A Faster Exact Algorithm to Count X3SAT Solutions

Gordon Hoi[1], Sanjay Jain[1(✉)], and Frank Stephan[1,2]

[1] School of Computing, National University of Singapore, 13 Computing Drive, Block COM1, Singapore 117417, Republic of Singapore
e0013185@u.nus.edu, sanjay@comp.nus.edu.sg

[2] Department of Mathematics, National University of Singapore, 10 Lower Kent Ridge Road, Block S17, Singapore 119076, Republic of Singapore
fstephan@comp.nus.edu.sg

Abstract. The Exact Satisfiability problem, XSAT, is defined as the problem of finding a satisfying assignment to a formula in CNF such that there is exactly one literal in each clause assigned to be "1" and the other literals in the same clause are set to "0". If we restrict the length of each clause to be at most 3 literals, then it is known as the X3SAT problem. In this paper, we consider the problem of counting the number of satisfying assignments to the X3SAT problem, which is also known as #X3SAT.

The current state of the art exact algorithm to solve #X3SAT is given by Dahllöf, Jonsson and Beigel and runs in $O(1.1487^n)$ time, where n is the number of variables in the formula. In this paper, we propose an exact algorithm for the #X3SAT problem that runs in $O(1.1120^n)$ time with very few branching cases to consider, by using a result from Monien and Preis to give us a bisection width for graphs with at most degree 3.

Keywords: #X3SAT · Counting models · Exponential time algorithms

1 Introduction

Given a propositional formula φ in conjunctive normal form (CNF), a common question to ask would be if there is a satisfying assignment to φ. This is known as the satisfiability problem, or SAT. Many other variants of the satisfiability problem have also been explored. An important variant is the Exact Satisfiability problem, XSAT, where it asks if one can find a satisfying assignment such that exactly one of the literals in each clause is assigned the value "1" and all other

Sanjay Jain and Frank Stephan are supported in part by the Singapore Ministry of Education Tier 2 grant AcRF MOE2019-T2-2-121/R146-000-304-112. Further, Sanjay Jain is supported in part by NUS grant number C252-000-087-001. We thank the anonymous referees of CP2020 for several helpful comments. An extended technical report is available at https://arxiv.org/pdf/2007.07553.pdf.

H. Simonis (Ed.): CP 2020, LNCS 12333, pp. 375–391, 2020.
https://doi.org/10.1007/978-3-030-58475-7_22

literals in the same clause are assigned "0". Another variant that has been heavily studied is the restriction of the number of literals allowed in each clause. In both SAT and XSAT, one allows arbitrary number of literals to be present in each clause. If we restrict the number of literals to be at most k in each clause, then the above problems are now known as kSAT and XkSAT respectively. The most famous of these variants are 3SAT and X3SAT. All the mentioned problems, SAT, 3SAT, XSAT and X3SAT are known to be NP-complete [1–3,10,17].

Apart from decision problems and optimization problems, one can also work on counting the number of different models that solves the decision problem. For example, we can count the number of different satisfying assignments that solves SAT, and this is known as #SAT. The problem #3SAT, #XSAT and #X3SAT are defined similarly. Counting problems seem much harder than their decision counterparts. One may use the output of a counting algorithm to solve the decision problem. Another convincing example can be seen in that 2SAT is known to be in P [11] but #2SAT is #P-complete [18]. In fact, #SAT, #3SAT, #X3SAT and #XSAT are all known to be in #P-complete [18,19]. The problem of model counting has found wide applications in the field of AI such as the use of inference in Bayesian belief networks or probabilistic inference [15,16]. In this paper, we will focus on the #X3SAT problem.

Let n denote the number of variables in the formula. Algorithms to solve #XSAT have seen numerous improvements [4,5,14,20] over the years. To date, the fastest #XSAT algorithm runs in $O(1.1995^n)$ time [21]. Of course, to solve the #X3SAT problem, one can rely on any of the mentioned algorithm that solves #XSAT to solve them directly. However, it is possible to exploit the structure of X3SAT and hence solve #X3SAT in a much faster manner. Dahllöf, Jonsson and Beigel gave an #X3SAT algorithm in $O(1.1487^n)$ time [5].

In this paper, we propose a faster and simpler algorithm to solve the #X3SAT problem in $O(1.1120^n)$ time. The novelty here lies in the use of a result by Monien and Preis [13] to help us to deal with a specific case. Also using a different way to analyze our algorithm allows us to tighten the analysis further.

2 Preliminaries

In this section, we will introduce some common definition needed by the algorithm and also the techniques needed to understand the analysis of the algorithm. The main design of our algorithm is a Davis Putnam Logemann Loveland (DPLL) [6,7] style algorithm, or also known as the branch and bound algorithm. Such algorithms are recursive in nature and have two kinds of rules associated with them: Simplification and Branching rules. Simplification rules help us to simplify a problem instance. Branching rules on the other hand, help us to solve a problem instance by recursively solving smaller instances of the problem. To illustrate the execution of the DPLL algorithm, a search tree is commonly used. We assign the root node of the search tree as the original problem. The subsequent child nodes are assigned whenever we invoke a branching rule. For more information, one may refer to [8]. Let μ denote our parameter of complexity.

To analyse the running time of the DPLL algorithm, one in fact just needs to bound the number of leaves generated in the search tree. This is due to the fact that the complexity of such algorithm is proportional to the number of leaves, modulo polynomial factors, i.e., $O(poly(|\varphi|, \mu) \times$ number of leaves in the search tree$) = O^*($number of leaves in the search tree$)$, where the function $poly(|\varphi|, \mu)$ is some polynomial based on $|\varphi|$ and μ, while $O^*(g(\mu))$ is the class of all functions f bounded by some polynomial $p(\cdot)$ times $g(\mu)$.

Then we let $T(\mu)$ denote the maximum number of leaf nodes generated by the algorithm when we have μ as the parameter for the input problem. Since the search tree is only generated by applying a branching rule, it suffices to consider the number of leaf nodes generated by that rule (as simplification rules take only polynomial time). To do this, we employ techniques in [12]. Suppose a branching rule has $r \geq 2$ children, with $t_1, t_2, \ldots, t_r$ number of variables eliminated for these children. Then, any function $T(\mu)$ which satisfies $T(\mu) \geq T(\mu - t_1) + T(\mu - t_2) + \ldots T(\mu - t_r)$, with appropriate base cases, would satisfy the bounds for the branching rule. To solve the above linear recurrence, one can model this as $x^{-t_1} + x^{-t_2} + \ldots + x^{-t_r} = 1$. Let β be the root of this recurrence, where $\beta \geq 1$. Then any $T(\mu) \geq \beta^{\mu}$ would satisfy the recurrence for this branching rule. In addition, we denote the branching factor $\tau(t_1, t_2, \ldots, t_r)$ as β. Tuple $(t_1, t_2, \ldots, t_r)$ is also known as the branching vector [8]. If there are k branching rules in the DPLL algorithm, then the overall complexity of the algorithm can be seen as the largest branching factor among all k branching rules; i.e. $c = max\{\beta_1, \beta_2, \ldots, \beta_k\}$, and therefore the time complexity of the algorithm is bounded above by $O^*(c^{\mu})$.

