# A Fast Algorithm for SAT in Terms of Formula Length

Junqiang Peng and Mingyu Xiao(✉)

School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China

**Abstract.** In this paper, we prove that the general CNF satisfiability problem can be solved in $O^*(1.0646^L)$ time, where $L$ is the length of the input CNF-formula (*i.e.*, the total number of literals in the formula), which improves the current bound $O^*(1.0652^L)$ given by Chen and Liu 12 years ago. Our algorithm is a standard branch-and-search algorithm analyzed by using the measure-and-conquer method. We avoid the bottleneck in Chen and Liu's algorithm by simplifying the branching operation for 4-degree variables and carefully analyzing the branching operation for 5-degree variables. To simplify case-analyses, we also introduce a general framework for analysis, which may be able to be used in other problems.

**Keywords:** Parameterized algorithms · Satisfiability · Measure-and-conquer

## 1 Introduction

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates to true. By SAT, we mean the problem of propositional satisfiability for formulas in conjunctive normal form (CNF) [5]. The SAT problem is the first problem proved to be NP-complete [4] and it plays an important role in computational complexity and artificial intelligence [1]. There are numerous investigations on this problem in different fields, such as approximation algorithms, randomized algorithms, heuristic algorithms, and exact and parameterized algorithms. In this paper, we study parameterized algorithms for SAT parameterized by the input length.

To measure the running time bound for the SAT problem, there are three frequently used parameters: the number of variables $n$, the number of clauses $m$, and the input length $L$. The input length $L$ is defined as the sum of the number of literals in each clause. The number of variables $n$ should be the most basic parameter. The simple brute force algorithm to try all $2^n$ possible assignments of the $n$ variables will get the running time bound of $O^*(2^n)$.[1] After decades of

[1] The $O^*$ notation supervises all polynomial factors, i.e., $f(n) = O^*(g(n))$ means $f(n) = O(g(n)n^{O(1)})$.


C.-M. Li and F. Manyà (Eds.): SAT 2021, LNCS 12831, pp. 436–452, 2021.
https://doi.org/10.1007/978-3-030-80223-3_30

**Table 1.** Previous and our upper bound for SAT

| Running time bounds | References |
|---|---|
| $O^*(1.0927^L)$ | Van Gelder 1988 [16] |
| $O^*(1.0801^L)$ | Kullmann and Luckhardt 1997 [12] |
| $O^*(1.0758^L)$ | Hirsch 1998 [9] |
| $O^*(1.074^L)$ | Hirsch 2000 [10] |
| $O^*(1.0663^L)$ | Wahlström 2005 [17] |
| $O^*(1.0652^L)$ | Chen and Liu 2009 [2] |
| $O^*(1.0646^L)$ | **This paper 2021** |

hard work, no one can break this trivial bound. The Strong Exponential Time Hypothesis conjectures that the SAT problem cannot be solved in time $O^*(c^n)$ for some constant $c < 2$ [11]. For a restricted version, the $k$-SAT problem (the length of each clause in the formula is bounded by a constant $k$), better results have been developed. For example, 3-SAT can be solved in $O^*(1.3279^n)$ time [13], 4-SAT can be solved in $O^*(1.4986^n)$ time [13], and $k$-SAT can be solved in $O^*(c(k)^n)$ time for some value $c(k)$ depending on $k$ [13]. When it comes to the parameter $m$, Monien *et al.* first gave an algorithm with time complexity $O^*(1.260^m)$ in 1981 [14]. Later, the bound was improved to $O^*(1.239^m)$ by Hirsch in 1998 [9], and then improved to $O^*(1.234^m)$ by Yamamoto in 2005 [19]. Now the best result is $O^*(1.2226^m)$ obtained by Chu, Xiao, and Zhang [3].

The input length $L$ is another important and frequently studied parameter. It is probably the most precise parameter to describe the input CNF-Formula. From the first algorithm with running time bound $O^*(1.0927^L)$ by Van Gelder in 1988 [16], the result was improved several times. In 1997, the bound was improved to $O^*(1.0801)$ by Kullmann and Luckhardt [12]. In 1998, the bound was improved to $O^*(1.0758^L)$ by Hirsch [9], and improved again by Hirsch to $O^*(1.074^L)$ in 2000 [10]. Then Wahlström gave an $O^*(1.0663^L)$-time algorithm in 2005 [17]. In 2009, Chen and Liu [2] used the measure-and-conquer method to analyze the running time bound and further improved the result to $O^*(1.0652^L)$. We list the major progress and our result in Table 1.

Our algorithm, as well as most algorithms for the SAT problem, is based on the branch-and-search process. The idea of branch-and-search is simple and practical: for a given CNF-formula $\mathcal{F}$, we iteratively branch on a variable or literal $x$ into two branches by assigning value 1 or 0 to it. Let $\mathcal{F}_{x=1}$ and $\mathcal{F}_{\overline{x}=1}$ be the resulted CNF-formula by assigning value 1 and 0 to $x$, respectively. It holds that $\mathcal{F}$ is satisfiable if and only if at least one of $\mathcal{F}_{x=1}$ and $\mathcal{F}_{\overline{x}=1}$ is satisfiable. To get a running time bound, we need to analyze how much the parameter $L$ can decrease in each branch. To break some bottlenecks in direct analysis, some references [2,17] analyzed the algorithm based on some other measures and gave the relation between the new measure and $L$. The current best result [2] was obtained by using the measure-and-conquer method, which is also to use a new

measure. This is the first time to bring the measure-and-conquer method to this research line. In this paper, we further improve the running time bound by still using the measure-and-conquer method. Similar to many measure-and-conquer algorithms, our algorithm and the algorithm in [2] deal with variables from high degree to low degree. The algorithm in [2] carefully analyzed branching operations for variables of degree 4. Our algorithm will simplify the branching operation for 4-degree variables and carefully analyze the branching operation for 5-degree variables. Finally, we can improve the bound to $O^*(1.0646^L)$.

Due to the limited space, the proofs of some lemmas marked with (*) are omitted, which can be found in the full version of this paper [15].

## 2 Preliminaries

Let $V = \{x_1, x_2, ..., x_n\}$ denote a set of $n$ boolean *variables*. Each variable $x_i$ ($i \in \{1, 2, ..., n\}$) has two corresponding *literals*: positive literal $x_i$ and negative literal $\overline{x_i}$ (we use $\overline{x}$ to denote the negation of a literal $x$, and $\overline{\overline{x}} = x$). A *clause* on $V$ consists of some literals on $V$. Note that we allow a clause to be empty. A clause $\{z_1, z_2, \ldots, z_q\}$ is also simply written as $z_1 z_2 \ldots z_q$. Thus, we use $zC$ to denote the clause containing literal $z$ and all literals in clause $C$. We also use $C_1C_2$ to denote the clause containing all literals in clauses $C_1$ and $C_2$. We use $\overline{C}$ to denote a clause that contains the negation of every literal in clause $C$. That is, if $C = z_1 z_2 ... z_q$, then $\overline{C} = \overline{z_1}\,\overline{z_2} ... \overline{z_q}$. A *CNF-formula* on $V$ is the conjunction of a set of clauses $\mathcal{F} = \{C_1, C_2, ..., C_m\}$. When we say a variable $x$ is contained in a clause (or a formula), it means that the clause (at least one clause of the formula) contains a literal $x$ or its negative $\overline{x}$.

