PDMI PREPRINT — 14/1998

APPROXIMATION ALGORITHMS FOR MAX SAT: A BETTER PERFORMANCE RATIO AT THE COST OF A LONGER RUNNING TIME

Evgeny Dantsin
Russian Academy of Sciences
Steklov Mathematical Institute
St. Petersburg Department
27 Fontanka, St. Petersburg, 191011, Russia
E-mail: dantsin@pdmi.ras.ru

Michael Gavrilovich
St. Petersburg State University
Universitetskaya emb., 7/9,
St. Petersburg, 199034, Russia
E-mail: misha@logic.pdmi.ras.ru

Edward A. Hirsch
Russian Academy of Sciences
Steklov Mathematical Institute
St. Petersburg Department
27 Fontanka, St. Petersburg, 191011, Russia
E-mail: hirsch@pdmi.ras.ru

Boris Konev
Russian Academy of Sciences
Steklov Mathematical Institute
St. Petersburg Department
27 Fontanka, St. Petersburg, 191011, Russia
E-mail: konev@pdmi.ras.ru

May 20, 1998

Abstract

We describe approximation algorithms for (unweighted) MAX SAT with performance ratios arbitrarily close to 1 (in particular, when performance ratios exceed the limit of polynomial-time approximation). Namely, we show how to construct an $(\alpha+\varepsilon)$-approximation algorithm $\mathcal{A}$ from a given polynomial-time α-approximation algorithm $\mathcal{A}_0$. The algorithm $\mathcal{A}$ runs in time of the order $\phi^{\varepsilon(1-\alpha)^{-1}K}$, where ϕ is the golden ratio (≈ 1.618) and K is the number of clauses in the input formula. Thus we estimate the cost of improving a performance ratio. Similar constructions for MAX 2SAT and MAX 3SAT are described too. We apply our constructions to some known polynomial-time algorithms taken as $\mathcal{A}_0$ and give upper bounds on the running time of the respective algorithms $\mathcal{A}$.

1 Introduction

In the MAX SAT problem we are given a Boolean formula represented by a set of clauses, 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 satisfied clauses. In recent years there are many achievements in the area of MAX SAT approximation. Most of them are connected with two directions: polynomial-time approximation algorithms for MAX SAT and limits of polynomial-time approximation. Here we mention only several recent results relevant to this paper, see [6, 1, 5] for detailed surveys.

A sequence of recent improvements on a polynomial-time 3/4-approximation algorithm for MAX SAT (Yannakakis [19]) led to a deterministic 0.758-approximation algorithm (Goemans and Williamson [11], Mahajan and Ramesh [17]) and a randomized 0.770-approximation algorithm (Asano [4]). Better performance ratios were obtained for MAX 2SAT and MAX 3SAT, version-s of MAX SAT in which clauses contain at most two and three literals respectively. Namely, MAX 2SAT can be solved by the randomized 0.931-approximation algorithm ([10, 11]) or by its derandomized version ([17]). For MAX 3SAT there are a randomized 7/8-approximation algorith-m (Karloff and Zwick [14]) and a deterministic 0.801-approximation algorithm (Trevisan, Sudan, Sorkin and Williamson [18]).

A characterization of NP in terms of probabilistically checkable proofs (Arora and Safra [3]) had important implications for MAX SAT approximation. Namely, there is a constant α such that the existence of a α-approximation polynomial-time algorithm for MAX SAT implies P = NP (Arora et al [2]). Håstad in [12] improved previous particular values of the limit of polynomial-time approximation for MAX 2SAT and MAX 3SAT: no $(7/8+\varepsilon)$-approximation algorithm for MAX 3SAT and no $(0.955+\varepsilon)$-approximation algorithm for MAX 2SAT exist unless P = NP.

Main results. What algorithm can we use if we need to find an α-approximation solution when α is arbitrarily close to 1, for example, when α is greater than the limit of approximation? Can we find such a solution faster than an exact solution (even if both take exponential time to be found)? In this paper we answer these questions.