We will introduce some known results about branching factors. If $k < k'$, then we have that $\tau(k', j) < \tau(k, j)$, for all positive k, j. In other words, comparing two branching factors, if one eliminates more variable, then this will result in a a smaller branching factor. Suppose that $i + j = 2\alpha$, for some α, then $\tau(\alpha, \alpha) \leq \tau(i, j)$. In other words, a more balanced tree will give a smaller branching factor.

Finally, suppose that we have a branching vector of (u, v) for some branching rule. Suppose that for the first branch, we immediately do a follow up branching to get a branching vector of (w, x), then we can apply branching vector addition to get a combined branching vector of $(u + w, u + x, v)$. This technique can sometimes help us to bring down the overall complexity of the algorithm further.

Finally, the correctness of DPLL algorithms usually follows from the fact that all cases have been covered. We now give a few definitions before moving onto the actual algorithm. We fix a formula φ:

Definition 1. Two clauses are called neighbours if they share at least a common variable. Two variables are called neighbours if they appear in some clause together. We say that a clause C is a degree k clause if C has k neighbours. Finally, a variable is a singleton if it appears only once in φ.

Suppose we have clauses $C_1 = (x \vee y \vee z)$, $C_2 = (x \vee a \vee b)$ and $C_3 = (y \vee a \vee c)$. Then C_1 is a neighbour to C_2 and C_3. In addition, all three are degree 2 clauses. Variables a, b, y, z are neighbours of x, while b, c, z are singletons.

Definition 2. We say that two variables, x and y, are linked when we can deduce either $x = y$ or $x = \bar{y}$. When this happens, we can proceed to remove one of the linked variable, either x or y, by replacing it with the other.

For example, in clause $(0 \vee x \vee y)$, we know that $x = \bar{y}$ to satisfy it. Thus, we can link x with $\bar{y}$ and remove one of the variables, say y.

Definition 3. We denote the formula $\varphi[x = 1]$ obtained from φ by assigning a value of 1 to the literal x. We denote the formula $\varphi[x = y]$ as obtained from φ by substituting all instances of x by y. Similarly, let δ be a subclause. We denote $\varphi[\delta = 0]$ as obtained from φ by substituting all literals in δ to 0.

Suppose we have $\varphi = (x \vee y \vee z)$. Then if we assign $x = 1$, then $\varphi[x = 1]$ gives us $(1 \vee y \vee z)$. On the other hand, if we have $\varphi[y = x]$, then we have $(x \vee x \vee z)$. If $\delta = (y \vee z)$, then $\varphi[\delta = 0]$ gives us $(x \vee 0 \vee 0)$.

Definition 4. A sequence of degree 2 clauses $C_1, C_2, \ldots, C_k$, $k \geq 1$ is called a chain if for $1 \leq j \leq k-1$, we have C_j is a neighbour to C_{j+1}. Given any two clauses C_e and C_f that are at least degree 3, we say that they are connected via a chain if we have a chain $C_1, C_2, \ldots, C_k$ such that C_1 is a neighbour of C_e (respectively C_f) and C_k is a neighbour of C_f (respectively C_e). Moreover, if we have a chain of degree 2 clauses $C_1, C_2, \ldots, C_k, C_1$, then we call this a cycle.

Suppose we have the following degree 3 clauses: $(a \vee b \vee c)$ and $(s \vee t \vee u)$, and the following chain: $(c \vee d \vee e)$, $(e \vee f \vee g)$, …, $(q \vee r \vee s)$. Then note that the degree 3 clause $(a \vee b \vee c)$ is a neighbour to $(c \vee d \vee e)$ and $(s \vee t \vee u)$ is a neighbour to $(q \vee r \vee s)$. Therefore, we say that $(a \vee b \vee c)$ and $(s \vee t \vee u)$ are connected via a chain.[1]

Definition 5. A path $x_1, x_2, \ldots, x_i$ is a sequence of variables such that for each $j \in \{1, \ldots, i-1\}$, the variables x_j and x_{j+1} are neighbours. A component is a maximal set of clauses such that any two variables, found in any clauses in the set has a path between each other. A formula is connected if any two variables have a path between each other. Else we say that the formula is disconnected, and consists of $k \geq 2$ components.

For example, let $\varphi = (x \vee y \vee z) \wedge (x \vee a \vee b) \wedge (e \vee c \vee d) \wedge (e \vee f \vee g)$. Then φ is disconnected and is made up of two components, since x has no path to e, while variables in the set $\{(x \vee y \vee z), (x \vee a \vee b)\}$ have a path to each other. Similarly, for $\{(e \vee c \vee d), (e \vee f \vee g)\}$. Therefore, $\{(x \vee y \vee z), (x \vee a \vee b)\}$ and $\{(e \vee c \vee d), (e \vee f \vee g)\}$ are two components.

Definition 6. Let I be a set of variables of a fixed size. We say that I is semi-isolated if there exists an $s \in I$ such that in any clause involving variables not in I, only s from I may appear.

[1] The definition of chains and cycles will be mainly used in Sect. 4.3 and Sect. 4.4.

For example consider the set $I = \{x, y, z, a, b\}$ and the clauses $(x \vee y \vee z)$, $(x \vee a \vee b)$, $(b \vee c \vee d)$, $(c \vee d \vee e)$. Since b is the only variable in I that appears in clauses involving variables not in I, I is semi-isolated.

Definition 7. Suppose $G = (V, E)$ is a simple undirected graph. A *balanced bisection* is a mapping $\pi : V \to \{0, 1\}$ such that, for $V_i = \{v : \pi(v) = i\}$, $|V_0|$ and $|V_1|$ differ by at most one. Let $cut(\pi) = |\{(v, w) : (v, w) \in E, v \in V_0, w \in V_1\}|$. The bisection width of G is the smallest $cut(\cdot)$ that can be obtained for a balanced bisection.

Theorem 8 (see Monien and Preis [13]). *For any $\varepsilon > 0$, there is a value $n(\varepsilon)$ such that the bisection width of any 3-regular graph $G = (V, E)$ with $|V| > n(\varepsilon)$ is at most $(\frac{1}{6} + \varepsilon)|V|$. This bisection can be found in polynomial time.*

The above result extends to all graphs G with maximum degree of 3 [9].

