Counting Satisfying Assignments in 2-SAT and 3-SAT

Vilhelm Dahllöf*, Peter Jonsson** and Magnus Wahlström***

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden
{vilda, petej, magwa}@ida.liu.se.

Abstract. We present an $O(1.3247^n)$ algorithm for counting the number of satisfying assignments for instances of 2-SAT and an $O(1.6894^n)$ algorithm for instances of 3-SAT. This is an improvement compared to the previously best known algorithms running in $O(1.381^n)$ and $O(1.739^n)$ time, respectively.

1 Introduction

A number of recent papers, including [2, 5, 7, 8, 14], have studied various counting problems from different perspectives. This growing interest is probably due to the fact that decision problems – "is there a solution?" – have been thoroughly studied whereas our knowledge about counting problems – "how many solutions?" – is in comparison much more modest. Counting problems are not only mathematically interesting, they arise naturally when asking questions such as "what is the probability that a formula in the propositional calculus is true?" [11] or "what is the probability that a graph remains connected, given a probability of failure on each edge?" [12].

The subject of this paper is the #2-SAT and #3-SAT problems, i.e. counting the number of satisfying assignments to instances of 2-SAT and 3-SAT. These problems are known to be #P-complete [9, 13]. Exact algorithms (with worst-case complexity analyses) have been studied earlier by Dubois [4], Zhang [14] and Littman *et al.* [10]. The algorithms by Dubois and Zhang run in $O(1.6180^n)$ time for #2-SAT and $O(1.8393^n)$ time for #3-SAT while the algorithms by Littman *et al.* run in $O(1.381^n)$ and $O(1.739^n)$ time, respectively.

We will here present an $O(1.3247^n)$ algorithm for the #2-SAT problem and an $O(1.6894^n)$ algorithm for the #3-SAT problem. Both algorithms run in polynomial space. These algorithms are (as the algorithms by Littman *et al.*) based

* The research is supported by CUGS – National Graduate School in Computer Science, Sweden.

** The research is partially supported by the *Swedish Research Council* (VR) under grant 221-2000-361.

*** The research is supported by CUGS – National Graduate School in Computer Science, Sweden.

on the Davis-Putnam-Logemann-Loveland algorithm for satisfiability modified to count all satisfying assignments [2] and using a connected-components approach [1]. The basic idea is to choose a variable x and recursively count the number of satisfying assignments where x is true and where x is false. The main difference between our algorithms and those by Littman *et al.* lies in the choice of variable; our algorithms make a more elaborate analysis of the formula and are able to choose a "better" variable. The #2-SAT algorithm is also improved by using certain ideas from an algorithm for counting independent sets in graphs [3].

In the next section some definitions will be given and the notation used will be explained. Then the paper will use two sections to focus on the #2-SAT and #3-SAT algorithms, respectively.

2 Preliminaries

A *propositional variable* can have the values *true* and *false* and a *literal* is either a propositional variable x or its negation $\neg x$. A propositional formula on *conjunctive normal form* (CNF) is a conjunction of disjunctions of literals. A k-SAT formula ($k > 0$) is a propositional formula in CNF such that each clause contains at most k literals. A *satisfying assignment* is an assignment to every propositional variable of a formula making the entire formula true. #2-SAT is the problem of computing the number of satisfying assignments of 2-SAT formulae and #3-SAT is the corresponding problem for 3-SAT formulae. We note that an empty formula (i.e. a formula containing no clauses) has 1 model while a formula containing an empty clause (i.e. a clause containing no literals) is not satisfiable and has 0 models. We define the size of a formula $|F|$ as the number of variables F contains.

Given a formula F over the variables V, we define the *constraint graph* as an undirected graph (V, E) where $E = \{(x, y) \mid x, y$ appear in the same clause of $F\}$. We say that x and y are *constrained* iff $(x, y) \in E$ and the *constraints of* F are E. The *neighbourhood* of x ($N_F(x)$) is the set $\{y \mid (x, y) \in E\}$, the *degree* of x ($\delta_F(x)$) is $|N_F(x)|$ and the degree of F is $\delta(F) = \max\{\delta_F(x) \mid x \in Var(F)\}$. Note that if $\delta(F) = k$, then $|E| \leq k \cdot |F|/2$. We say that a formula is a *cycle* or a *path* whenever the constraint graph is a cycle or a path. A *connected component* of such a graph is a maximal subgraph such that for every pair of vertices x, y in the subgraph, there is a path from x to y. We say that the *connected components* of a formula are the subformulas corresponding to the connected components of the constraint graph.