An assignment for $V$ is a map $A : V \to \{0, 1\}$. A clause $C_j$ is *satisfied* by an assignment if and only if there exists at least one literal in $C_j$ such that the assignment makes its value 1. A CNF-formula is *satisfied* by an assignment $A$ if and only if each clause in it is satisfied by $A$. We say a CNF-formula is *satisfiable* if it can be satisfied by at least one assignment. We may assign value 0 or 1 to a literal, which is indeed to assign a value to its variable to make the corresponding literal 0 or 1.

A literal $z$ is called an $(i, j)$-*literal* (resp., an $(i^+, j)$-*literal* or $(i^-, j)$-*literal*) in a formula $\mathcal{F}$ if $z$ appears $i$ (resp. at least $i$ or at most $i$) times and $\overline{z}$ appears $j$ times in the formula $\mathcal{F}$. Similarly, we can define $(i, j^+)$-literal, $(i, j^-)$-literal, $(i^+, j^+)$-literal, $(i^-, j^-)$-literal, and so on. Note that literal $z$ is an $(i, j)$-literal if and only if literal $\overline{z}$ is a $(j, i)$-literal. A variable $x$ is an $(i, j)$-*variable* if the positive literal $x$ is an $(i, j)$-literal. For a variable or a literal $x$ in formula $\mathcal{F}$, the *degree* of it, denoted by $deg(x)$, is the number of $x$ appearing in $\mathcal{F}$ plus the number of $\overline{x}$ appearing in $\mathcal{F}$, i.e., $deg(x) = i + j$ for an $(i, j)$-variable or $(i, j)$-literal $x$. A $d$-*variable* (resp., $d^+$-*variable* or $d^-$-*variable*) is a variable with the degree exactly $d$ (resp., at least $d$ or at most $d$). The degree of a formula $\mathcal{F}$ is the maximum degree of all variables in $\mathcal{F}$. For a clause or a formula $C$, the set of variables whose literal appears in $C$ is denoted by $var(C)$.

The *length* of a clause $C$, denoted by $|C|$, is the number of literals in $C$. A clause is a $k$-*clause* or $k^+$-*clause* if the length of it is $k$ or at least $k$. We use

$L(\mathcal{F})$ to indicate the length of a formula $\mathcal{F}$. It is the sum of the lengths of all clauses in $\mathcal{F}$, which is also the sum of the degrees of all variables in $\mathcal{F}$. A formula $\mathcal{F}$ is called *$k$-CNF formula* if each clause in $\mathcal{F}$ has a length of at most $k$.

In a formula $\mathcal{F}$, a literal $x$ is called a *neighbor* of a literal $z$ if there is a clause containing both $z$ and $x$. The set of neighbors of a literal $z$ in a formula $\mathcal{F}$ is denoted by $N(z, \mathcal{F})$. We also use $N^{(k)}(x, \mathcal{F})$ (resp., $N^{(k+)}(z, \mathcal{F})$) to denote the neighbors of $z$ in $k$-clauses (resp., $k^+$-clauses) in $\mathcal{F}$, *i.e.*, for any $z' \in N^{(k)}(z, \mathcal{F})$ (resp., $z' \in N^{(k+)}(z, \mathcal{F})$), there exists a $k$-clause (resp., $k^+$-clause) containing both $z$ and $z'$.

## 3 Branch-and-Search and Measure-and-Conquer

Our algorithm is a standard branch-and-search algorithm, which first applies some reduction rules to reduce the instance as much as possible and then searches for a solution by branching. The branching operations may exponentially increase the running time. We will use a measure to evaluate the size of the search tree generated in the algorithm. For the SAT problem, the number of variables or clauses of the formula is a commonly used measure. More fundamentals of branching heuristics about the SAT problem can be found in [1].

We use $T(\mu)$ to denote the maximum size or number of leaves of the search tree generated by the algorithm for any instance with the measure being at most $\mu$. For a branching operation that branches on the current instance into $l$ branches with the measure decreasing by at least $a_i$ in the $i$-th branch, we get a a recurrence relation

$$T(\mu) \leq T(\mu - a_1) + T(\mu - a_2) + \cdots + T(\mu - a_l).$$

The recurrence relation can also be simply 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 $\gamma$, then $T(\mu) = O(\gamma^\mu)$. If on each node of the search tree, the algorithm runs in polynomial time, then the total running time of the algorithm is $O^*(\gamma^\mu)$. For two branching vectors $\mathbf{a} = [a_1, a_2, \ldots, a_l]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_l]$, if $a_i \geq b_i$ holds for all $i = 1, 2 \ldots, l$, then the branching factor of $\mathbf{a}$ is not greater than that of $\mathbf{b}$. For this case, we say $\mathbf{b}$ *dominates* $\mathbf{a}$. This property will be used in many places to simplify some arguments in the paper. More details about analyzing recurrences can be found in the monograph [8].

The measure-and-conquer method [7] is a powerful tool to analyze branch-and-search algorithms. The main idea of the method is to adopt a new measure in the analysis of the algorithm. For example, instead of using the number of variables as the measure, it may set weights to different variables and use the sum of all variable weights as the measure. This method may be able to catch more structural properties and then get further improvements. Nowadays, the fastest exact algorithms for many NP-hard problems were designed by using this method. In this paper, we will also use the measure-and-conquer method.

We introduce a weight to each variable in the formula according to the degree of the variable, $w\colon \mathbb{Z}^+ \to \mathbb{R}^+$, where $\mathbb{Z}^+$ and $\mathbb{R}^+$ denote the sets of nonnegative integers and nonnegative reals, respectively. Let $w_i$ denote the weight of a variable with degree $i$. A variable with lower degree will not receive a higher weight. *i.e.*, $w_i \geq w_{i-1}$. In our algorithm, the measure of a formula $\mathcal{F}$ is defined as

$$\mu(\mathcal{F}) = \sum_x w_{deg(x)}. \tag{1}$$

In other words, $\mu(\mathcal{F})$ is the sum of the weight of all variables in $\mathcal{F}$. Let $n_i$ denote the number of $i$-variables in $\mathcal{F}$. Then we also have that $\mu(\mathcal{F}) = \sum_i w_i n_i$.