We show how to construct an $(\alpha+\varepsilon)$-approximation algorithm $\mathcal{A}$ from a given polynomial-time α-approximation algorithm $\mathcal{A}_0$. The algorithm $\mathcal{A}$ runs in time $|F|^{O(1)} \cdot \phi^{\varepsilon(1-\alpha)^{-1}K}$, where ϕ is the golden ratio (≈ 1.618) and K is the number of clauses in the input formula. Thus, we estimate the cost of improving a performance ratio.

The construction and bound remain the same for MAX 2SAT. For MAX 3SAT, we use a similar construction and obtain an algorithm running in time $|F|^{O(1)} \cdot 2^{\varepsilon(1-\alpha)^{-1}K}$.

We apply our construction, taking the known polynomial-time algorithms mentioned above as $\mathcal{A}_0$. For example, taking the 7/8-approximation algorithm ([14]), we obtain the $(7/8+\varepsilon)$-approximation algorithm for MAX 3SAT that runs in time $|F|^{O(1)} \cdot c^{\varepsilon K}$.

The techniques used. To construct an $(\alpha+\varepsilon)$-approximation algorithm from an α-approximation algorithm, we apply the *splitting method*, a method going back to [9, 8] and used in most SAT algorithms. In the simplest case, a SAT algorithm based on the splitting method transforms an input formula F into two formulas $F_{x=\mathit{True}}$ and $F_{x=\mathit{False}}$ obtained from F by substitution of the truth values for some variable x. Each of these formulas is split again, and splittings proceed until we come to formulas whose satisfiability can be checked easily. Many non-trivial upper bounds on the running time of such algorithms were obtained recently (for example [15, 16, 13]).

There are various ways of applying the splitting method to MAX SAT (for example [7]). We employ it as follows. Splittings and some other reductions (pure literal elimination and resolution) are used to transform an input formula F into formulas $F_1, \ldots, F_N$. Applying a polynomial-time MAX SAT approximation algorithm to $F_1, \ldots, F_N$, we find approximate solutions to them. These solutions are then used to construct a solution to F.

Organization of the paper. Section 2 contains basic definitions and notation we use. In Section 3 we describe an exact MAX SAT algorithm whose running time is $|F|^{O(1)} \cdot \phi^K$. The main purpose of this section is to show how we apply the splitting method to solving MAX SAT and how we estimate the running time of algorithms based on the splitting method. In Section 4 we show how to construct a deterministic $(\alpha+\varepsilon)$-approximation algorithm from a deterministic polynomial-time α-approximation algorithm, using the splitting method. Proofs of the corresponding upper bounds are given in this section too. Section 5 gives a similar construction for randomized algorithms.

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 set of clauses. 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 respect 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 clause 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.

Let $\mathcal{A}$ be an algorithm that produces on an input formula F an assignment A_F. We call $\mathcal{A}$ an *exact* algorithm for MAX SAT (or for related problems) if $\mathsf{Eval}(F, A_F) = \mathsf{Opt}(F)$ for all F. The algorithm $\mathcal{A}$ is called an *α-approximation* algorithm if $\mathsf{Eval}(F, A_F) \geq \alpha \cdot \mathsf{Opt}(F)$ for all F. The infimum of the ratio $\mathsf{Eval}(F, A_F) / \mathsf{Opt}(F)$ over all F is called the *performance ratio* of $\mathcal{A}$.

3 An 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 at most 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 built 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 3.1 *There exists a deterministic algorithm for* MAX SAT *whose running time does not exceed*

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

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 with building 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 the algorithm.

The first step of our algorithm is to build 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 a *best* of the assignments $A_{N'} \cup \{u\}$ and $A_{N'} \cup \{\overline{u}\}$, i.e. the assignment satisfying more clauses in F_N or any of them if they satisfy the same number of clauses in F_N.

