An Exact MaxSAT Algorithm: Further Observations and Further Improvements

Mingyu Xiao
University of Electronic Science and Technology of China, Chengdu, China
myxiao@uestc.edu.cn

Abstract

In the maximum satisfiability problem (MaxSAT), given a CNF formula with m clauses and n variables, we are asked to find an assignment of the variables to satisfy the maximum number of clauses. Chen and Kanj showed that this problem can be solved in $O^*(1.3248^m)$ time (DAM 2004) and the running time bound was improved to $O^*(1.2989^m)$ by Xu et al. (IJCAI 2019). In this paper, we further improve the result to $O^*(1.2886^m)$. By using some new reduction and branching techniques we can avoid several bottlenecks in previous algorithms and get the improvement on this important problem.

1 Introduction

The satisfiability problem (SAT) as the first proved NP-complete problem [Cook, 1971] is one of the most influential problems in computational complexity. Due to the large applications in engineering, artificial intelligence, and many other areas [Garey and Johnson, 1979], this problem and many variants have been extensively studied in the literature. Many practical solvers for these problems have been developed [Hutter *et al.*, 2017; Cai *et al.*, 2017]. Most of these algorithms are heuristic-based and can not solve all instances quickly due to the hardness of the problems. So diverse approaches have been investigated on these problems, including randomized, approximation, exact, parameterized algorithms, etc.

In exact and parameterized algorithms, we aim to find fast exponential algorithms that can optimally solve the problems. These running time bounds may be helpful for us to understand the computational complexity of NP-complete problems and the limitation to solving NP-complete problems. In practice, with the rapid development of hardware and computer techniques, many fast exact algorithms can be implemented to solve middle-size instances within an acceptable time. The most important thing is that many techniques and principles behind exact algorithms can be used to improve heuristic solvers. The reduction rules were used as the preprocessing procedures to simplify the input instance without losing the optimality and some branching rules were used to deal with some dense local structures [Xiao *et al.*, 2021]. All of the above motivates us to follow the hard research line of studying exact algorithms for important NP-complete problems with theoretical running time bounds. It is worth mentioning that for SAT and related problems, several running time bounds of decades ago were improved during the last two years [Chu *et al.*, 2021; Alferov and Bliznets, 2021; Peng and Xiao, 2021; Xu *et al.*, 2019].

In this paper, we will study exact algorithms for the maximum satisfiability problem (MaxSAT). In a CNF-formula, we are given n Boolean variables and m clauses on the variables. SAT asks whether we can give an assignment of the variables to satisfy all the clauses. MaxSAT asks us to satisfy the maximum number of clauses. We consider the decision version of MaxSAT: whether we can satisfy at least k clauses for k being part of the input. Clearly, MaxSAT is more general compared to the original SAT problem. MaxSAT has significant applications and has been extensively studied in theory and application [Morgado *et al.*, 2013; Bonet *et al.*, 2007; Li *et al.*, 2016]. For more progresses in MaxSAT, please see recent surveys [Li and Manyà, 2021; Bacchus *et al.*, 2021].

To exact solve SAT or MaxSAT, the trivial algorithm to check all possible assignments has the running time bound $O^*(2^n)$. However, no algorithm with running time bound $O^*(c^n)$ for constant $c < 2$ was found after decades of hard research, and the Strong Exponential Time Hypothesis (SETH) [Impagliazzo and Paturi, 2001] assumes that this kind of algorithm does not exist. So we turn to consider another natural measure – the number of clauses m. The running time bound under this measure can be significantly improved. The lines of research that attempt to solve SAT and MaxSAT in terms of m are shown in Tables 1 and 2. Previously, SAT and MaxSAT can be solved in $O^*(1.2226^m)$ and $O^*(1.2989^m)$ time, respectively. In this paper, we further improve the running time bound for MaxSAT to $O^*(1.2886^m)$. We also mention another frequently used measure L, which is the total length of all clauses. In terms of L, very recently, the running time bounds for SAT and MaxSAT were improved to $O^*(1.0646^L)$ [Peng and Xiao, 2021] and $O^*(1.0927^L)$ [Alferov and Bliznets, 2021], respectively.

Most previous algorithms are branch-and-search algorithms that will search for a solution by letting a variable (or a literal) be 1 or 0. For an (a, b)-literal x (x appears in a clauses and $\overline{x}$ appears in b clauses), the simple branch will decrease

Running times	References
$O^*(1.260^m)$	[Monien *et al.*, 1981]
$O^*(1.239^m)$	[Hirsch, 1998]
$O^*(1.234^m)$	[Yamamoto, 2005]
$O^*(1.2226^m)$	[Chu *et al.*, 2021]

Table 1: Progress for SAT in terms of m

Running times	References
$O^*(1.3803^m)$	[Niedermeier and Rossmanith, 1999]
$O^*(1.3413^m)$	[Bansal and Raman, 1999]
$O^*(1.3248^m)$	[Chen and Kanj, 2004]
$O^*(1.2989^m)$	[Xu *et al.*, 2019]
$O^*(1.2886^m)$	**This paper**

