2_sat.
- For any literal y, $i \leq \text{variable}(y) \leq n$, $u(y)$ is the number of unit clauses y in $S_0[A]$.

An inductive proof on i will prove these pre-conditions. □

Theorem 4.2 *The time complexity of dec_max_sat(S_0, n, g_0) is $O(n2^n)$ and the space complexity is $L/2 + O(n)$, where L is the size of the input.*

Proof. The time complexity is analogous to that of Theorem 3.2. Since for MAX-2-SAT, the maximum number of occurrences of any variable is bound by $O(n)$ and the algorithm takes constant time on literal selection, the total time is bounded by $O(n2^n)$.

For the space complexity, since only one literal in each binary clause is stored in the algorithm, we need $L/2$ to store the input. Adding the space for u and local variables in recursive calls gives us the result. □

To the best of our knowledge, the space complexity of other algorithms is bounded by cL, where $c > 1$, for the algorithms in [22,7,3].

5 Experimental Results

To obtain an efficient decision procedure, we have considered several techniques. One such technique is the so-called pure-literal deletion to prune some futile branches. A literal is said to be *pure* in the current clause set if its negation does not occur in the clause set. There are no need to assign false to

Fig. 2. A decision algorithm for MAX-2-SAT.

```
function max_2_sat2 ( S_0: clause set, n: variable, g_0: integer) return boolean
    // initiation
    for i := 1 to n
        compute B_0(i) and B_1(i) from S_0;
        u(i) := u_i(ī) := 0; // assuming no unit clauses in S_0
    end for
    return dec_max_2_sat(1, g_0, ∅);
end function

function dec_max_2_sat( i: variable, g: integer, A: assignment ) return Boolean
    if i > n then print_model(A); return true; end if // end of the search tree
    // decide if we want to set variable i to true
    if (u(ī) ≤ g) then
        record_unit_clauses(i, 0);
        if (dec_max_2_sat(i + 1, g − u(ī), A ∪ {i}) then return true; end if
        undo_record_unit_clauses(i, 0);
    end if
    // decide if we want to set variable i to false
    if (u(i) ≤ g) then
        record_unit_clauses(i, 1);
        if (dec_max_2_sat(i + 1, g − u(i), A ∪ {ī}) then return true; end if
        undo_record_unit_clauses(i, 1);
    end if
    return false;
end function

procedure record_unit_clauses ( i: variable, s: boolean )
    for y ∈ B_s(i) do u(y) := u(y) + 1 end for;
end procedure

procedure undo_record_unit_clauses ( i: variable, s: boolean )
    for y ∈ B_s(i) do u(y) := u(y) − 1 end for;
end procedure
```

a pure literal because doing so will not find an assignment which makes less clauses false.

For the data structure used in our algorithm, let us assume $b_0(i) = |B_0(i)|$ and $b_1(i) = |B_1(i)|$. Then a positive literal i is pure if $u(\bar{i}) + b_0(i) = 0$. Similarly, a negative literal $\bar{i}$ is pure if $u(i) + b_1(i) = 0$. We can easily check this condition before branching.

In [20], Niedermeier and Rossmanith used a rule called the *dominating unit-clause rule* to prune some search space. The dominating unit-clause rule can be easily checked using our data structure: There is no need to assign

Fig. 3. Modification to function dec_max_2_sat.

```
function dec_max_2_sat( i: variable, g: integer ) return Boolean
    if i > n then return true; // end of the search tree
    if (∑_{j=i}^{n} min(u(j̄), u(j)) > g) then return false end if
    // decide if we want to set variable i to true
    if (u(ī) ≤ g ∧ u(ī) < u(i) + b_1(i)) then
        record_unit_clauses(i, 0);
        if (dec_max_2_sat(i + 1, g − u(ī))) then return true; end if
        undo_record_unit_clauses(i, 0);
    end if
    // decide if we want to set variable i to false
    if (u(i) ≤ g ∧ u(i) ≤ u(ī) + b_0(i)) then
        record_unit_clauses(i, 1);
        if (dec_max_2_sat(i + 1, g − u(i))) then return true; end if
        undo_record_unit_clauses(i, 1);
    end if
    return false;
end function
```

variable i false (true) if $u(i) \geq u(\bar{i}) + b_0(i)$ ($u(\bar{i}) > u(i) + b_1(i)$). This rule covers the pure-literal checking because if literal i is pure, then $u(\bar{i}) + b_0(i) = 0$, hence $u(i) \geq u(\bar{i}) + b_0(i)$.

