

MAX-SAT for Formulas with Constant Clause Density Can Be Solved Faster Than in $\mathcal{O}(2^n)$ Time

Evgeny Dantsin and Alexander Wolpert

Roosevelt University, 430 S. Michigan Av., Chicago, IL 60605, USA
{edantsin, awolpert}@roosevelt.edu

Abstract. We give an exact deterministic algorithm for MAX-SAT. On input CNF formulas with constant clause density (the ratio of the number of clauses to the number of variables is a constant), this algorithm runs in $\mathcal{O}(c^n)$ time where $c < 2$ and n is the number of variables. Worst-case upper bounds for MAX-SAT less than $\mathcal{O}(2^n)$ were previously known only for k -CNF formulas and for CNF formulas with small clause density.

1 Introduction

When solving MAX-SAT, we can search for an approximate solution or we can search for an exact solution. Approximate algorithms are typically faster than exact ones. In particular, there are polynomial-time approximate algorithms for MAX-SAT, but they are limited by thresholds on approximation ratios (some of these algorithms achieve the thresholds, e.g. [9]). There are also exponential-time algorithms that give arbitrarily good approximation, but when a high precision is required they are not considerably faster than exact algorithms, see e.g. [6,8].

In this paper we deal with exact algorithms for MAX-SAT and worst-case upper bounds on their runtime. Beginning in the early 1980s [11], many such upper bounds were obtained, and now we have a spectrum of MAX-SAT upper bounds for various classes of input formulas such as 2-CNF, 3-CNF, k -CNF, formulas with constant clause density, etc. The upper bounds usually depend on the number of variables, the number of clauses, the maximum number of satisfiable clauses, etc. Majority of exact MAX-SAT algorithms use the DPLL approach, but other approaches are used too, for example local search.

We give exact deterministic algorithms that solve MAX-SAT for formulas with no restriction on clause length (the general case of MAX-SAT) and prove worst-case upper bounds on their runtime. Previously known upper bounds for this general case were obtained in [4,10,2,12,14]. The most recent “record” bounds are given in [5]:

$$\begin{aligned} &\mathcal{O}(1.3247^m \cdot |F|) \\ &\mathcal{O}(1.3695^K + |F|) \end{aligned}$$

where $|F|$ is the size of input formula F , m is the number of clauses in F , and K is the maximum number of satisfiable clauses (or the input parameter of

the decision version of MAX-SAT). Note that these bounds are better than the trivial upper bound $\mathcal{O}(2^n \cdot |F|)$, where n is the number of variables, only for small values of the clause density ($m/n < 3$). The algorithms in [5] use DPLL techniques; the proof of the bounds is based on case analysis and recurrence relations.

The contribution of this paper is twofold:

- Our bounds for MAX-SAT are better than $\mathcal{O}(2^n)$ for formulas with constant clause density, i.e., with $m/n < c$ where c is an arbitrary constant. Moreover, they are better than $\mathcal{O}(2^n)$ even if m/n is a slowly growing function.
- Our algorithms are based on a new method of solving MAX-SAT: we search for a partial assignment that satisfies a sufficient number of clauses and then we try to extend this assignment to satisfy more clauses.

Note that our method uses memoization which often occurs in recent algorithms for SAT and MAX-SAT, e.g. [3] and [13]. In particular, the randomized algorithm in [13] solves MAX- k SAT using DPLL combined with memoization. Its upper bound on the expected runtime is less than $\mathcal{O}(2^n)$ for k -CNF formulas with constant clause density.

Structure of the paper. Basic definitions and notation are given in Sect. 2. In Sect. 3 we discuss the idea of our algorithms and prove key lemmas. We describe the main algorithm in Sect. 4 and we prove an upper bound on its runtime in Sect. 5. In Sect. 6 we describe a modification of the main algorithm for the case when K is not too large compared to m . Sect. 7 summarizes our results.

2 Definitions and Notation

We deal with Boolean formulas in conjunctive normal form (CNF formulas). By a *variable* we mean a Boolean variable that takes truth values **true** or **false**. A *literal* is a variable x or its negation $\neg x$. A *clause* C is a set of literals such that C contains no complementary literals. When dealing with the satisfiability problem, a formula is typically viewed as a set of clauses. In the contexts of (weighted) MAX-SAT, it is more natural to view formulas as multisets. Therefore, we define a *formula* to be a multiset of clauses.