Table 2: Progress for MaxSAT in terms of m

the number of clauses by a and b, respectively, in the two subbranches. When $a+b$ is large, the numbers of clauses decrease greatly and we may be able to get a good complexity. So the bottleneck will happen when all the variables have a low degree. On the other hand, when the degree of a variable is at most 2, we can reduce the instance directly. For variables of degree 3, we may also have fast branching rules to deal with them. So usually the bottleneck cases are to deal with variables of degree 4 or 5. Chen and Kanj [2004] showed that their worst case was to branch on $(3, 2)$-literals and then got the running time bound $O^*(1.3248^m)$. Xu et al. [2019] showed that the previous bottleneck would not always happen and improved the bound to $O^*(1.2989^m)$. Their worst case is to deal with some $(2, 2)$-literals, which needs to consider many cases and combine several following subbranches. The bottleneck branching vector is $(14, 14, 12, 12, 7, 7, 5, 5)$ that contains eight subbranches. In this paper, we avoid the previous bottleneck of dealing with $(2, 2)$-literals by using new reduction rules and deep analysis. Our bottleneck comes back to deal with $(3, 2)$-literals. We get the following neat result that for the worst case: after branch on a $(3, 2)$-literal, we can branch with $(3, 3)$ at least in each subbranch. Thus, our worst branching case is $(6, 6, 5, 5)$, the corresponding complexity (branching factor) is 1.2886.

Our algorithm uses several reduction rules and techniques developed in previous papers. Compared to the work in [Chen and Kanj, 2004], we further show that after their bottleneck we always have good branchings. Compared to the work in [Xu *et al.*, 2019], our branching complexity to deal with $(2, 2)$-literals and $(3, 2)$-literals are improved to 1.2872 and 1.2886, respectively, while their results were 1.2989 and 1.2937.

2 Preliminaries

For a Boolean variable x, a *literal* is either x or the negation of it. We use $\overline{x}$ (resp., $\overline{C}$ for a set of literals C) to denote the negation of a literal x (resp., the set of negations of literals in C). Thus, we have $\overline{\overline{x}} = x$. A *clause* is a set of literals and it is satisfied if there is at least one literal that is assigned 1. Given a set V of n Boolean variables, a CNF formula $\mathcal{F}$ is given by a set of m clauses on V. The MaxSAT problem asks whether there is an assignment on the Boolean variables to satisfy at least k clauses, where k is also a part of the input. A MaxSAT instance is denoted by a pair $(\mathcal{F}, k)$.

For two sets of literals C_1 and C_2, we may simply use C_1C_2 to denote the union of them. A set of a single element $\{x\}$ may be simply written as x. For a set C of literals, $C = 1$ (resp., $C = 0$) means to assign 1 (resp., 0) to each literal in C, $\mathcal{F}_{C=1}$ means to delete all clauses containing a literal in C and delete literals in $\overline{C}$ from all clauses in $\mathcal{F}$, and $\mathcal{F}_{C=0} = \mathcal{F}_{\overline{C}=1}$. Sometimes, we may simply use $C = 1$ and $C = 0$ to denote $\mathcal{F}_{C=1}$ and $\mathcal{F}_{C=0}$, respectively.

If a literal appears in a clauses in $\mathcal{F}$, then it is called an *a-literal* and its *degree* is a. The *degree* of a variable x is the degree of the literal x plus the degree of the literal $\overline{x}$. A variable of degree a is also called an *a-variable*. If the degree of x is a (resp., at least a or at most a) and the degree of $\overline{x}$ is b, we say x is an *(a, b)-literal* (resp., an *(a^+, b)-literal* or an *(a^-, b)-literal*). Similarly, we can define (a, b^+)-literals, (a, b^-)-literals, (a^+, b^+)-literals, (a^-, b^-)-literals and so on. A literal x is an (a, b)-literal if and only if $\overline{x}$ is a (b, a)-literal. A clause containing c literals is called a *c-clause* and its *length* is c. A clause of length 1 is called a *unit* clause. An $(i, 1)$-literal x is called a *singleton* if the unique clause containing $\overline{x}$ is a unit clause and *non-singleton* otherwise.

Resolution [Robinson, 1965] is a frequently used technique in SAT problems. Resolution on a variable x in $\mathcal{F}$, denoted by $\mathcal{F}_{\backslash x}$, is the operation that for each pair of clauses xC and $\overline{x}D$ in $\mathcal{F}$, generate a new clause CD, and then remove all original clauses containing x or $\overline{x}$.

2.1 Framework of Branching Algorithms

We introduce the framework of our branching algorithm. It will first apply reduction rules to reduce the instance and then apply branching rules to search for a solution when the instance can not be further reduced.

A reduction rule will transfer an instance $I = (\mathcal{F}, k)$ to an equivalent instance $I' = (\mathcal{F}', k')$ in polynomial time, where I is yes if and only if I' is yes. We will use $(\mathcal{F}, k) \to (\mathcal{F}', k')$ to denote the reduction. Usually, the resulting instance I' has a smaller number of variables or clauses or $k' < k$. Reduction rules can simplify the input instance without exponentially increasing the running time bound. So they are frequently used as preprocessing in many algorithms. We will use some known reduction rules and also introduce some new rules.

When the instance can not be reduced any more, we will branch. The branching operation will generate a search tree and the size of the tree is related to the exponential part of the running time. We use $T(m)$ to denote the maximum size of the search tree on any instance with at most m clauses. In a branching operation, if it generates l sub-branches with $m_i < m$ clauses in the ith sub-branch, then we get a recurrence relation

$$T(m) \le T(m_1) + T(m_2) + \cdots + T(m_l).$$

Let $a_i = m - m_i$. The above recurrence is represented by a *branching vector* $(a_1, a_2, \ldots, a_l)$. The largest root of the function $f(x) = 1 - \sum_{i=1}^{l} x^{-a_i}$ is called the *branching factor* of the recurrence. If the maximum branching factor for all branching operations in the algorithm is at most γ, then $T(m) = O(\gamma^m)$. See [Fomin and Kratsch, 2010] for more details about analyzing branching algorithms. So we want to minimize the biggest branching factor in the algorithm. We will say a branching vector is not worse than another one if the corresponding branching factor is not greater than that of the latter.