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 a best of $A_{N'} \cup \{u\} \cup \{\overline{v}\}$ and $A_{N'} \cup \{\overline{u}\} \cup \{v\}$.

When any 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 a best of $A_{N_1} \cup \{x\}$ and $A_{N_2} \cup \{\overline{x}\}$.

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 built 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 $\mathsf{Opt}(F_N) = \mathsf{Opt}(F_{N'}) + \mathcal{K}(F_N) - \mathcal{K}(F_{N'})$. From 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

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

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

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

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 defined A_N as the best of $A_{N_1} \cup \{u\}$ and $A_{N_2} \cup \{\overline{u}\}$, the assertion holds in this case too. This completes the proof of soundness.

3.3 The 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 $l_k \le \mathcal{F}_{k+1}$. 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 $\mathcal{F}_k \le \mathcal{F}_{k+1}$, 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

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

Thus, the number of leaves in any reduction tree for F does not exceed $\mathcal{F}_{k+1}$ and we have $l_k \le \mathcal{F}_{k+1}$. It is well known that $\mathcal{F}_{k+1} \le \phi^k$ (easy to prove by induction on k). Hence, we obtain $l_k \le \phi^k$ 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 4.1 and its corollaries describe approximation algorithms for MAX SAT and MAX 2SAT. Theorem 4.4 and its corollary describe approximation algorithms for MAX 3SAT.

Theorem 4.1 *Assume that there is a deterministic polynomial-time α-approximation algorithm for* MAX SAT *(*MAX 2SAT*). Let $0 < \varepsilon \le 1 - \alpha$. Then one can construct a deterministic $(\alpha + \varepsilon)$-approximation algorithm for* MAX SAT *(respectively* MAX 2SAT*) whose running time does not exceed*

$$|I|^{O(1)} \cdot \phi^{\varepsilon(1-\alpha)^{-1}K},$$

where I is an input formula, K is the number of clauses in I, and ϕ is the golden ratio (≈ 1.618).

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

We apply this theorem, taking known algorithms as the α-approximation algorithm. Namely, we take the 0.758-approximation algorithm for MAX SAT ([11, 17]) and the 0.931-approximation algorithm for MAX 2SAT ([11, 10, 17]). Simple calculations yield the following corollaries.

Corollary 4.2 *For any ε such that $0 < \varepsilon \leq 0.242$, there is a deterministic $(0.758 + \varepsilon)$-approximation algorithm for* MAX SAT *whose running time does not exceed*

$$|I|^{O(1)} \cdot \phi^{4.133\varepsilon K},$$

where I is an input formula, K is the number of clauses in I, and ϕ is the golden ratio.

Corollary 4.3 *For any ε such that $0 < \varepsilon \leq 0.069$, there is a deterministic $(0.931 + \varepsilon)$-approximation algorithm for* MAX 2SAT *whose running time does not exceed*

$$|I|^{O(1)} \cdot \phi^{14.493\varepsilon K},$$

where I is an input formula, K is the number of clauses in I, and ϕ is the golden ratio.

Theorem 4.4 *Assume that there is a deterministic polynomial-time α-approximation algorithm for* MAX 3SAT*. Let $0 < \varepsilon \leq 1 - \alpha$. Then one can construct a deterministic $(\alpha + \varepsilon)$-approximation algorithm for* MAX 3SAT *whose running time does not exceed*

$$|I|^{O(1)} \cdot 2^{\varepsilon(1-\alpha)^{-1}K},$$

where I is an input formula and K is the number of clauses in I.

Proof. See Section 4.4.

Using the known 0.801-approximation algorithm for MAX 3SAT as the α-approximation algorithm, we obtain the following corollary.

Corollary 4.5 *For any ε such that $0 < \varepsilon \leq 0.199$, there is a deterministic $(0.801 + \varepsilon)$-approximation algorithm for* MAX 3SAT *whose running time does not exceed*

$$|I|^{O(1)} \cdot 2^{5.026\varepsilon K},$$

where I is an input formula, K is the number of clauses in I.