For a formula F , we write $|F|$ to denote the number of clauses in this multiset (we use letter m for this number). We use letters V and n to denote, respectively, the set of variables in F and its cardinality. The *clause density* of F is the ratio m/n . For any positive number Δ , we write $\mathcal{F}(\Delta)$ to denote the set of formulas such that their clause density is at most Δ .

We assign truth values to all or some variables in F . Let U be a subset of the variables of F . An *assignment* to the variables in U is a mapping from U to $\{\text{true}, \text{false}\}$. This mapping is extended to literals: each literal $\neg x$ is mapped to the complement of the truth value assigned to x . We say that a clause C is *satisfied* by an assignment a if a assigns **true** to at least one literal in C . The formula F is *satisfied* by A if every clause in F is satisfied by A .

For assignment a , we write $F(a)$ to denote the formula obtained from F as follows: any clause that contains a true literal is removed from F , all false literals are deleted from the other clauses. The empty clause is false; the empty formula is true.

The MAX-SAT problem is stated as follows: Given a formula, find an assignment that maximizes the number of satisfied clauses. We also consider the decision version of MAX-SAT: given a formula F and an integer K , is there an assignment that satisfies at least K clauses?

We use \mathcal{O}^* notation that extends big-Oh notation and suppresses all polynomial factors. For functions $f(n_1, \dots, n_k)$ and $g(n_1, \dots, n_k)$ we write

$$f(n_1, \dots, n_k) = \mathcal{O}^*(g(n_1, \dots, n_k))$$

if for some multivariate polynomial p , we have

$$f(n_1, \dots, n_k) = \mathcal{O}(p(n_1, \dots, n_k) \cdot g(n_1, \dots, n_k)).$$

We write $\log x$ to denote $\log_2 x$. The entropy function is denoted by H :

$$H(x) = -x \log x - (1 - x) \log(1 - x).$$

3 Idea of Algorithms and Key Lemmas

In the next sections we describe and analyze two algorithms for MAX-SAT. Both are based on an idea outlined in this section. This idea is suggested by Lemma 2 and Lemma 3. The former is a key observation behind our approach. Its weaker version was used by Arvind and Schuler in [1] for their quantum satisfiability-testing algorithm. The second lemma is a well known fact that any formula has an assignment satisfying at least a half of its clauses.

We consider the decision version of MAX-SAT. To solve this problem, we enumerate partial assignments and try to extend them to satisfy at least K clauses.

1. **Enumeration of partial assignments.** We select a set of partial assignments such that for some δ
 - each of them assigns truth values to δn variables;
 - each of them satisfies at least δK clauses.

It follows from Lemma 2 that this set contains an assignment a that can be extended to satisfy at least K clauses (if K clauses in F can be satisfied at all). Lemma 2 also shows that this set (containing a) can be constructed by processing $\mathcal{O}(2^{\delta n})$ partial assignments.

2. **Extension of partial assignments.** When processing a partial assignment, we try to extend it to satisfy more clauses in F . There are at most $m - \delta K$ unsatisfied clauses left and we need to satisfy at least $(1 - \delta)K$ of them. We consider two possibilities:
 - *There are “many” unsatisfied clauses.* Then $(1 - \delta)K$ clauses are satisfied due to the fact that any formula has an assignment satisfying at least a half of its clauses.

where $d \geq 2$ and $V_i \cap V_j = \emptyset$ for all i and j . For each i from 1 to d , let A_i denote the restriction of A to V_i and let F_i be the multiset of clauses satisfied by A_i , i.e.,

$$F_i = \{C \in F \mid C \text{ is satisfied by } A_i\}$$

Then there exists $i \in \{1, \dots, d\}$ such that

$$|F_1 \cup \dots \cup F_{i-1} \cup F_{i+1} \cup \dots \cup F_d| \geq (1 - 1/d)K$$