3 Reduction Rules

We first present the reduction rules.

R-Rule 1. $(\mathcal{F}' \wedge x\overline{x}C, k) \rightarrow (\mathcal{F}', k-1)$.

R-Rule 2. ([Bansal and Raman, 1999]) *For a $(1,1)$-literal x containing in two clause xC and $\overline{x}D$, let $(\mathcal{F}' \wedge xC \wedge \overline{x}D, k) \rightarrow (\mathcal{F} \wedge CD, k-1)$.*

Another easy case is to deal $(i,0)$-literal x. We only need to let $x = 1$. This case is included in the following rule as a special case.

R-Rule 3. ([Chen and Kanj, 2004]) *For an (i,j)-literal x in $\mathcal{F}$, if there are at least j unit clauses x, then $(\mathcal{F}, k) \rightarrow (\mathcal{F}_{x=1}, k-i)$.*

R-Rule 4. ([Bansal and Raman, 1999]) $(\mathcal{F}' \wedge xC \wedge \overline{x}C, k) \rightarrow (\mathcal{F}' \wedge C, k-1)$.

R-Rule 5. ([Bliznets and Golovnev, 2012]) *If there is a 2-clause xy containing a $(2,1)$-literal x, then $(\mathcal{F}' \wedge xy \wedge xC \wedge \overline{x}D, k) \rightarrow (\mathcal{F}' \wedge yD \wedge \overline{y}CD, k-1)$.*

R-Rule 6. ([Chen *et al.*, 2017]) *If there is an $(i,1)$-literal x such that the unit clause containing $\overline{x}$ also contains a $(j,1)$-literal y, then $(\mathcal{F}' \wedge xC_1 \wedge \cdots \wedge xC_i \wedge \overline{x}yD, k) \rightarrow (\mathcal{F}' \wedge yC_1D \wedge \cdots \wedge yC_iD, k-1)$.*

In $\mathcal{F}$, a literal x *dominates* another literal y if each clause containing y also contains x. We give the following new reduction rules.

R-Rule 7. *If there are two literals x and y such that x dominates y and $\overline{y}$ dominates $\overline{x}$, then replace y with $\overline{x}$ (and also $\overline{y}$ with x) in $\mathcal{F}$.*

R-Rule 8. *If there are two literals x and y such that $\overline{x}$ dominates $\overline{y}$ and there is a clause C containing both of x and y, then delete the clause C and decrease k by 1.*

R-Rule 9. *If there is a $(2,2)$-literal x such that the two clauses containing $\overline{x}$ contain the same singleton, then resolute on x, i.e., $(\mathcal{F}, k) \rightarrow (\mathcal{F}_{\backslash x}, k)$.*

The correctness of this rule is based on the following observation. Let $S = \{xC_1, xC_2, \overline{x}yD_1, \overline{x}yD_2\}$ be the four clauses containing x or $\overline{x}$, where y is a singleton. In $\mathcal{F}_{\backslash x}$, S is replaced with $S' = \{C_1yD_1, C_1yD_2, C_2yD_1, C_2yD_2\}$. For any optimal solution f to $\mathcal{F}$, at most one clause in $S \cup \{\overline{y}\}$ is not satisfied. If all of them are satisfied, then $y = 0$ and either $C_1 = C_2 = 1$ or $D_1 = D_2 = 1$ under f. For this case, all clauses in $S' \cup \{\overline{y}\}$ are also satisfied under f. If one clause in $S \cup \{\overline{y}\}$ is not satisfied under f, we can always assume that it is $\overline{y}$ (thus, $x = y = 1$). For this case, only one clause $\overline{y}$ in $S' \cup \{\overline{y}\}$ is not satisfied under f.

For any optimal solution f' to $\mathcal{F}_{\backslash x}$, also at most one clause in $S' \cup \{\overline{y}\}$ is not satisfied. If all of them are satisfied, then $y = 0$ and either $C_1 = C_2 = 1$ or $D_1 = D_2 = 1$ under f'. For this case, all clauses in $S \cup \{\overline{y}\}$ are also satisfied under f' and $x = 1$ for $D_1 = D_2 = 1$ and $x = 0$ for $C_1 = C_2 = 1$. If one clause in $S' \cup \{\overline{y}\}$ is not satisfied under f', then we can always assume that it is $\overline{y}$ (thus, $y = 1$). For this case, only one clause $\overline{y}$ in $S \cup \{\overline{y}\}$ is not satisfied under f' and $x = 1$.

Steps	Literal types	Vectors	Factors
Step 1	6^+-variables	(6,1)	1.2852
Step 2	$(i \geq 3, 1)$-nsin	(3,3)	1.2600
Step 3	3-variables	(8,1)	1.2321
Step 4	unit clauses	(5,2)	1.2366
Step 5	(3,2)-literals	Step 5.1: (4,2)	1.2721
		Step 5.2: (11,3,3)	1.2872
		Step 5.3: (6,1)	1.2852
		Step 5.4: (6,6,5,5)	1.2886
		Step 5.5: (11,10,4,3)	1.2801
Step 6	(2,2)-literals	Step 6.1: (4,2)	1.2721
		Step 6.2: (6,1)	1.2852
		Step 6.3: (11,3,3)	1.2872
		Step 6.4: (18,18,11,11,11,11,4,4)	1.2870
Step 7	(3,1)/(4,1)-sin	Lemma 5	1.2576

Table 3: An overview of branchings of the algorithm. In the table, 'nsin' stands for 'non-singletons' and 'sin' stands for 'singletons'.

