MAX SAT approximation beyond the limits of polynomial-time approximation

Evgeny Dantsin* Michael Gavrilovich† Edward A. Hirsch‡ Boris Konev§

Abstract

We describe approximation algorithms for (unweighted) MAX SAT with performance ratios arbitrarily close to 1, in particular, when performance ratios exceed the limits of polynomial-time approximation. Namely, given a polynomial-time α-approximation algorithm $\mathcal{A}_0$, we construct an $(\alpha+\epsilon)$-approximation algorithm $\mathcal{A}$. The algorithm $\mathcal{A}$ runs in time of the order $c^{\epsilon k}$, where k is the number of clauses in the input formula and c is a constant depending on α. Thus we estimate the cost of improving a performance ratio. Similar constructions for MAX 2SAT and MAX 3SAT are also described. Taking known algorithms as $\mathcal{A}_0$ (for example, the Karloff–Zwick algorithm for MAX 3SAT), we obtain particular upper bounds on the running time of $\mathcal{A}$.

1 Introduction

In the MAX SAT problem we are given a Boolean formula in conjunctive normal form and we seek a truth assignment that maximizes the number of satisfied clauses. An α-approximation algorithm for MAX SAT is an algorithm that finds an assignment satisfying at least a fraction α of the maximal number of simultaneously satisfied clauses. Most recent achievements in this area are connected with two directions: polynomial-time approximation algorithms for MAX SAT [10, 9, 15, 2] and limits of polynomial-time approximation ([1, 12] for example).

Karloff and Zwick [15] presented a 7/8-approximation algorithm for MAX 3SAT. On the other hand, Håstad [12] proved that for any $\epsilon > 0$, there is no polynomial-time $(7/8+\epsilon)$-approximation algorithm for MAX 3SAT unless $\mathbf{P} = \mathbf{NP}$. Thus any $(7/8+\epsilon)$-approximation algorithm for MAX 3SAT is believed to run in exponential time. How does the running time of such an algorithm depend on ϵ? Can we find a $(7/8+\epsilon)$-approximation solution faster than an exact solution (even if both take exponential time to be found)? We answer these questions by constructing a $(7/8+\epsilon)$-approximation algorithm for MAX 3SAT that runs in time $O(2^{8\epsilon k})$, where k is the number of clauses in the input formula.

Applying similar techniques to the MAX SAT problem, we construct an $(\alpha+\epsilon)$-approximation algorithm for MAX SAT from a given polynomial-time MAX SAT α-approximation algorithm. The constructed algorithm runs in time $l^{O(1)}\phi^{\epsilon(1-\alpha)^{-1}k}$, where ϕ is the golden ratio (≈ 1.618) and l

*Department of Computer Science, University of Manchester, UK. On leave from Steklov Institute of Mathematics, St.Petersburg, Russia. Email: dantsin@pdmi.ras.ru. Supported in part by grants from EPSRC and INTAS.

†St.Petersburg State University, Russia. Email: misha@logic.pdmi.ras.ru.

‡Steklov Institute of Mathematics, St.Petersburg, Russia. Web: http://logic.pdmi.ras.ru/~hirsch/, email: hirsch@pdmi.ras.ru. Supported in part by grants from INTAS and RFBR.

§Steklov Institute of Mathematics, St.Petersburg, Russia. Email: konev@pdmi.ras.ru. Supported in part by grants from INTAS and RFBR.

is the length of the input formula. In particular, taking the polynomial-time 0.784-approximation algorithm for MAX SAT by Asano and Williamson [2], we obtain a $(0.784+\epsilon)$-approximation algorithm for MAX SAT that runs in time $l^{O(1)}\phi^{4.630\epsilon k}$. As a by-product, we obtain an exact algorithm for MAX SAT running in time $l^{O(1)}\phi^k$.

For MAX 2SAT, we construct a $(0.931+\epsilon)$-approximation algorithm from the polynomial-time 0.931-approximation algorithm by Feige and Goemans [9] in a similar way.

To construct an $(\alpha+\epsilon)$-approximation algorithm from an α-approximation algorithm, we employ the *splitting method* going back to [8] and used in most SAT algorithms (this method was also applied to MAX SAT, for example, in [4]). The splitting method transforms the input formula F into (exponentially many) formulas $F_1,\ldots,F_m$. Applying a polynomial-time α-approximation algorithm to $F_1,\ldots,F_m$, we obtain α-approximation solutions for these formulas. These solutions are then used to construct an $(\alpha+\epsilon)$-approximation solution for F.