Given a formula F and a literal l, we denote by $F|_l$ the result of making literal l true in F and simplifying the result by (1) removing any clause that contains l; (2) removing $\neg l$ whenever it appears; and (3) performing unit propagation as far as possible. By unit propagation, we mean to repeatedly apply rules of the following two types:

1. $(p \vee q \vee r) \wedge (p) \longrightarrow true$; and
2. $(p \vee q \vee r) \wedge (\neg p) \longrightarrow (q \vee r)$.

We extend this notation to $F|_\Gamma$, where Γ is a set of literals.

Consider the following example: $F = (p \vee q \vee r)$ and note that $\#3\text{-SAT}(F) = 7$. It seems reasonable that $\#3\text{-SAT}(F|_q) + \#3\text{-SAT}(F|_{\neg q}) = \#3\text{-SAT}(F)$ but this is not the case. We see that $(p \vee q \vee r)|_q$ is simplified to the empty formula (which has 1 model by definition) and $\#3\text{-SAT}(F|_q) + \#3\text{-SAT}(F|_{\neg q}) = 4$. The problem is that the variables q and r (which can be given arbitrary values) are "eliminated" in the simplification process. Thus, we introduce a function $\Psi(F, R, \Gamma)$, where F is the formula to be simplified, R the set of eliminated variables and Γ is the literals to be assigned $true$, it returns (F', R'), where F' is the simplified formula and R' is the new set of eliminated variables; R' is computed as $R \cup (Var(F) - (Var(F') \cup \Gamma))$, where $Var(F)$ denotes the set of variables of F.

The algorithms we present are based on the Davis-Putnam-Logemann-Loveland algorithm for satisfiability modified for counting satisfying assignments and they build on two straightforward principles: assume F is an arbitrary SAT formula, R a set of eliminated variables and let $\#\text{SAT}(F, R)$ denote the number of satisfying assignments of F modulo the eliminated variables in R.

Principle 1. Arbitrarily choose $x \in Var(F)$. Then,

$$\#\text{SAT}(F, R) = \#\text{SAT}(\Psi(F, R, \{x\})) + \#\text{SAT}(\Psi(F, R, \{\neg x\})).$$

Principle 2. Let F_1, F_2 be disjoint components of F. Then,

$$\#\text{SAT}(F, R) = 2^{|R|} \cdot \#\text{SAT}(F_1, \emptyset) \cdot \#\text{SAT}(F_2, \emptyset).$$

Note that these disjoint components can be found in polynomial time. The correctness proofs of the algorithms are omitted since they can trivially be inferred from these two principles.

3 Algorithm for #2-SAT

Before presenting the algorithm for #2-SAT, we need one more definition: given a formula F and a variable x, we define

$$L_F(x) = |\{u \mid (\sim u, \sim v) \in F \wedge u \in N_F(x) \wedge v \notin N_F(x) \cup \{x\}\}|$$

where $\sim u$ denotes either u or $\neg u$. Informally, one can say that $L_F(x)$, $x \in V$, tells how many of the neighbours of x that appear in a constraint together with some variables not in $N_F(x) \cup \{x\}$. If $L_F(x) = 1$, then $B(x)$ denotes the unique variable in $N_F(x)$ that is adjacent to one or more vertices outside $N_F(x) \cup \{x\}$. In Figure 1 there is an example where $L_F(x) = 1$.