Definition 1. *A formula is* reduced *if none of the above reduction rules can be applied to the formula.*

Lemma 1. *Applying any reduction rule will not increase the number of clauses, and for any input formula, it takes polynomial time to apply the reduction rules to transfer it to a reduced one.*

4 Branching Operations

When no reduction rule can be applied to the formula, we will apply our branching rules. We have seven branching steps, each of which is going to deal with some type of literals. First of all, we give the overview of the branching vectors and branching factors of all steps in Table 3.

4.1 Three Special Cases

Before giving the detailed branching steps, we first analyze three special cases that will lead to good branching vectors. These three cases will frequently appear in some subbranches after a bad branching in our algorithm. So when we meet a bad branching, we may be able to combine it with the following good branchings to get an improved branching. These special cases are to deal with $(i,1)$-no-singletons, 3-variables, and literals appearing in many unit clauses. The first two special cases were deeply studied in the previous paper by Xu et al. [2019]. Note that in these lemmas, we do not require the formula to be reduced.

Lemma 2. ([Xu *et al.*, 2019]) *Let $\mathcal{F}$ be an arbitrary formula, x be an $(i,1)$-literal, $i > 0$, in $\mathcal{F}$, and $\overline{x}D$ be the unique clause containing $\overline{x}$. We can either (i) reduce at least one clause by applying reduction rules, or (ii) safely branch with $x = 1$ and $xD = 0$ and get a branching vector covered by $(i, 1 + 2|D|)$. If x is $(i,1)$-no-singleton with $i \geq 3$, then the branching vector is at least $(3,3)$.*

Lemma 2 can be used to deal $(i, 1)$-no-singletons. The next lemma will be used to deal with 3-variables.

Lemma 3. ([Xu *et al.*, 2019]) *Let $\mathcal{F}$ be an arbitrary formula containing a 3-variable x, where xC_1, xC_2 and $\bar{x}D$ are the three clauses containing the variable x. We can either (i) reduce the number of clauses by at least 1 by applying reduction rules, or (ii) safely branch with (B1) $C_1C_2D = 0$ and (B2) $\mathcal{F}_{\setminus x}$ and k decreases by 1, and get a branching vector covered by $(\tau, 1)$, where $\tau = \max\{8, 7 + 2|D|\}$.*

Lemma 3 can be obtained from Lemma 9 in [Xu *et al.*, 2019]. We slightly refine the form of the result. Note that if $|C_1| < 2$ or $|C_2| < 2$, we can directly reduce one clause by applying R-Rule 3 or 5. So we can get the branching vector $(7 + 2|D|, 1)$.

Remark: To obtain Lemmas 2 and 3, we may need to use more reduction rules that are omitted here. We present the two lemmas by citing the references and the absence of these reduction rules do not affect other parts but may improve the readability of the paper.

We also deeply analyze another structure where literals appear in many unit clauses. R-Rule 3 shows that if an (i, j)-literal x appears in at least j unit clauses x, then we can reduce the instance directly. Here we show that we can get a good branch if the condition relaxes a little.

Lemma 4. *Let x be an (i, j)-literal appearing in exactly $j-1$ unit clauses x, where $i \geq j \geq 2$. Let the $i+j$ clauses containing x and $\overline{x}$ be $x, \dots, x, xC_1, \dots, xC_{i-j+1}, \overline{x}D_1, \dots, \overline{x}D_j$. We can either (i) reduce the number of clauses by at least 1, or (ii) safely branch with (B1) $x = 1$ and (B2) $x = D_1 = \cdots = D_j = 0$, and get a branching vector not worse than $(i, 2j + 1)$.*

The proof is omitted here due to the limited space.

Next, we are going to describe our branching steps. When we introduce one, we assume that the current instance is a reduced instance and previous steps can not be applied.

4.2 Step 1: 6^+-Variables

We show that if the formula contains a 6^+-variable x, we can always branch with a branching vector not worse than $(6, 1)$. Directly branching on a 7^+-variable or a $(3, 3)$-literal or a $(4, 2)$-literal will result in a branching vector not worse than $(6, 1)$. Branching on a $(5, 1)$-singleton x will also lead to a branching vector $(6, 1)$. Because in the branching $x = 1$, the five clauses containing x will be satisfied and reduced, and the unit clause $\overline{x}$ is also reduced. So the number of clauses decreases by 6 at least in this branching. For $(5, 1)$-no-singletons, we will deal with them in the next step.

4.3 Step 2: $(i, 1)$-Non-Singletons With $i \geq 3$

We simply use the branching rule in Lemma 2, which will generate a branching vector $(i \geq 3, 1 + 2|D|)$. Since the variable is not a singleton, we have that $|D| \geq 1$ and then the branching vector is not worse than $(3, 3)$.

Note that $(i, 1)$-literals with $i \leq 1$ will be reduced by R-Rule 2 or R-Rule 3. For $(2, 1)$-literals, we will deal with them in the next step.

4.4 Step 3: 3-Variables

Assume there is a 3-variable x with three clauses xC_1, xC_2 and $\overline{x}D$. The instance is reduced and R-Rules 3 and 5 can not be applied. So we know that $|C_1|, |C_2| \geq 2$. We use the branching rule in Lemma 3. Then we can branch with $(8, 1)$ at least when $D = \emptyset$ and $(9, 1)$ at least when $D \neq \emptyset$.