Related results. The main results of this paper appeared in [6]. Independently of our work, Mahajan and Raman [17] suggested a $l^{O(1)}\phi^k$-time algorithm for MAX SAT. They also observed that this algorithm satisfies k' clauses in time $l^{O(1)}\phi^{k'}$ (even if k' is substantially less than the total number k of clauses). Since then, there has been a substantial progress in proving worst-case upper bounds for MAX SAT and its subproblems [3, 19, 11]. Currently, the best known bounds are $l^{O(1)}1.342^k$, $l^{O(1)}1.381^{k'}$, $l^{O(1)}1.106^l$ for MAX SAT [3], and $l^{O(1)}2^{k/5}$, $l^{O(1)}2^{l/10}$ for MAX 2SAT [11]. As indicated in [19], some techniques used in proving these bounds can be also applied in our setting to get better bounds for approximation algorithms.

Another relevant work [14] presents a $(1-\epsilon)$-approximation algorithm for MAX kSAT running in time $l^{O(1)}c^n$, where n is the number of variables, and $c<2$ is a constant that depends on k and ϵ.

The techniques underlying the mentioned results mostly come from proofs of worst-case upper bounds for SAT [5, 18, 16, 20, 13, 7].

Organization of the paper. Section 2 contains the basic definitions and notation we use. In Section 3 we demonstrate the application of the splitting method to MAX SAT and we estimate the running time of algorithms based on the splitting method. For this purpose, we describe an exact MAX SAT algorithm whose running time is $|F|^{O(1)}\cdot\phi^k$. In Section 4 we present our main result, namely we construct an $(\alpha+\epsilon)$-approximation algorithm from a polynomial-time α-approximation algorithm. Using known polynomial-time approximation algorithms for MAX SAT, MAX 2SAT and MAX 3SAT, we obtain the respective bounds on the running time of our algorithm.

2 Basic notation

Notation for Boolean formulas.

Literals are Boolean variables and their negations. If u is a literal, the complementary literal is denoted by $\overline{u}$. A *clause* is a finite set of literals that contains no pair of complementary literals. A *formula* is a finite multiset of clauses. Note that we define a formula as a multiset of clauses, not as a set. A clause is interpreted as the disjunction of its literals and a formula is thought of as the conjunction of the corresponding disjunctions.

Let F be a formula. The number of clauses in F is denoted by $\mathcal{K}(F)$. The *length* of F, denoted by $|F|$, is the sum of cardinalities of its clauses. By $\#_u(F)$ we denote the number of occurrences of

a literal u in F, i.e. exactly $\#_u(F)$ clauses in F contain u. A literal u is said to be an (m,n)-*literal* if $\#_u(F) = m$ and $\#_{\overline{u}}(F) = n$.

For a variable x, we define formulas $F[x]$ and $F[\overline{x}]$ as the respective results of substitution of the truth values *True* and *False* for x. Namely, for a literal u, we denote by $F[u]$ the formula obtained from F by removing all clauses that contain u and removing $\overline{u}$ from all other clauses.

Truth assignments and satisfiability.

Like a clause, an *assignment* is defined as a finite set of literals that contains no complementary literals, but the interpretation is different. Let A be an assignment and x be a variable. We consider that A assigns to x the truth value *True* if $x \in A$, and assigns *False* if $\overline{x} \in A$. If neither x nor $\overline{x}$ belongs to A, the value of x is undefined.

Let F be a formula and A be an assignment. We say that A *satisfies* a clause C if $A \cap C \neq \emptyset$. The number of clauses in F satisfied by A is denoted by $\mathsf{Eval}(F, A)$. If A satisfies all clauses in F, we say that A *satisfies* F. In particular, the empty formula is satisfied by any assignment and we denote this formula by *True*. The formula containing only the empty clauses is satisfied by no assignment and we denote it by *False*. The following problem is denoted by SAT: given a formula F, determine whether F is satisfiable and, if yes, find any satisfying assignment.

MAX SAT **and its approximation.**

An assignment A is said to be *optimal* for a formula F if A satisfies the maximal number of clauses in F, i.e. $\mathsf{Eval}(F, A) \geq \mathsf{Eval}(F, A')$ for any assignment A'. The number of clauses satisfied by an optimal assignment is denoted by $\mathsf{Opt}(F)$. By MAX SAT we mean the problem: given F, find any optimal assignment. The problems MAX 2SAT and MAX 3SAT differ from MAX SAT in restrictions on the input. Namely, every clause in F contains at most 2 literals in the case of MAX 2SAT and at most 3 literals in MAX 3SAT. An algorithm that produces on an input formula F an assignment A_F is called an *exact* algorithm for MAX SAT (or for a related problem) if $\mathsf{Eval}(F, A_F) = \mathsf{Opt}(F)$ for all F. A *randomized α-approximation* algorithm for MAX SAT (or for a related problem) is defined as a randomized algorithm that produces on an input formula F an assignment A_F such that

$$\mathbf{E}[\mathsf{Eval}(F, A_F)] \geq \alpha \cdot \mathsf{Opt}(F),$$

where $\mathbf{E}$ denotes the expectation. The infimum of the ratio $\mathbf{E}[\mathsf{Eval}(F, A_F)/\mathsf{Opt}(F)]$ over all F is called the *performance ratio* of $\mathcal{A}$.