Our algorithm for #2-SAT is presented in Figures 2 and 3. The function C_E computes #2-SAT by exhaustive search. It will be applied only to formulas of size ≤ 4 and can thus be safely assumed to run in $O(1)$ time. The correctness of the algorithm follows from the discussion in the previous section. When analysing the algorithm we will often encounter recurrences of the form

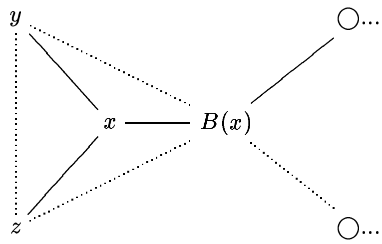

Fig. 1. A constraint graph where $L_F(x) = 1$; the filled lines indicate where there is a constraint by necessity, whereas a pointed line indicates a possible constraint

$$T(n) \leq \sum_{i=1}^{k} T(n - r_i) + \text{poly}(n).$$

They satisfy $T(n) \in O(\lambda(r_1, \ldots, r_k)^n)$ where $\lambda(r_1, \ldots, r_k)$ is the largest, real-valued root of the function $f(x) = 1 - \sum_{i=1}^{k} x^{-r_i}$. Note that we can ignore the polynomial factor so we will assume that all polynomial time computations take $O(1)$ time.

When inspecting C and C_3 one can see that every connected component of the input will be recursively solved by the algorithm. However, this will not increase the time required, since $\sum_{i=1}^{k} T(n_i) \leq T(n)$ when $n = \sum_{i=1}^{k} n_i$.

Given a 2-SAT formula F and a literal l, we define $Unit(F, l)$ to be the set of variables that appears in a clause of length 1 after having made literal l true in F and simplifying the result by (1) removing any clause that contains l; and (2) removing $\neg l$ whenever it appears (in other words, the usual simplification process employed by Ψ but without performing unit propagation). To illustrate $Unit$, consider the formula $F = (x \vee y) \wedge (x \vee z) \wedge (\neg x \vee w)$. It can easily be verified that $Unit(F, x) = \{w\}$ and $Unit(F, \neg x) = \{y, z\}$. In fact, if $\delta(x) = k$ in a 2-SAT formula F, then $|Unit(F, x)| + |Unit(F, \neg x)| \geq k$. This observation will be used several times in the complexity analysis.

Lemma 1. *Algorithm C_3 runs in $O(1.3045^n)$ time.*

Proof. In this lemma, we will measure the size of a formula G by the number m of constraints it contains. We will ultimately show that C_3 runs in $O(1.1939^m)$ time; by noting that a degree-3 formula with n vertices can contain at most $3n/2$ constraints, we get that the algorithm runs in $O(1.1939^{3n/2}) = O(1.3045^n)$ time.

We assume that the input formula F is connected and contains m constraints. Cases 0–2 are trivial and case 3 does not increase the running time.

Case 4: If F is a cycle, then F is immediately transformed into a path in both recursive calls. So, the interesting case is when the input is a path – then the time is bounded by the recurrence $T(m) \leq 4T(\lceil m/2 \rceil - 2)$ and straightforward calculations show that $T(m) \in O(4^{\log_2 m}) = O(m^2)$.

Algorithm $C_3(F, R)$
Case 0: F has an empty clause. Return 0.
Case 1: F is empty. Return $2^{|R|}$.
Case 2: $|F| \leq 4$. Return $C_E(F)$.
Case 3: F consists of two disjoint components F_1, F_2.
Return $2^{|R|} \cdot C_3(F_1, \emptyset) \cdot C_3(F_2, \emptyset)$.

Case 4: $\delta(F) \leq 2$. Pick a variable x in the following way:

1. If F is a path, choose x to be a variable that splits F into two paths of lengths $\lceil |F|/2 \rceil$ and $\lfloor |F|/2 \rfloor$.
2. If F is a cycle, choose x arbitrarily.

Return $C_3(\Psi(F, R, \{x\})) + C_3(\Psi(F, R, \{\neg x\}))$.

Case 5: $\delta(F) = 3$. Pick a variable x such that $\delta(x) = 3$ and do the following:

1. If $L_F(x) = 1$, then return $C_3(\Psi(F, R, \{B(x)\})) + C_3(\Psi(F, R, \{\neg B(x)\}))$.

2. If $L_F(x) > 1$, then return $C_3(\Psi(F, R, \{x\})) + C_3(\Psi(F, R, \{\neg x\}))$.

Fig. 2. Help function for computing #2-SAT

Case 5.1: $L_F(x) = 1$. We study the formula $F' = F|_{B(x)}$. First note that $\delta(B(x)) \geq 2$ since, otherwise, $|F| = 4$ and Case 2 would have applied. Now, $B(x)$ is given a fixed value and $\delta(B(x)) \geq 2$ implies that F' contains at least two constraints less than F. Furthermore, the neighbourhood of x forms a component F'' that is not connected to the rest of the formula. This component contains three variables and at least two constraints. Consequently, F'' will be taken care of in $O(1)$ time in the next recursive step of the algorithm (by component analysis and using C_E). The time needed in this case is thus bounded by $T(m) \leq 2T(m-4)$ since exactly the same situation arises when $B(x)$ is given the value false. It follows that $T(m) \in O(1.1892^m)$.