4.1 Algorithm

Like the exact algorithm in Section 3, our approximation algorithm starts with building a reduction tree T for an input formula I. However, this algorithm builds only a part T' of the entire tree T. Namely, the approximation algorithm starts with the root and proceeds with constructing descendants as in Section 3.1 while formulas consist of at least K_0 clauses, where

$$K_0 = \lfloor K - \varepsilon(1-\alpha)^{-1}K \rfloor$$

(the choice of this value of K_0 will become clear below). Otherwise, i.e. when $\mathcal{K}(F_N) < K_0$, 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. At each leaf N, the assignment A_N is computed by the polynomial-time α-approximation algorithm from the statement of the theorem. Namely, we define A_N to be the output of this algorithm on input F_N. 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.

4.2 Soundness

The following two lemmas prove that the assignment computed for the root of T' satisfies at least $(\alpha+\varepsilon)\cdot\mathsf{Opt}(I)$ clauses in I.

Lemma 4.6 *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 $\mathsf{Eval}\,(F_N, A_N) - \mathsf{Eval}\,(F_{N'}, A_{N'}) \geq \mathcal{K}(F_N) - \mathcal{K}(F_{N'})$. The latter inequality as well as the inequality (2) are proved by straightforward consideration of the types of reductions.

Lemma 4.7 *Let R be the root of T'. Then $(\alpha+\varepsilon)\cdot\mathsf{Opt}(F_R) \leq \mathsf{Eval}\,(F_R, A_R)$.*

Proof. Let L be a leaf such that (1) and (2) hold. Since A_L is the assignment computed by the α-approximation algorithm for F_L, we have

$$\mathsf{Eval}\,(F_L, A_L) \geq \alpha\cdot\mathsf{Opt}(F_L). \tag{3}$$

We thus obtain

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

We estimate the last expression using our choice of K_0:

$$\begin{aligned}
&\alpha\cdot\mathsf{Opt}(F_R) + (1-\alpha)(\mathcal{K}(F_R) - \mathcal{K}(F_L)) \geq \\
&\alpha\cdot\mathsf{Opt}(F_R) + (1-\alpha)(K - K_0) \geq \\
&\alpha\cdot\mathsf{Opt}(F_R) + \varepsilon K \geq \\
&(\alpha+\varepsilon)\cdot\mathsf{Opt}(F_R).
\end{aligned}$$

4.3 The 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 $K_0 = \lfloor K - \varepsilon(1-\alpha)^{-1}K \rfloor$. 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 3.1. 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.

4.4 Proof of Theorem 4.4

The $(\alpha+\varepsilon)$-approximation algorithm for MAX 3SAT is similar to its counterparts for MAX SAT and MAX 2SAT in Theorem 4.1 and differs only in the following. While the algorithm in Theorem 4.1 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 builds 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 $l_{k+1} \le l_k + l_{k-1}$ does not hold. Instead, the inequality $l_{k+1} \le l_k + l_k$ obviously holds. Therefore, we obtain the required bound with 2 instead of ϕ in the base of exponent.

5 Randomized versions

A *randomized α-approximation* algorithm for MAX SAT (or related problems) is defined as a randomized algorithm that produces on an input formula F an assignment A_F such that

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

where E denotes the expectation. The algorithms that we construct in this section are similar to the algorithms in the previous section. The only difference is that α-approximation algorithms are randomized and, accordingly, we obtain randomized $(\alpha+\varepsilon)$-approximation algorithms.

Theorem 5.1 (a randomized version of Theorem 4.1) *Assume that there is a polynomial-time randomized α-approximation algorithm for* MAX SAT *(*MAX 2SAT*). Let $0 < \varepsilon \le 1-\alpha$. Then one can construct a randomized $(\alpha+\varepsilon)$-approximation algorithm for* MAX SAT *(respectively* MAX 2SAT*) whose running time is*

$$|I|^{O(1)} \cdot \phi^{\varepsilon(1-\alpha)^{-1}K},$$

where I is an input formula, K is the number of clauses in I, and ϕ is the golden ratio (≈ 1.618).