For simplicity, we assume that the input formula does not contain empty clauses.

3 Exact algorithm for MAX SAT

Many algorithms for SAT are based on successive reductions of formulas to simpler ones. The most natural example is the following: satisfiability of a formula F is reduced to satisfiability of formulas $F[x]$ and $F[\overline{x}]$ for some variable x in F, each of $F[x]$ and $F[\overline{x}]$ reduces in a similar way, and further reductions proceed until we obtain formulas *True* or *False* without variables. Such a process can be represented by a tree, called a *reduction tree*, in which the root corresponds to the input formula and other nodes correspond to formulas obtained by reductions. In the example above, any reduction tree is a binary tree consisting of less than 2^{n+1} nodes, where n is the number of variables in the input formula.

Using successive reductions, the satisfiability problem for F can be solved in time $|F|^{O(1)} \cdot t$, where t is the number of nodes in the constructed reduction tree. Various upper bounds on t were obtained in recent years, for example Hirsch [13] showed that any formula F in CNF has a reduction tree with at most $O(1.2389^{\mathcal{K}(F)})$ nodes (see [16] for a survey of related results).

In this section we use the approach of successive reductions to find an exact solution for MAX SAT. We present an algorithm and prove its soundness and an upper bound on its running time. The technique used in the proof will be developed in next sections to obtain upper bounds for approximation algorithms.

Theorem 1. *There exists a deterministic algorithm for* MAX SAT *whose running time is*

$$|I|^{O(1)} \cdot \phi^k,$$

where I is an input formula, k is the number of clauses in I, and ϕ is the golden ratio ($\phi = \frac{1}{2}(1+\sqrt{5}) \approx 1.618$).

Proof. The required algorithm is described in Section 3.1. Its soundness and the bound on its running time are proven in Sections 3.2 and 3.3 respectively. □

3.1 Algorithm

The algorithm starts by constructing a *reduction tree* for the input formula I. To define such a tree, we describe three kinds of operations on formulas.

Splitting. Let F be a formula and u be a literal in F. We say that formulas $F[u]$ and $F[\overline{u}]$ are obtained from F by *splitting* F with respect to u.

Pure literal elimination. A literal u is said to be *pure* in a formula F if at least one clause in F contains u and no clause in F contains $\overline{u}$. If u is a pure literal in F, we say that $F[u]$ is obtained from F by *pure literal elimination.*

Elimination of a $(1,1)$-literal. Let u be a $(1,1)$-literal in a formula F. Let C_1 and C_2 be clauses in F such that $u \in C_1$ and $\overline{u} \in C_2$. If the set $(C_1 \setminus \{u\}) \cup (C_2 \setminus \{\overline{u}\})$ contains no pair of complementary literals, this set is called a *resolvent* of C_1 and C_2. In this case, we define a formula F' as the result of replacing C_1 and C_2 in F by their resolvent. Otherwise, i.e. when $(C_1 \setminus \{u\}) \cup (C_2 \setminus \{\overline{u}\})$ contains a pair of complementary literals, we define F' as the formula obtained by removing C_1 and C_2 from F. In both cases we say that F' is obtained from F by $(1,1)$*-literal elimination.*

Now we define a *reduction tree* as follows. Let F be a formula. Consider a tree T in which each node N is labeled by a formula F_N such that the following conditions hold:

1. The root of T is labeled by F.
2. Each leaf is labeled by *True* or *False*.
3. For each non-leaf node N, there are two alternatives:
   (a) The node N has one child N'. The formula $F_{N'}$ is obtained from F_N by either pure literal elimination or $(1,1)$-literal elimination.

(b) The node N has two children N_1 and N_2. The formulas F_{N_1} and F_{N_2} are obtained from F_N by splitting with respect to an (m,n)-literal, where both m and n are positive and at least one of them is greater than 1.

Any such tree T is said to be a *reduction tree* for F.

Description of algorithm.

The first step of our algorithm is to construct a reduction tree T for the input formula I. At the second step, the algorithm puts one more label on each node of T. Namely, moving from the leaves to the root, the algorithm labels each node N by an assignment A_N computed as follows.

Each leaf in T is labeled by the empty assignment. For a non-leaf node N, there are the following three cases.

Case 1. The node N has a single child N' and $F_{N'}$ is obtained from F_N by elimination of a pure literal u. Then we define A_N as $A_{N'} \cup \{u\}$.