4.5 Step 4: Literals Appearing in Many Unit Clauses

If there is an $(i, j \geq 2)$-literal x that appears in at least $j-1$ unit clauses x, we directly use the rule in Lemma 4 to decrease the number of clauses or branch with $(5, 2)$ at least.

This step will deal with $(2, 2)/(3, 2)$-literals x containing in one unit clause. If x is a $(2, 2)$-literal, the branching vector is at least $(2, 5)$. If x is a $(3, 2)$-literal, the branching vector is at least $(3, 5)$.

4.6 Step 5: $(3, 2)$-Literals

In this step, we dealing with $(3, 2)$-literals. Now the formula only contains $(3, 2)/(2, 3)/(2, 2)$-literals and $(4, 1)/(3, 1)$-singletons. Let x be a $(3, 2)$-literal with five clauses xC_1, xC_2, xC_3, $\overline{x}D_1$, and $\overline{x}D_2$. Note that none of C_1, C_2, and C_3 can be empty since this case will be handled in Step 4.

Step 5.1: There is a unit clause $\overline{x}$. For this case, we can branch on x with a branching vector not worse than $(4, 2)$.

Note that no matter whether $x = 1$ or 0, all unit clauses $\overline{x}$ will be reduced in both subbranches. We can directly get a branching vector $(3, 2)$ by branching on x. In the branching $x = 1$, we can further reduce one unit clause $\overline{x}$ and get a branching vector $(4, 2)$.

Next we assume that none of C_1, C_2, C_3, D_1 and D_2 is empty.

Step 5.2: D_1 or D_2 contains a (2,2)-literal y. We show that we can branch with a branching vector not worse than $(11, 3, 3)$.

We branch on x. First, we look at the branching $x = 0$. After deleting the two clauses containing $\overline{x}$, if y becomes a $(0, 1)$/(0,2)/(1,1)-literal, we can reduce one more clause by applying R-Rules 2-3 on y. For this case, we get a branching vector $(3, 3)$. Otherwise y will become a (1,2)-literal. Let yE be the clause containing y. If E is not empty, then By Lemma 3 (with $|D| \geq 1$), we can either further reduce one clause by reductions or further branch with a branching vector (1,9) at least. Thus, we can get a branching vector $(3, 2 + 1, 2 + 9) = (3, 3, 11)$ at least. If E becomes empty, then we know that $E = x$ (it is impossible that before branching the clause containing y is a unit clause, otherwise Step 4 would be applied on y). For this case, in each branching of $x = 0$ and $x = 1$, y will become a variable of degree at most 3. We can either reduce the number of clauses by one or branch with $(1, 8)$ at least by Lemma 3. So we can branch with $(3 + 1, 3 + 8, 2 + 1, 2 + 8) = (4, 11, 3, 10)$ at least.

Step 5.3: One of D_1 and D_2 contains a 5-variable and the other contains a singleton. We show that we can branch with a branching vector not worse than $(6, 1)$.

Without loss of generality, we assume that D_1 contains a singleton y_1 and D_2 contains an (i_1, i_2)-literal y_2, where $i_1 +$

$i_2 = 5$. Since R-Rule 3 and Step 5.2 can not be applied now, we have that $i_1 \geq 2$ and $i_2 \geq 1$. If $y_2 = \overline{y_1}$, then we branch on x. In the branching $x = 0$, we can further reduce at least one clause by applying R-Rule 3 on y_1. Thus, we can always get $(3,3)$ for this case. Next, we assume that $y_1 \neq \overline{y_2}$. We will branch on y_2. In the branching $y_2 = 0$, we reduce i_2 clauses containing $\overline{y_2}$ directly. In the branching $y_2 = 1$, we first reduce i_1 clauses containing y_2. After deleting the i_1 clauses, if x becomes a (0,1) or ($i > 0$,0)-literal, then we can further reduce at least one clause by applying R-Rule 3. Otherwise x becomes an ($i > 0$,1)-literal. Since $y_1 \neq \overline{y_2}$, we know that y_1 is a singleton still left in the clause containing $\overline{x}$. Thus, the condition of R-Rule 6 holds and we can further reduce one clause by applying R-Rule 6. So we can always branching with a branching vector $(i_1 + 1, i_2)$. If $i_1 = 2$ or 3, the branching vector is $(3,3)$ or $(4,2)$. Next we consider $i_1 = 4$. Now y_2 is a singleton since Step 2 could be applied on y_2 now. Then in the branching $y_2 = 0$, we can further reduce the unit clause $\overline{y_2}$. Thus, we can indeed get $(6,1)$.

Step 5.4: D_1 and D_2 contain only 5-variables. We show that we can branch with $(6,6,5,5)$ at least.

A (3,2)-literal x' is called *good* if a variable of degree at most 3 or an $(i,1)$-literal is created after deleting the three clauses containing x'. In this step, we select x as a good (3,2)-literal if this kind of literals exists.

Assume that $y_1 \in D_1$ and $y_2 \in D_2$ are 5-variables. If $D_1 = y_1$ and $D_2 = \overline{y_2}$, then R-Rule 4 would be applied. So D_1 or D_2 must contain another 5-variable and then we can assume that $y_1 \neq \overline{y_2}$. Furthermore, y_1 and y_2 can only be $(2,3)$- and $(3,2)$-literals.

