Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Discrete Applied Mathematics 130 (2003) 139–155

DISCRETE APPLIED MATHEMATICS

www.elsevier.com/locate/dam

Worst-case upper bounds for MAX-2-SAT with an application to MAX-CUT

Jens Gramm[a,1,2], Edward A. Hirsch[b,3], Rolf Niedermeier[a,1,*], Peter Rossmanith[c,4]

[a] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, 72076 Tübingen, Germany
[b] Steklov Institute of Mathematics at St.Petersburg, 27 Fontanka, 191011 St.Petersburg, Russia
[c] Institut für Informatik, Technische Universität München, Arcisstr. 21, 80290 München, Germany

Received 31 May 2000; received in revised form 3 March 2002; accepted 19 March 2002

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 (polynomial-time) approximation algorithms and (exponential-time) exact algorithms. In this paper, we present an exact algorithm solving MAX-2-SAT in time $\text{poly}(L) \cdot 2^{K/5}$, where K is the number of clauses and L is their total length. In fact, the running time is only $\text{poly}(L) \cdot 2^{K_2/5}$, where K_2 is the number of clauses containing two literals. This bound implies the bound $\text{poly}(L) \cdot 2^{L/10}$. Our results significantly improve previous bounds: $\text{poly}(L) \cdot 2^{K/2.88}$ (J. Algorithms 36 (2000) 62–88) and $\text{poly}(L) \cdot 2^{K/3.44}$ (implicit in Bansal and Raman (Proceedings of the 10th Annual Conference on Algorithms and Computation, ISAAC'99, Lecture Notes in Computer Science, Vol. 1741, Springer, Berlin, 1999, pp. 247–258.))

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 $\text{poly}(M) \cdot 2^{M/3}$, where M is the number of edges in the graph. This is of special interest for graphs with low vertex degree.

* Corresponding author. Tel.: +49-7071-29-77568; fax: +49-7071-29-78962.

E-mail addresses: gramm@informatik.uni-tuebingen.de (J. Gramm), hirsch@pdmi.ras.ru (E.A. Hirsch), rossmani@in.tum.de (P. Rossmanith), niedermr@informatik.uni-tuebingen.de (R. Niedermeier).

[1] Web: `http://www-fs.informatik.uni-tuebingen.de/~gramm` or `http://www-fs.informatik.uni-tuebingen.de/~niedermr`

[2] Work supported by DFG projects KOMET, LA 618/3-3, and OPAL, NI 369/2-1.

[3] Web: `http://logic.pdmi.ras.ru/~hirsch`. Work is partially supported by INTAS YSF 99-4044, RFBR 99-01-00113, and grant #1 of the 6th contest-expertise of RAS young scientists projects 1999.

[4] Web: `http://wwwbrauer.in.tum.de/~rossmani`.

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. [1,3,31]), it is known that the performance ratio of a polynomial-time algorithm cannot be better than some constant ζ (inapproximability ratio) unless $P = NP$. For example, for MAX-2-SAT (for formal definitions, see below), $\alpha = 0.931$ [17] and $\zeta = 0.955$ [20].

Recently, there was an explosion in proving (exponential) worst-case time upper bounds for NP-hard problems and, in particular, for the exact solution of MAX-SNP-hard problems. Most 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 [29] and independently by Dantsin [10] (e.g., a $2^{N/1.44}$ bound[5] was proved). After that, many upper bounds for SAT [23,27], k-SAT [12,13,26,32,36,37], MAX-SAT [4,28,30], MAX-2-SAT [4,30], 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 $2^{L/9.7}$ [23], while, w.r.t. the number of variables, nothing better than trivial 2^N is known. In contrast, for 3-SAT, randomized 1.3303^N [37] and deterministic 1.481^N [12,13] 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[6] and MAX-SNP-complete, even if each clause contains at most two literals (*MAX-2-SAT*; see, e.g. [31, Theorem 13.11]). MAX-SAT and MAX-2-SAT are well-studied in the context of approximation algorithms (see, e.g. [2,11,17,20,25,38]). Recently, numerous results appeared in the domain of worst-case time bounds for the exact solution of MAX-SAT

[5] 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 representation of the input.