In dec_max_2_sat(i, g, A), if $(\sum_{j=i}^{n} min(u_i(\bar{j}), u(j))) > g$, then there is no solution (dec_max_2_sat(i, g, A) will return false). We found later that this technique of pruning is also used in [3,4] (named LB2) and is crucial to the high performance of their implementation.

Combining the ideas in the above discussion, we present in Fig. 3 a modified dec_max_2_sat(i, k, A). The same ideas apply to bb_max_sat(i, k, A) in Fig. 1 as well.

We have implemented the algorithm presented in Fig. 3 in C++ and preliminary experimental results seem promising. Table 1 shows some results of Borchers and Furman's program (BF) [7], Alsinet et al.'s (AMP, the option LB2-I+JW is used), and our implementation (New) on the problems of 50 variables distributed by Borchers and Furman. Note that min_false_clauses in our algorithm is at first set to the number found by the first phase of Borchers and Furman's local search procedure and then decreased by one until the optimal value is decided.

Problems p100-p500 have 50 variables and problems p2200-p2400 have 100 variables. Times (in seconds) are collected on a Pentium 4 linux machine with 256Mb memory. "–" indicates an incomplete run after running for two hours. It is clear that our algorithm runs consistently faster than both Borchers and Furman's program and Alsinet et al.'s modification. We have also generated several thousands of random instances of MAX-2-SAT and the results remains the same. Some of the experimental results are depicted in Fig. 4.

Table 1
Experimental results on Borchers and Furman's examples.

Problem	false clauses	BF	AMP	New
p100	4	0.035	0.03	0.02
p150	8	0.091	0.04	0.02
p200	16	6.425	0.51	0.12
p250	22	37	0.36	0.05
p300	32	530	6.53	0.85
p350	41	3866	14	1.45
p400	45	3467	6.05	0.54
p450	63	–	86	4.68
p500	66	–	25	1.78
p2200	5	0.191	0.18	0.53
p2300	15	763	41	7.67
p2400	29	–	1404	172

Fig. 4. Running time for BF [7], AMP [4], and our new algorithm (New). We considered the following cases: $n = 50$ variables and $m = cn$ clauses, where $c = 1.6, 1.7, ..., 5.9, 6$. For each case, we generated 100 random problems. The total run time of 100 instances is depicted (excluding the time reading the input from the file).

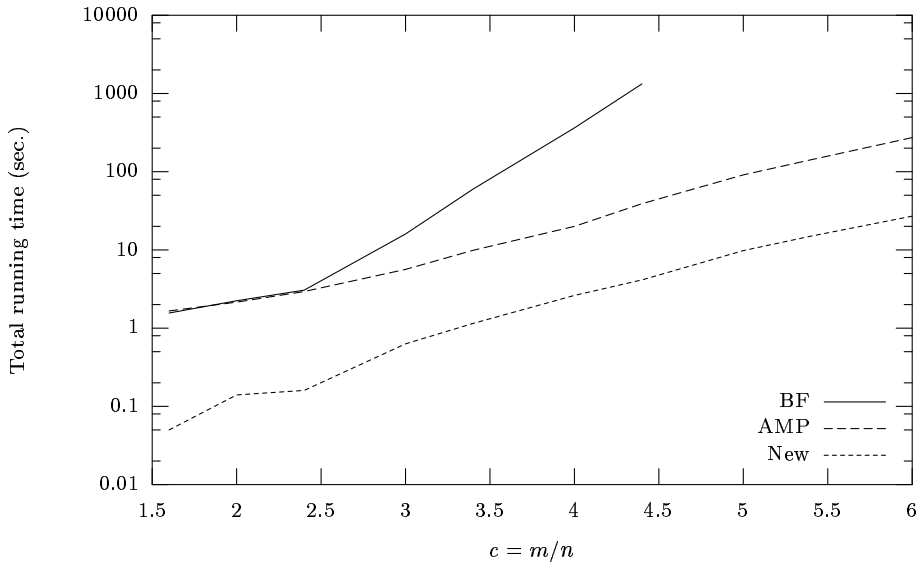
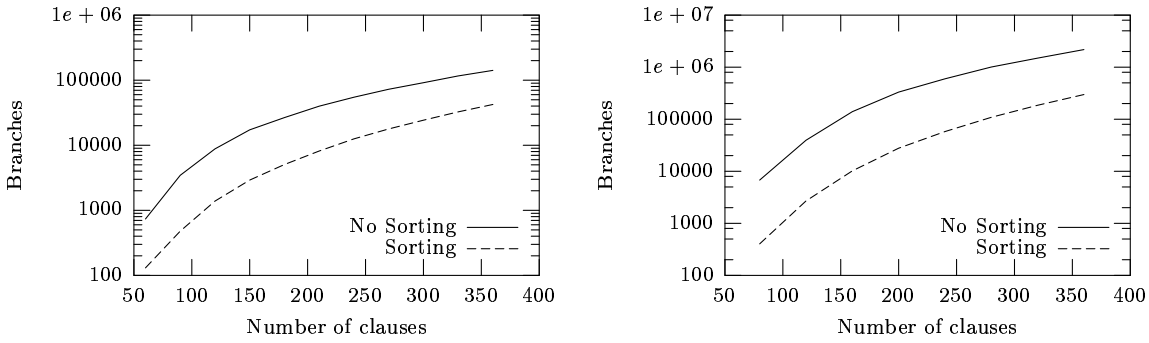