One important step is to set the value of weight $w_i$. Different values of $w_i$ will generate different branching vectors and factors. We need to find a good setting of $w_i$ so that the worst branching factor is as small as possible. We will get the value of $w_i$ by solving a quasiconvex program after listing all our branching vectors. However, we pre-specify some requirements of the weights to simplify arguments. Some similar assumptions were used in previous measure-and-conquer algorithms. We set the weight such that

$$\begin{aligned} &w_1 = w_2 = 0, \\ &0 < w_3 < 2, w_4 = 2w_3, \text{ and} \\ &w_i = i \text{ for } \ i \geq 5. \end{aligned} \tag{2}$$

We use $\delta_i$ to denote the difference between $w_i$ and $w_{i-1}$ for $i > 0$, *i.e.*, $\delta_i = w_i - w_{i-1}$. By (2), we have

$$w_3 = \delta_3 = \delta_4. \tag{3}$$

We also assume that

$$\begin{aligned} &\delta_i \leq \delta_{i-1} \text{ for } i \geq 3, \text{ and} \\ &w_3 \geq \delta_5. \end{aligned} \tag{4}$$

Under these assumptions, it holds that $w_i \leq i$ for each $i$. Thus, we have

$$\mu(\mathcal{F}) \leq L(\mathcal{F}). \tag{5}$$

This tells us that if we can get a running time bound of $O^*(c^{\mu(\mathcal{F})})$ for a real number $c$, then we also get a running time bound of $O^*(c^{L(\mathcal{F})})$ for this problem. To obtain a running time bound in terms of the formula length $L(\mathcal{F})$, we consider the measure $\mu(\mathcal{F})$ and show how much the measure $\mu(\mathcal{F})$ decreases in the branching operations of our algorithm and find the worst branching factor among all branching vectors.

## 4 The Algorithm

We will first introduce our algorithm and then analyze its running time bound by using the measure-and-conquer method. Our algorithm consists of reduction operations and branching operations. When no reduction operations can be applied anymore, the algorithm will search for a solution by branching. We first introduce our reduction rules.

### 4.1 Reduction Rules

We have ten reduction rules. They are well-known and frequently used in the literature (see [2,17] for examples). So we may omit the proofs of the correctness of some rules. We introduce the reduction rules in the order as stated and a reduction rule will be applied in our algorithm only when all previous reduction rules can not be applied on the instance.

**R-Rule 1 (Elimination of duplicated literals).** *If a clause $C$ contains duplicated literals $z$, remove all but one $z$ in $C$.*

**R-Rule 2 (Elimination of subsumptions).** *If there are two clauses $C$ and $D$ such that $C \subseteq D$, remove clause $D$.*

**R-Rule 3 (Elimination of tautology).** *If a clause $C$ contains two opposite literals $z$ and $\overline{z}$, remove clause $C$.*

**R-Rule 4 (Elimination of 1-clauses and pure literals).** *If there is a 1-clause $\{x\}$ or a $(1^+, 0)$-literal $x$, assign $x = 1$.*

*Davis-Putnam Resolution*, proposed in [6], is a classic and frequently used technology for SAT. Let $\mathcal{F}$ be a CNF-formula and $x$ be a variable in $\mathcal{F}$. Assume that clauses containing literal $x$ are $xC_1, xC_2, ..., xC_a$ and clauses containing literal $\overline{x}$ are $\overline{x}D_1, \overline{x}D_2, ..., \overline{x}D_b$. A *Davis-Putnam resolution* on $x$ is to construct a new CNF-formula $DP_x(\mathcal{F})$ by the following method: initially $DP_x(\mathcal{F}) = \mathcal{F}$; add new clauses $C_iD_j$ for each $1 \leq i \leq a$ and $1 \leq j \leq b$; and remove $xC_1, xC_2, ..., xC_a, \overline{x}D_1, \overline{x}D_2, ..., \overline{x}D_b$ from the formula. It is known that

**Proposition 1 ([6]).** *A CNF-formula $\mathcal{F}$ is satisfiable if and only if $DP_x(\mathcal{F})$ is satisfiable.*

In the resolution operation, each new clause $C_iD_j$ is called a *resolvent*. A resolvent is *trivial* if it contains both a literal and the negation of it. Since trivial resolvents will always be satisfied, we can simply delete trivial resolvents from the instance directly. So when we do resolutions, we assume that all trivial resolvents will be deleted.

**R-Rule 5 (Trivial resolution).** *If there is a variable $x$ with at most one non-trivial resolvent, then apply resolution on $x$.*

**R-Rule 6 ([2]).** *If there are a 2-clause $z_1z_2$ and a clause $C$ containing both $z_1$ and $\overline{z_2}$, then remove $\overline{z_2}$ from $C$.*

**R-Rule 7.** *If there are two clauses $z_1z_2C_1$ and $z_1\overline{z_2}C_2$, where literal $\overline{z_2}$ appears in no other clauses, then remove $z_1$ from clause $z_1z_2C_1$.*

**Lemma 1.** *(*) Let $\mathcal{F}$ be a CNF-formula and $\mathcal{F}'$ be the resulting formula after applying R-Rule 7 on $\mathcal{F}$. Then $\mathcal{F}$ is satisfiable if and only if $\mathcal{F}'$ is satisfiable.*

**R-Rule 8 ([2]).** *If there is a 2-clause $z_1z_2$ and a clause $\overline{z_1z_2}C$ such that literal $\overline{z_1}$ appears in no other clauses, remove the clause $z_1z_2$ from $\mathcal{F}$.*

**R-Rule 9** ([2]). *If there is a 2-clause $z_1 z_2$ such that either literal $z_1$ appears only in this clause or there is another 2-clause $\overline{z_1} z_2$, then replace $z_1$ with $\overline{z_2}$ in $\mathcal{F}$ and then apply R-Rule 3 as often as possible.*

**R-Rule 10** ([2]). *If there are two clauses $CD_1$ and $CD_2$ such that $|D_1|, |D_2| \geq 1$ and $|C| \geq 2$, then remove $CD_1$ and $CD_2$ from $\mathcal{F}$, and add three new clauses $xC$, $\overline{x}D_1$, and $\overline{x}D_2$, where $x$ is a new 3-variable.*

This is like the Davis-Putnam resolution in reverse and thus it is correct.

**Definition 1 (Reduced formulas).** *A CNF-formula $\mathcal{F}$ is called reduced, if none of the above reduction rules can be applied on it.*

Our algorithm will first iteratively apply above reduction rules in the order to get a reduced formula. We will use $R(\mathcal{F})$ to denote the resulting reduced formula obtained from $\mathcal{F}$. Next, we show some properties of reduced formulas.