[6] 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".

and MAX-2-SAT [4,11,19,21,22,28,30]. The currently best bounds for MAX-SAT are $2^{K/2.36}$ and $2^{L/6.89}$ [4]. For MAX-2-SAT, the considerably better bounds $2^{K/2.88}$ [30] and $2^{K/3.44}$ (implicit in [4]) 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 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 $d \geqslant 5$, our algorithm does not improve a simple $2^{nd/(d+1)}$ bound [39]. We are not aware of previous non-trivial worst-case upper bounds for the exact solution of MAX-CUT, except for the parameterized bounds given by Mahajan and Raman [28]. Their results are a bound of 2^{2k} for the question of whether a given graph has a cut 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 $\lceil m/2 \rceil + k$.

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 (or edge, resp.) 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 [14,15]. In short, this procedure reduces the problem for a formula F to the problem for two formulas $F[v]$ and $F[\bar{v}]$ (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 labeled by F has two children labeled 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 poly($|F|$) 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 [4,30]. 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[7]

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

2-clauses (and maybe some 1-clauses) can be eliminated in polynomial time[8]. 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 $T(K) \leqslant 2T(K-5)$ 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, i.e., only variables occurring in four 2-clauses, corresponds to the recurrence inequality $T(K) \leqslant 2T(K-4)$.

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 $T(K) \leqslant 2T(K-4)$. Usually, this situation is handled by looking to the next level of splitting and showing that the obtained two instances are not "bottleneck" [23,30] which gives an inequality with an "intermediate" solution. 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 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 [18,19,21,22], 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 the DIMACS Workshop "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 organized 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 closely related 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 $\bar{v}$. *Literals* are variables and their negations. If l denotes a negated variable $\bar{v}$, then $\bar{l}$ denotes the variable v.

[8] 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. [6,34]).

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 non-empty 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. Thus, the operators $+$ and $-$ are defined:

$$F + G = \{(\omega_1 + \omega_2, S) \mid (\omega_1, S) \in F \text{ and } (\omega_2, S) \in G, \text{ and } \omega_1 + \omega_2 > 0\},$$

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

In other words, $+$ and $-$ denote the union and the difference of formulas considered as multisets of clauses.

Example 1. If

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

and

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

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

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

(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 $l\neq\bar{l}'$). For short, we write $F[l_1,\ldots,l_s]$ instead of $F[\{l_1,\ldots,l_s\}]$.

Example 2. If

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

then

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

The optimal value of a maximum weight assignment for formula F is defined as $\mathrm{OptVal}(F)=\max_A\{\,\omega\mid(\omega,\mathbb{T})\in F[A]\}$, 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 l 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 l. 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 $\bar{v}$.

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

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

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

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

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

if $l_2 \neq \bar{l_3}$, 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. [35]).

Lemma 4. *If F contains 2-clauses $C=(\omega_1,\{v,l_1\})$ and $D=(\omega_2,\{\bar{v},l_2\})$ 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 $\mathbf{T}_{\mathrm{DP}}$ replaces F with $(F-\{C,D\})+\mathfrak{R}(C,D)$ if F, C, and D satisfy the conditions of Lemma 4.

Dominating 1-clause: The following fact was observed in [30].

Lemma 5 (Niedermeier and Rossmanith [30]). *If for a literal l and a formula F, $\#_l^{(1)} \geqslant \#_{\bar{l}}$, then*

$$\mathrm{OptVal}(F)=\mathrm{OptVal}(F[l]). \tag{3}$$

Rule $\mathbf{T}_{\mathrm{dom}}$ replaces F with $F[l]$ 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,

$$\mathrm{OptVal}(F)=\mathrm{OptVal}(F-G)+\mathrm{OptVal}(G). \tag{4}$$

Rule $\mathbf{T}_{\mathrm{small}}$ 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 $\#_a^{(2)}=2$, $\#_{\bar{a}}^{(2)}=\#_a^{(1)}=0$, and $\#_{\bar{a}}^{(1)}=1$. Consider a 2-clause $(\omega,\{a,b\})$ in F. Rule $\mathbf{T}_{\mathrm{rare}}$ replaces this clause with $(\omega,\mathbb{T})$ and replaces literal a with literal $\bar{b}$ and literal $\bar{a}$ with literal b in all other clauses.

