Electronic Colloquium on Computational Complexity, Report No. 37 (2000)

New Worst-Case Upper Bounds for MAX-2-SAT with Application to MAX-CUT

Jens Gramm* Edward A. Hirsch† Rolf Niedermeier‡ Peter Rossmanith§

May 26, 2000

Abstract

The maximum 2-satisfiability problem (MAX-2-SAT) is: given a Boolean formula in 2-CNF, find a truth assignment that satisfies the maximum possible number of its clauses. MAX-2-SAT is **MAX-SNP**-complete. Recently, this problem received much attention in the contexts of approximation (polynomial-time) algorithms and exact (exponential-time) algorithms. In this paper, we present an exact algorithm solving MAX-2-SAT in time $\mathrm{poly}(L) \cdot 2^{K/5}$, where K is the number of clauses and L is their total length. Since, in our analysis, we count only clauses containing exactly two literals, this bound implies the bound $\mathrm{poly}(L) \cdot 2^{L/10}$.

Our results significantly improve previous bounds: $\mathrm{poly}(L) \cdot 2^{K/2.88}$ [27] and $\mathrm{poly}(L) \cdot 2^{K/3.44}$ (implicit in [3]). Concerning the bound w.r.t. L, nothing better than the bound $\mathrm{poly}(L) \cdot 2^{L/6.89}$ [3] for (general) MAX-SAT was known. Our algorithm, together with its analysis, is much simpler than recent MAX-SAT algorithms (previous MAX-2-SAT bounds were obtained by general MAX-SAT algorithms using inequalities relating different input formula parameters).

As an application, we derive upper bounds for the (**MAX-SNP**-complete) maximum cut problem (MAX-CUT), showing that it can be solved in time $\mathrm{poly}(M) \cdot 2^{M/3}$, where M is the number of edges in the given graph. This is of special interest for graphs with low vertex degree.

1 Introduction

Worst-Case Upper Bounds for NP-hard Problems. Various **NP**-hard optimization problems arise naturally in many areas of computer science while no polynomial-time algorithms for them are known. For some of these problems, there are polynomial-time approximation algorithms that give solutions within a factor of some performance ratio α of the optimal solution. However, for those problems that are **MAX-SNP**-hard (see, e.g., [1, 28]), it is known that the performance ratio of a polynomial-time algorithm cannot be better than some constant ζ (inapproximability ratio) unless $\mathbf{P} = \mathbf{NP}$. For example, for MAX-2-SAT (for formal definitions, see below), $\alpha = 0.931$ [14] and $\zeta = 0.955$ [17].

*Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Fed. Rep. of Germany. Email: gramm@informatik.uni-tuebingen.de. WWW: http://www-fs.informatik.uni-tuebingen.de/~gramm.

†Steklov Institute of Mathematics at St.Petersburg, 27 Fontanka, 191011 St.Petersburg, Russia. Email: hirsch@pdmi.ras.ru. WWW: http://logic.pdmi.ras.ru/~hirsch. Work is partially supported by grants from INTAS and RFBR.

‡Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Fed. Rep. of Germany. Email: niedermr@informatik.uni-tuebingen.de. WWW: http://www-fs.informatik.uni-tuebingen.de/~niedermr.

§Institut für Informatik, Technische Universität München, Arcisstr. 21, D-80290 München, Fed. Rep. of Germany. Email: rossmani@in.tum.de. WWW: http://wwwbrauer.in.tum.de/~rossmani.

ISSN 1433-8092

Recently, there was an explosion in proving (exponential-time) worst-case upper bounds for **NP**-hard problems and, in particular, for the exact solution of **MAX-SNP**-hard problems. The main results in the area concentrate around *SAT*, the problem of satisfiability of a propositional formula in conjunctive normal form (*CNF*), which can be easily solved in time of the order 2^N, where N is the number of variables in the input formula. In the early 1980s, this trivial bound was improved for formulas in 3-CNF (every clause contains at most three literals) by Monien and Speckenmeyer [26] and independently by Dantsin [8] (e.g., a $2^{N/1.44}$ bound[1] was proved). After that, many upper bounds for k-SAT [10, 20, 23, 24, 29, 33], MAX-SAT [3, 25, 27], MAX-2-SAT [3, 27], and other **NP**-hard problems were obtained.

Previous Research and Our Results. Concerning the problems for formulas in CNF, most authors consider bounds w.r.t. three main parameters:

- the length L of the input formula (i.e., the number of literal occurrences),
- the number K of its clauses, and
- the number N of the variables occurring in it.

The best currently known bounds for SAT are $2^{K/3.23}$ and $2^{L/9.7}$ [20], while, w.r.t. the number of variables, nothing better than trivial 2^N is known. In constrast, for 3-SAT, randomized $(4/3)^N$ [33] and deterministic 1.481^N [10] are known, while the bounds w.r.t. K and L are the same as for SAT.

The maximum satisfiability problem (*MAX-SAT*) is an important generalization of SAT. Here, we are given a formula in CNF, and the answer is the maximum number of simultaneously satisfiable clauses. This problem is **NP**-complete[2] and **MAX-SNP**-complete, even if each clause contains at most two literals (*MAX-2-SAT*; see, e.g., [28, Theorem 13.11]). MAX-SAT and MAX-2-SAT are well-studied in the context of approximation algorithms (see, e.g., [2, 9, 14, 17, 22, 34]). Recently, numerous results appeared in the domain of worst-case time bounds for the exact solution of MAX-SAT and MAX-2-SAT [3, 9, 16, 18, 19, 25, 27]. The currently best bounds for MAX-SAT are $2^{K/2.36}$ and $2^{L/6.89}$ [3]. For MAX-2-SAT, the considerably better bounds $2^{K/2.88}$ [27] and $2^{K/3.44}$ (implicit in [3]) follow from MAX-SAT algorithms. In this paper we prove a much better $2^{K/5}$ bound by giving a direct (and much simpler!) algorithm for MAX-2-SAT. Our result still holds if K in the exponent is the number of 2-clauses (i.e., unit clauses are not counted). Therefore, the bound $2^{L/10}$ follows, which is the first bound w.r.t. L that is better for MAX-2-SAT than for MAX-SAT.