3 Algorithm

Our algorithm takes in a total of 4 parameters: a formula φ, a cardinality vector $\boldsymbol{c}$, two sets L and R.

The second parameter, a cardinality vector $\boldsymbol{c}$, maps literals to $\mathbb{N}$. The idea behind introducing this cardinality vector $\boldsymbol{c}$ is to help us to keep track of the number of models while applying simplification and branching rules. At the start, $\boldsymbol{c}(l) = 1$ for all literals in φ and will be updated along the way whenever we link variables together or when we remove singletons. Since linking of variables is a common operation, we introduce a function to help us perform this procedure. The function $Link(.)$, takes as inputs the cardinality vector and two literals involving different variables to link them[2]. It updates the information of the eliminated variable (y) onto the surviving variable (x) and after which, drops the entries of eliminated variable (y and $\bar{y}$) in the cardinality vector $\boldsymbol{c}$. When we link x and y as $x = y$ (respectively, $x = \bar{y}$), then we call the function $Link(\boldsymbol{c}, x, y)$ (respectively, $Link(\boldsymbol{c}, x, \bar{y})$). We also use a function $MonienPreis(.)$ to give us partition based on Theorem 8.

Function: $Link(.)$
Input: A Cardinality Vector $\boldsymbol{c}$, literal x, literal y
Output: An updated Cardinality Vector $\boldsymbol{c'}$

- Update $\boldsymbol{c}(x) = \boldsymbol{c}(x) \times \boldsymbol{c}(y)$, and $\boldsymbol{c}(\bar{x}) = \boldsymbol{c}(\bar{x}) \times \boldsymbol{c}(\bar{y})$. After which, drop entries of y and $\bar{y}$ from $\boldsymbol{c}$ and update it as $\boldsymbol{c'}$. Finally, return $\boldsymbol{c'}$

Function: $MonienPreis(.)$
Input: A graph G_φ with maximum degree 3
Output: L and R, the left and right partitions of minimum bisection width

For the third and fourth parameter, we have the sets of clauses L and R. L and R will be used to store partitions of clauses after calling $MonienPreis(.)$,

[2] As seen in Definition 2.

based on the minimum bisection width. Initially, L and R are empty sets and will continue to be until we first come to Line 17 of the algorithm.[3]

We call our algorithm $CountX3SAT(\cdot)$. Whenever a literal l is assigned a constant value, we drop both the entries l and $\bar{l}$ from the cardinality vector and multiply the returning recursive call by $\boldsymbol{c}(l)$ if $l = 1$, or $\boldsymbol{c}(\bar{l})$ if $\bar{l} = 1$. In each recursive call, we ensure that the cardinality vector is updated to contain only entries where variables in the remaining formula have yet to be assigned a constant value. By doing so, we guarantee the following invariant: For any given φ, let $S_\varphi = \{h : h \text{ is an exact-satisfiable assignment for } \varphi\}$. Now for any given φ and a cardinality vector $\boldsymbol{c}$, the output of $CountX3SAT(\varphi, \boldsymbol{c}, L, R)$ is given as $\sum_{h \in S_\varphi} \prod_{l:l \text{ is assigned true in } h} \boldsymbol{c}(l)$. Initial call to our algorithm would be $CountX3SAT(\varphi, \boldsymbol{c}, \emptyset, \emptyset)$, where the cardinality vector $\boldsymbol{c}$ has $\boldsymbol{c}(l) = 1$ for all literals at the start. The correctness of the algorithm follows from the fact that each step will maintain the invariant that $CountX3SAT(\varphi, \boldsymbol{c}, L, R)$ returns $\sum_{h \in S_\varphi} \prod_{l:l \text{ is assigned true in } h} \boldsymbol{c}(l)$, where if φ is not exactly satisfiable, it returns 0. Note that in the algorithm below possibilities considered are exhaustive.

Algorithm: CountX3SAT(.)
Input: A formula φ, a cardinality vector $\boldsymbol{c}$, a set L, a set R
Output: $\sum_{h \in S_\varphi} \prod_{l:l \text{ is assigned true in } h} \boldsymbol{c}(l)$

1: If any clause is not exact satisfiable (by analyzing this clause itself) then return 0. If all clauses consist of constants evaluating to 1 or no clause is left then return 1.
2: If there is a clause $(1 \vee \delta)$, then let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries of the variables in δ. Drop this clause from φ.
Return $CountX3SAT(\varphi[\delta = 0], \boldsymbol{c'}, L, R) \times \prod_{i \text{ is a literal in } \delta} \boldsymbol{c}(\bar{i})$
3: If there is a clause $C = (0 \vee \delta)$, then update $C = \delta$ in φ.
Return $CountX3SAT(\varphi, \boldsymbol{c}, L, R)$.
4: If there is a single literal x in a clause, then let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries x and $\bar{x}$ from $\boldsymbol{c}$.
Return $CountX3SAT(\varphi[x = 1], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(x)$.
5: If there is a 2-literal clause $(x \vee y)$, for some literals x and y with $x \neq y$ and $x \neq \bar{y}$, then $\boldsymbol{c'} = Link(\boldsymbol{c}, x, \bar{y})$. Return $CountX3SAT(\varphi[y = \bar{x}], \boldsymbol{c'}, L, R)$.
6: If there is a clause $(x \vee \bar{x})$, for some variable x. Check if x appears in other clauses. If yes, then drop this clause from φ and return $CountX3SAT$ $(\varphi, \boldsymbol{c}, L, R)$. If no, then let $\boldsymbol{c'}$ be the new cardinality vector by dropping x and $\bar{x}$. Drop this clause from φ and return $CountX3SAT(\varphi, \boldsymbol{c'}, L, R) \times (\boldsymbol{c}(x) + \boldsymbol{c}(\bar{x}))$.
7: If there are $k \geq 2$ components in φ and there are no edges between L and R, then let $\varphi_1, \ldots, \varphi_k$ be the k components of φ. Let $\boldsymbol{c}_i$ be the cardinality vector for φ_i by only keeping the entries of the literals involving variables appearing in φ_i, and dropping the rest. Let $L = R = \emptyset$. Return $CountX3SAT(\varphi_1, \boldsymbol{c}_1, L, R) \times \ldots \times CountX3SAT(\varphi_k, \boldsymbol{c}_k, L, R)$.
8: If there exists a clause $(x \vee x \vee y)$, for some literals x and y, then let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries x and $\bar{x}$ from $\boldsymbol{c}$.
Return $CountX3SAT(\varphi[x = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x})$

[3] More details about their role will be given in Sect. 4.3.