Lemma 6. *Rule $\mathbf{T}_{\mathrm{rare}}$ is correct.*

Proof. Let F' be the obtained formula. It is trivial that $\mathrm{OptVal}(F') \leqslant \mathrm{OptVal}(F)$. 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 $\mathbf{T}_{\mathrm{dom}}$ to the literal $\bar{a}$ in this formula, i.e.,

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

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

$$\begin{aligned}\text{OptVal}(F) &= \text{OptVal}(F[A]) \leqslant \text{OptVal}(F[\bar{b}])\\ &= \text{OptVal}(F[a,\bar{b}]) = \text{OptVal}(F'[a,\bar{b}]) \leqslant \text{OptVal}(F'). \quad \square\end{aligned}$$

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 $\text{poly}(|F|)\cdot 2^{K_2/5}$, 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) and $|F|$ is the length of representation of the input. We first present the algorithm and then estimate its running time and show its correctness using several lemmas.

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

Method: (A1) Apply $\mathbf{T}_{\text{pure}}$, $\mathbf{T}_{\text{ann}}$, $\mathbf{T}_{\text{DP}}$, $\mathbf{T}_{\text{dom}}$, $\mathbf{T}_{\text{small}}$, $\mathbf{T}_{\text{rare}}$ 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)[9], and return $\text{OptVal}(H_1) + \text{OptVal}(H_2) - 2$.

(A4) If F contains a variable v of weight at least five, then return $\max(\text{OptVal}(F[v]), \text{OptVal}(F[\bar{v}]))$.

(A5) If each variable has weight exactly four, then choose a variable v and return $\max(\text{OptVal}(F[v]), \text{OptVal}(F[\bar{v}]))$.

(A6) If F contains only variables of weight three and weight four, and both possibilities are realized, then choose[10] a variable v and determine correct transformation rules that modify $F[v]$ and $F[\bar{v}]$ into formulas F_1 and F_2 satisfying $K_2(F) - K_2(F_i) \geqslant 5$ $(i = 1,2)$ and containing a variable of weight at most three each; return $\max(\text{OptVal}(F_1), \text{OptVal}(F_2))$.

(A7) Choose[11] a variable v such that transformation rules modify $F[v]$ and $F[\bar{v}]$ into formulas F_1 and F_2 satisfying $K_2(F) - K_2(F_i) \geqslant 5$ $(i=1,2)$; return $\max(\text{OptVal}(F_1), \text{OptVal}(F_2))$.

[9] For the ease of presentation, we introduce new variables u and v not occurring in F in order to maintain the induction hypothesis in the proof of the following Theorem 10. Theorem 10 states our main result concerning the correctness and running time of Algorithm 1. Note that omitting these new variables here would not change the behavior of the algorithm, but would make it more involved to prove a bound on the worst-case running time in Theorem 10.

[10] Lemma 8 shows that a variable and transformation rules satisfying the requirements of step (A6) can be found in polynomial time.

[11] Lemma 9 shows that a variable and transformation rules satisfying the requirements of step (A7) can be found in polynomial time.

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

Lemma 7. *Let F be a formula, and let x be a variable of weight one or two. Then repeated application of transformation rules to x*

(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 8 shows how to find an appropriate variable and transformation rules at step (A6) of the algorithm. Lemma 9 shows the same for step (A7).

Lemma 8. *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 find a variable v and determine correct transformation rules that modify the formulas $F[v]$ and $F[\bar{v}]$ into formulas F_1 and F_2 such that for each $i=1,2$,

(1) $K_2(F)-K_2(F_i)\geqslant 5$, *and*
(2) F_i *contains a variable of weight exactly one, two, or three.*

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) of weight three that only occurs in the 2-clauses where y occurs. 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 $\mathbf{T}_{\text{small}}$ to $F[z]$ and $F[\bar{z}]$, resulting in F_1 and F_2. In this way, $K_2(F)-K_2(F_i)\geqslant 6$ for $i=1,2$, because we can eliminate all 2-clauses containing the variables y and z. If claim (2) is violated, i.e., F_i only contains variables that occur at least four times in 2-clauses, then we replace F_i with $F_i-(1,\mathbb{T})+(1,\{u_1,u_2\})$, where u_1 and u_2 are new variables (clearly, this modification is a correct transformation rule). Note that we can "subtract" $(1,\mathbb{T})$ because we can "spend" one of the $\mathbb{T}$-clauses that appear due to the last substitution: since we had $K_2(F)-K_2(F_i)\geqslant 6$ before, we have $K_2(F)-K_2(F_i)\geqslant 5$ after the modification, and claim (1) is still true. Note that u_1 now fulfills claim (2).