Using our MAX-2-SAT algorithm, we easily obtain the bound $2^{M/3}$ for the MAX-CUT problem (given a graph with M edges, find a cut of maximum size in it). This is of particular interest for graphs with bounded degree: If the maximum vertex degree is 3, then MAX-CUT can be solved in time $2^{n/2}$ (where n is the number of vertices) and, if the maximum vertex degree is 4, then MAX-CUT can be solved in time $2^{2n/3}$. For larger degree $d \geq 5$, our algorithm does not improve a simple $2^{nd/(d+1)}$ bound [35]. We are not aware of previous non-trivial worst-case upper bounds for the exact solution of MAX-CUT, except for the parametrized bounds given by Mahajan and Raman [25]. Their results are a bound of 2^{2k} for the question of whether a given graph has a cut

[1] For brevity, we usually omit a polynomial factor in this paper: e.g., if we write $2^{N/1.44}$, we mean $\text{poly}(|F|)\cdot 2^{N/1.44}$, where $|F|$ is the length of the input.

[2] A more precise **NP**-formulation is, of course, "given a formula in CNF and an integer k, decide whether there is an assignment that satisfies at least k clauses."

of size k, and a bound of 2^{4k} for the question of whether a given graph with m edges has a cut of size $\lceil \frac{m}{2} \rceil + k$.

Our results w.r.t. K and w.r.t. M, also hold for the versions of MAX-2-SAT and MAX-CUT where each clause (resp., edge) is assigned an integer weight. In this case, K and M in the above bounds denote the total weight of all clauses (resp., edges).

Splitting Algorithms. Most of the algorithms corresponding to the bounds mentioned above, as well as the algorithms presented in this paper, use a kind of Davis-Putnam-Logemann-Loveland procedure [11, 12]. In short, this procedure reduces the problem for a formula F to the problem for two formulas $F[v]$ and $F[\overline{v}]$ (where v is a propositional variable). This is called "splitting." Before the algorithm splits each of the obtained two formulas, it can transform them into simpler formulas F_1 and F_2 using *transformation rules*. In a *splitting tree* corresponding to the execution of such an algorithm, the node labelled by F has two children labelled by F_1 and F_2. The algorithm does not split a formula if it is trivial to solve the problem for it; these formulas are the leaves of the splitting tree. The running time of the algorithm is within a $\mathrm{poly}(L)$ factor of the number of leaves.

Sources of Our Improvements. Our MAX-2-SAT algorithm is a typical splitting algorithm, i.e., to describe it we need to specify: a set of formulas corresponding to the leaves of our tree, a heuristic determining the choice of a variable for splitting, and transformation rules. Worst-case analysis of such algorithms usually contains a huge amount of case enumeration. The number of cases we need to consider in our proof is tremendously smaller than in the current results for general MAX-SAT [3, 27]. Our MAX-2-SAT algorithm makes use of two main ideas.

The leaves of our splitting tree are formulas containing only unit clauses (clearly, MAX-1-SAT is trivial). Therefore, in the analysis of the running time of our algorithm *we count only 2-clauses.* We prove that every variable occurring in at most two[3] 2-clauses (and maybe some 1-clauses) can be eliminated in polynomial time[4]. It is not very hard to see that if there is a variable occurring in three 2-clauses, then we can make a splitting such that each of the formulas F_1 and F_2 has at least five 2-clauses less than F (this situation corresponds to the recurrence inequality $T(K) \leq 2T(K-5)$ for the running time). Clearly, we can say the same about F containing a variable occurring in at least five 2-clauses. If our splitting tree contains only formulas of these types, then the running time is at most $2^{K/5}$. The remaining case corresponds to the recurrence inequality $T(K) \leq 2T(K-4)$.

The second idea is connected to a general point in splitting algorithms for **NP**-hard problems: usually, a problem has "bottleneck" instances, i.e., the instances corresponding to the "worst" recurrence inequality. For example, for the algorithm described above, these are the formulas for which our splitting corresponds to the inequality $T(K) \leq 2T(K-4)$. Usually, this situation is handled by looking to the next level of splitting and showing that the obtained two instances are not "bottleneck" [20, 27]. In this paper, we handle this situation in a different way. Namely, we show that we can build a splitting tree such that *each branch contains at most one "bottleneck" instance.* Therefore, we can omit the corresponding recurrence inequality from asymptotic analysis.

For the MAX-CUT problem, there is an easy translation of any of its instances with M edges into a MAX-2-SAT instance with $2M$ clauses. This would already give us a $2^{2M/5}$ bound. However, the formulas given by the translation satisfy a very specific condition. Moreover, this condition is

[3]For simplicity, we give here our ideas in the unweighted case.

[4]In fact, it follows easily that MAX-2-SAT is solvable in polynomial time when every variable occurs in at most two 2-clauses (and maybe some 1-clauses). Note that MAX-2-SAT is **NP**-complete and **MAX-SNP**-complete, even if the number of occurrences of every variable is bounded by three (see, e.g., [5, 31]).

preserved by our transformation rules. For such formulas, our algorithm runs with small modifications in the time $2^{K/6}$, i.e., MAX-CUT can be solved in the time $2^{M/3}$.

History of the Paper. The present work started from [15, 16, 18, 19], where parts of the ideas of this paper already appeared. The authors thank DIMACS for financial support that gave them an opportunity to meet at DIMACS Workshop on Faster Exact Algorithms for NP-Hard Problems, where the ideas from earlier discussions between them were implemented into better algorithms with significantly better bounds.

Organization of the Paper. Our paper is structured as follows. In Section 2, we give basic definitions. In Section 3, we describe the transformation rules we use. In Section 4, we present our new MAX-2-SAT algorithm and its analysis. Section 5 shows the application to MAX-CUT. Conclusions, open questions, and comparison to close research are given in Section 6.

2 Background

Let V be a set of Boolean variables. The negation of a variable v is denoted by $\overline{v}$. *Literals* are variables and their negations. If l denotes a negated variable $\overline{v}$, then $\overline{l}$ denotes the variable v.

Algorithms for finding the exact solution of MAX-SAT are usually designed for the unweighted MAX-SAT problem. However, the formulas are usually represented by multisets (i.e., formulas in CNF with positive integer weights). In this paper, we consider the weighted MAX-SAT problem with positive integer weights. A *(weighted) clause* is a pair (ω, S) where ω is a strictly positive integer number and S is a nonempty finite set of literals which does not contain, simultaneously, any variable together with its negation. We call ω the *weight* of a clause (ω, S).