Proof. We prove this lemma using Lemma 1. Let S denote the multiset of all clauses in F satisfied by A . This multiset can be represented as the union of d multisets:

$$S = S_1 \cup \dots \cup S_d$$

where $S_j = F_j - (F_1 \cup \dots \cup F_{j-1})$. Since $|S| \geq K$, we have $|S_1| + \dots + |S_d| \geq K$. Applying Lemma 1, we obtain that there exists i such that

$$|S_1| + \dots + |S_{i-1}| + |S_{i+1}| + \dots + |S_d| \geq (1 - 1/d)K. \quad (3)$$

The multisets S_1, \dots, S_d are pairwise disjoint, therefore

$$|S_i| + \dots + |S_{i-1}| + |S_{i+1}| + \dots + |S_d| = |S_i \cup \dots \cup S_{i-1} \cup S_{i+1} \cup \dots \cup S_d|. \quad (4)$$

By the definition of multisets S_j , we have $S_j \subseteq F_j$ for every j and consequently

$$S_1 \cup \dots \cup S_{i-1} \cup S_{i+1} \cup \dots \cup S_d \subseteq F_1 \cup \dots \cup F_{i-1} \cup F_{i+1} \cup \dots \cup F_d. \quad (5)$$

Taking (3), (4), and (5) together we get

$$\begin{aligned} |F_1 \cup \dots \cup F_{i-1} \cup F_{i+1} \cup \dots \cup F_d| &\geq \\ |S_i \cup \dots \cup S_{i-1} \cup S_{i+1} \cup \dots \cup S_d| &= \\ |S_i| + \dots + |S_{i-1}| + |S_{i+1}| + \dots + |S_d| &\geq \\ (1 - 1/d)K. &\quad \square \end{aligned}$$

Lemma 3 (well known fact). *Any formula F has an assignment that satisfies at least a half of its clauses.*

Proof. Consider an arbitrary assignment a . If a satisfies less than a half of clauses in F , we flip the values of the variables in a . It is easy to see that the new assignment satisfies at least a half of clauses in F . \square

4 Main Algorithm

In this section we describe the main algorithm based on the approach discussed informally in Sect. 3.

Algorithm \mathcal{A}

Input: Formula F with m clauses over n variables, integers K and d such that $2 \leq d \leq n$ and $m/2 \leq K \leq m$.

Output: “yes” if there is an assignment that satisfies at least K clauses, “no” otherwise.

1. **Partition.** We partition the set V of variables in F into d subsets V_1, \dots, V_d of (approximately) same size. More exactly, we partition V into d subsets such that each of them has either $\lfloor n/d \rfloor$ or $\lceil n/d \rceil$ elements.
2. **Database preparation.** We prepare a “database” needed for the next steps. Namely, we build tables (arrays) T_1, \dots, T_d defined as follows:
 - (a) Let $F|_{V_j}$ denote the formula obtained from F by removing all clauses that contain no variables from V_j and deleting variables from $V - V_j$ in the remaining clauses. By a *short subformula* of $F|_{V_j}$ we mean any subformula of $F|_{V_j}$ that has length at most $2K/d$.
 - (b) We define T_j to be a table consisting of all pairs (S, M) where S is a short subformula of $F|_{V_j}$ and M is the maximum number of satisfiable clauses in S . For every S , we compute M by brute-force, which takes time $\mathcal{O}^*(2^{n/d})$.
 - (c) Fixing an order on short subformulas of $F|_{V_j}$, we sort each table T_j according to this order. Therefore, given a short subformula, we can find the corresponding record in logarithmic time.
3. **Partial assignments.** For each $j \in \{1, \dots, d\}$ we consider all assignments to the variables in $V - V_j$. Given such an assignment, we check whether it satisfies at least $(1 - 1/d)K$ clauses. For each assignment that passes this test, we perform the next step.
4. **Extension.** Let a be a partial assignment selected at the previous step. Let K_a denote the number of clauses satisfied by a . If $K_a \geq K$, return “yes”. Otherwise, we try to extend a to an assignment that satisfies at least K clauses. Consider the formula $F(a)$ obtained from F by substitution of the truth values corresponding to a . To extend a , we need to satisfy $K - K_a$ clauses in $F(a)$. Two cases are possible:
 - (a) *Case 1.* If $|F(a)| \geq 2(K - K_a)$ then by Lemma 3 there exists an assignment that satisfies at least $K - K_a$ clauses in $F(a)$. Therefore, a can be extended to an assignment that satisfies at least K clauses. Return “yes”.
 - (b) *Case 2.* The formula $F(a)$ is shorter than $2(K - K_a)$. We need to check whether there is an assignment that satisfies at least $K - K_a$ clauses in $F(a)$. Note that $F(a)$ must occur in the table T_j . Therefore, we can find (in logarithmic time) the maximum number of satisfiable clauses in $F(a)$. If this number is at least $K - K_a$ then return “yes”.
5. **“No” answer.** The output is “no” if we failed to find an appropriate assignment as described above.