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 (2), we can choose any variable z occurring together with y, and occurring also together with a variable different from x and y. Note that if such z exists, then it has weight at most three in $F[y]$, but still has at least one occurrence there together with a variable different from x. Therefore, by Lemma 7(3), after the elimination of x, the variable z still occurs in the formula and has weight at most three.

Suppose now that such z does not exist. Then it cannot be the case that y occurs in 2-clauses together with x and at least two other variables, because one of the latter variables would have an occurrence together with a variable different from x and y. We now have that y occurs in 2-clauses together with x and only one other variable z'. Since F contains no closed subformulas, y must occur together with z' in 2-clauses of total weight three. Since the special case above does not apply, F should contain one more occurrence of z', and this is an occurrence together with x. Let z'' be the variable occurring in the remaining 2-clause containing x. If x is eliminated by $\mathbf{T}_{\mathrm{DP}}$ or $\mathbf{T}_{\mathrm{rare}}$, then z' still occurs in F_1 fulfilling claim (2). If, however, x is eliminated by $\mathbf{T}_{\mathrm{pure}}$ or $\mathbf{T}_{\mathrm{dom}}$, then exactly one 2-clause containing z'' disappears and therefore z'' fulfills claim (2).

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

Lemma 9. *Let F be a formula being split at step* (A7). *Then for any variable v we can find in polynomial time transformation rules that modify the formulas $F[v]$ and $F[\bar{v}]$ into formulas F_1 and F_2 satisfying $K_2(F) - K_2(F_i) \geqslant 5$ $(i = 1, 2)$.*

Proof. Note that by Lemma 7 and the conditions of steps (A1)–(A6), at step (A7) the formula F contains only variables of weight three. Therefore, $K_2(F) - K_2(F[v]) = 3$ and the formula $F[v]$ must contain two variables u and w such that u has weight exactly two and w has weight either one or two (note that F does not contain small closed subformulas). We now show how to find transformation rules that produce from $F[v]$ a formula F_1 such that $K_2(F_1) - K_2(F[v]) \geqslant 2$. (Modifying the formula $F[\bar{v}]$ into F_2 can be handled identically.)

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

Otherwise, we can apply $\mathbf{T}_{\mathrm{rare}}$ or $\mathbf{T}_{\mathrm{DP}}$ 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 w occurring in 2-clauses of weight one or two (see Lemma 7). Hence, we can now apply transformation rules to w that eliminate another 2-clause of weight one. □

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

Theorem 10. *Given a formula F in* 2-*CNF*, *Algorithm* 1 *finds* OptVal(F) *in time* $\mathrm{poly}(|F|) \cdot 2^{K_2/5}$, *where K_2 is the total weight of* 2-*clauses in F and $|F|$ is the length of representation of the input.*

Proof (Running time). Each of the transformation rules $\mathbf{T}_{\mathrm{pure}}$, $\mathbf{T}_{\mathrm{ann}}$, $\mathbf{T}_{\mathrm{DP}}$, $\mathbf{T}_{\mathrm{dom}}$, $\mathbf{T}_{\mathrm{small}}$, and $\mathbf{T}_{\mathrm{rare}}$ 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) in one of the steps (A3)–(A6), or (A7). 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 2-clauses is $O(2^{K_2/5})$.

First consider a formula F with K_2 2-clauses that forces our algorithm to make a recursive call at step (A3), (A4), (A6), or (A7). 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 (A5). At 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, we prove below that, for most such formulas, we still have $2\lambda(K_2 - 5)$ leaves and a different "odd" formula can occur at most once on each path from the root to a leaf. They can increase the size of the tree at most by a factor of 4. Therefore, we get the desired bound.

We now prove this claim about (A5). What may cause the application of (A5) 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 steps (A3)–(A6), or (A7).

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 (A5) 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 (A5): Setting the truth value of a variable clearly implies that, afterwards, another variable has weight 1, 2, or 3, because, at step (A5), F does not have non-trivial closed subformulas. It also cannot originate from (A7), since at this step the formula contains only variables of weight three, and weights cannot increase.