Case 2. The node N has a single child N' and $F_{N'}$ is obtained from F_N by elimination of a $(1,1)$-literal u. Then there are exactly two clauses C_1 and C_2 such that $u \in C_1$ and $\overline{u} \in C_2$. The assignment A_N is defined depending on whether C_1 and C_2 contain another pair of complementary literals, different from u and $\overline{u}$.

Assume first that there is no other pair of complementary literals. Then A_N is one of the assignments $A_{N'} \cup \{u\}$ and $A_{N'} \cup \{\overline{u}\}$ chosen so as to maximize the number of satisfied clauses.

Assume now that there are literals v and $\overline{v}$, different from u and $\overline{u}$, such that $v \in C_1$ and $\overline{v} \in C_2$. If neither v nor $\overline{v}$ belongs to $A_{N'}$, we define A_N as one of the assignments $A_{N'} \cup \{u\} \cup \{\overline{v}\}$ and $A_{N'} \cup \{\overline{u}\} \cup \{v\}$ chosen so as to maximize the number of satisfied clauses. When either of v and $\overline{v}$ belongs to $A_{N'}$, we define A_N as follows: A_N is $A_{N'} \cup \{u\}$ if $\overline{v} \in A_{N'}$ and A_N is $A_{N'} \cup \{\overline{u}\}$ if $v \in A_{N'}$.

Case 3. The node N has two children N_1 and N_2 labeled by formulas $F_N[x]$ and $F_N[\overline{x}]$. Then we define A_N as one of the assignments $A_{N_1} \cup \{x\}$ and $A_{N_2} \cup \{\overline{x}\}$ chosen so as to maximize the number of satisfied clauses.

Finally, the algorithm returns the assignment computed for the root of T. It is easy to implement this algorithm as an algorithm running within polynomial space and in time $|I|^{O(1)} \cdot t$, where t is the number of nodes in T.

3.2 Soundness

Let N be the root of the constructed reduction tree T. Using induction on the height of T, we prove that A_N is an optimal assignment for F_N, i.e. $\mathsf{Opt}(F_N) = \mathsf{Eval}\,(F_N, A_N)$. The assertion is trivial when T consists of a single node. Consider three cases corresponding to the types of reductions.

Pure literal elimination. The root N has a single child N' and $F_{N'}$ is obtained from F_N by elimination of a pure literal u. Since we have $\mathsf{Opt}(F_{N'}) = \mathsf{Eval}\,(F_{N'}, A_{N'})$ by the inductive assumption, we obtain

$$\mathsf{Opt}(F_N) = \mathsf{Opt}(F_N[u]) + \#_u(F_N) =$$
$$\mathsf{Eval}\,(F_{N'}, A_{N'}) + \#_u(F_N) = \mathsf{Eval}\,(F_N, A_N)\,.$$

Elimination of a $(1,1)$-literal. The root N has a single child N' and $F_{N'}$ is obtained from F_N by $(1,1)$-literal elimination. A simple consideration of possible cases shows that $\mathsf{Opt}(F_N) = \mathsf{Opt}(F_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'})$. On the other hand, we have $\mathsf{Eval}\,(F_N, A_N) = \mathsf{Eval}\,(F_{N'}, A_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'})$ by the construction of A_N from $A_{N'}$. Using the inductive assumption, we obtain the required equality $\mathsf{Opt}(F_N) = \mathsf{Eval}\,(F_N, A_N)$.

Splitting. The root N has two children N_1 and N_2 labeled by formulas $F_N[u]$ and $F_N[\overline{u}]$. Since F_N has an optimal assignment containing u or $\overline{u}$, we have

$$\mathsf{Opt}(F_N) = \max\left(\mathsf{Opt}(F_N[u]) + \#_u(F_N),\ \mathsf{Opt}(F_N[\overline{u}]) + \#_{\overline{u}}(F_N)\right).$$

By the inductive assumption, $\mathsf{Opt}(F_N[u]) = \mathsf{Eval}\,(F_N[u], A_{N_1})$ and $\mathsf{Opt}(F_N[\overline{u}]) = \mathsf{Eval}\,(F_N[\overline{u}], A_{N_2})$. Therefore, $\mathsf{Opt}(F_N)$ is equal to

$$\max\left(\mathsf{Eval}\,(F_N[u], A_{N_1}) + \#_u(F_N),\ \mathsf{Eval}\,(F_N[\overline{u}], A_{N_2}) + \#_{\overline{u}}(F_N)\right),$$

and we have

$$\mathsf{Opt}(F_N) = \max(\mathsf{Eval}\,(F_N, A_{N_1} \cup \{u\})\,, \mathsf{Eval}\,(F_N, A_{N_2} \cup \{\overline{u}\})).$$

Since we choose A_N so as to maximize the number of satisfied clauses by $A_{N_1} \cup \{u\}$ and $A_{N_2} \cup \{\overline{u}\}$, the assertion holds in this case too. This completes the proof of soundness.