Proof. We describe the required $(\alpha+\varepsilon)$-approximation algorithm in Section 5.1 and prove its soundness and the bound on the running time in Section 5.2.

Corollary 5.2 *For any ε such that $0 < \varepsilon \le 0.230$, there is a randomized $(0.770+\varepsilon)$-approximation algorithm for* MAX SAT *whose running time is*

$$|I|^{O(1)} \cdot \phi^{4.348\varepsilon K},$$

where I is an input formula, K is the number of clauses in I, and ϕ is the golden ratio.

Proof. We obtain this corollary, using the polynomial-time randomized 0.770-approximation algorithm for MAX SAT ([4]) as the α-approximation algorithm.

Theorem 5.3 (a randomized version of Theorem 4.4) *Assume that there is a polynomial-time randomized α-approximation algorithm for* MAX 3SAT*. Let $0 < \varepsilon \le 1 - \alpha$. Then one can construct a randomized $(\alpha + \varepsilon)$-approximation algorithm for* MAX 3SAT *whose running time is*

$$|I|^{O(1)} \cdot 2^{\varepsilon(1-\alpha)^{-1}K}$$

where I is an input formula and K is the number of clauses in I.

Proof. See Section 5.3.

Corollary 5.4 *For any ε such that $0 < \varepsilon \le 1/7$, there is a randomized $(7/8 + \varepsilon)$-approximation algorithm for* MAX 3SAT *whose running time is*

$$|I|^{O(1)} \cdot 2^{8\varepsilon K}$$

where I is an input formula and K is the number of clauses in I.

Proof. We use the polynomial-time randomized 7/8-approximation algorithm for MAX 3SAT ([14]) as the α-approximation algorithm.

5.1 Algorithm

Consider the randomized α-approximation algorithm mentioned in the condition of Theorem 5.1. We can think of it 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 $(\alpha+\varepsilon)$-approximation algorithm in Section 4.1, our randomized $(\alpha+\varepsilon)$-algorithm starts with building a tree T' which is a part of a reduction tree. Let $L_1, \ldots, L_s$ be the leaves of T' labeled by the formulas $F_{L_1}, \ldots, F_{L_s}$. The next step is computation of the assignments $A_{L_1}, \ldots, A_{L_s}$. Namely, our algorithm generates a string τ of random bits and runs $\mathcal{A}_0$ on all $F_{L_1}, \ldots, F_{L_s}$ to compute

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

Further assignments A_N for non-leaf nodes N are computed in the same way as in the deterministic case (Section 4.1). The output of our randomized $(\alpha + \varepsilon)$-algorithm is the assignment computed for the root.

5.2 Soundness and the running time

Soundness. Like the deterministic case, the proof of soundness is based on assertions similar to the assertions in Lemmas 4.6–4.7. In fact, Lemma 4.6 holds for the randomized case without any changes.

Lemma 5.5 (a randomized version of Lemma 4.6) *Let T' be the tree constructed by the randomized $(\alpha+\varepsilon)$-algorithm. Let R be the root of T'. The tree T' has a leaf L such that*

$$\begin{aligned} &\mathsf{Eval}\,(F_R, A_R) \ge \mathsf{Eval}\,(F_L, A_L) + \mathcal{K}(F_R) - \mathcal{K}(F_L) \\ &\mathsf{Opt}(F_R) \le \mathsf{Opt}(F_L) + \mathcal{K}(F_R) - \mathcal{K}(F_L) \end{aligned}$$

Proof. Our algorithm constructs T' in completely the same way as in the deterministic case. Like (1) in Lemma 4.6, the first inequality depends only on the method of computation of assignments for all non-leaf nodes of T' and does not depend on the assignments at the leaves. Therefore, this inequality holds. The second inequality holds because (2) depends only on the construction of T'.