Note that the algorithm can be also implemented using DPLL combined with memoization: partial assignments can be enumerated in the DPLL manner and the tables can be built on the run with memoization.

5 Runtime of Algorithm \mathcal{A}

In this section we give a worst-case upper bound on the runtime of Algorithm \mathcal{A} (Theorem 1). Applying this algorithm to solve MAX-SAT, we get essentially the same upper bound (Corollary 1). Then we show that if we restrict input formulas to formulas with constant clause density, we have the $\mathcal{O}(c^n)$ upper bound where $c < 2$ (Corollary 2).

Theorem 1. *If Algorithm \mathcal{A} is run with d such that*

$$d \geq 4 \text{ and } d \geq 2 \left(\frac{m}{n} \log \left(\frac{ed}{2} \right) + 1 \right) \tag{6}$$

where m and n are, respectively, the number of clauses and the number of variables in the input formula, then the runtime of Algorithm \mathcal{A} is

$$\mathcal{O}^* \left(2^{n(1-1/d)} \right).$$

Proof. First, we note that for any m and n , there exists d that meets the inequalities in (6). Consider the execution of Algorithm \mathcal{A} on input formula F with any $K \geq m/2$ and with d chosen according to (6). After partitioning of the variables in F into d subsets, we build d database tables. Each table contains $\sum_{i=1}^{2K/d} \binom{m}{i}$ records and can be built in time

$$\mathcal{O}^* \left(\sum_{i=1}^{2K/d} \binom{m}{i} \cdot 2^{n/d} \right).$$

After building the tables, we take d identical steps. At each step, we enumerate all assignments to $n(1 - 1/d)$ variables. For each assignment a , we compute the formula $F(a)$ and (if needed) look it up using binary search in the corresponding sorted table. Since the table size is $2^{\mathcal{O}^*(m)}$, the lookup can be done in polynomial time. Therefore, the overall runtime of Algorithm \mathcal{A} is bounded by

$$\mathcal{O}^* \left(\sum_{i=1}^{2K/d} \binom{m}{i} \cdot 2^{n/d} \right) + \mathcal{O}^* \left(2^{(1-1/d)n} \right). \tag{7}$$

We show that for any d that meets (6), the first term in this sum is asymptotically smaller than the second term. The sum of binomial coefficients in (7) can be approximated using the binary entropy function, see e.g. [7, exercise 9.42], so we can approximate the first term in (7) as

$$\mathcal{O}^* \left(2^{H(2K/md) m + n/d} \right).$$

Since the function H is increasing on the interval $(0, \frac{1}{2}]$, it follows from $d \geq 4$ and $K \leq m$ that

$$H \left(\frac{2K}{md} \right) \leq H \left(\frac{2}{d} \right).$$

Using the well known inequality $\ln(1+x) \leq x$, we have

$$\begin{aligned} H\left(\frac{2}{d}\right) &= -\frac{2}{d} \log\left(\frac{2}{d}\right) - \left(1 - \frac{2}{d}\right) \log\left(1 - \frac{2}{d}\right) \\ &= \frac{2}{d} \log\left(\frac{d}{2}\right) + \left(1 - \frac{2}{d}\right) \log e \ln\left(1 + \frac{2}{d-2}\right) \\ &\leq \frac{2}{d} \log\left(\frac{d}{2}\right) + \frac{2}{d} \log e \\ &= \frac{2}{d} \log\left(\frac{ed}{2}\right). \end{aligned}$$

Therefore, (7) is not greater than

$$\mathcal{O}^*\left(2^{(2m/d) \log(ed/2) + n/d}\right) + \mathcal{O}^*\left(2^{(1-1/d)n}\right)$$