**Lemma 2.** *(*) In a reduced CNF-formula $\mathcal{F}$, all variables are $3^+$-variables.*

**Lemma 3.** *(*) In a reduced CNF-formula $\mathcal{F}$, if there is a 2-clause $xy$, then no other clause in $\mathcal{F}$ contains $xy$, $\overline{x}y$, or $x\overline{y}$.*

**Lemma 4.** *(*) In a reduced CNF-formula $\mathcal{F}$, if there is a clause $xyC$, then*

*(i) no other clause contains $xy$;*
*(ii) no other clause contains $\overline{x}y$ or $\overline{xy}$ if $x$ is a 3-variable.*

**Lemma 5.** *(*) In a reduced CNF-formula $\mathcal{F}$, if there is $(1,i)$-literal $x$ and $xC$ is the only clause containing $x$, then*

*(i) $|C| \geq 2$;*
*(ii) all variables in $C$ are different from all variables in $N^{(2)}(\overline{x}, F)$, that is, if $y \in N^{(2)}(\overline{x}, F)$, then $y, \overline{y} \notin C$.*

### 4.2 Branching Rules and the Algorithm

After getting a reduced formula, we will search for a solution by branching. In a branching operation, we will generate two smaller CNF-formulas such that the original formula is satisfiable if and only if at least one of the two new formulas is satisfiable. The two smaller formulas are generated by specifying the value of a set of literals in the original formula.

The simplest branching rule is that we pick up a variable or literal $x$ from $\mathcal{F}$ and branch into two branches $\mathcal{F}_{x=1}$ and $\mathcal{F}_{x=0}$, where $\mathcal{F}_{x=1}$ and $\mathcal{F}_{x=0}$ are the formulas after assigning $x = 1$ and $x = 0$ in $\mathcal{F}$, respectively. When the picked literal $x$ is a $(1, 1^+)$-literal, we will apply a stronger branching. Assume that $xC$ is the only clause containing $x$. Then we branch into two branches $\mathcal{F}_{x=1\ \&\ C=0}$ and $\mathcal{F}_{x=0}$, where $\mathcal{F}_{x=1\ \&\ C=0}$ is the resulting formula after assigning 1 to $x$ and 0 to all literals in $C$ in $\mathcal{F}$. The correctness of this branching operation is also easy to observe. Only when all literals in $C$ are assigned 0, we need to assign 1 to $x$.

**Algorithm 1:** SAT($\mathcal{F}$)

**Input**: a CNF-formula $\mathcal{F}$
**Output**: 1 or 0 to indicate the satisfiability of $\mathcal{F}$
**Step 1.** If $\mathcal{F} = \emptyset$, return 1. If $\mathcal{F}$ contains an empty clause, return 0.
**Step 2.** If $\mathcal{F}$ is not a reduced CNF-formula, iteratively apply the reduction rules to reduce it.
**Step 3.** If there is a $d$-variable $x$ with $d \geq 6$, return SAT($\mathcal{F}_{x=1}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 4.** If there is a $(1,4)$-literal $x$ (assume $xC$ is the only clause containing $x$), return SAT($\mathcal{F}_{x=1\ \&\ C=0}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 5.** If there is a 5-variable $x$ contained in a 2-clause, return SAT($\mathcal{F}_{x=1}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 6.** If there is a 5-variable $x$ contained in a $4^+$-clause, return SAT($\mathcal{F}_{x=1}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 7.** If there is a clause containing both a 5-variable $x$ and a $4^-$-variable, return SAT($\mathcal{F}_{x=1}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 8.** If there are still some 5-variables, then $\mathcal{F} = \mathcal{F}^* \wedge \mathcal{F}'$, where $\mathcal{F}^*$ is a 3-CNF with $var(\mathcal{F}^*)$ be the set of 5-variables in $\mathcal{F}$ and $var(\mathcal{F}^*) \cap var(\mathcal{F}') = \emptyset$. We return SAT($\mathcal{F}^*$) $\wedge$ SAT($\mathcal{F}'$) and solve $\mathcal{F}^*$ by using the 3-SAT algorithm by Liu [13].
**Step 9.** If there is a $(1,3)$-literal $x$ (assume $xC$ is the only clause containing $x$), return SAT($\mathcal{F}_{x=1\ \&\ C=0}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 10.** If there is a $(2,2)$-literal $x$, return SAT($\mathcal{F}_{x=1}$)$\vee$SAT($\mathcal{F}_{x=0}$).
**Step 11.** Apply the algorithm by Wahlström [18] to solve the instance.

The main steps of our algorithm for the SAT problem are given in Algorithm 1. The algorithm will execute one step only when all previous steps can not be applied. In Step 2, the algorithm first reduces the formula by applying the reduction rules. Step 3 will branch on a variable of degree $\geq 6$ if it exists. Steps 4–8 deal with 5-variables. Note that if Steps 1–7 do not apply, then $\mathcal{F}$ can be written as $\mathcal{F} = \mathcal{F}^* \wedge \mathcal{F}'$, where $\mathcal{F}^*$ is a 3-CNF with $var(\mathcal{F}^*)$ be the set of 5-variables in $\mathcal{F}$ and $var(\mathcal{F}^*) \cap var(\mathcal{F}') = \emptyset$. So we can do Step 8. Steps 9–10 deal with 4-variables. When the algorithm comes to the last step, all variables must have a degree of 3 and the algorithm deals with this special case.

We compare our algorithm with the previous algorithm by Chen and Liu [2]. We can see that they used a simple and uniform branching rule to deal with variables of degree at least 5 and used careful and complicated branching rules for 4-variables. Their bottlenecks contain one case of branching on $(2,3)$-variables (or $(3,2)$-variables) and one case of dealing with 4-variables. To get further improvements, we carefully design and analyze the branching rules for 5-variables to avoid one previous bottleneck, and also refine the branching rules for 4-variables.

## 5 Framework of the Analysis

We use the measure-and-conquer method to analyze the running time bound of our algorithm, and adopt $\mu(\mathcal{F})$ defined in (1) as the measure to construct

recurrence relations for our branching operations. Before analyzing each detailed step of the algorithm, we first introduce the general framework of our analysis.

In each sub-branch of a branching operation, we assign value 1 or 0 to some literals and remove some clauses and literals. If we assign value 1 to a literal $x$ in the formula $\mathcal{F}$, then we will remove all clauses containing $x$ and all $\overline{x}$ literals from the clauses containing $\overline{x}$. The assignment and removing together are called an *assignment operation*. We may assign values to more than one literal and we do assignment operations for each literal. Let $S$ be a subset of literals. We use $\mathcal{F}_{S=1}$ to denote the resulting formula after assigning 1 to each literal in $S$ and doing assignment operations. Note that $\mathcal{F}_{S=1}$ may not be a reduced formula and we will apply our reduction rules to reduce it. We use $\mathcal{F}'_{S=1}$ to denote the reduced formula obtained from $\mathcal{F}_{S=1}$, i.e., $\mathcal{F}'_{S=1} = R(\mathcal{F}_{S=1})$. We analyze how much we can reduce the measure in each branch by establishing some lower bounds for

$$\Delta_S = \mu(\mathcal{F}) - \mu(\mathcal{F}'_{S=1}).$$

We also define

$$\xi_S^{(1)} = \mu(\mathcal{F}) - \mu(\mathcal{F}_{S=1});$$

$$\xi_S^{(2)} = \mu(\mathcal{F}_{S=1}) - \mu(\mathcal{F}'_{S=1}).$$

Thus, $\Delta_S = \xi_S^{(1)} + \xi_S^{(2)}$.