3.3 Running time

Our algorithm runs in time $|I|^{O(1)} \cdot t$, where t is the number of nodes in T. Since each reduction decreases the number of clauses, the number t does not exceed $|I| \cdot l(T)$, where $l(T)$ denotes the number of leaves in T. To find an upper bound on $l(T)$, we use Fibonacci numbers $\mathcal{F}_i$ defined by the equalities $\mathcal{F}_0 = 0$, $\mathcal{F}_1 = 1$ and $\mathcal{F}_{i+2} = \mathcal{F}_{i+1} + \mathcal{F}_i$.

Consider all formulas with k clauses and all possible reduction trees for them. Let l_k denote the maximal number of leaves in these trees. We prove that $l_k \leq \mathcal{F}_{k+1}$. Indeed, if $k = 1$, then $l_k = 1$ and $\mathcal{F}_{k+1} = 1$. Let F be a formula with $k > 1$ clauses and T be a reduction tree for F. Then one of the following alternatives holds.

One child. The root of T has one child labeled by a formula F'. Obviously, the number of clauses in F' does not exceed $k-1$. Therefore, the number of leaves in T is not greater than l_{k-1}, which does not exceed $\mathcal{F}_k$ by the inductive assumption. Since $\mathcal{F}_k \leq \mathcal{F}_{k+1}$, the number of leaves in such a tree is bounded by $\mathcal{F}_{k+1}$.

Two children. The root of T has two children labeled by $F[u]$ and $F[\overline{u}]$, where u is an (m,n)-literal. It is easy to see that the number of clauses in $F[u]$ is at most $k-m$ and the number of clauses in $F[\overline{u}]$ is at most $k-n$. Therefore, the number of leaves in T does not exceed $l_{k-m} + l_{k-n}$. Since both m and n are positive and at least one of them is greater than 1, the number of leaves in T is at most

$$l_{k-m} + l_{k-n} \leq l_{k-1} + l_{k-2} \leq \mathcal{F}_k + \mathcal{F}_{k-1} = \mathcal{F}_{k+1}.$$

Thus, the number of leaves in any reduction tree for F does not exceed $\mathcal{F}_{k+1}$ and we have $l_k \leq \mathcal{F}_{k+1}$. It is well known that $\mathcal{F}_{k+1} \leq \phi^k$ (easy to prove by induction on k). Hence, we obtain $l_k \leq \phi^k$ and the required bound on the running time.

4 Approximation algorithms for MAX SAT, MAX 2SAT, and MAX 3SAT

In this section we prove two theorems. Theorem 2 and its corollaries describe approximation algorithms for MAX SAT and MAX 2SAT. Theorem 3 and its corollary describe an approximation algorithm for MAX 3SAT.

Recall that by I and k we denote the input formula and the number of clauses in it. Also recall that ϕ denotes the golden ratio (≈ 1.618).

Theorem 2. *Suppose we are given a polynomial-time randomized α-approximation algorithm for* MAX SAT *(*MAX 2SAT*). Then, for any positive $\epsilon \leq 1 - \alpha$, we can construct a randomized $(\alpha+\epsilon)$-approximation algorithm for* MAX SAT *(respectively* MAX 2SAT*) whose running time is*

$$|I|^{O(1)} \cdot \phi^{\epsilon(1-\alpha)^{-1}k}.$$

Proof. We describe the required $(\alpha + \epsilon)$-approximation algorithm in Section 4.1 and prove its soundness and the bound on the running time in Sections 4.2 and 4.3. □

Corollary 1. *For any positive $\epsilon \leq 0.216$, there is a randomized $(0.784+\epsilon)$-approximation algorithm for* MAX SAT *whose running time is*

$$|I|^{O(1)} \cdot \phi^{4.630\epsilon k}.$$

Proof. We obtain this corollary, using the polynomial-time randomized 0.784-approximation algorithm for MAX SAT ([2]) as the α-approximation algorithm. □

Corollary 2. *For any positive $\epsilon \leq 0.069$, there is a randomized $(0.931+\epsilon)$-approximation algorithm for* MAX 2SAT *whose running time is*

$$|I|^{O(1)} \cdot \phi^{14.493\epsilon k}.$$

Proof. In [9] a 0.931-approximation algorithm for MAX 2SAT is presented. □

Theorem 3. *Suppose we are given a polynomial-time randomized α-approximation algorithm for* MAX 3SAT*. Then, for any positive $\epsilon \leq 1-\alpha$, we can construct a randomized $(\alpha+\epsilon)$-approximation algorithm for* MAX 3SAT *whose running time is*

$$|I|^{O(1)} \cdot 2^{\epsilon(1-\alpha)^{-1}k}.$$

Proof. See Section 4.4. □

Corollary 3. *For any positive $\epsilon \leq 1/8$, there is a randomized $(7/8 + \epsilon)$-approximation algorithm for* MAX 3SAT *whose running time is*

$$|I|^{O(1)} \cdot 2^{8\epsilon k}.$$

Proof. We use the polynomial-time randomized 7/8-approximation algorithm for MAX 3SAT ([15]) as the α-approximation algorithm. □