An *assignment* is a finite set of literals that does not contain any variable together with its negation. Informally speaking, if an assignment A contains a literal l, then the literal l has the value *True* in A. In addition to usual clauses, we allow a special *true clause* $(\omega, \mathbb{T})$ which is satisfied by every assignment. (We also call it a $\mathbb{T}$-*clause*.)

The length of a clause (ω, S) is the cardinality of S. A *k-clause* is a clause of length exactly k. In this paper, a *formula in (weighted) CNF* (or simply *formula*) is a finite set of (weighted) clauses (ω, S), with at most one clause for each S. If a formula contains only one clause, for short we write this clause instead of the formula. A formula is in *2-CNF* if it contains only 2-clauses, 1-clauses and a $\mathbb{T}$-clause. The *length of a formula* is the sum of the lengths of all its clauses. The total weight of all 2-clauses of a formula F is denoted by $K_2(F)$ and by K_2 when the formula is clear from the context.

The pairs $(0, S)$ are *not* clauses: for simplicity, however, we write $(0, S) \in F$ for all S and all F. Therefore, the operators $+$ and $-$ are defined:

$$\begin{aligned} F + G &= \{(\omega_1 + \omega_2, S) \mid (\omega_1, S) \in F \text{ and } (\omega_2, S) \in G, \text{ and } \omega_1 + \omega_2 > 0\}, \\ F - G &= \{(\omega_1 - \omega_2, S) \mid (\omega_1, S) \in F \text{ and } (\omega_2, S) \in G, \text{ and } \omega_1 - \omega_2 > 0\}. \end{aligned}$$

Example 1. If

$$F = \{\ (2, \mathbb{T}),\ (3, \{x, y\}),\ (4, \{\overline{x}, \overline{y}\})\ \}$$

and

$$G = \{\ (2, \{x, y\}),\ (4, \{\overline{x}, \overline{y}\})\ \},$$

then

$$F - G = \{\ (2, \mathbb{T}),\ (1, \{x, y\})\ \}.$$

For a literal l and a formula F, the formula $F[l]$ is obtained by setting the value of l to *True*. More precisely, we define

$$\begin{aligned} F[l] \;=\; & (\{(\omega, S) \mid (\omega, S) \in F \text{ and } l, \bar{l} \notin S\} + \\ & \{(\omega, S \setminus \{\,\bar{l}\,\}) \mid (\omega, S) \in F \text{ and } S \neq \{\,\bar{l}\,\} \text{ and } \bar{l} \in S\} + \\ & \{(\omega, \mathbb{T}) \mid \omega \text{ is the sum of the weights } \omega' \\ & \qquad \text{of all clauses } (\omega', S) \text{ of } F \text{ such that } l \in S\}. \end{aligned}$$

(Note that no $(\omega, \emptyset)$ or $(0, S)$ is included in $F[l]$, $F+G$ or $F-G$.) For an assignment $A = \{l_1, \ldots, l_s\}$ and a formula F, we define $F[A] = F[l_1][l_2]\ldots[l_s]$ (evidently, $F[l][l'] = F[l'][l]$ for every pair of literals l, l' with $l \neq \overline{l'}$). For short, we write $F[l_1, \ldots, l_s]$ instead of $F[\{l_1, \ldots, l_s\}]$.

Example 2. If

$$F = \{\ (1, \mathbb{T}),\ (1, \{x, y\}),\ (5, \{\overline{y}\}),\ (2, \{\overline{x}, \overline{y}\}),\ (10, \{\overline{z}\}),\ (2, \{\overline{x}, z\})\ \},$$

then

$$F[x, \overline{z}] = \{\ (12, \mathbb{T}),\ (7, \{\overline{y}\})\ \}.$$

The optimal value of a maximum weight assignment for formula F is defined as $\mathrm{OptVal}(F) = \max_A\{\,\omega \,|\, (\omega, \mathbb{T}) \in F[A]\,\}$, where A is taken over all possible assignments. An assignment A is *optimal* if $F[A]$ contains only one clause $(\omega, \mathbb{T})$ (or does not contain any clause, in this case $\omega = 0$) and $\mathrm{OptVal}(F) = \omega$ $(= \mathrm{OptVal}(F[A]))$.

If we say that a *literal* v *occurs* in a clause or in a formula, we mean that this clause (more formally, its second component) or this formula (more formally, one of its clauses) contains the literal v. However, if we say that a *variable* v *occurs* in a clause or in a formula, we mean that this clause or this formula either contains the literal v or it contains the literal $\overline{v}$.

For a literal l, we write $\#_l(G)$ to denote the total weight of the clauses of a formula G in which l occurs. We omit G when the meaning of G is clear from the context. We also write $\#_l^{(k)}$ to denote the total weight of k-clauses in which l occurs. The *weight of a variable* is the total sum of the weights of the 2-clauses the variable occurs in.

A *closed subformula* G is a subset of a formula F such that none of the variables occurring in G occurs in $F - G$. We use this term only for non-trivial subformulas, i.e. both G and $F - G$ contain at least one variable.

3 Transformation rules

A *correct* transformation rule replaces a formula F with a "simpler" formula F' such that F *has an optimal assignment with weight* ω *iff* F' *has an optimal assignment with weight* ω, i.e., a correct transformation rule preserves OptVal. In this section, we present the transformation rules we use and show their correctness. Note that these rules do not increase neither the weight of any variable nor the total weight of the 2-clauses.

Pure literal. A literal is *pure* in a formula F if it occurs in F, and its negation does not occur in F. The following lemma is well-known and straightforward.

Lemma 1. If b is a pure literal in F, then $\mathrm{OptVal}(F) = \mathrm{OptVal}(F[b])$.

Rule $\mathbf{T_{pure}}$ replaces F with $F[b]$ if b is a pure literal.

Annihilation of 1-clauses. Rule $\mathbf{T_{ann}}$ "annihilates" opposite 1-clauses, i.e., it replaces F with $(F - \{ (\omega, \{a\}) , (\omega, \{\overline{a}\}) \}) + (\omega, \mathbb{T})$ if F contains clauses $(\omega_1, \{a\})$ and $(\omega_2, \{\overline{a}\})$ and $\omega = \min(\omega_1, \omega_2)$.