It remains to observe that the second term in this sum dominates over the first term if

$$\frac{2m}{d} \log\left(\frac{ed}{2}\right) + \frac{n}{d} \leq n - \frac{n}{d},$$

which is equivalent to the condition on d in (6):

$$d \geq 2\left(\frac{m}{n} \log\left(\frac{ed}{2}\right) + 1\right). \quad \square$$

Corollary 1. *There is an exact deterministic algorithm that solves MAX-SAT in $\mathcal{O}^*(2^{n(1-1/d)})$ time, where d meets condition (6).*

Proof. We repeatedly apply Algorithm \mathcal{A} to find K such that the algorithm returns “yes” for K and returns “no” for $K+1$. This can be done using either binary search or straightforward enumeration. \square

Remark 1. What value of d minimizes the upper bound in Corollary 1? We can approximate the optimum value of d as follows:

$$d = \mathcal{O}\left(\frac{m}{n} \log\left(\frac{m}{n} + 2\right)\right) \quad (8)$$

It is straightforward to check that this approximation meets condition (6).

Corollary 2. *There is an exact deterministic algorithm that solves MAX-SAT for formulas with constant clause density in $\mathcal{O}(c^n)$ time where $c < 2$. More exactly, for any constant Δ , there is a constant $c < 2$ such that on formulas in $\mathcal{F}(\Delta)$, the algorithm runs in $\mathcal{O}(c^n)$ time.*

Proof. The $\mathcal{O}^*(2^{n(1-1/d)})$ upper bound can be written as

$$\mathcal{O}^*\left(2^{n(1-1/d)}\right) = \mathcal{O}\left(2^{n - n/d + \mathcal{O}(\log n)}\right) \quad (9)$$

If $\Delta = m/n$ is a constant, it immediately follows from (6) that d can be taken as a constant too. Therefore n/d is asymptotically larger than $\mathcal{O}(\log n)$, which yields the $\mathcal{O}(c^n)$ upper bound. \square

Remark 2. Note that the proof above can be used to obtain the $\mathcal{O}(2^n)$ bound for some classes $\mathcal{F}(\Delta)$ where Δ is not a constant. For example, we can allow Δ to be $\mathcal{O}(\sqrt{n})$. Using the approximation (8), we have

$$d = \mathcal{O}(\sqrt{n} \log n).$$

Then n/d is asymptotically larger than $\mathcal{O}(\log n)$ and, therefore, (9) gives the $\mathcal{O}(2^n)$ bound.

6 Algorithm That Needs No Tables

An obvious disadvantage of Algorithm \mathcal{A} is that the algorithm has to build an exponential-size database. We could avoid it if we chose d so that $F(a)$ is always “long” enough. Then a can be extended due to Lemma 3 and no lookup is needed. Using this observation, we define Algorithm \mathcal{B} that does not require building tables. Theorem 2 gives a worst-case upper bound on its runtime:

$$\mathcal{O}^* \left(2^{n \left(1 - \frac{m-K}{K}\right)} \right).$$

Algorithm \mathcal{B}

Input: Formula F with m clauses over n variables, integer K such that

$$m/2 < K \leq m \frac{n}{n+1}. \quad (10)$$

Output: “yes” if there is an assignment that satisfies at least K clauses, “no” otherwise.

1. **Partition.** We take an integer d defined by

$$d = \left\lceil \frac{K}{m-K} \right\rceil.$$

Then, similarly to the partition step in Algorithm \mathcal{A} , we partition the set V of variables in F into d subsets V_1, \dots, V_d . Note that it follows from (10) that $d \leq n$.

2. **Partial assignments.** This step is similar to the partial assignment step in Algorithm \mathcal{A} : for each $j \in \{1, \dots, d\}$ we enumerate all assignments to the variables in $V - V_j$. For each such assignment, we check whether it satisfies at least $(1 - 1/d)K$ clauses. In Theorem 2 below we show that any assignment a that satisfies at least $(1 - 1/d)K$ clauses can be extended to an assignment that satisfies at least K clauses. Therefore, as soon as we find a satisfying $(1 - 1/d)K$ clauses, we return “yes”.
3. **“No” answer.** If we failed to find an appropriate assignment at the previous step, “no” is returned.