9: If there is a clause $(x \vee \bar{x} \vee y)$, then let $\boldsymbol{c'}$ be the new cardinality vector by removing the entries y and $\bar{y}$. Return $CountX3SAT(\varphi[y = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{y})$

10: If there exists a clause containing two singletons x and y, then update $\boldsymbol{c}$ as: $\boldsymbol{c}(x) = \boldsymbol{c}(x) \times \boldsymbol{c}(\bar{y}) + \boldsymbol{c}(\bar{x}) \times \boldsymbol{c}(y)$, $\boldsymbol{c}(\bar{x}) = \boldsymbol{c}(\bar{x}) \times \boldsymbol{c}(\bar{y})$.
Let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries y and $\bar{y}$ from $\boldsymbol{c}$. Drop y from φ. Return $CountX3SAT(\varphi, \boldsymbol{c'}, L, R)$.

11: There are two clauses $(x \vee y \vee z)$ and $(x \vee y \vee w)$, for some literals x, y, z and w. Then in this case, let $\boldsymbol{c'} = Link(\boldsymbol{c}, z, w)$. Drop one of the clauses. Return $CountX3SAT(\varphi[w = z], \boldsymbol{c'}, L, R)$.

12: There are two clauses $(x \vee y \vee z)$ and $(x \vee \bar{y} \vee w)$, for some literals x, y, z and w. Then let $\boldsymbol{c'}$ be the new cardinality vector by dropping entries of x and $\bar{x}$. Return $CountX3SAT(\varphi[x = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x})$.

13: There are two clauses $(x \vee y \vee z)$ and $(\bar{x} \vee \bar{y} \vee w)$, for some literals x, y, z and w. Then $\boldsymbol{c'} = Link(\boldsymbol{c}, x, \bar{y})$. Return $CountX3SAT(\varphi[y = \bar{x}], \boldsymbol{c'}, L, R)$.

14: If there exists a semi-isolated set I, with $3 \leq |I| \leq 20$, then let x be the variable appearing in further clauses with variables not in I. Let $\boldsymbol{c'}$ be the new cardinality vector by updating the entries of x and $\bar{x}$, dropping of entries of variables in $I - \{x\}$. Drop all the entries of $I - \{x\}$ from φ. Return $CountX3SAT(\varphi, \boldsymbol{c'}, L, R)$.[4]

15: This rule is not analyzed for all cases, but only specific cases as mentioned in Sections 4.1 and 4.2 (more specifically this applies only when some variable appears in at least 3 clauses). If there exists a variable x such that branching $x = 1$ and $x = 0$ allows us to either remove at least 7 variables on both branches, or at least 8 on one and 6 on the other, or at least 9 on one and 5 on the other, then branch x. Let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries x and $\bar{x}$. Return $CountX3SAT(\varphi[x = 1], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(x) + CountX3SAT(\varphi[x = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x})$[5].

16: If there exists a variable x appearing at least 3 times, then let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries x and $\bar{x}$. Return $CountX3SAT(\varphi[x = 1], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(x) + CountX3SAT(\varphi[x = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x})$[5].

17: If there is a degree 3 clause in φ, then check if $\exists$ an edge between L and R. If no, then construct G_φ and let $(L', R') \leftarrow MonienPreis(G_\varphi)$. Then return $CountX3SAT(\varphi, \boldsymbol{c}, L', R')$. If $\exists$ an edge between L and R, apply only the simplification rules (if any) as stated in Section 4.3. Choose an edge e between L and R. Then branch the variable x_e represented by e. Let the cardinality vector $\boldsymbol{c'}$ be the new cardinality vector by dropping off entries x_e and $\bar{x}_e$. Return $CountX3SAT(\varphi[x_e = 1], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(x_e) + CountX3SAT(\varphi[x_e = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x}_e)$[3].

18: If every clause in the formula is degree 2, choose any variable x and we branch $x = 1$ and $x = 0$. Let $\boldsymbol{c'}$ be the new cardinality vector by dropping the entries x and $\bar{x}$. Return $CountX3SAT(\varphi[x = 1], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(x) + CountX3SAT(\varphi[x = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x})$[5].

[4] More details on the updating of $\boldsymbol{c'}$ below in this section.

[5] More details on this branching rule is given in Section 4.

Note that every line in the algorithm has descending priority; Line 1 has higher priority than Line 2, Line 2 than Line 3 etc.

Line 1 of the algorithm is our stopping condition. If any clause is not exact satisfiable, immediately return 0. When no variables are left, then check if every clause is exactly satisfied. If yes, then return 1, else 0.

Line 2 of the algorithm deals with any clause that contains a constant 1. In this case, all the other literals in the clause must be assigned 0 and we can safely drop off this clause after that. Line 3 deals with any clause with a constant 0 in it. We can then safely drop the constant 0 from the clause. Line 4 deals with single-literal clauses. This literal must be assigned 1. Line 5 deals with two literal clauses when the two literals involve two different variables. Line 6 deals with two literal clauses when they come from the same variable, say x. Now if x does not appear elsewhere, then either $x = 1$ or $x = 0$ will satisfy this clause. Thus as done in Line 6, multiplying $CountX3SAT(\varphi, \boldsymbol{c'}, L, R)$ by the sum of $(\boldsymbol{c}(x) + \boldsymbol{c}(\bar{x}))$ would give us the correct value. Regardless of whether x appears elsewhere or not, drop this clause.

After Line 6, we know that all clauses are of length 3. In Line 7, if the formula is disconnected, then we deal with each components separately. Line 7 has some relation with Line 17. If the algorithm is not currently processing Line 17, then basically we just call the algorithm on different components. The explicit relationship between Line 7 and Line 17 will be given in Sect. 4.3. In Line 8, we deal with a literal that appears twice in a clause. Then we can assign that literal as 0. In Line 9, we have a literal and its negation appearing in the same clause, then we assign the last literal to be 0. In Line 10, we deal with clauses having two singletons and we need to update the cardinality vector $\boldsymbol{c}$ before we are allowed to remove one. Suppose we have two singletons x and y and we wish to remove say y, then we need to update the entries of $\boldsymbol{c}(x)$ and $\boldsymbol{c}(\bar{x})$ to retain the information of $\boldsymbol{c}(y)$ and $\boldsymbol{c}(\bar{y})$. Note that in the updated x, when $x = 0$, this means that both the original x and y are 0. On the other hand, when we have $x = 1$ in the updated x, this means that we can either have $x = 1$ in the original x, or $y = 1$. Thus, this gives us the following update: $\boldsymbol{c}(x) = \boldsymbol{c}(x) \times \boldsymbol{c}(\bar{y}) + \boldsymbol{c}(\bar{x}) \times \boldsymbol{c}(y)$ when x is assigned "1", and $\boldsymbol{c}(\bar{x}) = \boldsymbol{c}(\bar{x}) \times \boldsymbol{c}(\bar{y})$ when x is assigned "0". After which, we can then safely remove the entries of y and $\bar{y}$ from the cardinality vector $\boldsymbol{c}$.