If F originated from (A4), 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).

Finally, we show that F could not originate from (A6). Assume that it did. Let G be the formula from which F originated. Then, F would contain a variable of weight one, two or three which contradicts the assumption that it contains only variables of weight four.

Correctness: The correctness of transformation rules $\mathbf{T}_{\text{pure}}$, $\mathbf{T}_{\text{ann}}$, $\mathbf{T}_{\text{DP}}$, $\mathbf{T}_{\text{dom}}$, $\mathbf{T}_{\text{small}}$, and $\mathbf{T}_{\text{rare}}$ is shown in Section 3. The correctness of steps (A2)–(A5) is trivial. At steps (A6) and (A7), we can find an appropriate variable v and determine correct transformation rules by Lemmas 8 and 9, respectively. □

In the case of unweighted[12] MAX-2-SAT, we have $L \geqslant 2K_2$. This directly implies the following corollary.

Corollary 11. *Given a formula F in unweighted 2-CNF of length L, Algorithm 1 finds* $\mathrm{OptVal}(F)$ *in time* $\mathrm{poly}(L) \cdot 2^{L/10}$.

Remark 12. Of course, in Corollary 11, only the number of literal occurrences in 2-clauses is essential in the exponent.

Remark 13. 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 that 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 [33]. 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 [33], 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.

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

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

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

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

Lemma 14. *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 $\mathbf{T}_{\text{rare}}$, 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 $\mathbf{T}_{\text{pure}}$ and $\mathbf{T}_{\text{dom}}$, we simply assign values to certain variables. Hence, the above discussion shows that these rules preserve (MCC). Rule $\mathbf{T}_{\text{ann}}$ 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 $\mathbf{T}_{\text{small}}$ also does not violate the property.

Only regarding $\mathbf{T}_{\text{DP}}$, 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, $\mathbf{T}_{\text{DP}}$ replaces these two clauses with $(w_1 + w_2, \mathbb{T})$ and, thus, (MCC) is not violated. □

To simplify the following proof of the worst-case time bound, 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; steps (A4)–(A6) are omitted; and at step (A7), the inequality now requires $K_2(F) - K_2(F_i) \geqslant 6$ (thus, we cannot use Lemma 9 and will have to show again how to find an appropriate variables and transformation rules).

We observe that the modifications covered in Lemma 14 are exactly those applied by our algorithm 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).

Theorem 15. *Given a formula F in 2-CNF satisfying* (*MCC*), *the modified Algorithm* 1 *finds* $\text{OptVal}(F)$ *in time* $\text{poly}(|F|) \cdot 2^{K_2/6}$, *where K_2 is the total weight of 2-clauses in F and $|F|$ is the length of representation of the input.*

Proof. In the proof of Theorem 10, 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 remains to prove that an appropriate variable and transformation rules at the modified step (A7) can be found.

In Lemma 14, 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 labeled 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 7(1)), because these are handled by the transformation rules. Therefore, every formula labeling a node of our splitting tree either contains a variable of weight at least six (this directly implies the required inequality $K_2(F) - K_2(F_i) \geqslant 6$ 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 $(2, \{\bar{a}, \bar{b}\})$ and, thus, there are no other 2-clauses containing variables x and y. In this situation, however, $\mathbf{T}_{\text{small}}$ 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 $\mathbf{T}_{\text{dom}}$ 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 (A6). □

Theorem 15 gives an upper bound for the running time of the modified algorithm on 2-CNF formulas derived from MAX-CUT instances. We now translate this result into numbers of vertices and edges of a graph.

Corollary 16. *Given a graph G having n vertices and edges of total weight M, we can solve (weighted) MAX-CUT in time* $\text{poly}(|G|) \cdot 2^{M/3}$, *where $|G|$ is the length of representation of the input. If an unweighted graph has maximum vertex degree three, then MAX-CUT is solvable in time* $\text{poly}(|G|) \cdot 2^{n/2}$, *and if the graph has maximum vertex degree four, it is solvable in time* $\text{poly}(|G|) \cdot 2^{2n/3}$.