Resolution. In this paper, the *resolvent* $\mathfrak{R}(C, D)$ of 2-clauses $C = (\omega_1, \{l_1, l_2\})$ and $D = (\omega_2, \{\overline{l_1}, l_3\})$ is the formula

$$\{ (\max(\omega_1, \omega_2), \mathbb{T}), \ (\min(\omega_1, \omega_2), \{l_2, l_3\}) \} \tag{1}$$

if $l_2 \neq \overline{l_3}$, and it is the formula $\{(\omega_1 + \omega_2, \mathbb{T})\}$, otherwise. This definition is slightly non-traditional, but it is very useful in the MAX-SAT context.

The following lemma is a straightforward generalization of a statement about usual resolution (see, e.g., [32]).

Lemma 2. If F contains 2-clauses $C = (\omega_1, \{v, l_1\})$ and $D = (\omega_2, \{\overline{v}, l_2\})$ such that the variable v does not occur in other clauses of F, then

$$\mathrm{OptVal}(F) = \mathrm{OptVal}(\ (F - \{C, D\}) + \mathfrak{R}(C, D) \ ). \tag{2}$$

Rule $\mathbf{T_{DP}}$ replaces F with $(F - \{C, D\}) + \mathfrak{R}(C, D)$ if F, C, and D satisfy the conditions of Lemma 2.

Dominating 1-clause. The following fact was observed in [27].

Lemma 3 ([27]). If for a literal b and a formula F, $\#_b^{(1)} \geq \#_{\overline{b}}$, then

$$\mathrm{OptVal}(F) = \mathrm{OptVal}(F[b]).$$

Rule $\mathbf{T_{dom}}$ replaces F with $F[b]$ in such a case.

Small closed subformula. We can easily compute the optimal value for a closed subformula G containing at most, say, 12 variables. Clearly,

$$\mathrm{OptVal}(F) = \mathrm{OptVal}(F - G) + \mathrm{OptVal}(G).$$

Rule $\mathbf{T_{small}}$ replaces F with $(F - G) + (\mathrm{OptVal}(G), \mathbb{T})$ in such a case.

Rare variable. Let F be a formula, and let a be a literal such that $\#_a^{(2)} = 2$, $\#_{\overline{a}}^{(2)} = \#_a^{(1)} = 0$, and $\#_{\overline{a}}^{(1)} = 1$. Consider a 2-clause $(\omega, \{a, b\})$ in F. Rule $\mathbf{T_{rare}}$ replaces this clause with $(\omega, \mathbb{T})$ and replaces literal a with literal $\overline{b}$ and literal $\overline{a}$ with literal b in all other clauses.

Lemma 4. Rule $\mathbf{T_{rare}}$ is correct.

Proof. Let F' be the obtained formula. It is trivial that $\mathrm{OptVal}(F') \leq \mathrm{OptVal}(F)$. We now prove the opposite inequality.

Let A be an optimal assignment for F. Let $b \in A$. Consider $F[b]$. Note that we can apply $\mathbf{T_{dom}}$ to the literal $\overline{a}$ in this formula, i.e.,

$$\begin{aligned} &\mathrm{OptVal}(F) = \mathrm{OptVal}(F[A]) \leq \mathrm{OptVal}(F[b]) \\ &= \mathrm{OptVal}(F[\overline{a}, b]) = \mathrm{OptVal}(F'[\overline{a}, b]) \leq \mathrm{OptVal}(F'). \end{aligned}$$

Let now $\overline{b} \in A$. Consider $F[\overline{b}]$. Note that we can apply $\mathbf{T_{ann}}$ and then $\mathbf{T_{pure}}$ to the literal a in this formula, i.e.,

$$\begin{aligned} &\mathrm{OptVal}(F) = \mathrm{OptVal}(F[A]) \leq \mathrm{OptVal}(F[\overline{b}]) \\ &= \mathrm{OptVal}(F[a, \overline{b}]) = \mathrm{OptVal}(F'[a, \overline{b}]) \leq \mathrm{OptVal}(F'). \end{aligned}$$

□

4 A $2^{K/5}$-time Algorithm for MAX-2-SAT

In this section, we present Algorithm 1 which solves MAX-2-SAT in time $\text{poly}(L) \cdot 2^{K_2/5}$, where K_2 is the total weight of 2-clauses of the input formula (in the case of unweighted MAX-2-SAT, K_2 is the number of 2-clauses). We first state the algorithm and then show its running time and its correctness using several lemmas.

Algorithm 1.
Input: A formula F in weighted 2-CNF.
Output: OptVal(F).

Method.

(A1) Apply $\mathbf{T_{pure}}$, $\mathbf{T_{ann}}$, $\mathbf{T_{DP}}$, $\mathbf{T_{dom}}$, $\mathbf{T_{small}}$, $\mathbf{T_{rare}}$ to F as long as at least one of them is applicable.

(A2) If F contains only a $\mathbb{T}$-clause, return the weight of this clause.

(A3) If F consists of several closed subformulas, then decompose F into two closed subformulas H_1 and H_2, apply Algorithm 1 to each of the formulas $H_1 + (1, \{u, v\})$ and $H_2 + (1, \{u, v\})$ (where u and v are new variables), and return the sum of the results minus 2.

(A4) If each variable has weight exactly four, then choose a variable v, apply Algorithm 1 to $F[v]$ and $F[\bar{v}]$ and return the maximum of the results.

(A5) If F satisfies the conditions of Lemma 6 below, then compute F_1 and F_2 with the properties stated in the lemma. Next, recursively apply Algorithm 1 to F_1 and F_2 and return the maximum of the results.

Otherwise, find a variable v such that the following works: Form the formulas $F_1' := F[v]$ and $F_2' := F[\overline{v}]$. For each $i = 1, 2$, apply $\mathbf{T_{pure}}$, $\mathbf{T_{ann}}$, $\mathbf{T_{DP}}$, $\mathbf{T_{dom}}$, $\mathbf{T_{rare}}$ to F_i' in such order that, for the obtained formula F_i,

$$K_2(F) - K_2(F_i) \geq 5. \tag{3}$$

Execute Algorithm 1 for the formulas F_1 and F_2 and return the maximum of its answers.