Case 1: x is not good. According to the choice of x, we know that all $(3,2)$-literals are not good. Now y_1 and y_2 are $(2,3)$-literals since if they were $(3,2)$-literals they would be good. We also know that y_1 and y_2 are not in $C_1C_2C_3$, otherwise x would be good. If D_2 contains y_1 or $\overline{y_1}$, we branch on x. In the branching $x = 0$, y_1 will become a 3-variable and the clause containing y_1 has length at least 2 otherwise Step 5.1 would be applied before this step. So we can further reduce one clause or branch on y_1 with at least $(1,9)$ by Lemma 3 (with $|D| \geq 1$). Thus, we can branch with $(3, 2+1, 2+9) = (3,3,11)$ at least. It is the same of the case that D_1 contains y_2 or $\overline{y_2}$.

Next we further assume that $y_1, \overline{y_1} \notin D_2$ and $y_2, \overline{y_2} \notin D_1$. We branch on y_1. In $\mathcal{F}_{y_1=1}$, x is a (3,1)-lateral and y_2 is a variable of degree at least 4 by the above assumptions. If y_2 is a (1,3)/(2,3)-literal, we further branch with (B1) $x = 1$ and (B2) $x = y_2 = 0$, which create a branching vector $(3,4)$. In total, we get a branching vector not worse than $(2+3, 2+4, 3) = (5,6,3)$. Else y_2 is a (2,2)-literal, we further branch on y_2. In the branching $y_2 = 1$, we can further reduce the three clauses containing x (which do not contain y_2 by the above assumptions). So we can get branching vector $(5,2)$ at least. In total, we get a branching vector not worse than $(2+5, 2+2, 3) = (7,4,3)$.

Case 2: x is good. We say x is *excellent* if a variable of degree at most 3 or an $(i,1)$-literal is created after deleting the two clauses containing $\overline{x}$. We first consider the case that x is excellent. For this case, we branch on x. In each subbranch, we can either reduce one more clause or branch with either (3,3) by Lemma 2 or (1,8) by Lemma 3. The worst case is to branch with (3,3) in both subbranches. Thus, we can always branch with $(2+3, 2+3, 3+3, 3+3) = (5,5,6,6)$ at least.

Next we assume that x is not excellent. Now Both of y_1 and y_2 are $(3,2)$-literals, $y_1, \overline{y_1} \notin D_2$, and $y_2, \overline{y_2} \notin D_1$, otherwise x would be excellent. We further distinguish two cases.

Case 2.1: one of $|D_1|$ and $|D_2|$, say $|D_1|$ is one, i.e., $y_1 = D_1$. We branch on y_1. In the branching $y_1 = 1$, $\overline{x}$ will become a $(1, i \leq 3)$-literal. We can either reduce one more clause or branch with either (3,3) by Lemma 2 or (1,8) by Lemma 3. In the branching $y_1 = 0$, $\overline{x}$ becomes a $(2,i)$-literal with one unit clause $\overline{x}$. By Lemma 4, we can either further reduce one clause or branch with $(2,5)$ at least. In the worst case, we can branch with $(3+3, 3+3, 2+2, 2+5) = (6,6,4,7)$ at least.

Case 2.2: $|D_1|, |D_2| \geq 2$. We branch on y_1. In the branching $y_1 = 1$, $\overline{x}$ will become a $(1, i \leq 3)$-literal with the clause $\overline{x}D_2$ containing $\overline{x}$ unchanged. If $\overline{x}$ is a $(1, i \leq 1)$-literal, then we can reduce one more clause directly by applying R-Reductions 1 or 2. If $\overline{x}$ is a $(1,2)$-literal, then we can branch with at least $(1,11)$ by Lemma 3 (with $|D| \geq 2$). If $\overline{x}$ is a $(1,3)$-literal, then we can branch with at least $(3,5)$ by Lemma 2 (with $|D| \geq 2$). In the worst case, we can branch with $(3+1, 3+11, 2) = (4,14,2)$ at least.

Step 5.5: D_1 and D_2 contain only (3,1)-singletons. We will get a branching vector not worse than $(11,10,4,3)$.

None of D_1 and D_2 is empty now, and D_1 and D_2 can not contain the same singleton otherwise R-Rule 9 would be applied. We assume that $y_1 \in D_1$ and $y_2 \in D_2$ are two different (3,1)-singletons. We will first branch on x. In the branching $x = 1$, we reduce three clauses. In the branching $x = 0$, the two clauses containing $\overline{x}$ will be removed and both of y_1 and y_2 become (2,1)-singletons. Then, we can reduce one clause directly or branch on y_1 with (1,8) at least by Lemma 3. Note that in the first branch of $(1,8)$ it is to resolute the (2,1)-singleton y_2, which will not change the degree of any other literals. So y_2 will still be left as a (2,1)-singleton. So we can further reduce one clause directly or branch on y_2 with (1,8) at least by Lemma 3. For the worst case, we can branch with $(3, 3+1, 3+8, 10) = (3,4,11,10)$.

4.7 Step 6: Dealing With $(2,2)$-Literals

After Step 5, there are only (2,2)-literals, $(3,1)$-singletons, and $(4,1)$-singletons. Let x be a $(2,2)$-literal with the four clauses xC_1, xC_2, $\overline{x}D_1$, and $\overline{x}D_2$. None of C_1, C_2, D_1, and D_2 is empty, otherwise there would be a unit clause containing a (2,2)-literal, and Step 4 could be applied.

Step 6.1: D_1 and D_2 contain the same literal. Assume that D_1 and D_2 contain the same literal y. We can branch on x with a branching vector not worse than $(4,2)$.

