New upper bound for the #3-SAT problem

Konstantin Kutzkov

Department of Computer Science, University of Munich
Oettingenstr. 67
80538 München, Germany
kutzkov@gmail.com

Abstract

We present a new deterministic algorithm for the #3-SAT problem, based on the DPLL strategy. It uses a new approach for counting models of instances with low density. This allows us to assume the adding of more 2-clauses than in previous algorithms. The algorithm achieves a running time of $O(1.6423^n)$ in the worst case which improves the current best bound of $O(1.6737^n)$ by Dahllöf et al.

1 Introduction

The canonical NP-complete problem SAT has received a lot of attention in the last decades. As a natural extension of the problem one is interested not in finding *a* satisfying assignment for a given SAT instance but in counting *the number* of all possible solutions.
The counting complexity class #P has been presented by Valiant in [9]. He also proved that both #2-SAT and #3-SAT are #P-complete [10].
It is obvious that the standard DPLL procedure solves the problem in $O(2^n)$ steps. Algorithms with better bounds have been presented ([1, 2, 5, 8, 11]). Dahllöf et al. [2] achieved the best known bounds for #2-SAT ($O(1.2561^n)$) and #3-SAT ($O(1.6737^n)$). Slight refinement of their analysis performed in [6] yields the new record running time for #2-SAT of $O(1.246^n)$.
Our algorithm is a variant of the DPLL procedure [3, 4] and improves the bound for #3-SAT to $O(1.6423^n)$. It profits from the very good result for #2-SAT from [2, 6] and a new lemma which allows us to assume that the number of models of 3-SAT instances over n variables with density less than or equal to 5/3 can be determined in $O(1.6409^n)$.

2 Preliminaries

As usual, n denotes the number of variables, m the number of clauses and $d := m/n$ the density of the formula. Variables are written as x_i and literals as x_i (for positive ones) and $\neg x_i$ (for negative ones). V denotes the set of variables and $L := V \cup \overline{V}$ the set of the literals.
An unsatisfied clause with exactly one unassigned variable is called a unit clause,

and an unsatisfied clause with two (resp. three) unassigned variables is called 2-clause (resp. 3-clause.) A 3-clause c_3 is subsumed by the 2-clause c_2 if all literals in c_2 occur in c_3.
An assignment α is called to be total if it assigns a boolean value to each $v \in V$. A partial assignment assigns a value to a given subset of V.
F_3 stands for the subformula of F that contains exactly the 3-clauses from F and F_2 for the subformula where every unsatisfied clause contains at most two unassigned literals. F_2 and F_3 both depend on the current partial assignment and are not fixed during the execution of the algorithm. The density of F_3 is defined as the number of 3-clauses divided by the number of unassigned variables in the formula (this definition implies that it is possible that F_3 has density less than 1/3).
The complexity of a given SAT instance is written as $\mu(F) = n - \lambda(F_2)$ where $\lambda(F_2)$ denotes the weight of F_2. (The exact way for the weighting of F_2 will be explained later.)
The degree of each variable x, i.e. $d(x)$, gives the number of occurrences of the variable x in the formula. $d_2(x)$ and $d_3(x)$ denote the number of occurrences of the variable x in F_2 and F_3, respectively.
A set of clauses is called *variable-disjoint* if no two clauses in the set share a variable.
A satisfying assignment (or model) α is an assignment making each clause in the formula *true*. A formula containing the empty clause has no model, and an empty formula has one model.

3 The algorithm

3.1 The algorithm for 3-SAT instances with low density

Let's consider the following problem: Given a 3-SAT instance F in which no variable occurs more than five times. What is the number of models for F?
The following **Algorithm 1** solves the problem:

1. Initialize $F_3 := F$, $F_2 := \emptyset$
2. Find a variable with the most occurrences in F_3, recursively branch on it and update F_3 and F_2.
3. Repeat step 2) until F_3 becomes empty.
4. Solve the #2-SAT problem in the current branch and update the number of models for F.

Lemma 1. **Algorithm 1** runs in $O(1.6796^n)$.