Fig. 5. Run time comparison: no sorting vs. sorting at pre-processing.



(a) $n = 30$ **(b)** $n = 40$

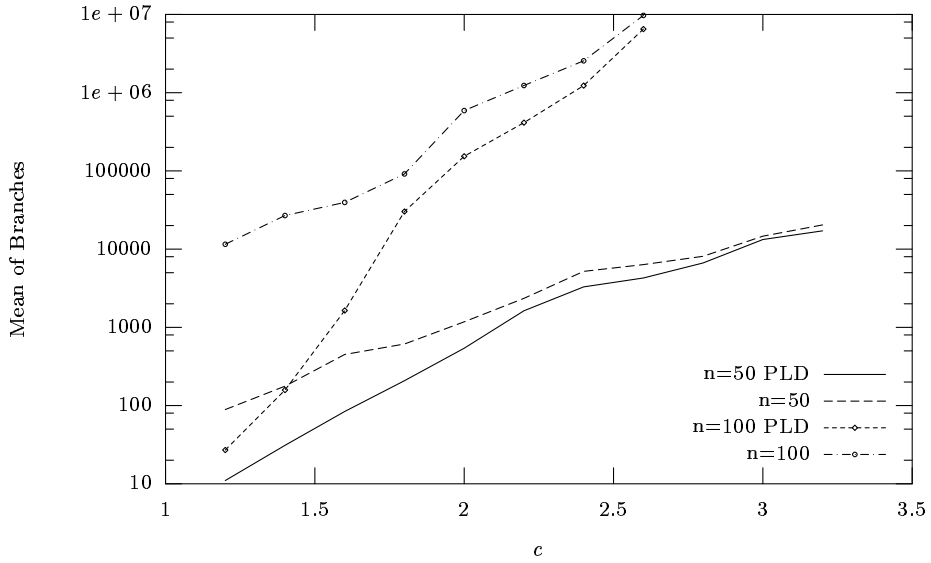
Note that our algorithm takes a fixed order, i.e., from 1 to n, to assign truth value to variables. We found that it is helpful to sort the variables first according their occurrences in the input in non-increasing order and then assign variables in that order. Fig 5 shows the impact of sorting n variables for $n = 30, 40$ variables and $m = 80, ..., 360$ clauses.

Another pre-processing technique we found useful is to delete pure literals from the input when $c = m/n \leq 3$, where n is the number of variables and m is the number of clauses. Fig. 5 shows the impact of using pure literal deletion (PLD) at pre-processing when $n = 50, 100$. After removing pure literals from the input, the remaining variables are sorted according to their occurrences. This ordering of the variables is then used in the algorithm in Fig. 3. From the figure, we see clearly that the smaller the value of c, the more the size of the search tree (number of branches) is reduced. The running time, which is proportional to the size of the search tree, is also reduced. This figure also shows the importance of literal selection heuristics. In [3,4], it shown that MOM and JW work well in some cases of MAX-SAT. We plan to further investigate some effective, easy to compute heuristics.