Note that y can not be a $(3,1)$/$(4,1)$-singleton, otherwise R-Rule 9 could be applied. So y is a $(2,2)$-literal. Furthermore, literal $\overline{y}$ is not in C_1 or C_2 otherwise R-Rules 7 or 8 would be applied. In the branching $x = 1$, we reduce two clauses. In the branching $x = 0$, we can reduce four clauses containing $\overline{x}$ or $\overline{y}$. We can branch with $(2,4)$ at least.

Step 6.2: each of D_1 and D_2 contains a singleton and at least one is a $(4,1)$-singleton. Assume that $y_1 \in D_1$ is a

(4,1)-singleton and $y_2 \in D_2$ is a singleton. We can branch on y_1 with a branching vector not worse than $(6, 1)$.

In the branching $y_1 = 1$, four clauses containing y_1 and the unit clause $\overline{y}$ are reduced. Furthermore, x becomes a (2,1)-literal with singleton y_2 in the clause containing $\overline{x}$. We can further reduce one clause by applying R-Rule 6. Thus, we can get a branching vector $(6, 1)$ at least.

The proofs of the following two properties were omitted due to the limited space.

Property 1. *If D_1D_2 (resp.,C_1C_2) contains only $(3, 1)$-singletons, then in the branching $x = 0$ (reps., $x = 1$), we can branch with $(18, 11, 11, 4)$ at least.*

Property 2. *If D_1D_2 (resp.,C_1C_2) contains a (2,2)-literal, then in the branching $x = 0$ (reps., $x = 1$) we can branch with $(11, 3)$ or $(11, 10, 4)$ at least.*

It seems that we can use Properties 1 and 2 to handle the remaining cases. However, direct application of them can not get a good enough branching rule. For example, we branch on x, and in each branching, we get a branching vector $(11, 3)$. Then we can only get $(11, 11, 3, 3)$, which has a branching factor $1.3 > 1.2886$. So we still consider the following two cases.

Step 6.3: D_1 contains a $(2, 2)$-literal y_1 and D_2 contains a singleton y_2. We show that we can branch with a branching vector not worse than $(11, 3, 3)$.

We branch on the $(2, 2)$-literal y_1. In the branching $y_1 = 1$, we first reduce the two clauses containing y_1. After this, x will become a (2,1)-literal such that the clause containing $\overline{x}$ also contains a singleton y_2 and we can further reduce at least one clause by applying R-Rule 6. So m decreases by at least 3 in this branching. In the branching $y_1 = 0$, we can branch with either $(11, 3)$ or $(11, 10, 4)$ or $(18, 11, 11, 4)$ by Properties 1 and 2. For the worst case, we can branch with $(11, 3, 3)$ for this step.

By the symmetry, Step 6.3 will also deal with the case that one of C_1 and C_2 contains a $(2, 2)$-literal and the other contains a singleton.

Step 6.4: There still exists a $(2, 2)$-literal x. For this case, we can branch with $(18, 18, 11, 11, 11, 11, 4, 4)$ at least.

We branch on the $(2, 2)$-literal x. We analyze the branching $x = 0$. Assume there is a $(2, 2)$-literal $y_1 \in D_1$. Since Step 6.3 is not applicable now, we know that there is another $(2, 2)$-literal $y_2 \in D_2$. Both of y_1 and y_2 will become 3-variables after assigning $x = 0$, where literal y_1 is a (1,2)-literal such that the clause containing y has length at least 2 otherwise R-Rule 4 would be applied before assigning $x = 0$. By Lemma 3 with $|D| \geq 1$, we can branch on y_1 with $(9, 1)$ first, and in the branching $\mathcal{F}_{\backslash y_1}$, we still can branch on 3-variable y_2 with $(8, 1)$ at least. So we get the branching vector $(2 + 9, 2 + 1 + 8, 2 + 1 + 1) = (11, 11, 4)$. Next we can assume that none of D_1 and D_2 contains $(2, 2)$-literals. By Property 1, we can branch with $(18, 11, 11, 4)$ at least. So we can always branch with $(18, 11, 11, 4)$ at least. For the branching $x = 1$, we can get the same result by symmetry. So we can always branch with $(18, 18, 11, 11, 11, 11, 4, 4)$.

4.8 Step 7: Only $(3, 1)/(4, 1)$-Singletons

This case can be reduced to a set cover problem. Assume in the MaxSAT instance, there are m clauses and n $(i, 1)$-singletons x $(i = 3, 4)$. Besides the n clauses $\overline{x}$, all the other $m - n$ clauses are considered as the elements to be covered. Each $(i, 1)$-singleton x is regarded as a set that covers exactly i elements corresponding to the i clauses containing x. So the MaxSAT problem is equal to find a minimum number of sets to cover all elements. As shown in [Bliznets and Golovnev, 2012], this case can be quickly solved by using fast algorithms for the set cover problem.

Lemma 5. ([Bliznets and Golovnev, 2012]) *MaxSAT with the formula containing only $(3, 1)$/$(4, 1)$-singletons can be solved in $O^*(1.2576^m)$ time.*

5 The Final Result

Now we present our main result that MaxSAT can be solved in $O^*(1.2886^m)$ time. The algorithm will apply the above seven branching steps in order. However, before applying each branching step, we first iteratively apply our reduction rules in order until the instance becomes reduced. We only execute our branching operations on reduced instances. All the reductions can be done in polynomial time. For the first six branching steps, the biggest branching vector is 1.2886, which is corresponding to the branching vector (6,6,5,5) in Step 5.4. In the last step, we solve the remaining instance in $O^*(1.2576^m)$ time directly by Lemma 5. So we can solve the whole instance in time $O^*(1.2886^m)$. It is clear that each step of the algorithm uses only polynomial space.