Proof. Let $a_{F_3}^{(i)}$ be the number of the 3-clauses in the i-th step in F_3. We initialize $a_{F_3}^{(0)} := m$. The following recursion gives an obvious lower bound for the number of 3-clauses which can be deleted from F_3 in the i-th step (*deleted* means that these either become satisfied or 2-clauses):

$$a_{F_3}^{(i+1)} = a_{F_3}^{(i)} - \lceil 3a_{F_3}^{(i)}/(n-i) \rceil = \lfloor a_{F_3}^{(i)} - 3a_{F_3}^{(i)}/(n-i) \rfloor$$

We are interested in the value for i when $a_{F_3}^{(i)} = 0$ becomes true.
Let's first generalize the recursion for the case when each of the n variables

occurs at most k times, implying that we have at most $n \cdot k/3$ clauses in F_3. Let x denote the number of the variables we need to assign until a state with the density $(k-1)/3$ is reached . By solving the linear equation

$$(k/3)n - kx = ((k-1)/3)(n-x)$$

we compute $x = 1/(2k+1)n$. Therefore we have $n(1 - 1/(2k+1))$ unassigned variables. By starting with $k = 5$ we compute $n(1-10/11 \cdot 8/9 \cdot \ldots \cdot 2/3) < 0.631n$. Therefore we easily compute an upper bound for the running time of the algorithm to $2^{0.631n} \cdot 1.246^{0.369n} = O(1.6796^n)$. □

Let's now cast a more detailed look on *Algorithm 1.* We concentrate on the case when F_3 has density $\leq 1/3$.
Before starting with the analysis we consider the following simplification:

Lemma 2. Let $(a \vee b)$ be a 2-clause in F_2. If there exists a 3-clause of the kind $(a \vee \neg b \vee c)$ we replace it with the 2-clause $(a \vee c)$.

Proof. If we branch as $\neg a, \neg c$ and $(a \vee c)$ we observe that we can cut the first branch. Thus, we need to consider only the branch where we add the 2-clause $(a \vee c)$. □

3-clauses of the above kind are called quasisubsumed 3-clauses.

Before a branching is performed we first check for subsumed or quasisubsumed 3-clauses. (It is clear that after this simplification the number of satisfying assignments remains unchanged.)
We distinguish now the following cases:

1. There is a variable a with $d_3(a) > 1$. We branch on the found variable.

2. The variables a, b, c don't occur in F_2. We delete the clause $(a \vee b \vee c)$ and after termination of the recursion we multiply the computed value for the number of models in this branch by 7.

3. Now we assume that at least one of the considered variables occur in F_2, w.l.o.g. a and at least one of the variables b, c not occur in F_2, w.l.o.g. c. Then we branch on a and since it occurs in F_2 we conclude that in the one branch two variables are assigned. In one of the two branches c does not occur any more. We assume the worst case that in one branch three variables are assigned and in the other only one.

4. Now we assume that all three variables a, b, c are present in F_2. Let's consider the 2-clauses $([\neg]a \vee l)$. ($[\neg]a$ means that we generalize the cases a and $\neg a$.)

   (a) Let's first assume that the variable l is different from the variables b and c. We consider first the situation $(a \vee l)$ and branch in the following way: $(a \vee b)$ and $\neg a, \neg b$. The setting $\neg a, \neg b$ obviously implies the literals c and l.
   If we have the 2-clause $(\neg a \vee l)$ then we branch as follows: $(b \vee c)$ and $\neg b, \neg c$. The literals $\neg b, \neg c$ imply a and this implies the literal l. So

in the two cases the number of 3-clauses in each branch decreases by one. In one branch the number of the unassigned variables remains unchanged and in the other we assign four variables. (Unfortunately, we can't conclude that also one further 3-clause is eliminated since it is possible that the implied variable l occurs only in F_2.)

(b) Let's now assume that no variable different from a, b and c occurs in the 2-clauses in which a, b and c occur. Lemma 2 guarantees that the existence of 2-clauses of the kind $(a \vee \neg b)$ simplifies $(a \vee b \vee c)$ to $(a \vee c)$ without branching. Since subsumed 3-clauses are also eliminated it is a simple observation that the single possibility that the considered case remains is when w.l.o.g. the 2-clauses $(\neg a \vee \neg b), (\neg a \vee \neg c)$ are present in F_2. Then we branch on a and in the two branches we delete one 3-clause and in one branch we assign one variable, $\neg a$, and in the other one three $(a, \neg b, \neg c)$.

Now we update *Algorithm 1* to **Algorithm 2** as follows:

1. Initialize $F_3 := F$, $F_2 := \emptyset$
2. Find a variable with the most occurrences in F_3, recursively branch on it and update F_3 and F_2.
3. Repeat step 2) until F_3 has density no more than 1/3.
4. Check which of the above described cases remains and split the formula according to it until F_3 becomes empty.
5. Solve the #2-SAT problem in the current branch and update the number of models for F.

Lemma 3. The number of solutions of a 3-SAT instance over n variables with density less than or equal to 5/3 can be examined in $O(1.6409^n)$.