6 Conclusion

We have analyzed a branch-and-bound algorithm for MAX-SAT and showed that the complexity of this algorithm is substantially better than the known results. We also presented an efficient decision algorithm for MAX-2-SAT and showed that it is fast both in theory and in practice. The high performance of our algorithm for MAX-2-SAT may be due to the fact that we have special data structure for binary clauses. As future work, we will implement the algorithm in Fig. 1 for general MAX-SAT (and weighted MAX-SAT [7]) and compare our implementation with [7,3]. We will also use them to study properties of MAX-SAT.

Fig. 6. Computing cost for our decision algorithm with and without pure literal deletion at pre-processing. We generated 100 random problems for each case. The computing cost is the mean of branches of the search tree.



References

[1] J. Alber, J. Gramm, R. Niedermeier, Faster exact algorithms for hard problems: A parameterized point of view. Preprint, submitted to Elsevier, August, 2000

[2] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, Generic ILP versus specialized 0-1 ILP: An update. Technical Report CSE-TR-461-02, University of Michigan, Ann Arbor, Michigan, Aug., 2002.

[3] T. Alsinet, F. Manyà, J. Planes, Improved branch and bound algorithms for Max-SAT. Proc. of 6th International Conference on the Theory and Applications of Satisfiability Testing, SAT2003, pages 408-415.

[4] T. Alsinet, F. Manyà, J. Planes, Improved branch and bound algorithms for Max-2-SAT and Weighted Max-2-SAT. Submitted.

[5] S. Arora, C. Lund. Hardness of approximation. In D. Hochbaum (ed.): Approximation algorithms for NP-hard problems, Chapter 10, pages 399-446. PWS Publishing Company, Boston, 1997.

[6] N. Bansal, V. Raman, Upper bounds for MaxSat: Further improved. In Aggarwal and Rangan (eds.): *Proceedings of 10th Annual conference on Algorithms and Computation*, ISSAC'99, volume 1741 of Lecture Notes in Computer Science, pages 247-258, Springer-Verlag, 1999.

[7] B. Borchers, J. Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2(4):299-306, 1999.

[8] J. Cheriyan, W.H. Cunningnham, L. Tuncel, Y. Wang. A linear programming and rounding approach to Max 2-Sat. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:395–414, 1996.

[9] E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k–SAT based on local search. *Theoretical Computer Science*, 2002.

[10] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery 7,* 3 (July 1960), 201–215.

[11] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 7 (July 1962), 394–397.

[12] J.W. Freeman, Improvements to propositional satisfiability search algorithms. Ph.D. Dissertation, Dept. of Computer Science, University of Pennsylvania, 1995.

[13] J. Gramm, E.A. Hirsch, R. Niedermeier, P. Rossmanith: New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. Preprint, submitted to Elsevier, May, 2001

[14] P. Hansen, B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279-303, 1990.

[15] E.A. Hirsch. A new algorithm for MAX-2-SAT. In *Proceedings of 17th International Symposium on Theoretical Aspects of Computer Science*, STACS 2000, vol. 1770, Lecture Notes in Computer Science, pages 65-73. Springer-Verlag.

[16] E.A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397-420, 2000.

[17] R.G. Jeroslow, J. Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1:167-187, 1990.

[18] M. Mahajan, V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999.

[19] R. Niedermeier, Some prospects for efficient fixed parameter algorithms. In *Proc. of the 25th Conference on Current Trends in Theory and Practice of Informatics* (SOFSEM'98), Springer, LNCS 1521, pages 168–185, November, 1998.

[20] R. Niedermeier, P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63-88, 2000.

[21] I. Schiermeyer. Pure literal look ahead: An $O(1,497^n)$ 3-satisfiability algorithm. In Franco et al (eds.) Proceedings of *Workshop on the Satisfiability Problem*, Report No. 96-230, Universität zu Köln, pages 127-136, Siena, April 1996.

[22] R. Wallace, E. Freuder. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In D. Johnson and M. Trick (eds.) *Cliques, Coloring and Satisfiability*, volume 26, pages 587-615, 1996.

[23] H. Xu, R.A. Rutenbar, K. Sakallah, sub-SAT: A formulation for related boolean satisfiability with applications in routing. ISPD'02, April, 2002, San Diego, CA.