In a branching operation, we will branch into two sub branches. Assume that the set of literals in $S_1$ are assigned the value in the first sub branch and the set of literals in $S_2$ are assigned the value in the second sub branch. If we can show

$$\min(\Delta_{S_1}, \Delta_{S_2}) \geq a \quad \text{and} \quad \Delta_{S_1} + \Delta_{S_2} \geq b,$$

then we can always get a branching vector covered by one of

$$[a, b-a] \quad \textit{and} \quad [b-a, a].$$

This technique will be frequently used in our analysis.

### 5.1 Some Lower Bounds

Next, we show some detailed lower bounds for $\Delta_S$ (as well as for $\Delta_{S_1} + \Delta_{S_2}$). We first consider $\xi_S^{(1)}$ and $\xi_S^{(2)}$.

According to the assignment operation, we know that all variables of the literals in $S$ will be deleted in $\mathcal{F}_{S=1}$. So we have a trivial bound

$$\xi_S^{(1)} \geq \sum_{v \in S} w_{deg(v)}. \tag{6}$$

To get better bounds, we first define some notations. For a literal $x$ in a reduced formula $\mathcal{F}$, we define:

– $n_i(x)$: the number of $i$-variables whose literals appear in $N(x, \mathcal{F})$;

– $n'_i(x)$: the number of $i$-variables whose literals appear in $N^{(2)}(x, \mathcal{F})$;
– $n''_i(x)$: the number of $i$-variables whose literals appear in $N^{(3+)}(x, \mathcal{F})$.

Note that by the definition, we always have that $n_i(x) = n'_i(x) + n''_i(x)$.

Next, we give some lower bounds on $\xi_S^{(1)}$, $\xi_S^{(2)}$, and $\Delta_{S_1} + \Delta_{S_2}$, which will be used to prove our main results.

**Lemma 6.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula. Let $S = \{x\}$, where $x$ is a literal in $\mathcal{F}$. It holds that*

$$\xi_S^{(1)} \geq w_{deg(x)} + \sum_{i \geq 3} n_i(x)\delta_i. \tag{7}$$

**Lemma 7.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula of degree $d$. Let $S = \{x\}$, where $x$ is a $(j, d-j)$-literal in $\mathcal{F}$. It holds that*

$$\xi_S^{(1)} \geq w_d + j\delta_d. \tag{8}$$

**Lemma 8.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula of degree $d$. Let $S = \{x\}$, where $x$ is a literal in $\mathcal{F}$. It holds that*

$$\xi_S^{(2)} \geq n'_3(\overline{x})w_3 + \sum_{4 \leq i \leq d} n'_i(\overline{x})w_{i-1}. \tag{9}$$

**Lemma 9.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula of degree $d$. Let $S_1 = \{x\}$ and $S_2 = \{\overline{x}\}$, where the corresponding variable of $x$ is a $d$-variable in $\mathcal{F}$. It holds that*

$$\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2d\delta_d + (n'_3(x) + n'_3(\overline{x}))(2w_3 - 2\delta_d) + \sum_{4 \leq i \leq d} (n'_i(x) + n'_i(\overline{x}))(w_i - 2\delta_d). \tag{10}$$

**Lemma 10.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula of degree $d$. Let $x$ be a $(1, d-1)$-literal and $xC$ be the only clause containing $x$ in $\mathcal{F}$. Let $S = \{x\} \cup \overline{C}$. It holds that*

$$\Delta_S \geq w_d + 2w_3 + \sum_{3 \leq i \leq d} n'_i(\overline{x})w_i. \tag{11}$$

**Lemma 11.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula of degree $d$. Let $x$ be a $(1, d-1)$-literal and $xC$ be the only clause containing $x$ in $\mathcal{F}$. Let $S_1 = \{x\} \cup \overline{C}$ and $S_2 = \{\overline{x}\}$. It holds that*

$$\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2w_3 + 2(d-1)\delta_d. \tag{12}$$

**Lemma 12.** *(*) Assume that $\mathcal{F}$ is a reduced CNF-formula of $d = 5$. Let $S_1 = \{x\}$ and $S_2 = \{\overline{x}\}$, where the corresponding variable of $x$ is a 5-variable in $\mathcal{F}$. If all clauses containing $x$ or $\overline{x}$ are $3^+$-clauses, it holds that*

$$\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + (\sum_{3 \leq i \leq 5} (n_i(x) + n_i(\overline{x})))\delta_5 + (\sum_{3 \leq i \leq 4} (n_i(x) + n_i(\overline{x})))(w_3 - \delta_5). \tag{13}$$

## 6 Step Analysis

Equipped with the above lower bounds, we are ready to analyze the branching vector of each step in the algorithm.

### 6.1 Step 2

In this step, we do not branch and only apply reduction rules to reduce the formula. However, it is still important to show that the measure will never increase when applying reduction rules, and reduction operations use only polynomial time.

**Lemma 13.** *(*) For any CNF-formula $\mathcal{F}$, it holds that*

$$\mu(R(\mathcal{F})) \leq \mu(\mathcal{F}).$$

**Lemma 14.** *(*) For any CNF-formula $\mathcal{F}$, we can apply the reduction rules in polynomial time to transfer it to $R(\mathcal{F})$.*

### 6.2 Step 3

In this step, we branch on a variable $x$ of degree at least 6. The two sub-branches are: $S_1 = \{x\}$; $S_2 = \{\overline{x}\}$. We have the following result:

**Lemma 15.** *The branching vector generated by Step 3 is covered*

$$[w_6 + \delta_6, w_6 + 11\delta_6] \text{ or } [w_6 + 11\delta_6, w_6 + \delta_6]. \tag{14}$$

*Proof.* Since R-Rule 4 is not applicable, both $x$ and $\overline{x}$ are $(1^+, 1^+)$-literals. By the condition of this case, we have $d \geq 6$ and $\delta_d = \delta_6$ by (2).

By Lemma 7, we can get that $\Delta_{S_1} \geq \xi_{S_1}^{(1)} \geq w_d + j\delta_d \geq w_6 + \delta_6$ since $x$ is a $(j, d-j)$-literal with $j \geq 1$. Also, we can get $\Delta_{S_2} \geq w_6 + \delta_6$ by the same method.

By Lemma 9, we have that $\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2d\delta_d + (n_3'(x) + n_3'(\overline{x}))(2w_3 - 2\delta_d) + \sum_{4 \leq i \leq d} (n_i'(x) + n_i'(\overline{x}))(w_i - 2\delta_d) \geq 2w_6 + 12\delta_d$ since $w_3 > \delta_d$ and $w_i > 2\delta_d$ for $4 \leq i \leq d$.