Proof. Let's first denote the number of variables we need to assign until the density of F_3 becomes 1/3 with $n_{1/3}$. Let's now assume that we have already branched on the required $n_{1/3}$ variables. In order to analyze the running time for the remaining formula we introduce a new complexity measure:
Let $n_r := n - n_{1/3}$ denote the number of unassigned variables in the formula. Then the complexity of the given instance is $1.246^{n_r + w|F_3|}$ where $|F_3|$ is the cardinality of F_3 and $w \in \mathbb{R}$ is an unknown parameter, i.e. the weight of F_3. The measure $n_r + w|F_3|$ is correct since the emptiness of F_3 implies the running time for the #2-SAT problem and $n_r + w|F_3| = 0 \Rightarrow n_r = 0$ (since $|F_3|$ depends on n_r).
If we assign d variables and delete f 3-clauses we write the decreased complexity as $n_r - d + w(|F_3| - f)$.
Since $1.246^{n_r + w|F_3|}$ denotes the number of the recursion branches, we are interested in the value of w so that each inequality of the kind $1.246^{n_r + w|F_3|} \geq \sum_{i=1}^{k} 1.246^{n_r - d_i + w(|F_3| - f_i)}$ holds, where $(d_i, f_i)_{1 \leq i \leq k}$ are the values for the complexity decrease in the two components in the i-branch. In the following we use the notation $\gamma(d_1 + w \cdot f_1, .., d_k + w \cdot f_k)$ for inequalities of the above kind. Now we distinguish the different recursion cases and define the corresponding inequality:

1. $\gamma(1 + 2w, 1 + 2w)$

2. $\gamma(3+w)$

3. $\gamma(1+w, 3+w)$

4. a) $\gamma(w, 4+w)$

4. b) $\gamma(1+w, 3+w)$

In the same way as in Lemma 1 we compute $n_{1/3} \leq 0.446n$ and following $n_r \geq 0.554n$. It is also clear that $|F_3| \leq \lfloor n_r/3 \rfloor$. For $w = 1.58$ we observe that each of the recursion inequalities holds. Therefore, we compute the running time to $2^{n_{1/3}} \cdot 1.246^{n_r + 1.58 n_r/3} = O(1.6409^n)$. □

3.2 The complexity measure

Let's now explain how we compute the complexity of a given boolean formula F. As already mentioned the complexity is written as $\mu(F) = n - \lambda(F_2)$. The weight of the 2-clauses, $\lambda(F_2)$, is built in the following way: The set of all 2-clauses is divided in three subsets: $\mathcal{S}$ (for single), $\mathcal{T}$ (for two) and $\mathcal{R}$ (for rest). $\mathcal{S}$ is built from arbitrary chosen 2-clauses in F_2, so that no variable in the $\mathcal{S}$-clauses occurs more than once in $\mathcal{S}$, $\mathcal{T}$ may contain any two (but only two!) 2-clauses and all 2-clauses which don't fit in $\mathcal{S}$ or $\mathcal{T}$ are in $\mathcal{R}$. The $\mathcal{S}$-clauses and $\mathcal{T}$-clauses are weighted with a positive real constant, denoted as ϵ. The $\mathcal{R}$-clauses are not weighted. It is clear that for $\epsilon < 1$ for the weight of F_2 follows $\lambda(F_2) \leq (n/2 + 2)\epsilon < n/2 + 2$. Therefore the complexity $\mu(F) = 0$ implies that we have no more than constant number of unassigned variables in the formula. (This is a similar model to the one of Zhang [11] for the 3-SAT problem.)

3.3 The #3−SAT algorithm

In this section we show how the algorithm for instances with low density leads to an improvement of the running time for the general #3-SAT problem.
We always choose a variable with a maximal degree in $\mathcal{S}$ and $\mathcal{T}$ and branch on it. We consider all possible combinations which can occur.

Algorithm 3

- Case 0: F_2 is empty. Find a variable x with the maximum degree $d(x)$ in F_3. If $d(x) < 6$ apply *Algorithm 2*. Otherwise branch on x. For each branch:
  Branch on a chosen variable in the added 2-clauses until all the remaining 2-clauses can be put into $\mathcal{S} \cup \mathcal{T}$. (The exact way for choosing the variable is explained in the analysis.)

- Case 1: All 2-clauses in F_2 are in the subset $\mathcal{S}$. ($\mathcal{T}$ is still empty.) Find the variable in F_2 of maximum degree and recursively branch on it. If in one branch we have a variable of degree 1, e.g. if a occurs only in the clause $(a \vee l_1 \vee l_2)$, branch on l_1.