Lemma 5.6 (a randomized version of Lemma 4.7) *Let R be the root of T'. Then*

$$(\alpha + \varepsilon) \cdot \mathsf{Opt}(I) \leq E[\mathsf{Eval}\,(F_R, A_R)].$$

Proof. Let $\mathcal{A}(I, \tau)$ denote the output of constructed $(\alpha+\varepsilon)$-approximation algorithm on an input formula I and a string τ of random bits. By the first inequality in Lemma 5.5, the number of clauses satisfied by $\mathcal{A}(I, \tau)$ is greater than or equal to $\mathsf{Eval}\,(F_{L_i}, \mathcal{A}_0(F_{L_i}, \tau)) + \mathcal{K}(I) - \mathcal{K}(F_{L_i})$ for any τ. Hence we have

$$\mathrm{E}[\mathsf{Eval}\,(F_R, A_R)] \geq \mathrm{E}[\mathsf{Eval}\,(F_{L_i}, \mathcal{A}_0(F_{L_i})) + \mathcal{K}(I) - \mathcal{K}(F_{L_i})],$$

an “expectation” version of the inequality (3) in the proof of Lemma 4.7. Similarly, “expectation” versions of the other inequalities in the proof of Lemma 4.7 holds.

Running time. Computation of assignments for non-leaf nodes in T' does not depend on the random string τ. Therefore, the bound on the running time is the same as in Theorem 4.1.

The method of building T' by the randomized algorithm is exactly the same as the method of building T' by the deterministic algorithm. Therefore, the running time of the randomized algorithm can be estimated exactly as in the deterministic case (see Section 4.3).

5.3 Proof of Theorem 5.3

This is an “expectation” version of Theorem 4.4. We use the same arguments as in the proof of Theorem 5.1.

6 Acknowledgments

E. Dantsin, E. A. Hirsch, and B. Konev are supported in part by grants from INTAS (International Association for the promotion of cooperation with scientists from the independent States of the former Soviet Union) and RFBR (Russian Foundation for Basic Research). M. Gavrilovich is supported by a grant from ISSEP (International Soros Science Education Program). We thank these foundations for support.

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] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[3] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–13, 1992.

[4] T. Asano. Approximation algorithms for MAX SAT: Yannakakis vs. Goemans-Williamson. In *Proceedings of the 5th Israel Symposium on Theory and Computing Systems*, pages 24–37, 1997.

[5] L. Babai. Transparent proofs and limits to approximation. In *Proceedings of the First European Congress of Mathematicians, Paris, 1992, vol. 1*, pages 31–92. Birkhäuser Verlag, 1994.

[6] M. Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and non-approximability: Towards tight results. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 422–431, October 1995. Revised July 1997, 5th version. To appear in *SIAM Journal of Computing.*

[7] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Technical report, New Mexico Tech, Mathematical Department, Socorro, NM 87801, October 1995. To appear in *Journal of Combinatorial Optimization.*

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

[10] 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.

[11] 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.

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

[13] E. A. Hirsch. Two new upper bounds for SAT. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 521–530, 1998.

[14] 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*, pages 406–415, 1997.

[15] O. Kullmann. Worst-case analysis, 3-SAT decision and lower bounds: approaches for improved SAT algorithms. In *DIMACS Proceedings of SAT Workshop 1996*, pages 261–313, 1996.

[16] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Submitted to *Information and Computation*, 1997.

[17] S. Mahajan and H. Ramesh. Derandomizing semidefinite programming based approximation algorithms. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 162–169, 1995.

[18] L. Trevisan, G. B. Sorkin, M. Sudan, and D. P. Williamson. Gadgets, approximation, and linear programming. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 1996.

[19] M. Yannakakis. On the approximation of maximum satisfiability. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–9, 1992. Full version in *Journal of Algorithms* 17, pp. 475-502, 1994.