Theorem 2. *Algorithm \mathcal{B} is correct for all K that meet (10). Its runtime is*

$$\mathcal{O}^* \left(2^{n(1-\frac{m-K}{K})} \right)$$

where m and n are, respectively, the number of clauses and the number of variables in the input formula.

Proof. Let a be a partial assignment selected at Step 3 and K_a be the number of clauses satisfied by a . Consider $F(a)$ that has $m - K_a$ clauses. If $F(a)$ is “long” enough, namely

$$m - K_a \geq 2(K - K_a), \tag{11}$$

then it follows from Lemma 3 that a can be extended to an assignment satisfying at least K clauses (cf. Case 1 in Algorithm \mathcal{A}). It remains to show that the choice of d in the algorithm guarantees (11).

Since $d = \lceil K/(m - K) \rceil$, we have

$$d \geq K/(m - K).$$

This inequality can be rewritten as

$$m - \frac{K}{d} \geq K.$$

Adding K to both sides we get

$$m + K - \frac{K}{d} \geq 2K.$$

Since $K_a \geq (1 - 1/d)K$,

$$m + K_a \geq 2K.$$

Subtracting $2K_a$ from both sides, we obtain (11).

Clearly, the runtime of the algorithm is determined by the time needed to enumerate all assignments to the variables in each $V - V_j$. Since each such subset of variables contains at most $\lceil (1 - 1/d)n \rceil$ variables, we obtain the claimed runtime. \square

7 Summary of Results

1. We give an exact deterministic algorithm that solves MAX-SAT for CNF formulas with no limit on clause length. Its runtime is

$$\mathcal{O}^* \left(2^{n(1-1/d)} \right) \text{ where } d = \mathcal{O} \left(\frac{m}{n} \log \left(\frac{m}{n} + 2 \right) \right).$$

2. This algorithm solves MAX-SAT for formulas with constant clause density in time

$$\mathcal{O}(c^n) \text{ where } c < 2.$$

3. We give another exact deterministic algorithm that solves MAX-SAT in time

$$\mathcal{O}^* \left(2^{n(1-\frac{m-K}{K})} \right)$$

where K is the maximum number of satisfiable clauses. This algorithm is faster than the first one when K is not too large compared to m .

Acknowledgement

We thank anonymous referees for their useful comments.

References

1. V. Arvind and R. Schuler. The quantum query complexity of 0-1 knapsack and associated claw problems. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation, ISAAC 2003*, volume 2906 of *Lecture Notes in Computer Science*, pages 168–177. Springer, December 2003.
2. N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation, ISAAC'99*, volume 1741 of *Lecture Notes in Computer Science*, pages 247–258. Springer, December 1999.
3. P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity, CCC 2003*, pages 225–236, 2003.
4. L. Cai and J. Chen. On fixed-parameter tractability and approximability of NP optimization problems. *Journal of Computer and System Sciences*, 54(3):465–474, 1997.
5. J. Chen and I. Kanj. Improved exact algorithm for Max-Sat. *Discrete Applied Mathematics*, 142(1-3):17–27, August 2004.
6. E. Dantsin, M. Gavrilovich, E. A. Hirsch, and B. Konev. MAX SAT approximation beyond the limits of polynomial-time approximation. *Annals of Pure and Applied Logic*, 113(1-3):81–94, December 2001.
7. R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994.
8. E. A. Hirsch. Worst-case study of local search for MAX- k -SAT. *Discrete Applied Mathematics*, 130(2):173–184, 2003.
9. H. Karloff and U. Zwick. A 7/8-approximation algorithm for MAX 3SAT? In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97*, pages 406–415, 1997.
10. M. Mahajan and V. Raman. Parameterizing above guaranteed values: Maxsat and maxcut: An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 31(2):335–354, 1999.
11. B. Monien and E. Speckenmeyer. Upper bounds for covering problems. Technical Report Bericht Nr. 7/1980, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn, 1980.
12. R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36(1):63–88, July 2000.
13. R. Williams. On computing k -CNF formula properties. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 330–340. Springer, May 2003.
14. H. Zhang, H. Shen, and F. Manyà. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*, 86(1), May 2003.