- Case 2: The subset $\mathcal{T}$ is not empty and no variable occurs more than twice in F_2. Find a variable x with $d_2(x) = 2$ which will cause the deletion of minimal number of 2-clauses and recursively branch on it. If in one branch four weighted 2-clauses are deleted, then in the other we have an appropriate variable and branch on it. (The existence of such a variable is proven in the analysis section.)

- Case 3: There is a variable occurring altogether three times in $\mathcal{S}$ and $\mathcal{T}$. Branch recursively on this variable.

The analysis

Let's first explain how we compute the complexity of a splitting algorithm [7]. Let $(d_1, .., d_k)$ be a given splitting where d_i denotes the complexity decrease in the i-th branch for $1 \leq i \leq k$. Then *the branching number* $\tau(d_1, .., d_k)$ is defined as the unique positive root of the equation $x^n = \sum_{i=1}^{k} x^{n-d_i}$. The running time of a splitting algorithm does not exceed $\tau_{\max}^{\mu(F)}$ where $\tau_{\max}$ is the maximal splitting number of this algorithm.

Case 0: Since the number of the added 2-clauses is unknown, we consider only six of the added 2-clauses. We analyze only the worst case when all six 2-clauses are added in one branch. It is easy to verify that the more symmetric is the complexity decrease in the two branches the better is the running time [7]. The analysis for the other cases can be immediately concluded from this one.
If a variable occurs $3 \leq k \leq 6$ times in the added 2-clauses then we branch on it and the corresponding branching number is computed to $\tau(1, 2+(6-k)\epsilon, k+2)$. (It is clear that up to three 2-clauses always fit in the $\mathcal{S}$- and $\mathcal{T}$-slots since F_2 is empty.)
Assume that no variable occurs more than twice in the added six 2-clauses. First of all we observe that each four of the considered 2-clauses will fit in $\mathcal{S}$ and $\mathcal{T}$ (There always exist two variable-disjoint 2-clauses which can be put into $\mathcal{S}$.). If we have the 2-clauses $(l_1 \vee l_2), (\neg l_1 \vee l_3)$ we easily obtain $\tau(1, 3+3\epsilon, 3+3\epsilon)$.
Let's now consider the 2-clauses $(l_1 \vee l_2), (l_1 \vee l_3)$. If at least one of l_2, l_3 does not occur in another of the six 2-clauses or l_2 and l_3 share a 2-clause we branch on l_1. In the following we call such variables of degree two *suitable* variables. The corresponding branching number is easily computed to $\tau(1, 2+4\epsilon, 4+3\epsilon)$ by branching on l_1 in the right branch.
If there exists no suitable variable and not all six 2-clauses fit in $\mathcal{S}$ and $\mathcal{T}$, we conclude a "deadlock" situation, i.e. six variables forming six 2-clauses so that each variable occurs twice: e.g. $(l_1 \vee l_2), (l_3 \vee l_4), (l_1 \vee l_3), (l_2 \vee l_5), (l_4 \vee l_6), (l_5 \vee l_6)$. Then the following observation is trivial: We can branch on a variable of degree two and in the branch where we set one literal one of the variables becomes suitable. Therefore we compute the branching number $\tau(1, 2+4\epsilon+1-2\epsilon, 2+4\epsilon+3-3\epsilon, 4+2\epsilon) = \tau(1, 3+2\epsilon, 5+\epsilon, 4+2\epsilon)$ by branching on the suitable variable.
Otherwise all the six 2-clauses can be weighted with ϵ. Thus, we compute the branching number to $\tau(1, 1+6\epsilon)$.

Case 1: The goal in this case is to show that we have a worst case branching number $\tau(1-\epsilon, 2+\epsilon)$.
Let's consider the 2-clause $(a \vee b)$. If one of the two variables occurs only in this clause then we branch on the other variable and easily compute the branching number $\tau(2-\epsilon, 2-\epsilon)$.
If $d_3(a) = d_3(b) = 1$ then after branching on one of the variables, w.l.o.g.

a, we obtain for the worst case $\tau(1-\epsilon, 2)$. In the a-branch the variable b has degree 1, w.l.o.g. it occurs as b in the 3-clause $(b \vee l_1 \vee l_2)$. We branch on l_1 and in one branch we add one 2-clause and in the other the variable b does not occur any more. Thus the corresponding branching number is $\tau(1-\epsilon+1+\epsilon, 1-\epsilon+2, 2) = \tau(2, 3-\epsilon, 2)$. (We weight the added 2-clauses with ϵ since the $\mathcal{T}$-slots are still empty.)
Otherwise one of the variables has degree 3 and the branching number can not exceed $\tau(1-\epsilon, 2+\epsilon)$ (or $\tau(1,2)$ which obviously yields a better bound).