Proof. Generating 2-CNF formulas from MAX-CUT instances, i.e., graphs with n vertices and edges of total weight M, gives 2-clauses of total weight $2M$ with n different variables. Then, the bound shown in Theorem 15 translates into a bound of $\text{poly}(|G|) \cdot 2^{2M/6} = \text{poly}(|G|) \cdot 2^{M/3}$ with respect to the total weight of the edges. The other two bounds follow from the inequality $m \leqslant dn/2$ relating n to the number m of edges and the maximum degree d. □

6. Discussion and open questions

Our bounds vs. parameterized 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 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 [8,16,28], there has been much research for *parameterized* 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 [4] give the best-known parameterized bound $2^{k/2.15}$ which is better than their "unparameterized" bound $2^{K/2.36}$ when $k < 0.92K$, where K is the total weight of all clauses. In [19], the parameterized bound $2^{k/2.73}$ for MAX-2-SAT has been proved. However, our present "unparameterized" bound $2^{K_2/5}$, where K_2 is the total weight of 2-clauses, is better for all reasonable values of k: the parameterized bound is better only when $k < 0.55K_2$, while an assignment satisfying $0.5K + 0.25K_2 \geqslant 0.75K_2$ clauses can be found in a polynomial time [28,38]. It seems like the idea of counting only 2-clauses does not work for parameterized bounds.

As $\lceil K/2 \rceil$ clauses can be easily satisfied, Mahajan and Raman [28] propose to ask in the parameterized version of the problem for an assignment satisfying $\lceil K/2 + k' \rceil$ clauses. Taking the parameterized bound shown in [19] 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 $\lceil K/2 + k' \rceil$ clauses or one can determine that no such assignment exists. However, for $k' \leqslant \lceil K_2/4 \rceil$, this question still can be handled in polynomial time. Comparing for $k' > \lceil K_2/4 \rceil$ the bound $2^{k'/0.45}$ to the bound shown for Algorithm 1, we see, again, that the parameterized 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 $\lceil K/2 + K_2/4 \rceil + k''$ 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 [23,30]. 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 $\varepsilon > 0$, an assignment satisfying $(1-\varepsilon)\mathrm{OptVal}(F)$ 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 [24].)

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 [9]. 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 [5].

From a more practical point of view, it would also be challenging to examine experimentally the efficiency of our algorithms. Previous results for exact MAX-2-SAT algorithms having guaranteed worst-case time bounds compared with an exact, heuristic algorithm [7] lacking guaranteed worst-case time bounds have shown encouraging results in this direction [18,19]. It is also interesting whether polynomial-time approximation algorithms (such as [17]) could be used in practice for pruning the search tree for some formulas; however, it is not clear if it is possible to use such algorithms for proving better worst-case upper bounds.

Acknowledgements

We thank the anonymous referees of *Discrete Applied Mathematics* who helped to improve the presentation of this paper.

References

[1] S. Arora, C. Lund, Hardness of approximation, in: D. Hochbaum (Ed.), Approximation Algorithms for NP-Hard Problems, Chapter 10, PWS Publishing Company, Boston, 1997, pp. 399–446.
[2] T. Asano, D.P. Williamson, Improved approximation algorithms for MAX SAT, in: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'00, 2000, pp. 96–105.
[3] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi. Complexity and Approximation—Combinatorial Optimization Problems and their Approximability Properties, Springer, Berlin, 1999.
[4] N. Bansal, V. Raman, Upper bounds for MaxSat: further improved, in: A. Aggarwal, C. Pandu Rangan (Eds.), Proceedings of the 10th Annual Conference on Algorithms and Computation, ISAAC'99, Lecture Notes in Computer Science, Vol. 174, Springer, Berlin, 1999, pp. 247–258.
[5] 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, 1999, pp. 856–857.
[6] P. Berman, M. Karpinski, On some tighter inapproximability results, in: Proceedings of the 26th International Colloquium on Automata, Languages and Programming, ICALP'99, Lecture Notes in Computer Science, Vol. 1644, Springer, Berlin, 1999, pp. 200–209.
[7] B. Borchers, J. Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems, J. Combin. Optim. 2 (4) (1999) 299–306.
[8] L. Cai, J. Chen, On fixed-parameter tractability and approximability of NP optimization problems, J. Comput. System Sci. 54 (1997) 465–474.
[9] J. Cheriyan, W.H. Cunningham, L. Tunçel, Y. Wang, A linear programming and rounding approach to Max 2-Sat, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 26 (1996) 395–414.
[10] E. Dantsin, Two propositional proof systems based on the splitting method, Zapiski Nauchnykh Seminarov LOMI, 105 (1981) 24–44 (in Russian). English translation: J. Soviet Math. 22 (3) (1983) 1293–1305 .
[11] E. Dantsin, M. Gavrilovich, E.A. Hirsch, B. Konev, MAX SAT approximation beyond the limits of polynomial-time approximation, Ann. Pure Appl. Logic 113 (2001) 81–94.
[12] 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, Theoret. Comput. Sci. (2002), to appear.
[13] E. Dantsin, A. Goerdt, E.A. Hirsch, U. Schöning, Deterministic algorithms for k-SAT based on covering codes and local search, in: Proceedings of the 27th International Colloquium on Automata, Languages and Programming, ICALP'00, Lecture Notes in Computer Science, Vol. 1853, Springer, Berlin, 2000, pp. 236–243.
[14] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Comm. ACM 5 (7) (1962) 394–397.
[15] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (3) (1960) 201–215.
[16] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer-Verlag, Berlin, 1999.