Theorem 1. *MaxSAT can be solved in $O^*(1.2886^m)$ time and polynomial space.*

6 Conclusion

In this paper, we give a new algorithm for MaxSAT and improve the running time bound from $O^*(1.2989^m)$ to $O^*(1.2886^m)$. For such a classic problem, any small progress is not easy nowadays. To get the improvement, we need to well understand previous techniques and bottleneck cases, and also need to use new reduction and analysis techniques. While improving the running time bound, we reveal more properties of the problem and refine the branching strategies for $(2, 2)$-variables and $(3, 2)$-variables.

Our running time bound was obtained by purely worst-case analysis. In practice, some reduction and branching operations may have a good performance on practical instances, which deserves further study. As mentioned in [Morgado *et al.*, 2013] in some cases branch-and-search algorithms are the best option to solve the problem. So it is also interesting to integrate our reduction and branching techniques to fast solvers.

Acknowledgements

The work is supported by the National Natural Science Foundation of China, under grant 61972070.

References

[Alferov and Bliznets, 2021] Vasily Alferov and Ivan Bliznets. New length dependent algorithm for maximum satisfiability problem. In *AAAI 2021, Virtual Event, February 2-9, 2021*, pages 3634–3641, 2021.

[Bacchus *et al.*, 2021] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In *Handbook of Satisfiability (Second Edition)*, pages 929–991. IOS Press, 2021.

[Bansal and Raman, 1999] Nikhil Bansal and Venkatesh Raman. Upper bounds for maxsat: Further improved. In Alok Aggarwal and C. Pandu Rangan, editors, *ISAAC '99*, volume 1741 of *LNCS*, pages 247–258. Springer, 1999.

[Bliznets and Golovnev, 2012] Ivan Bliznets and Alexander Golovnev. A new algorithm for parameterized MAX-SAT. In Dimitrios M. Thilikos and Gerhard J. Woeginger, editors, *Parameterized and Exact Computation - 7th International Symposium, IPEC 2012*, volume 7535 of *LNCS*, pages 37–48. Springer, 2012.

[Bonet *et al.*, 2007] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for max-sat. *Artif. Intell.*, 171(8-9):606–618, 2007.

[Cai *et al.*, 2017] Shaowei Cai, Chuan Luo, and Haochen Zhang. From decimation to local search and back: A new approach to maxsat. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 571–577. ijcai.org, 2017.

[Chen and Kanj, 2004] Jianer Chen and Iyad A. Kanj. Improved exact algorithms for MAX-SAT. *Discret. Appl. Math.*, 142(1-3):17–27, 2004.

[Chen *et al.*, 2017] Jianer Chen, Chao Xu, and Jianxin Wang. Dealing with 4-variables by resolution: An improved maxsat algorithm. *Theor. Comput. Sci.*, 670:33–44, 2017.

[Chu *et al.*, 2021] Huairui Chu, Mingyu Xiao, and Zhe Zhang. An improved upper bound for SAT. In *AAAI 2021, Virtual Event, February 2-9, 2021*, pages 3707–3714, 2021.

[Cook, 1971] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[Fomin and Kratsch, 2010] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Hirsch, 1998] Edward A. Hirsch. Two new upper bounds for SAT. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1998, San Francisco, California, USA*, pages 521–530, 1998.

[Hutter *et al.*, 2017] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger H. Hoos, and Kevin Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artif. Intell.*, 243:1–25, 2017.

[Impagliazzo and Paturi, 2001] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.

[Li and Manyà, 2021] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability (Second Edition)*, pages 903–927. IOS Press, 2021.

[Li *et al.*, 2016] Chu Min Li, Felip Manyà, and Joan Ramon Soler. A clause tableau calculus for maxsat. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 766–772, 2016.

[Monien *et al.*, 1981] Burkhard Monien, Ewald Speckenmeyer, and Oliver Vornberger. Upper bounds for covering problems. *Methods of operations research*, 43:419–431, 1981.

[Morgado *et al.*, 2013] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013.

[Niedermeier and Rossmanith, 1999] Rolf Niedermeier and Peter Rossmanith. New upper bounds for maxsat. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP'99*, volume 1644 of *LNCS*, pages 575–584. Springer, 1999.

[Peng and Xiao, 2021] Junqiang Peng and Mingyu Xiao. A fast algorithm for SAT in terms of formula length. In Chu-Min Li and Felip Manyà, editors, *SAT 2021 - 24th International Conference*, volume 12831 of *LNCS*, pages 436–452. Springer, 2021.

[Robinson, 1965] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Xiao *et al.*, 2021] Mingyu Xiao, Sen Huang, Yi Zhou, and Bolin Ding. Efficient reductions and a fast algorithm of maximum weighted independent set. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Ljubljana, Slovenia, April 19-23, 2021*, pages 3930–3940, 2021.

[Xu *et al.*, 2019] Chao Xu, Wenjun Li, Yongjie Yang, Jianer Chen, and Jianxin Wang. Resolution and domination: An improved exact maxsat algorithm. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1191–1197, 2019.

[Yamamoto, 2005] Masaki Yamamoto. An improved $O(1.234^m)$-time deterministic algorithm for SAT. In Xiaotie Deng and Ding-Zhu Du, editors, *Algorithms and Computation, 16th International Symposium, ISAAC 2005*, volume 3827 of *LNCS*, pages 644–653. Springer, 2005.