4.1 Algorithm

Like the exact algorithm in Section 3, our approximation algorithm starts by constructing a reduction tree T for an input formula I. However, this algorithm constructs only a part T' of the entire tree T. Namely, the approximation algorithm starts with the root and constructs descendants as in Section 3.1 while formulas consist of at least k_0 clauses and at least one of the reductions is applicable, where

$$k_0 = \lfloor k - \epsilon(1-\alpha)^{-1}k \rfloor$$

(the choice of this value of k_0 will become clear below). Otherwise, i.e. when $\mathcal{K}(F_N) < k_0$ or all clauses in F_N are empty, the node N is considered to be a leaf in the tree T'. Thus, T' can be viewed as a part of a reduction tree obtained by deleting some subtrees.

For each node N of T', our approximation algorithm computes an assignment A_N. This is similar to the case of the exact algorithm and the only difference is in assignments for the leaves. If no reduction is applicable, then the formula F_N contains no non-empty clause, i.e., F_N is *False*. In this case we set A_N to be the empty assignment. Note that this assignment is optimal. Otherwise, if F_N contains a non-empty clause, the assignment A_N is computed by the polynomial-time α-approximation algorithm from the statement of the theorem.

Computation of the assignments for the non-leaf nodes is performed in the same way as in the exact algorithm. The assignment computed for the root of T' is considered to be the output of our approximation algorithm. Like the exact algorithm, the approximation one can be implemented as an algorithm running within polynomial space.

We think of the randomized α-approximation algorithm as a deterministic algorithm computing a function of two arguments: an input formula F and a string τ of random bits. Thus, we have a polynomial-time deterministic algorithm $\mathcal{A}_0$ that outputs an assignment $\mathcal{A}_0(F,\tau)$ on given F and τ. Like the deterministic algorithm, the randomized algorithm first constructs the tree T' for input formula. Let $L_1, \ldots, L_s$ denote the leaves of the tree T'. Then the algorithm generates a string τ of random bits and runs $\mathcal{A}_0$ on all the formulas $F_{L_1}, \ldots, F_{L_s}$ corresponding to the leaves $L_1, \ldots, L_s$ to compute

$$\mathcal{A}_0(F_{L_1}, \tau), \ldots, \mathcal{A}_0(F_{L_s}, \tau)$$

and uses these outputs as $A_{L_1}, A_{L_2}, \ldots, A_{L_s}$.

Note that we construct the splitting tree in a deterministic way. Thus if the α-approximation algorithm that we use at the leaves can be derandomized, our algorithm can be derandomized as well.

4.2 Soundness

From Lemmas 1 and 2 below it follows that the assignment computed for the root of T' satisfies at least $(\alpha+\epsilon)\cdot \mathsf{Opt}(I)$ clauses in I.

Lemma 1. *Let R be the root of T'. The tree T' has a leaf L such that*

$$\mathsf{Eval}\,(F_R, A_R) \geq \mathsf{Eval}\,(F_L, A_L) + \mathcal{K}(F_R) - \mathcal{K}(F_L) \tag{1}$$

$$\mathsf{Opt}(F_R) \leq \mathsf{Opt}(F_L) + \mathcal{K}(F_R) - \mathcal{K}(F_L) \tag{2}$$

Proof. The inequality (1) follows from the fact that for each node N and its child N', we have

$$\mathsf{Eval}\,(F_N, A_N) - \mathsf{Eval}\,(F_{N'}, A_{N'}) \geq \mathcal{K}(F_N) - \mathcal{K}(F_{N'}). \tag{3}$$

Similarly, to prove the inequality (2), it suffices to prove that each non-leaf N has a child N' such that

$$\mathsf{Opt}(F_N) \leq \mathsf{Opt}(F_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'}). \tag{4}$$

These inequalities are proved by straightforward consideration of the types of reductions similarly to Section 3.2. We put the proofs in an appendix. □

Lemma 2. *Let R be the root of T'. Then $(\alpha+\epsilon)\cdot\mathsf{Opt}(F_R) \leq \mathbf{E}[\mathsf{Eval}\,(F_R, A_R)]$.*

Proof. Let L be a leaf such that (1) and (2) hold, and let $\mathcal{A}(I,\tau)$ denote the output of the constructed $(\alpha+\epsilon)$-approximation algorithm on an input formula I and a string τ of random bits. By the first inequality in Lemma 1, the number of clauses satisfied by $\mathcal{A}(I,\tau)$ is greater than or equal to $\mathsf{Eval}\,(F_L, \mathcal{A}_0(F_L,\tau)) + \mathcal{K}(I) - \mathcal{K}(F_L)$ for any τ.