In Lines 11, 12 and 13, we deal with two overlapping variables (in different permutation) between any two clauses. After which, any two clauses can only have at most only 1 overlapping variable between them. In Line 14, we deal with semi-isolated sets I such that we can remove all but one of its variable. In Line 15, if we can find a variable x such that by branching it, we can remove that amount of variables as stated, then we proceed to do so. The goal of introducing Line 14 and Line 15 is to help us out for Line 16, where we deal with variables that appear at least 3 times. Their relationship will be made clearer in the latersections. After which, all variables will appear at most 2 times and each clause must have at most degree 3. In Line 17, the remaining formula must consist of clauses of degree 2 and 3. Then we construct a graph G_φ, apply $MonienPreis(.)$ to it and choose a variable to branch, followed by applying simplification rules. We'll continue doing so until no degree 3 clauses exist. Lastly in Line 18, the formula will only consist of degree 2 clauses, and we will select any variable and branch $x = 1$ and $x = 0$. Hence, we have covered all cases in the algorithm.

Now, we give the details of Line 14. As I is semi-isolated, let x be the variable in I, such that x appears in further clauses containing variables not in I. Note that when $x = 1$ or when $x = 0$, the formula becomes disconnected and clauses involving $I - \{x\}$ become a component of constant size. Therefore, we can use brute force (requiring constant time), to check which assignments to the $|I| - 1$ variables satisfy the clauses involving variables from I, and then correspondingly update $\boldsymbol{c}(x)$ and $\boldsymbol{c}(\bar{x})$, and drop all variables in $I - \{x\}$ from φ. We call such a process *contraction* of I into x. Details given below.

Updating of Cardinality Vector in Line 14 (Contracting Variables). Let S be the set of clauses which involve only variables in I. δ below denotes assignments to variables in $I - \{x\}$. For $i \in \{0, 1\}$, let

$Z_i = \{\delta$: all clauses in S are satisfied when variables in I are set according to δ and $x = i\}$.

The following formulas update the cardinality vector for coordinate x and $\bar{x}$, by considering the different possibilities of δ which make the clauses in S satisfiable. This is done by summing over all such δ in Z_i (for $i = x = 0$ and $i = x = 1$), the multiplicative factor formed by considering the cardinality vector values at the corresponding true literals in δ. Here the literals ℓ in the formula range over literals involving the variables in $I - \{x\}$.

Let $\boldsymbol{c}(x) = \boldsymbol{c}(x) \times \sum_{\delta \in Z_1} \prod_{\ell \text{ is true in } \delta} \boldsymbol{c}(\ell)$.
Let $\boldsymbol{c}(\bar{x}) = \boldsymbol{c}(\bar{x}) \times \sum_{\delta \in Z_0} \prod_{\ell \text{ is true in } \delta} \boldsymbol{c}(\ell)$.

4 Analysis of the Branching Rules of the Algorithm

Note that Lines 1 to 14 are simplification rules and Lines 15 to 18 are branching rules. For Line 7, note that since the time of our algorithm is running in $O^*(c^n)$, for some c, then calling our algorithm onto different components will still give us $O^*(c^n)$. Therefore, we will analyse Lines 15 to 18 of the algorithm.

4.1 Line 15 of the Algorithm

The goal of introducing Lines 14 and 15 is to ultimately help us to simplify our cases when we deal with Line 16 of the algorithm. In Line 16, there can be some ugly overlapping cases which we don't have to worry after adding Lines 14 and 15 in the algorithm. The cases we are interested in are as follows.

(A) There exists a variable which appears in at least four clauses.

Suppose the variable is x_0, and the four clauses it appears in are $(x_0' \vee x_1 \vee x_2)$, $(x_0'' \vee x_3 \vee x_4)$, $(x_0''' \vee x_5 \vee x_6)$, $(x_0'''' \vee x_7 \vee x_8)$, where $x_0', x_0'', x_0''', x_0''''$ are either x_0 or $\bar{x_0}$. Note that $x_0, x_1, x_2, \ldots, x_8$ are literals involving different variables (by Lines 8,9,11,12,13). Note that setting literal x_0' to 1 will correspondingly set both x_1 and x_2 to 0; when x_0' is set to 0 correspondingly x_1 and $\bar{x_2}$ get linked. Similarly, when we set x_0'', x_0''', x_0''''. Thus, setting x_0 to 1 or 0 will give us removal of i variables on one setting and $12 - i$ variables on the other setting, where $4 \leq i \leq 8$. Thus, including x_0, this gives us, in the worst case, a branching factor of $\tau(9, 5)$.

(B) There exists a variable which appears in exactly three clauses.

Suppose x_0 is a variable appearing in the three clauses $(x_0' \vee x_1 \vee x_2)$, $(x_0'' \vee x_3 \vee x_4)$, $(x_0''' \vee x_5 \vee x_6)$ where x_0', x_0'', x_0''' are either x_0 or $\bar{x_0}$. Note that $x_0, x_1, x_2, \ldots, x_6$ are literals involving different variables. Let $I = \{x_0, v_1, v_2, \ldots, v_6\}$, where v_i is the variable for the literal x_i.

(B.1) If I is semi-isolated, or $I \cup \{u\}$ is semi-isolated for some variable u, then Line 14 takes care of this.

(B.2) If there are two other variables u, w which may appear in any clause involving variables from I, then we can branch on one of the variables u and then do contraction as in Line 14 for $I \cup \{w\}$ to w. Thus, we will have a branching factor of at least $\tau(8, 8)$.

(B.3) If there are at most two clauses $C1$ and $C2$ which involve variables from I and from outside I and these two together involve at least three variables from outside I, then consider the following cases.

Case 1: If both $C1$ and $C2$ have two variables from outside I. Then, let $C1$ have literal x_i' and $C2$ have literal x_j', where x_i' is either x_i or $\bar{x_i}$ and x_j' is either x_j or $\bar{x_j}$, and $i, j \in \{0, 1, \ldots, 6\}$. Now, one can branch on literal x_i' being 1 or 0. In both cases, we can contract the remaining variables of I into x_j (using Line 14). Including the two literals set to 0 in $C1$ when x_i' is 1, we get branching factor of $\tau(8, 6)$.