□

We first formulate the additional straightforward properties of our transformation rules that we use in our proofs.

Lemma 5. Let F be a formula, and let x be a variable of weight one or two. Then repeated application of transformation rules to this variable

1. eliminates this variable from F;
2. decreases the total weight of 2-clauses of F; and
3. does not change clauses that do not contain x (in particular, it does not change the weights of the variables that do not occur together with x in a clause).

The following two lemmas address special cases that will be needed in our main theorem which states the correctness of Algorithm 1 and proves the claimed running time.

Lemma 6. Let F be a formula such that there are no closed subformulas and all variables are of weight either three or four, where both these possibilities are realized. Furthermore, let us assume that no transformation rule is applicable.

Then, we can easily compute two formulas F_1 and F_2 such that for each $i = 1, 2$,

1. F_i contains a variable of weight exactly one, two, or three,
2. $K_2(F) - K_2(F_i) \geq 5$, and
3. $\mathrm{OptVal}(F) = \max\{\mathrm{OptVal}(F_1), \mathrm{OptVal}(F_2)\}$.

Proof. Let x be a variable of weight three and let y be a variable of weight four. Furthermore, let x and y occur together in a clause. Such variables must exist, since there are no closed subformulas.

As a special case, let us first assume that there is a variable v ($v = x$ is possible) that exclusively occurs together with y in clauses of total weight three. Then, take the variable z ($z = x$ is possible) that only occurs together with y in a clause of weight one and look at $F[z]$ and $F[\bar{z}]$: In both formulas, all clauses that contain y form a small closed subformula. Hence, we apply $\mathbf{T_{small}}$ to $F[z]$ and $F[\bar{z}]$, resulting in F_1 and F_2. Note that $K_2(F) - K_2(F_i) \geq 6$. If claim (1) of Lemma 6 is violated, i.e., F_i contains a variable that does not occur four times in 2-clauses, then we replace F_i with $F_i - (1, \mathbb{T}) + (1, \{u, v\})$, where u and v are new variables. Thus, v fulfills claim (1), and claim (2) and (3) are still true.

If the previous special case does not apply, then x occurs in $F[y]$ in 2-clauses of weight one or two. We now produce a formula F_1 from $F[y]$ by applying transformation rules to x in $F[y]$ until x is eliminated. To fulfill claim (1), we choose some other variable z that occurs in F together in a clause with y and that still occurs in F_1 in 2-clauses, which, however, have weight at most three. Note that by Lemma 5(3), such a z always exists and meets claim (1) because, otherwise, the special case would have applied.

In the same way, we get F_2 from $F[\bar{y}]$. □

Lemma 7. Let F be a formula containing two variables u and v such that u has weight exactly two and v has weight either one or two. Then, we can find easily transformation rules that produce from F another formula F' such that $K_2(F) - K_2(F') \geq 2$.

Proof. First apply $\mathbf{T_{ann}}$ to F as long as possible. If we can now apply $\mathbf{T_{pure}}$ or $\mathbf{T_{dom}}$ to u then we are done, since this eliminates 2-clauses of weight two.

Otherwise, we can apply $\mathbf{T_{rare}}$ or $\mathbf{T_{DP}}$ to u which eliminates 2-clauses of weight one or two and, if it eliminates a 2-clause of weight only one, then it leaves v occurring in 2-clauses of weight one or two (see Lemma 5). Hence, we can now apply transformation rules to v that eliminate another 2-clause of weight one. □

Using the above lemmas, we are now ready to prove our main result:

Theorem 1. Given a formula F in 2-CNF, Algorithm 1 always correctly finds $\mathrm{OptVal}(F)$ in time $\mathrm{poly}(L) \cdot 2^{K_2/5}$, where L is the length of F and K_2 is the total weight of 2-clauses in F.

Proof. Running time. Every transformation rule takes polynomial time and does not increase the total weight of non-$\mathbb{T}$-clauses. When the condition of a rule is satisfied, the rule decreases the total weight of non-$\mathbb{T}$-clauses. Thus, the transformation rules are executed a polynomial number of times during step (A1).

After applying transformation rules to F, Algorithm 1 makes two recursive calls for formulas with smaller total weight of 2-clauses (unless F becomes trivial), either at step (A3), (A4), or

(A5). Clearly, the total running time of the algorithm is the total running time of the two recursive calls plus a polynomial time spent to make these calls. Therefore, the running time is within a polynomial factor of the number of nodes (or leaves) of the recursion tree. In the following we show that the number $\lambda(K_2)$ of these leaves for a formula F with K_2 is $O(2^{K_2/5})$.

First consider a formula F with K_2 clauses that forces our algorithm to make a recursive call at step (A3) or (A5). The number of leaves in the recursion tree corresponding to this formula is at most $2\lambda(K_2 - 5)$. If all nodes of our tree for the input formula would be of this type, then we would have a straightforward $2^{K_2/5}$ bound on the number of leaves.

However, there may be also recursive calls at step (A4). On first glance, the number of leaves in a tree corresponding to such a call is bounded only by $2\lambda(K_2 - 4)$. To avoid worsening our bound, in the following we prove that, for most such formulas, we still have $2\lambda(K_2 - 5)$ leaves and a different "odd" formula can occur at most once in each branch of our tree (i.e., on each path from the root to a leaf). Therefore, we get the desired bound.

We now prove this claim about (A4). What may cause the application of (A4) to a formula F? In principle, F may be the input, F can originate from a transformation rule in (A1), or from a recursive call at step (A3), (A4), or (A5).

If F originated from applying a transformation rule at step (A1), then we have the desired bound on the number of leaves, since the transformation rule reduces K_2 at least by 1 and (A4) then reduces it by 4 (in both branches).

Note that F cannot originate from (A3), since (A3) adds weight one variables to each of the two produced formulas.

Such F also cannot originate from (A4): Setting the truth value of a variable clearly implies that, afterwards, another variable has weight 1, 2 or 3, because at step (A4), F does not have non-trivial closed subformulas.