If all clauses in F_L are empty, then by Lemma 1 we have

$$\begin{aligned}&\mathsf{Opt}(F_R) \leq \mathsf{Opt}(F_L) + \mathcal{K}(F_R) - \mathcal{K}(F_L) = \\ &\mathcal{K}(F_R) - \mathcal{K}(F_L) \leq \mathsf{Eval}\,(F_R, A_R)\,.\end{aligned}$$

Now assume that F_L contains a non-empty clause. Since A_L is the assignment computed by the α-approximation algorithm for F_L, we have

$$\mathbf{E}[\mathsf{Eval}\,(F_L, A_L)] \geq \alpha\cdot\mathsf{Opt}(F_L). \tag{5}$$

We thus obtain

$$\begin{aligned}&\mathbf{E}[\mathsf{Eval}\,(F_R, A_R)] \geq \alpha\cdot\mathsf{Opt}(F_L) + \mathcal{K}(F_R) - \mathcal{K}(F_L) \geq \\ &\alpha\cdot(\mathsf{Opt}(F_R) - (\mathcal{K}(F_R) - \mathcal{K}(F_L))) + \mathcal{K}(F_R) - \mathcal{K}(F_L) = \\ &\alpha\cdot\mathsf{Opt}(F_R) + (1-\alpha)(\mathcal{K}(F_R) - \mathcal{K}(F_L)).\end{aligned}$$

Therefore,

$$\begin{aligned}&\alpha\cdot\mathsf{Opt}(F_R) + (1-\alpha)(\mathcal{K}(F_R) - \mathcal{K}(F_L)) \geq \\ &\alpha\cdot\mathsf{Opt}(F_R) + (1-\alpha)(k-k_0) \geq \\ &\alpha\cdot\mathsf{Opt}(F_R) + \epsilon k \geq \\ &(\alpha+\epsilon)\cdot\mathsf{Opt}(F_R).\end{aligned}$$

□

4.3 Running time

As in the case of the exact algorithm in Section 3.3, we estimate the running time of our algorithm by estimating the number of leaves in T'. We show that the number of leaves in T' is not greater than $\mathcal{F}_{k+1-k_0}$, where $k_0 = \lfloor k - \epsilon(1-\alpha)^{-1}k\rfloor$. To show it, we prove the inequality $l_k \leq \mathcal{F}_{k+1-k_0}$ (for $k \geq k_0$) instead of $l_k \leq \mathcal{F}_{k+1}$ in the proof of Theorem 1. This inequality is proved by induction on k, beginning with k_0.

Let $k = k_0$, i.e. the root of T' is labeled by a formula having at most k_0 clauses. Then the tree T' consists of a single node. Therefore, $l_{k_0} = \mathcal{F}_{k+1-k_0} = 1$ and the induction basis is proved.

For $k > k_0$, we have the same reductions as in the tree T in Section 3.3. Therefore, the inequality $l_{k+1} \leq l_{k-1} + l_k$ can be proved in the same way. Thus, we obtain $l_k \leq \mathcal{F}_{k+1-k_0} \leq \phi^{k-k_0}$. Substituting the value of k_0, we come to the required bound. The running time does not depend on choice of τ.

4.4 Proof of Theorem 3

The $(\alpha+\epsilon)$-approximation algorithm for MAX 3SAT is similar to its counterparts for MAX SAT and MAX 2SAT in Theorem 2 and differs only in the following. While the algorithm in Theorem 2 uses three kinds of reductions (namely splitting, pure literal elimination and $(1,1)$-literal elimination), the MAX 3SAT algorithm uses only the first two of them. This is connected with the fact that resolution may increase the number of literals in clauses. Thus, the algorithm for MAX 3SAT constructs a reduction tree by means of splitting and pure literal elimination. Splitting is allowed for any (m,n)-literal where m and n are positive.

The proof of soundness is similar to the proof in Section 4.2. The proof of the running time bound differs. Since $(1,1)$-literal elimination is not used now, the inequality $l_{k+1} \leq l_k + l_{k-1}$ does not hold. Instead, the inequality $l_{k+1} \leq l_k + l_k$ obviously holds. Therefore, we obtain the required bound with 2 instead of ϕ in the base of exponent.

References

[1] S. Arora and C. Lund. Hardness of approximation. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, chapter 10. 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, *Proceedings of the 10th International Symposium on Algorithms and Computation, ISAAC'99*, volume 1741 of *Lecture Notes in Computer Science*, pages 247–258. Springer, December 1999.

[4] 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.

[5] 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.