Case 2: $C1$ and $C2$ together have three variables from outside I. Without loss of generality assume $C1$ has one variable from outside I and $C2$ has two variables from outside I. Then let $C1$ have literal y which is outside I and $C2$ have literal x_j', where x_j' is either x_j or $\bar{x_j}$. Now, one can branch on literal y being 1 or 0. In both cases, we can contract the variables of I into x_j (using Line 14). Including the literal y we get branching factor of $\tau(7, 7)$.

(B.4) Case 2.3 and Case 2.4 in Lemma 10 for Line 16.

Lemma 9. *Branching the variable in Line 15 takes $O(1.1074^n)$ time. (The worst branching factor is $\tau(9, 5)$).*

4.2 Line 16 of the Algorithm

In this case, we deal with variables that appear exactly 3 times.

Lemma 10. *The time complexity of branching variables appearing 3 times is $O(1.1120^n)$.*

Proof. Suppose x_0 appears three times. Then we let the clauses that x_0 appear in be $(x_0' \vee x_1 \vee x_2)$, $(x_0'' \vee x_3 \vee x_4)$, $(x_0''' \vee x_5 \vee x_6)$, where the primed versions of x_0 denote either x_0 or $\bar{x_0}$.

Let $I = \{x_0, v_1, \ldots, v_6\}$, where v_i is the variable in the literal x_i.

Note that when x_0' is set to 1, then x_1 and x_2 are also set to 0. When x_0' is set to 0 then x_1 and x_2 get linked. Similarly, for setting of x_0'' and x_0'''. Thus, setting of x_0 to 1 or 0 allows us to remove i variables and $9 - i$ variables respectively among $v_1, \ldots, v_6$, where $3 \leq i \leq 6$ (the worst case for us thus happens with removal of 3 variables on one side and 6 on the other). We will show how to remove three further variables outside I in the following cases (these may fall on either side of setting of x_0 to 1 or 0 above). Including x_0, we get the worst case branching factor of $\tau(10, 4)$.

Let the variables outside I be called outside variables for this proof. Let a clause involving both variables from I and outside I be called a mixed clause. By Line 14 and 15 of the algorithm, there are at least 3 mixed clauses, and at least three outside variables which appear in mixed clauses.

Consider 3 mixed clauses $C1 = (x_i' \vee a_1 \vee a_2)$, $C2 = (x_j' \vee a_3 \vee a_4)$ and $C3 = (x_k' \vee a_5 \vee a_6)$, where a_2, a_4, a_6 are literals involving outside variables, and x_i', x_j', x_k' are literals involving variables from I.

Case 1: It is possible to select the three mixed clauses such that a_4 involves a variable not appearing in $C1$ and a_6 involves a variable not appearing in $C1, C2$.

Note that this can always be done when there are at least four outside variables which appear in some mixed clauses.

In this case, x_i' is set in at least one of the cases of x_0 being set to 1 or 0. Similarly for x_j' and x_k'. In the case when x_i' is set, one can either set a_2 or link it to a_1. In the case when x_j' is set, one can either set a_4 or link it to a_3. In the case when x_k' is set, one can either set a_6 or link it to a_5. Note that the above linkings are not cyclic as the variable for a_4 is different from that of a_1 and a_2. and the variable for a_6 is different from that of a_1, a_2, a_3, a_4. Thus, in total three outside variables are removed when x_0 is set to 1 and 0.

Case 2: Not Case 1. Here, the number of outside variables which appear in some mixed clause is exactly three. Choose some mixed clauses $C1, C2, C3$ such that exactly three outside variables are present in them. Suppose these variables are a, b, c. Suppose the number of outside variables in C1, C2, C3 is given by triple (s_1, s_2, s_3) (without loss of generality assume $s_1 \leq s_2 \leq s_3$). We assume that the clauses chosen are so as to have the earlier case applicable below. That is, if all three variables a, b, c appear in some mixed clause as only outside variable, then Case 2.1 is chosen; Otherwise, if at least 2 mixed clauses involving 2 outside variables are there and a mixed clause involving only one outside variable is there then Case 2.2. is chosen. Otherwise, if only one mixed clause involving two outside variable is there then Case 2.3 is chosen. Else, case 2.4 is chosen.

Case 2.1: $(s_1, s_2, s_3) = (1, 1, 1)$. This would fall in Case 1, as all three outside variables are different.

Case 2.2: $(s_1, s_2, s_3) = (1, 2, 2)$. As two variables cannot overlap in two different clauses, one can assume without loss of generality that the outside variables in $C1$ is a or b, in $C2$ are (a, b) and $C3$ are (b, c). But then this falls in Case 1.

Case 2.3: $(s_1, s_2, s_3) = (1, 1, 2)$. For this not to fall in Case 1, we must have the same outside variable in $C1$ and $C2$. Suppose a appears in $C1, C2$ and b, c in $C3$. Furthermore, to not fall in Case 1, we must have that all other outside clauses must have a only as the outside variable (they cannot have both b, c as outside variable, as overlapping of two variables is not allowed). Thus, by branching on a, and then contracting, using Line 14, I to x_k, will allow us to have a worst case branching factor $\tau(7, 7)$. Thus, this is covered under Line 15.

Case 2.4: $(s_1, s_2, s_3) = (2, 2, 2)$. Say a, b are the outside variables in C1, a, c are the outside variables in $C2$ and b, c are the outside variables in $C3$. Furthermore, no other mixed clauses are there (as no two clauses can overlap in two literals).

Case 2.4.1: At least one of a, b, c appears both as positive and negative literal in $C1, C2, C3$.

Suppose without loss of generality that a appears as positive in $C1$ and negative in $C2$. Then, setting a to be 1, allows us to set b as well as contract all of I to c using Line 14. Setting a to be 0, allows us to set c as well as contract all of I to b using Line 14. Thus, we get a worst case branching factor of $\tau(9, 9)$.

Thus, this is covered under Line 15.

Case 2.4.2: None of a, b, c appears both as positive and negative literal in $C1, C2, C3$. Without loss of generality assume a, b, c all appear as positive literals in $C1, C2, C3$.

When, we set $x'_i = 1$, we have that $a = b = 0$ and we can contract rest of I to c using Line 14. This gives us removal of 9 variables. When we set $x'_i = 0$, we have that $a = \bar{b}$, and thus c must be 0 (from $C2$ and $C3$), and thus we can contract rest of I into a using Line 14. Thus we get a worst case branching factor of $\tau(9, 9)$. Thus, this is covered under Line 15.

Therefore, the worst case time complexity is $O(\tau(10, 4)^n) \subseteq O(1.1120^n)$.