Finally, if F originated from (A5), then we have two distinct cases. Let G be a formula from which F originated (i.e., F is obtained at step (A5) as $G[v]$ or $G[\overline{v}]$, or possibly from applying transformation rules to $G[v]$ or $G[\overline{v}]$). First, if G contains a weight 5 variable, then we do not need to worry, because this can happen only once on each path in the recursion tree from the root to one of its leaves (note that weights never increase and, thus, none of the successors will have a variable of weight greater than 4). Otherwise, Lemma 6 applies to G (note that by Lemma 5(1), at step (A5), the formula contains only weight 3 and weight 4 variables, and it must contain a weight 4 variable because weights do not increase).

Correctness. The correctness of the transformation rules is shown in Section 3. The correctness of steps (A2)–(A4) is trivial. It remains to show that at step (A5), appropriate rules can be easily found.

If a variable has weight at least five, then we can simply choose that variable. If each variable has weight exactly three, then we proceed as follows: Choose some variable x. Then it is easy to see that Lemma 7 applies to $F[x]$ (and to $F[\overline{x}]$) and, thus, we can eliminate 2-clauses of weight two from $F[x]$ (and from $F[\overline{x}]$) making a total of at least 5.

Otherwise, Lemma 6 applies and shows how to proceed. □

In the case of unweighted[5] MAX-2-SAT, we have $L \geq 2K_2$. This directly implies the following corollary.

Corollary 1. Given a formula F in unweighted 2-CNF of length L, Algorithm 1 always correctly finds $\mathrm{OptVal}(F)$ in time $\mathrm{poly}(L) \cdot 2^{L/10}$.

[5] In other words, all weights equal 1.

Remark 1. Of course, in Corollary 1, only the number of literal occurrences in 2-clauses is essential in the exponent.

Remark 2. Algorithm 1 can be easily redesigned so that it finds the optimal assignment (or one of them, if there are several assignments satisfying the same number of clauses) instead of only $\mathrm{OptVal}(F)$.

5 Application to MAX-CUT

Our results can be applied to other **NP**-complete problems which are easily reducible to MAX-2-SAT. For instance, we consider the **NP**-complete graph problem MAX-CUT: Given an undirected graph $G = (V, E)$ where edges are assigned integer weights, we ask for a cut of maximum weight, i.e., for a partition of V into V_1 and V_2 such that we maximize the sum of weights over those edges $(s, t) \in E$ for which $s \in V_1$ and $t \in V_2$. For a survey on MAX-CUT refer to Poljak and Tuza [30]. We can easily reduce MAX-CUT to MAX-2-SAT. The resulting formulas expose a very special structure. After presenting the reduction, we formulate, in the following, a condition that tries to capture this structure. We take advantage of it, and refine the analysis of Algorithm 1 when processing these formulas. Thereby, we improve the bounds, compared to the general case, and derive upper bounds for MAX-CUT.

For the reduction of MAX-CUT to MAX-2-SAT [30], we translate a graph $G = (V, E)$ into a 2-CNF formula having the vertices as variables and having clause set

$$\begin{aligned} C \;=\; & \{\, (w, \{i, j\}) \mid \text{edge } (i, j) \in E \text{ having weight } w \,\} \\ & \cup \{\, (w, \{\bar{i}, \bar{j}\}) \mid \text{edge } (i, j) \in E \text{ having weight } w \,\}. \end{aligned}$$

In this way, a graph having n vertices and m edges of total weight M results in a formula having n variables and $2m$ clauses of total weight $2M$. All these clauses are 2-clauses. The graph G has a cut of weight k iff the formula has simultaneously satisfiable clauses of weight $M + k$; every optimal assignment to the formula translates into a maximum cut, namely with all vertices corresponding to satisfied variables on one side and all vertices corresponding to falsified variables on the other side. An assignment satisfying a maximum number of clauses in the resulting formula will satisfy at least one of the clauses $(w, \{i, j\})$ and $(w, \{\bar{i}, \bar{j}\})$, which are created for an edge (i, j) of weight w, but will satisfy both clauses only if the edge is in the cut.

As we can see, the formulas created by this reduction initially exhibit a characteristic structure which we call *MAX-CUT Condition*:

> For each 2-clause of weight w containing literals x and y, there is also a 2-clause of weight w containing literals $\bar{x}$ and $\bar{y}$. **(MCC)**

In the following, we show that the steps applied by Algorithm 1 preserve this structure of the formulas.

Lemma 8. Let a formula satisfy (MCC). After applying a transformation rule or after assigning a value to a variable, the formula still fulfills (MCC).

Proof. For assigning a value to a variable, the claim is trivial; the 2-clauses of the new formula are exactly those 2-clauses of the old formula that do not contain the assigned variable. To prove the rest of this statement, we show for all transformation rules that, applied to a formula satisfying (MCC), they preserve this property. Rule $\mathbf{T_{rare}}$, however, cannot apply at all to formulas having (MCC). To apply this rule, we would need a literal x occurring in 2-clauses of weight two without $\bar{x}$ occurring in any 2-clauses. This contradicts (MCC).

When applying rules $\mathbf{T_{pure}}$ and $\mathbf{T_{dom}}$, we simply assign values to certain variables. Hence, the above discussion shows that these rules preserve (MCC). Rule $\mathbf{T_{ann}}$ does not affect the 2-clauses and, thus, does no harm to (MCC). As the statement formulated in (MCC) is valid or not only within a closed subformula, rule $\mathbf{T_{small}}$ also does not violate the property.

Only regarding $\mathbf{T_{DP}}$, it is not so obvious that the rule maintains (MCC). Let a variable x have occurrences in 2-clauses only in clauses $(w_1, \{x, l_1\})$ and $(w_2, \{\bar{x}, l_2\})$. We infer from (MCC) that $l_2 = \bar{l_1}$. Therefore, $\mathbf{T_{DP}}$ replaces these two clauses with $(w_1 + w_2, \mathbb{T})$ and, thus, (MCC) is not violated. □

We observe that the modifications covered in Lemma 8 are exactly those applied by Algorithm 1 to the input formula while processing it. We conclude that the special structure of the formula is preserved in every step of the algorithm. Compared with arbitrary formulas, the number of possible occurrence patterns for a variable is, thereby, reduced. Using this, we can improve the analysis of Algorithm 1 when the input is a formula satisfying (MCC).