With $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_6 + \delta_6$ and $\Delta_{S_1} + \Delta_{S_2} \geq 2w_6 + 12\delta_6$, we can know that the branching vector of this case is covered by $[w_6 + \delta_6, w_6 + 11\delta_6]$ or $[w_6 + 11\delta_6, w_6 + \delta_6]$. □

### 6.3 Step 4

In this step, the algorithm will consider a $(1, 4)$-literal $x$. Assume that $xC$ is the only clause containing $x$. The two sub-branches are: $S_1 = \{x\} \cup \overline{C}$; $S_2 = \{\overline{x}\}$. We have the following result:

**Lemma 16.** *The branching vector generated by step 4 is covered by*

$$[w_5 + 2w_3, w_5 + 8\delta_5] \text{ or } [w_5 + 8\delta_5, w_5 + 2w_3]. \tag{15}$$

*Proof.* By Lemma 10, we get that $\Delta_{S_1} \geq w_d + 2w_3 + \sum_{3\leq i\leq d} n'_i(\overline{x})w_i \geq w_5 + 2w_3$. By Lemma 7, we have that $\Delta_{S_2} \geq \xi^{(1)}_{S_2} \geq w_d + j\delta_d = w_5 + 4\delta_5$ since $\overline{x}$ is a $(4,1)$-literal.

By Lemma 11, we can get that $\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2w_3 + 2(d-1)\delta_d = 2w_5 + 2w_3 + 8\delta_5$.

Since $w_3 < 2$ and $w_5 = 5$ by (2), we have $2w_5 > 5w_3$, i.e., $2w_5 - 4w_3 > w_3$. Since $w_4 = 2w_3$ by (4), we have $2w_5 - 2w_4 > w_3 \Rightarrow 2\delta_5 > w_3$. So $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_5 + 2w_3$. Since $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + 2w_3 + 8\delta_5$, we know that the branching vector of this case is covered by $[w_5 + 2w_3, w_5 + 8\delta_5]$ or $[w_5 + 8\delta_5, w_5 + 2w_3]$. □

### 6.4 Step 5

In this step, we branch on a 5-variable $x$ such that either $x$ or $\overline{x}$ is in a 2-clause. The two sub-branches are: $S_1 = \{x\}$; $S_2 = \{\overline{x}\}$. We have the following result:

**Lemma 17.** *The branching vector generated by step 5 is covered by one of*

$$[w_5 + 2\delta_5, w_5 + 4w_3 + 4\delta_5],\ [w_5 + 4w_3 + 4\delta_5, w_5 + 2\delta_5],$$
$$[w_5 + 3\delta_5, w_5 + 2w_3 + 5\delta_5],\ \textit{and}\ [w_5 + 2w_3 + 5\delta_5, w_5 + 3\delta_5].$$

*Proof.* We will consider two subcases:

**Case 1.** There are at least two 2-clause containing literal $x$ or $\overline{x}$. Now it holds that $\sum_{3\leq i\leq d} n'_i(x) + n'_i(\overline{x}) \geq 2$. By Lemma 7, we get that $\Delta_{S_1} \geq \xi^{(1)}_{S_1} \geq w_d + j\delta_d \geq w_5 + 2\delta_5$ since $x$ is a $(2,3)$-literal or $(3,2)$-literal. In a similar way, we can get that $\Delta_{S_2} \geq w_5 + 2\delta_5$.

By Lemma 9, we have $\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2d\delta_d + (n'_3(x) + n'_3(\overline{x}))(2w_3 - 2\delta_d) + \sum_{4\leq i\leq d}(n'_i(x) + n'_i(\overline{x}))(w_i - 2\delta_d) \geq 2w_5 + 10\delta_5 + \sum_{3\leq i\leq 5}(n'_i(x) + n'_i(\overline{x}))(2w_3 - 2\delta_5) \geq 2w_5 + 10\delta_5 + 2(w_3 - 2\delta_5) \geq 2w_5 + 4w_3 + 6\delta_5$ since $w_4, w_5 \geq 2w_3$ and $\sum_{3\leq i\leq 5} n'_i(x) + n'_i(\overline{x}) \geq 2$.

By $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_5 + 2\delta_5$ and $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + 4w_3 + 6\delta_5$, we know that the branching vector of this case is covered by $[w_5 + 2\delta_5, w_5 + 4w_3 + 4\delta_5]$ or $[w_5 + 4w_3 + 4\delta_5, w_5 + 2\delta_5]$.

**Case 2.** There is only one 2-clause containing literal $x$ or $\overline{x}$. Note that $\sum_{3\leq i\leq d} n'_i(x) + n'_i(\overline{x}) = 1$. For literal $x$, it is contained in at least two clauses and at most one of them is 2-clause. So $\sum_{3\leq i\leq d} n_i(x) \geq 3$ holds.

By Lemma 6, we get that $\Delta_{S_1} \geq \xi^{(1)}_{S_1} \geq w_5 + \sum_{3\leq i\leq d} n_i(x)\delta_i \geq w_5 + (\sum_{3\leq i\leq d} n_i(x))\delta_5 \geq w_5 + 3\delta_5$. Similarly, we also can get that $\Delta_{S_2} \geq w_5 + 3\delta_5$. By Lemma 9, we get that $\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2d\delta_d + (n'_3(x) + n'_3(\overline{x}))(2w_3 - 2\delta_d) + \sum_{4\leq i\leq d}(n'_i(x) + n'_i(\overline{x}))(w_i - 2\delta_d) \geq 2w_5 + 10\delta_5 + \sum_{3\leq i\leq 5}(n'_i(x) + n'_i(\overline{x}))(2w_3 - 2\delta_5) \geq 2w_5 + 10\delta_5 + (2w_3 - 2\delta_5) \geq 2w_5 + 2w_3 + 8\delta_5$ since $w_4, w_5 \geq 2w_3$ and $\sum_{3\leq i\leq d} n'_i(x) + n'_i(\overline{x}) = 1$.

Since $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_5 + 3\delta_5$ and $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + 2w_3 + 8\delta_5$, we know that the branching vector of this subcase is covered by $[w_5 + 3\delta_5, w_5 + 2w_3 + 5\delta_5]$ or $[w_5 + 2w_3 + 5\delta_5, w_5 + 3\delta_5]$.

These two cases complete the proof. □

### 6.5 Step 6

In this step, all clauses containing a 5-variable are $3^+$-clauses now. We branch on a 5-variable $x$ contained in a $4^+$-clause. The two sub-branches are: $S_1 = \{x\}$; $S_2 = \{\overline{x}\}$. We have the following result:

**Lemma 18.** *The branching vector generated by step 6 is covered by*

$$[w_5 + 4\delta_5, w_5 + 7\delta_5] \text{ or } [w_5 + 7\delta_5, w_5 + 4\delta_5]. \tag{16}$$

*Proof.* Literal $x$ is contained in at least two $3^+$-clauses. So $\sum_{3\leq i\leq d} n_i(x) \geq 4$ holds. By Lemma 6, we get that $\Delta_{S_1} \geq \xi^{(1)}_{S_1} \geq w_5 + (\sum_{3\leq i\leq d} n_i(x))\delta_d \geq w_5 + 4\delta_5$. Similarly, we get that $\Delta_{S_2} \geq w_5 + 4\delta_5$.