4.3 Line 17 of the Algorithm

We now deal with degree 3 clauses.

17: If there is a degree 3 clause in φ, then check if $\exists$ an edge between L and R. If no, then construct G_φ and let $(L', R') \leftarrow MonienPreis(G_\varphi)$. Then return $CountX3SAT(\varphi, \boldsymbol{c}, L', R')$. If $\exists$ an edge between L and R, apply only the simplification rules (if any) as stated in this section (Section 4.3). Choose an edge e between L and R. Then branch the variable x_e represented by e. Let the cardinality vector $\boldsymbol{c'}$ be the new cardinality vector by dropping off entries x_e and $\bar{x}_e$. Return $CountX3SAT(\varphi[x_e = 1], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(x_e) + CountX3SAT(\varphi[x_e = 0], \boldsymbol{c'}, L, R) \times \boldsymbol{c}(\bar{x}_e)$.

Now, we discuss Line 17 of the algorithm in detail. As long as a degree 3 clause exists in the formula, we repeat this process. First, we describe how to construct the graph G_φ.

Construction. We construct a graph $G_\varphi = (V, E)$, where $V = \{v_C : C$ is a degree 3 clause in $\varphi\}$. Given any vertices $v_{C'}$ and $v_{C''}$, we add an edge between them if any of the below conditions occur on clauses C' and C'', where C' and C'' are clauses with 3 neighbours:

1. If a common variable appears in both C' and C''
2. C' and C'' are connected by a chain of 2-degree clauses.

By construction, the graph G_φ has maximum degree 3. Let m_3 denote the number of degree 3 clauses in φ. This gives us $|V| = m_3$. We can therefore apply the result by Monien and Preis, with the size of the bisection width $k \leq m_3(\frac{1}{6} + \varepsilon)$.

We construct the graph G_φ when there are no edges between L and R, and then apply $MonienPreis(.)$ to get our new partitions L' and R', which are sets of clauses. These partitions will remain connected until all edges between them are removed. In other words, the variables represented by them are branched. Now instead of bruteforcing all the variables in the bisection width at the same time, we branch them edge by edge. After each branching, we apply simplification rules before branching again. By our construction, we will not increase the degree of our clauses or variables (except temporarily due to linking; the corresponding clause will then be removed via Line 6). Therefore, we never need to resort to the earlier branching rules (Line 15 and 16) that deal with variables appearing at least 3 times again. In other words, once we come into Line 17, we will be repeating this branching rule in a recursive manner until all degree 3 clauses have been removed. Applying the simplification rules could mean that some variables have been removed directly or via linking, or some degree 3 clauses have now been dropped to a degree 2 clause etc. In other words, the clauses in the sets L and R have changed. Therefore, we need to update L and R correspondingly to reflect these changes before we repeat the branching again.

After branching the last variable between the two partitions, the formula becomes disconnected with two components and Line 7 handles this. Recall that in Line 7, we gave an additional condition to check for any edges between L and R. During the course of applying simplification rules or branching the variables, it could be that additional components can be created before all the edges between L and R have been removed. Therefore, this condition to check for any edges between the partition is to ensure that Line 7 will not be called prematurely until all edges have been removed. We will now give in detail the choosing of the variable to branch below.

Choosing of Variables to Branch. Based on the construction earlier, an edge is added if any of the two possibilities mentioned above happen in the formula. Let e be an edge in the bisection width. We choose a specific variable to branch in the different scenarios listed.

1. Case 1: The edge e represents a variable sitting on two degree 3 clauses. Branch this variable.
2. Case 2: The edge e represents a chain of 2 degree clauses. We alternate the branchings between the variables that appear in a degree 3 clause and a

degree 2 clause at both ends whenever Case 2 arises for symmetry reasons. For example, if we have degree 3 clause $(a \vee b \vee c)$ in the left partition connected to degree 3 clause $(s \vee t \vee u)$ in the right partition via a chain $(c, d, e), \ldots, (q, r, s)$, and it is left partition end turn, then we branch on variable c; if it is right partition end turn then we branch on variable s. These branchings will remove the whole chain, and convert the two degree 3 clauses into degree two or lower clause by compression as described below.

Compression. Suppose C' and C'' are two degree 3 clauses connected via a chain $C_1, C_2, \ldots, C_k$, where c is a common variable between C' and C_1, and s is a common variable between C'' and C_k. When s is assigned either a value of 0 or 1, C'' drops to a clause of degree at most 2. C_k becomes a 2-literal clause (in the worst case) and we can link the two remaining literals in it together and the clause is dropped. Therefore, the neighbouring clause C_{k-1} has now become a degree 1 clause. By Line 10 of the algorithm, we can remove 1 singleton and C_{k-1} drops to a 2-literal clause. Continuing the process of linking, dropping of clause and removing of singletons, the degree 3 clause at the end, C', will drop to become a clause of at most degree 2 when C_1 is removed. Therefore, C' and C'' will drop to a clause of at most degree 2.

With the Compression method, we now have the following. Let C be a degree 3 clause. Since C is a degree 3 clause, it has an edge to three other degree 3 clauses, say E_1, E_2, E_3. Choose any edge, say between E_1 and C. Now this edge can either represent a variable appearing in both C and E_1, or a chain between E_1 and C with variables at both ends appearing in E_1 and C. Therefore, assigning a value of 0 or 1 to this chosen variable represented by the edge will cause C to drop to a clause of degree at most 2.

Self-loop. Note that such a case can arise, where a degree 3 clause can be connected via a degree 2 chain to itself. The idea to handle this is similar to Line 14 and by adopting the idea in Compression. Due to space constraints, details are omitted. More information is available at https://arxiv.org/abs/2007.07553.

Based on the choice of variables as mentioned above, we now give the time analysis for Line 17 of the algorithm. Note that the measure of complexity for our branching factors here is m_3, the number of degree 3 clauses.

Lemma 11. *The time complexity of dealing of branching variables in the bisection width is $O(1.1092^n)$.*

Proof. For m_3, the current number of degree 3 clauses, we have that each variable in a degree 3 clause occurs in exactly one further clause and that there are three variables per clause. Thus $3m_3 \leq 2n$ and $m_3 \leq \frac{2}{3}n$, where n is the current number of variables. Note that the bisection width has size $k \leq m_3(\frac{1}{6} + \varepsilon)$.