To simplify the proof, we slightly modify Algorithm 1: step (A3) now does not add new variables and makes a recursive call directly for H_1 and H_2; step (A4) is omitted; and at step (A5), the inequality now requires $K_2(F) - K_2(F_i) \geq 6$. Also, F satisfying (MCC) cannot fulfill the condition of Lemma 6 and, thus, the first part of step (A5) can be omitted.

Theorem 2. Given a formula F in 2-CNF satisfying (MCC), the modified Algorithm 1 always correctly finds $\mathrm{OptVal}(F)$ in time $\mathrm{poly}(L) \cdot 2^{K_2/6}$, where L is the length of F and K_2 is the total weight of 2-clauses in F.

Proof. In the proof of Theorem 1, we have seen that every step of the recursion takes polynomial time. The size of the splitting tree is now guaranteed by the conditions of the steps of the modified algorithm. It only remains to prove that an appropriate variable and transformation rules at the modified step (A5) can be easily found.

In Lemma 8, we have shown that every step of Algorithm 1 (and also of its modified version) preserves (MCC). Thus, we can assume that every node of our splitting tree is labelled by a formula satisfying (MCC). Note that (MCC) implies that F does not contain variables of odd weights. Also, it does not contain variables of weight two (Lemma 5(1)), because these are handled by the transformation rules. Therefore, every formula labelling a node of our splitting tree either contains a variable of weight at least six (this directly implies the required inequality $K_2(F) - K_2(F_i) \geq 6$ for $i = 1, 2$), or each of its variables is of weight exactly four. We now prove that, even in this case, we can find transformation rules such as to fulfill the required inequality.

Take any clause of literals a and b corresponding to variables x and y. This clause has to have weight one: If it would have weight two, (MCC) would imply that there is also a clause $(2, \{\overline{a}, \overline{b}\})$ and, thus, there are no other 2-clauses containing variables x and y. In this situation, however, $\mathbf{T_{small}}$ would apply. Therefore, (MCC) implies that there is another literal c (corresponding to a variable z) such that there are, besides $(1, \{a, b\})$, also clauses $(1, \{\bar{a}, \bar{b}\})$, $(1, \{a, c\})$, and $(1, \{\bar{a}, \bar{c}\})$. Assigning a value to x eliminates four 2-clauses and causes $\mathbf{T_{dom}}$ to apply to y and z (again by (MCC)). This eliminates two more 2-clauses because, otherwise, x, y, and z would form a small closed subformula of F. Summarizing, we have that we can always fulfill the modified inequality of the step (A5). □

Theorem 2 gives upper bounds for the running time of the modified algorithm on 2-CNF formulas derived from MAX-CUT instances. We now translate these results into numbers of vertices and edges of a graph.

Corollary 2. Given a graph G having n vertices and edges of total weight M, we can solve (weighted) MAX-CUT in time $\text{poly}(n) \cdot 2^{M/3}$. If an unweighted graph has maximum vertex degree three, then MAX-CUT is solvable in time $\text{poly}(n) \cdot 2^{n/2}$, and if the graph has maximum vertex degree four, it is solvable in time $\text{poly}(n) \cdot 2^{2n/3}$.

Proof. Generating 2-CNF formulas from MAX-CUT instances, i.e., graphs with n vertices and edges of total weight M, results in 2-clauses of total weight $2M$ with n different variables. Then, the bound shown in Theorem 2 translates into a bound of $\text{poly}(n) \cdot 2^{2M/6} = \text{poly}(n) \cdot 2^{M/3}$ with respect to the total weight of the edges. The other two bounds follow from the inequality $m \leq dn/2$ relating n to the number m of edges and the maximum degree d. □

6 Discussion and Open Questions

Our Bounds vs Parametrized Bounds. In this paper, we proved the upper bound of the order $2^{K_2/5}$ for MAX-2-SAT with positive integer weights, where K_2 is the total weight of 2-clauses of the input formula (or the number of 2-clauses for unweighted MAX-2-SAT) and L is the number of literal occurrences. This also implies the bound $2^{L/10}$ for unweighted MAX-2-SAT. From this, we also derived upper bounds for MAX-CUT.

Our bounds depend neither on the weight of an optimal solution nor on a required minimal weight of solution. In contrast, beginning from [13, 25], there has been much research for *parametrized* bounds for MAX-SAT, MAX-2-SAT and MAX-CUT: in terms of k, how much time do we need to find a solution of weight at least k? For MAX-SAT, Bansal and Raman [3] give the best known parametrized bound $2^{k/2.15}$ which is better than their "unparametrized" bound $2^{K/2.36}$ when $k < 0.92K$, where K is the total weight of all clauses. In one of the papers [16] that inspired this paper, the parameterized bound $2^{k/2.73}$ for MAX-2-SAT has been proved. However, our present "unparametrized" bound $2^{K_2/5}$, where K_2 is the total weight of 2-clauses, is better for all reasonable values of k: the parametrized bound is better only when $k < 0.55K_2$, while an assignment satisfying $0.5K + 0.25K_2 \geq 0.75K_2$ clauses can be found in a polynomial time [25, 34]. It seems like the idea of counting only 2-clauses does not work for parametrized bounds.

As $\lceil \frac{K}{2} \rceil$ clauses can be easily satisfied, Mahajan and Raman [25] propose to ask in the parameterized version of the problem for an assignment satisfying $\lceil \frac{K}{2} + k' \rceil$ clauses. Taking the parameterized bound shown in [16] and plugging it into the results by Mahajan and Raman, we can translate it into a bound with respect to this new parameter k'; in time $2^{6k'/2.73} = 2^{k'/0.45}$ one can find an assignment to the variables that satisfies at least $\lceil \frac{K}{2} + k' \rceil$ clauses or one can determine that no such assignment exists. However, for $k' \leq \lceil \frac{K_2}{4} \rceil$, this question still can be handled in polynomial time. Comparing for $k' > \lceil \frac{K_2}{4} \rceil$ the bound $2^{k'/0.45}$ to the bound shown for Algorithm 1, we see, again, that the parametrized bound is worse for every parameter value.

It would be interesting, however, to consider, for a given k'', the parameterized complexity of the question whether there is an assignment satisfying $\lceil \frac{K}{2} + \frac{K_2}{4} \rceil + k''$ clauses.