Let $m_4$ be the number of $4^+$-clauses containing $x$. We have that $\sum_{3\leq i\leq 5} n_i(x) + n'_i(x) \geq 2(5 - m_4) + 3m_4 \geq 10 + m_4 \geq 11$ since $m_4 \geq 1$. By Lemma 12, We get that $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + (\sum_{3\leq i\leq 5}(n_i(x) + n_i(\overline{x})))\delta_5 + (\sum_{3\leq i\leq 4}(n_i(x) + n_i(\overline{x})))(w_3 - \delta_5) \geq 2w_5 + 11\delta_5$ since $w_3 \geq \delta_5$.

Since $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_5 + 4\delta_5$ and $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + 11\delta_5$, we know that the branching vector of this case is covered by $[w_5 + 4\delta_5, w_5 + 7\delta_5]$ or $[w_5 + 7\delta_5, w_5 + 4\delta_5]$. □

### 6.6 Step 7

In Step 7, all clauses containing a 5-variable are 3-clauses. We branch on a 5-variable $x$ whose literal and a literal of a $4^-$-variable are in the same clause. The two sub-branches are: $S_1 = \{x\}$; $S_2 = \{\overline{x}\}$. We have that

**Lemma 19.** *The branching vector generated by step 7 is covered by*

$$[w_5 + 4\delta_5, w_5 + w_3 + 5\delta_5] \text{ or } [w_5 + w_3 + 5\delta_5, w_5 + 4\delta_5]. \tag{17}$$

*Proof.* There is at least one $4^-$-variable whose literal is in $N(x, \mathcal{F}) \cup N(\overline{x}, \mathcal{F})$. So it holds that $\sum_{3\leq i\leq 4}(n_i(x) + n_i(\overline{x})) \geq 1$.

For literal $x$, it is contained in at least two 3-clauses, which means that $\sum_{3\leq i\leq d} n_i(x) \geq 4$ holds. By Lemma 6, we get that $\Delta_{S_1} \geq \xi^{(1)}_{S_1} \geq w_5 + \sum_{3\leq i\leq 5} n_i(x)\delta_i \geq w_5 + (\sum_{3\leq i\leq d} n_i(x))\delta_5 = w_5 + 4\delta_5$. Similarly, we can get that $\Delta_{S_2} \geq w_5 + 4\delta_5$.

By Lemma 12, we get that $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + (\sum_{3\leq i\leq 5}(n_i(x) + n_i(\overline{x})))\delta_5 + (\sum_{3\leq i\leq 4}(n_i(x) + n_i(\overline{x})))(w_3 - \delta_5) \geq 2w_5 + w_3 + 9\delta_5$ since $\sum_{3\leq i\leq 4} n_i(x) + n_i(\overline{x}) \geq 1$.

Since $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_5 + 4\delta_5$ and $\Delta_{S_1} + \Delta_{S_2} \geq 2w_5 + 2w_3 + 8\delta_5$, we know that the branching vector is covered by $[w_5 + 4\delta_5, w_5 + w_3 + 5\delta_5]$ or $[w_5 + w_3 + 5\delta_5, w_5 + 4\delta_5]$. □

### 6.7 Step 8

In Step 8, the literals of all 5-variables form a 3-SAT instance $\mathcal{F}^*$. We apply the $O^*(1.3279^n)$-time algorithm in [13] for 3-SAT to solve our problem, where $n$ is the number of variables in the instance. Since $w_5 = 5$, we have that $n = \mu(\mathcal{F}^*)/w_5 = \mu(\mathcal{F}^*)/5$. So the running time for this part will be

$$O^*(1.3279^{\mu(\mathcal{F}^*)/w_5}) = O^*(1.0584^{\mu(\mathcal{F}^*)}).$$

### 6.8 Step 9

In this step, we branch on a $(1,3)$-literal $x$. The two sub-branches are: $S_1 = \{x\}$; $S_2 = \{\overline{x}\}$. We have the following result:

**Lemma 20.** *(*) The branching vector generated by step 9 is covered by*

$$[w_4 + 2w_3, w_4 + 6\delta_4] \text{ or } [w_4 + 6\delta_4, w_4 + 2w_3]. \tag{18}$$

### 6.9 Step 10

In this step, we branch on a $(2,2)$-literal $x$. The two sub-branches are: $S_1 = \{x\}$; $S_2 = \{\overline{x}\}$. We have the following result:

**Lemma 21.** *The branching vector generated by step 10 is covered by*

$$[w_4 + 2\delta_4, w_4 + 6\delta_4] \text{ or } [w_4 + 6\delta_4, w_4 + 2\delta_4]. \tag{19}$$

*Proof.* By Lemma 7, we get that $\Delta_{S_1} \geq \xi^{(1)}_{S_1} \geq w_d + j\delta_d = w_4 + 2\delta_4$ since $x$ is a $(2,2)$-literal. Similarity, we can get that $\Delta_{S_2} \geq w_4 + 2\delta_4$.

By Lemma 9, we have $\Delta_{S_1} + \Delta_{S_2} \geq 2w_d + 2d\delta_d + (n'_3(x) + n'_3(\overline{x}))(2w_3 - 2\delta_d) + \sum_{4 \leq i \leq d}(n'_i(x) + n'_i(\overline{x}))(w_i - 2\delta_d) = 2w_4 + 8\delta_4 + (n'_3(x) + n'_3(\overline{x}))(2w_3 - 2\delta_4) + (n'_4(x) + n'_4(\overline{x}))(w_4 - 2\delta_4) = 2w_4 + 8\delta_4$.

Since $\min(\Delta_{S_1}, \Delta_{S_2}) \geq w_4 + 2\delta_4$ and $\Delta_{S_1} + \Delta_{S_2} \geq 2w_4 + 8\delta_4$, we know that the branching vector is covered by $[w_4 + 2\delta_4, w_4 + 6\delta_4]$ or $[w_4 + 6\delta_4, w_4 + 2\delta_4]$. □

### 6.10 Step 11

All variables are 3-variables now. We apply the $O^*(1.1279^n)$-time algorithm by Wahlström [18] to solve this special case, where $n$ is the number of variables. For this case, we have that $n = \mu(\mathcal{F})/w_3$. So the running time of this part is

$$O^*((1.1279^{1/w_3})^{\mu(\mathcal{F})}).$$

## 7 The Final Result

Each one of the branching vectors above will generate a constraint in our quasiconvex program to solve the best value for $w_3$ and $w_4$. Let $\alpha_i$ denote the branching factor for branching vector $(i)$ where $14 \leq i \leq 21$. We want to find the minimum value $\alpha$ such that $\alpha \leq \alpha_i$ and $\alpha \leq 1.1279^{1/w_3}$ (generated by Step 11) under the assumptions (2) and (4). By solving this quasiconvex program, we get that $\alpha = 1.0646$ by letting $w_3 = 1.9234132344759123$ and $w_4 = 3.8468264689518246$. Note that $\alpha = 1.0646$ is greater than 1.0584 the branching factor generated in Step 8. So 1.0646 is the worst branching factor in the whole algorithm. By (5), we get the following result.