Case 2: The case when a literal and its negation both occur in F_2 has an obvious bound of $\tau(2-3\epsilon, 2-3\epsilon)$ since the setting of a variable deletes at most two 2-clauses and one of the 2-clauses is shared by the assigned variables.
Otherwise we have a literal which occurs twice. If one of the variables of degree two is suitable (see the analysis of case 0) we branch on it and compute the branching number $\tau(1-2\epsilon, 3-3\epsilon)$. Otherwise we consider the 2-clauses $(a \vee b), (a \vee c)$. Since the partial assignment $\neg a, b, c$ deletes four two 2-clauses and we have no suitable variable we conclude the existence of the 2-clauses $(b \vee d), (c \vee d)$. Then we branch first on a and in the a-branch we branch again on d. So we obtain $\tau(2-4\epsilon, 4-4\epsilon, 3-4\epsilon)$.

Case 3: It is trivial that the branching number can not exceed $\tau(1-3\epsilon, 4-5\epsilon)$ since we delete all $\mathcal{T}$-clauses.

3.4 The bound $O(1.6423^n)$

Theorem 1. The number of satisfying assignments of a given 3-SAT instance can be examined in $O(1.6423^n)$ steps.

Proof. Let's now list the corresponding branching numbers for each case:

1. Case 0:
   $\tau(1, 2+(6-k)\epsilon, k+2)$ for $3 \leq k \leq 6$, $\tau(1, 3+3\epsilon, 3+3\epsilon)$, $\tau(1, 2+4\epsilon, 4+3\epsilon)$, $\tau(1, 3+2\epsilon, 5+\epsilon, 4+2\epsilon)$, $\tau(1, 1+6\epsilon)$
2. Case 1:
   $\tau(2-\epsilon, 2-\epsilon)$, $\tau(2, 3-\epsilon, 2)$, $\tau(1-\epsilon, 2+\epsilon)$
3. Case 2:
   $\tau(2-3\epsilon, 2-3\epsilon)$, $\tau(1-2\epsilon, 3-3\epsilon)$, $\tau(2-4\epsilon, 4-4\epsilon, 3-4\epsilon)$
4. Case 3:
   $\tau(1-3\epsilon, 4-5\epsilon)$

For $\epsilon = 0.149$ each branching number is minor than the desired 1.6423. □

4 Conclusions

The main goal of the article is to present the algorithm for 3-SAT instances with low density. We have chosen a complexity model that is easy to analyze. A better running time can be achieved when more refined complexity measure is considered.

But the more intriguing question is a possible improvement of *Algorithm 2*. It is clear that an improved bound for #2-SAT will automatically improve the current bound. A more precise analysis may lead to a better running time, especially about the number of occurrences of a given variable in F_2 when it occurs only once in F_3. This would allow to assume the existence of a variable of degree more than six in F_3.

Acknowledgments

The author thanks to the unknown referees for many valuable comments and suggestions. The author also thanks to Anna Ganeva for help in writing this paper.

References

[1] V. Dahllöf, P. Jonsson, M. Wahlström. *Counting satisfying assignments in 2-SAT and 3-SAT*, Proc. 8th International Computing and Combinatorics Conference, 535-543, 2002

[2] V. Dahllöf, P. Jonsson, M. Wahlström. *Counting models for 2-SAT and 3-SAT formulae*, Theoretical Computer Science 332:265-291, 2005

[3] M. Davis, G. Logemann, and D. Loveland. *A machine program for theorem-proving.* Communications of ACM, 5:394-397, 1962

[4] Martin Davis and Hilary Putnam. *A computing procedure for quantification theory.* Journal of the ACM, 7:201-215, 1960

[5] O. Dubois. *Counting the number of solutions for instances of satisfiability*, Theoretical Computer Science 81:49-64, 1991

[6] Martin Fürer and Shiva Prasad Kasiviswanathan. *Algorithms for Counting 2-SAT Solutions and Colorings with Applications.* ECCC TR05-033, 2005

[7] Oliver Kullmann. *New methods for 3-SAT decision and worst-case analysis.* Theoretical Computer Science 223:1-72, 1999

[8] M.L.Littman, T. Pitassi, R. Impagliazzo. *On the complexity of counting satisfying assignments*, The working notes of the LICS 2001, Workshop on Satisfiability, 2001

[9] L. Valiant. *The complexity of computing the permanent*, Theoretical Computer Science 8:189-201, 1979

[10] L. Valiant. *The complexity of enumeration and reliability problems*, SIAM 8:410-421, 1979

[11] Wenhui Zhang. *Number of models and satisfiability of sets of clauses.* Theoretical Computer Science, 155:277-288, 1996