Possible Applications of Our Ideas. The key idea of our MAX-2-SAT algorithm is to count only 2-clauses (we can do this, since MAX-1-SAT instances are trivial). It would be interesting to apply this idea to SAT, for example, by counting only 3-clauses in 3-SAT (since 2-SAT instances are easy). Also, it would be interesting to apply our idea of handling "bottleneck" cases to the analysis of other algorithms with such cases [20, 27]. Also, it remains a challenge to find a "less-than-2^N" algorithm for MAX-SAT, or even for MAX-2-SAT, where N is the number of variables. (Note that

for any fixed $\epsilon > 0$, an assignment satisfying $(1-\epsilon)\mathrm{OptVal}(F)$ clauses of a formula F in k-CNF can be found in randomized c^N time, where $c < 2$ is a constant depending only on k and ϵ [21].)

In a similar way as we did for MAX-CUT, we can apply our results to the **NP**-complete unweighted INDEPENDENT SET problem which also has an easy reduction to MAX-2-SAT [7]. The problem is, for a given graph $G = (V, E)$, to find the maximum number of vertices sharing no edge. The resulting bound with respect to the number of edges m, however, does not improve the bound of $2^{m/8.77}$ given by Beigel [4].

From a more practical point of view, it would also be challenging to experimentally examine the efficiency of our algorithms. Previous results for exact MAX-2-SAT algorithms having guaranteed worst-case time bounds compared with a heuristic algorithm [6] lacking guaranteed worst-case time bounds have shown encouraging results in this direction [15, 16].

References

[1] S. Arora and C. Lund. Hardness of approximation. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, chapter 10, pages 399–446. PWS Publishing Company, Boston, 1997.

[2] T. Asano and D. P. Williamson. Improved approximation algorithms for MAX SAT. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'00*, pages 96–105, 2000.

[3] N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In A. Aggarwal and C. Pandu Rangan, editors, *Algorithms and Computation (Proceedings of ISAAC'99)*, volume 1741 of *Lecture Notes in Computer Science*, pages 247–258. Springer-Verlag, December 1999.

[4] R. Beigel. Finding maximum independent sets in sparse and general graphs. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'99*, pages 856–857, 1999.

[5] P. Berman and M. Karpinski. On some tighter inapproximability results. In *Automata, Languages and Programming (Proceedings of ICALP'99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 200–209. Springer-Verlag, 1999.

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

[7] J. Cheriyan, W. H. Cunningham, L. Tunçel, and 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.

[8] E. Dantsin. Two propositional proof systems based on the splitting method (in Russian). *Zapiski Nauchnykh Seminarov LOMI*, 105:24–44, 1981. English translation: *Journal of Soviet Mathematics*, 22(3):1293–1305, 1983.

[9] E. Dantsin, M. Gavrilovich, E. A. Hirsch, and B. Konev. Approximation algorithms for MAX SAT: a better performance ratio at the cost of a longer running time. Technical Report PDMI preprint 14/1998, Steklov Institute of Mathematics at St.Petersburg, 1998. Electronic address: ftp://ftp.pdmi.ras.ru/pub/publicat/preprint/1998/14-98.ps.gz.

[10] E. Dantsin, A. Goerdt, E. A. Hirsch, and U. Schöning. Deterministic algorithms for k-SAT based on covering codes and local search. In *Proceedings of ICALP'2000*. To appear.

[11] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[13] R. G. Downey and M. R. Fellows. *Parametrized Complexity*. Springer-Verlag, 1999.

[14] U. Feige and M. X. Goemans. 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 and Computing Systems*, pages 182–189, 1995.

[15] J. Gramm. Exact algorithms for Max2Sat and their applications. Diploma thesis, WSI für Informatik, Universität Tübingen, October 1999. Available from http://www-fs.informatik.uni-tuebingen.de/~gramm/publications/.

[16] J. Gramm and R. Niedermeier. Faster exact solutions for Max-2-Sat. In G. Bongiovanni, G. Gambosi, and R. Petreschi, editors, *Algorithms and Complexity (Proceedings of CIAC 2000)*, volume 1767 of *Lecture Notes in Computer Science*, pages 174–186. Springer-Verlag, March 2000.

[17] J. Håstad. Some optimal inapproximability results. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing, STOC'97*, pages 1–10, 1997.

[18] E. A. Hirsch. A $2^{K/4}$-time algorithm for MAX-2-SAT: Corrected version. Technical Report 99-036, Revision 02, Electronic Colloquim on Computational Complexity, February 2000. Electronic address: ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/1999/TR99-036/revisn02.ps.

[19] E. A. Hirsch. A new algorithm for MAX-2-SAT. In H. Reichel and S. Tison, editors, *Proceedings of STACS 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 65–73. Springer-Verlag, February 2000. Contains an error, fixed in [18].

[20] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, Special Issue II on Satisfiability in Year 2000, 2000. To appear. A preliminary version appeared in *Proceedings of SODA'98*.

[21] E. A. Hirsch. Worst-case time bounds for MAX-k-SAT w.r.t. the number of variables using local search. In *Proceedings of RANDOM 2000*, 2000. To appear. Preliminary version available as Technical Report 00-019, Electronic Colloquim on Computational Complexity, 2000. Electronic address: ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/2000/TR00-019/index.html.

[22] 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.

[23] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[24] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages, available from http://www.cs.toronto.edu/~kullmann. A journal version is submitted to Information and Computation, January 1997.

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

[26] B. Monien and E. Speckenmeyer. Solving satisfiability in less then 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[27] R. Niedermeier and P. Rossmanith. New upper bounds for MaxSat. *Journal of Algorithms*, 2000. To appear. Preliminary version appeared in *Proceedings of ICALP'99*.

[28] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[29] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS'98*, pages 628–637, 1998.

[30] S. Poljak and Z. Tuza. Maximum cuts and large bipartite subgraphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 20:181–244, 1995.

[31] V. Raman, B. Ravikumar, and S. Srinivasa Rao. A simplified NP-complete MAXSAT problem. *Information Processing Letters*, 65(1):1–6, 1998.

[32] J. A. Robinson. Generalized resolution principle. *Machine Intelligence*, 3:77–94, 1968.

[33] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS'99*, pages 410–414, 1999.

[34] M. Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17(3):457–502, November 1994.

[35] U. Zwick. Personal communication, 2000.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'