[6] E. Dantsin, M. Gavrilovich, E. Hirsch, and B. Konev. Approximation algorithms for MAX SAT: a better performance ratio at the cost of a longer running time. Preprint 14/1998, Steklov Institute of Mathematics at St.Petersburg, May 1998. http://www.pdmi.ras.ru/preprint/1998/index.html.

[7] E. Dantsin, A. Goerdt, E. A. Hirsch, and U. Schöning. Deterministic algorithms for k-SAT based on covering codes and local search. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming, ICALP'2000*, volume 1853 of *Lecture Notes in Computer Science*. Springer, 2000.

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

[9] U. Feige and M. X. Goemans. Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT. In *Proceeding of the 3rd Israel Symposium on Theory and Computing Systems*, pages 182–189, 1995.

[10] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, November 1995.

[11] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. Technical Report 00-037, Electronic Colloquim on Computational Complexity, June 2000. ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/2000/TR00-037/index.html.

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

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

[14] 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, ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/2000/TR00-019/index.html.

[15] 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.

[16] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages, http://www.cs.toronto.edu/~kullmann, January 1997.

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

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

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

[20] 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.

5 Appendix: Proof of Lemma 1

5.1 Proof of inequality (3)

Pure literal elimination. The root N has a single child N' and $F_{N'}$ is obtained from F_N by elimination of a pure literal u. Then

$$\begin{aligned}&\mathsf{Eval}\,(F_N, A_N) = \mathsf{Eval}\,(F_{N'}, A_{N'}) + \#_u(F_N) = \\ &\mathsf{Eval}\,(F_{N'}, A_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'}).\end{aligned}$$

Therefore, in this case the inequality (3) holds.

Elimination of a $(1,1)$-literal. The root N has a single child N' and $F_{N'}$ is obtained from F_N by $(1,1)$-literal elimination. In this case

$$\begin{aligned}&\mathsf{Eval}\,(F_N, A_N) = \mathsf{Eval}\,(F_{N'}, A_{N'}) + 1 = \\ &\mathsf{Eval}\,(F_{N'}, A_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'}).\end{aligned}$$

Therefore, in this case the inequality (3) holds.

Splitting. The root N has two children N_1 and N_2 labeled by formulas $F_N[u]$ and $F_N[\overline{u}]$. It is easy to see that

$$\begin{aligned}\mathsf{Eval}\,(F_N, A_{N_1} \cup \{u\}) &= \mathsf{Eval}\,(F_N[u], A_{N_1}) + \#_u(F_N),\\ \mathsf{Eval}\,(F_N, A_{N_2} \cup \{\overline{u}\}) &= \mathsf{Eval}\,(F_N[\overline{u}], A_{N_2}) + \#_{\overline{u}}(F_N).\end{aligned}$$

In addition,

$$\begin{aligned}\#_u(F_N) &= \mathcal{K}(F_N) - \mathcal{K}(F_{N_1}),\\ \#_{\overline{u}}(F_N) &= \mathcal{K}(F_N) - \mathcal{K}(F_{N_2}).\end{aligned}$$

Since by definition A_N satisfies not less clauses than either of $A_{N_1} \cup \{u\}$ and $A_{N_2} \cup \{\overline{u}\}$, we see that in this case the inequality (3) holds. This completes the proof of the inequality (3).

5.2 Proof of inequality (4)

Pure literal elimination. The root N has a single child N' and $F_{N'}$ is obtained from F_N by elimination of a pure literal u. We have

$$\begin{aligned}&\mathsf{Opt}(F_N) = \mathsf{Opt}(F_N[u]) + \#_u(F_N) = \\ &\mathsf{Opt}(F_N[u]) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'}).\end{aligned}$$

Therefore, the inequality (4) holds.

Elimination of a $(1,1)$-literal. The root N has a single child N' and $F_{N'}$ is obtained from F_N by $(1,1)$-literal elimination. A simple consideration of possible cases shows that in this case we also have $\mathsf{Opt}(F_N) = \mathsf{Opt}(F_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'})$. Therefore, in this case the inequality (4) holds.

Splitting. The root N has two children N_1 and N_2 labeled by formulas $F_N[u]$ and $F_N[\overline{u}]$. Since F_N has an optimal assignment containing either u or $\overline{u}$, we have

$$\begin{aligned}&\mathsf{Opt}(F_N) = \max(\mathsf{Opt}(F_N[u]) + \#_u(F_N), \mathsf{Opt}(F_N[\overline{u}]) + \#_{\overline{u}}(F_N)) \leq \\ &\max(\mathsf{Opt}(F_N[u]) + \mathcal{K}(F_N) - \mathcal{K}(F_N), \mathsf{Opt}(F_N[\overline{u}]) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'})).\end{aligned}$$

This completes the proof of the inequality (4).