[17] U. Feige, M.X. Goemans, Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT, in: Proceedings of the Third Israel Symposium on Theory and Computing Systems, 1995, pp. 182–189.

[18] 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/.

[19] J. Gramm, R. Niedermeier, Faster exact solutions for Max-2-Sat, in: Proceedings of the Fourth Italian Conference on Algorithms and Complexity, CIAC 2000, Lecture Notes in Computer Science, Vol. 1767, Springer, Berlin, 2000, pp. 174–186.

[20] J. Håstad, Some optimal inapproximability results, in: Proceedings of the 29th Annual ACM Symposium on Theory of Computing, STOC'97, Journal of the ACM 48 (2001) 798–859.

[21] 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.

[22] E.A. Hirsch, A new algorithm for MAX-2-SAT, in: Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science, STACS 2000, Lecture Notes in Computer Science, Vol. 1770, Springer-Verlag, Berlin, February 2000, pp. 65–73 (contains an error, Fixed in [21]).

[23] E.A. Hirsch, New worst-case upper bounds for SAT, J. Automat. Reason. 24 (4) (2000) 397–420.

[24] 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, ICALP Workshops 2000, Proceedings in Informatics, Vol. 8, 2000, pp. 69–76.

[25] H. Karloff, 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, 1997, pp. 406–415.

[26] O. Kullmann, New methods for 3-SAT decision and worst-case analysis, Theoret. Comput. Sci. 223 (1–2) (1999) 1–72.

[27] O. Kullmann, H. Luckhardt, Algorithms for SAT/TAUT decision based on various measures. preprint, 71 pages, available from http://www.cs.toronto.edu/∼kullmann February 1999.

[28] M. Mahajan, V. Raman, Parameterizing above guaranteed values: MaxSat and MaxCut, J. Algorithms 31 (1999) 335–354.

[29] B. Monien, E. Speckenmeyer, Solving satisfiability in less then 2^n steps, Discrete Appl. Math. 10 (1985) 287–295.

[30] R. Niedermeier, P. Rossmanith, New upper bounds for MaxSat, J. Algorithms 36 (2000) 63–88.

[31] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.

[32] R. Paturi, P. Pudlák, M.E. Saks, 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, 1998, pp. 628–637.

[33] S. Poljak, Z. Tuza, Maximum cuts and large bipartite subgraphs, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 20 (1995) 181–244.

[34] V. Raman, B. Ravikumar, S. Srinivasa Rao, A simplified NP-complete MAXSAT problem, Inform. Process. Lett. 65 (1) (1998) 1–6.

[35] J.A. Robinson, Generalized resolution principle, Mac. Intell. 3 (1968) 77–94.

[36] 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, 1999, pp. 410–414.

[37] R. Schuler, U. Schöning, O. Watnabe, T. Hofemister, A probabilistic {3-SAT} algorithm further improved, Proceedings of 19th International Symposium on Theoretical Aspects of Computer Science, STACS 2002, Lecture Notes in Computer Science, 2285, Springer, 2002, 192–202.

[38] M. Yannakakis, On the approximation of maximum satisfiability, J. Algorithms 17 (3) (1994) 457–502.

[39] U. Zwick, personal communication, 2000.