**Theorem 1.** *Algorithm 1 solves the SAT problem in* $O^*(1.0646^L)$ *time.*

**Table 2.** The weight setting

| | |
|---|---|
| $w_1 = w_2 = 0$ | |
| $w_3 = 1.9234132344759123$ | $\delta_3 = 1.9234132344759123$ |
| $w_4 = 3.8468264689518246$ | $\delta_4 = 1.9234132344759123$ |
| $w_5 = 5$ | $\delta_5 = 1.1531735310481754$ |
| $w_i = i (i \geq 6)$ | $\delta_i = 1 (i \geq 6)$ |

**Table 3.** The branching vector and factor for each step

| Steps | Branching vectors | Branching factors |
|---|---|---|
| Step 3 | $[w_6 + \delta_6, w_6 + 11\delta_6]$ | 1.0636 |
| Step 4 | $[w_5 + 2w_3, w_5 + 8\delta_5]$ | 1.0632 |
| Step 5 | $[w_5 + 3\delta_5, w_5 + 2w_3 + 5\delta_5]$ | 1.0618 |
| | $[w_5 + 2\delta_5, w_5 + 4w_3 + 4\delta_5]$ | 1.0636 |
| Step 6 | $[w_5 + 4\delta_5, w_5 + 7\delta_5]$ | 1.0636 |
| Step 7 | $[w_5 + 4\delta_5, w_5 + 5\delta_5 + w_3]$ | 1.0646 |
| Step 8 | $O^*((1.3279^{1/w_5})^\mu)$ | 1.0584 |
| Step 9 | $[w_4 + 2w_3, w_4 + 6\delta_4]$ | 1.0646 |
| Step 10 | $[w_4 + 2\delta_4, w_4 + 6\delta_4]$ | 1.0646 |
| Step 11 | $O^*((1.1279^{1/w_3})^\mu)$ | 1.0646 |

We also show the whole weight setting in Table 2 and the branching vector of each step under the setting in Table 3. From Table 3, we can see that we have four bottlenecks: Steps 7, 9, 10, and 11. In fact, Steps 9, 10, and 11 have the same branching vector $[4w_3, 8w_3]$ under the assumption that $w_4 = 2w_3$ (for Step 11, the worst branching vector in [18] is $[4, 8]$). The branching factor for these

three steps will decrease if the value of $w_3$ increases. On the other hand, the branching factor for Step 7 will decrease if the value of $w_3$ decreases. We set the best value of $w_3$ to balance them. If we can either improve Step 7 or improve Steps 9, 10, and 11 together, then we may get a further improvement. However, the improvement is very limited and several other bottlenecks will appear.

## 8 Concluding Remarks

In this paper, we show that the SAT problem can be solved in $O^*(1.0646^L)$ time, improving the previous bound in terms of the input length obtained more than 10 years ago. Nowadays, improvement becomes harder and harder. However, SAT is one of the most important problems in exact and parameterized algorithms, and the state-of-the-art algorithms are frequently mentioned in the literature. Furthermore, in order to give a neat and clear analysis, we introduce a general analysis framework, which can even be used to simplify the analysis for other similar algorithms based on the measure-and-conquer method.

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

## References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
2. Chen, J., Liu, Y.: An improved SAT algorithm in terms of formula length. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 144–155. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03367-4_13
3. Chu, H., Xiao, M., Zhang, Z.: An improved upper bound for SAT. Proc. AAAI Conf. Artif. Intell. **35**(5), 3707–3714 (2021). https://ojs.aaai.org/index.php/AAAI/article/view/16487
4. Cook, S.A.: The complexity of theorem-proving procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, USA, 3–5 May 1971, pp. 151–158. ACM (1971). https://doi.org/10.1145/800157.805047
5. Cook, S.A., Mitchell, D.G.: Finding hard instances of the satisfiability problem: a survey. In: Du, D., Gu, J., Pardalos, P.M. (eds.) Satisfiability Problem: Theory and Applications, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, 11–13 March 1996. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, pp. 1–17. DIMACS/AMS (1996). https://doi.org/10.1090/dimacs/035/01
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960). https://doi.org/10.1145/321033.321034
7. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. J. ACM **56**(5), 25:1–25:32 (2009). https://doi.org/10.1145/1552285.1552286

8. Fomin, F.V., Kratsch, D.: Exact Exponential Algorithms. TTCSAES. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16533-7
9. Hirsch, E.A.: Two new upper bounds for SAT. In: Karloff, H.J. (ed.) Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California, USA, 25–27 January 1998, pp. 521–530. ACM/SIAM (1998). http://dl.acm.org/citation.cfm?id=314613.314838
10. Hirsch, E.A.: New worst-case upper bounds for SAT. J. Autom. Reason. **24**(4), 397–420 (2000). https://doi.org/10.1023/A:1006340920104
11. Impagliazzo, R., Paturi, R.: On the complexity of k-sat. J. Comput. Syst. Sci. **62**(2), 367–375 (2001). https://doi.org/10.1006/jcss.2000.1727
12. Kullmann, O., Luckhardt, H.: Deciding propositional tautologies: Algorithms and their complexity. preprint 82 (1997)
13. Liu, S.: Chain, generalization of covering code, and deterministic algorithm for k-sat. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, Prague, Czech Republic, 9–13 July 2018. LIPIcs, vol. 107, pp. 88:1–88:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ICALP.2018.88
14. Monien, B., Speckenmeyer, E., Vornberger, O.: Upper bounds for covering problems. Methods Oper. Res. **43**, 419–431 (1981)
15. Peng, J., Xiao, M.: A fast algorithm for SAT in terms of formula length (2021). https://arxiv.org/abs/2105.06131
16. Van Gelder, A.: A satisfiability tester for non-clausal propositional calculus. Inf. Comput. **79**(1), 1–21 (1988). https://doi.org/10.1016/0890-5401(88)90014-4
17. Wahlström, M.: An algorithm for the SAT problem for formulae of linear length. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 107–118. Springer, Heidelberg (2005). https://doi.org/10.1007/11561071_12
18. Wahlström, M.: Faster exact solving of SAT formulae with a low number of occurrences per variable. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 309–323. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_23
19. Yamamoto, M.: An improved $\tilde{\mathcal{O}}(1.234^m)$-time deterministic algorithm for SAT. In: Deng, X., Du, D.-Z. (eds.) ISAAC 2005. LNCS, vol. 3827, pp. 644–653. Springer, Heidelberg (2005). https://doi.org/10.1007/11602613_65