Once we remove the edges in the bisection width, the two sides (call them left (L) and right (R)) get disconnected, and thus each component can be solved independently. Here note that after the removal of all the edges in the bisection width, we have at most $m_3/2$ degree 3 clauses in each partition. As we ignore polynomial factors in counting the number of leaves, it suffices to concentrate on one (say left) partition. We consider two kinds of reductions: (i) a degree 3 clause on the left partition is removed or becomes of degree less than three due to a branching, and (ii) the degree 3 clauses on the right partition are not part of the left partition. The reduction due to (ii) is called bookkeeping reduction because we spread it out over the removal of all the edges in the bisection width. Note that after all the edges between L and R have been removed, $\frac{m_3}{2}$ many clauses are reduced due to the right partition not being connected to the left partition. As the number of edges in the bisection width is at most $\frac{m_3}{6}$, in the worst case, we can count at least $\frac{m_3}{2} \div \frac{m_3}{6} = 3$ degree 3 clauses for each edge in the bisection width that we remove. For the removal of degree 3 clauses in the left partition, we analyze as follows.

Let an edge be given between L and R. We let the degree 3 clause $C = (a \vee b \vee c)$ be on the left partition, and the degree 3 clause $T = (s \vee t \vee u)$ be on the right partition. Then the edge can be represented by c, with $s = c$ or $s = \bar{c}$, or the edge is represented by a chain of degree 2 clauses, with the ends being c and s. We branch the variable $c = 1$ and $c = 0$.

When $c = 0$, C gets dropped to a degree 2 clause. Now this also means that the given edge gets removed (either directly or via Compression). Counting an additional 3 degree 3 clauses from the bookkeeping process, we remove a total of 4 degree 3 clauses here.

When $c = 1$, then $a = b = 0$. Since C is a degree 3 clause, it is connected to 3 other degree 3 clauses. Now all 3 degree 3 clauses will either be removed, or will drop to a degree 2 clause (again either directly, or via Compression). Hence, this allows us to remove $1 + 3i + (3 - i)$ degree 3 clauses, where removing C counts as 1, i is the number of neighbours of C in the right partition (bookkeeping) while $(3 - i)$ be the number of neighbours on the left. Since $i \in \{1, 2, 3\}$, the minimum number of degree 3 clauses we can remove here happens to be for $i = 1$, giving us 6 degree 3 clauses for this branch. This gives us a branching factor of $\tau(6, 4)$.

When we branch the variable $s = 1$ and $s = 0$, C gets dropped to a degree 2 clause via Compression, and in both branches, the edge gets removed and we can count 3 additional clauses from the bookkeeping process. In both branches, we remove 4 degree 3 clauses. This gives us a branching factor of $\tau(4, 4)$. Since we are always doing alternate branching for Case 2 (branching at point c and then at point t), we can apply branching vector addition on $(6, 4)$ to $(4, 4)$ on both branches to get a branching vector of $(8, 8, 10, 10)$.

Hence, Case 1 takes $O(\tau(6, 4)^{m_3})$ time, while Case 2 takes $O(\tau(8, 8, 10, 10)^{m_3})$ time. Since Case 2 is the bottleneck, this gives us $O(\tau(8, 8, 10, 10)^{m_3}) \subseteq O(\tau(8, 8, 10, 10)^{\frac{2}{3}n}) \subseteq O(1.1092^n)$, which absorbs all subexponential terms.

4.4 Line 18 of the Algorithm

In Line 18, the formula φ is left with only degree 2 clauses in the formula. Now suppose that no simplification rules apply, then we know that the formula must consist of cycles of degree 2 because of Lines 2, 3, 5, 6 and 10 of the algorithm. Now if φ consists of many components, with each being a cycle, then we can handle this by Line 7 of the algorithm. Therefore, φ consists of a cycle.

Now, we choose any variable x in this cycle and branch $x = 1$ and $x = 0$. Since all the clauses are of degree 2, we can repeatedly apply Line 10 and other simplification rules to solve the remaining variables (same idea as in Compression). Therefore, we would only need to branch one variable in this line. This, and repeatedly applying the simplification rules, will only take polynomial time.

Putting everything together, we have the following result.

Theorem 12. *The whole algorithm runs in* $O(1.1120^n)$ *time.*

References

1. Byskov, J., Madsen, B., Skjernaa, B.: New algorithms for exact satisfiability. Theor. Comput. Sci. **332**, 513–541 (2005)
2. Cook, S.: The complexity of theorem proving procedures. In: Third Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151–158 (1971)
3. Dahllöf, V.: Exact algorithms for exact satisfiability problems. Linköping Studies in Science and Technology, Ph.D. dissertation no 1013 (2006)
4. Dahllöf, V., Jonsson, P.: An algorithm for counting maximum weighted independent sets and its applications. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), pp. 292–298 (2002)
5. Dahllöf, V., Jonsson, P., Beigel, R.: Algorithms for four variants of the exact satisfiability problem. Theor. Comput. Sci. **320**(2–3), 373–394 (2004)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Commun. ACM **5**(7), 394–397 (1962)
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
8. Fomin, F.V., Kratsch, D.: Exact Exponential Algorithms. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16533-7
9. Gaspers, S., Sorkin, G.B.: Separate, measure and conquer: faster polynomial-space algorithms for Max 2-CSP and counting dominating sets. ACM Trans. Algorithms (TALG) **13**(4), 44:1–44:36 (2017)
10. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds.) Complexity of Computer Computations. IRSS, pp. 85–103. Springer, Boston (1972). https://doi.org/10.1007/978-1-4684-2001-2_9
11. Krom, M.R.: The decision problem for a class of first-order formulas in which all disjunctions are binary. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik **13**(1–2), 15–20 (1967)
12. Kullmann, O.: New methods for 3-SAT decision and worst-case analysis. Theor. Comput. Sci. **223**(1–2), 1–72 (1999)
13. Monien, B., Preis, R.: Upper bounds on the bisection width of 3- and 4-regular graphs. J. Discret. Algorithms **4**(3), 475–498 (2006)
14. Porschen, S.: On some weighted satisfiability and graph problems. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 278–287. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30577-4_31
15. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**, 273–302 (1996)

16. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI, vol. 5, pp. 475–481 (2005)
17. Schaefer, T.J.: The complexity of satisfiability problems. In: Tenth Annual Symposium on Theory of Computing (STOC 1978), pp. 216–226 (1978)
18. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)
19. Leslie, G.: Valiant the complexity of computing the permanent. Theor. Comput. Sci. **8**(2), 189–201 (1979)
20. Wahlström, M.: Algorithms, measures and upper bounds for satisfiability and related problems. Ph.D. thesis, Department of Computer and Information Science, Linköpings Universitet (2007)
21. Zhou, J., Weihua, S., Wang, J.: New worst-case upper bound for counting exact satisfiability. Int. J. Found. Comput. Sci. **25**(06), 667–678 (2014)