Algorithm $C(F, R)$
Case 0: F has an empty clause. Return 0.
Case 1: F is empty. Return $2^{|R|}$.
Case 2: F consists of two disjoint components F_1, F_2.
Return $2^{|R|} \cdot C(F_1, \emptyset) \cdot C(F_2, \emptyset)$.
Case 3: $\delta(F) \leq 3$. Return $C_3(F, R)$.
Case 4: Pick a variable x s.t. $\delta(x) \geq 4$ and return $C(\Psi(F, R, \{x\})) + C(\Psi(F, R, \{\neg x\}))$.

Fig. 3. Function for computing #2-SAT

Case 5.2: $L_F(x) > 1$ – that is, of the variables $\{v, y, z\} = N_F(x)$, at least two are related to variables not in $\{v, x, y, z\}$. In this worst-case analysis, we assume that only v and y are related to variables not in $\{v, x, y, z\}$.

Let $S = Unit(F, x) \cap \{v, y\}$ and $S' = Unit(F, \neg x) \cap \{v, y\}$. The running time $T(m)$ of the algorithm satisfies the recursive relation

$$T(m) \le T(m - 3 - |S|) + T(m - 3 - |S'|)$$

Here, three constraints are removed due to giving x a fixed value and at least one more constraint is removed for each variable v, y that appear in a unit clause. It is easy to see that $|S| + |S'| = 2$ and that the worst case occurs when $|S|$ or $|S'|$ equals 0. The time needed in this case is bounded by $T(m) \le T(m-3) + T(m-5)$ and $T(m) \in O(1.1939^m)$.

Theorem 1. *Algorithm C runs in $O(1.3247^n)$ time.*

Proof. The result follows from Lemma 1 if $\delta(F) \le 3$. Otherwise, let $S = Unit(F, x)$ and $S' = Unit(F, \neg x)$. The running time $T(n)$ of the algorithm satisfies the recursive relation

$$T(n) \le T(n - 1 - |S|) + T(n - 1 - |S'|)$$

Since $\delta(x) \ge 4$, we know that $|S| + |S'| \ge 4$. Obviously, $\lambda(a, b) = \lambda(b, a)$ and $\lambda(a, b) \ge \lambda(a + 1, b)$ so $\lambda(a, b) \le \lambda(1, 5) \approx 1.3247$ whenever $a, b \ge 5$. It can easily be checked that the small number of remaining cases also satisfy $\lambda(a, b) \le \lambda(1, 5)$ and $T(n) \in O(1.3247^n)$.

4 Algorithm for #3-SAT

In this section we present an algorithm D for #3-SAT and provide an upper bound on its running time.

To find this upper time bound, we will use a different method than the one described in Section 3. We will need to solve systems of recurrences like this:

$$f(n) = f(n - 1) + g(n - 1) \tag{1}$$
$$g(n) = f(n - 1) + g(n - 2) \tag{2}$$

To do this, we rewrite (2) so that the right hand side is only dependent upon $n - 1$, by adding a new equation to the system:

$$f(n) = f(n - 1) + g(n - 1) \tag{3}$$
$$g(n) = f(n - 1) + h(n - 1) \tag{4}$$
$$h(n) = g(n - 1) \tag{5}$$

We then write the equations into a matrix T, where each row represents a line in the system. An upper bound for the solution to the system is $f(n) \in O(\lambda^n)$,

Algorithm $D(F,R)$
Case 0: F has an empty clause. Return 0.
Case 1: F is empty. Return $2^{|R|}$.
Case 2: F consists of disjoint components F_1, F_2.
Return $2^{|R|} \cdot D(F_1, \emptyset) \cdot D(F_2, \emptyset)$.
Case 3: There exists some 1-clause $(x) \in F$. Return $D(\Psi(F, R, \{x\}))$.
Case 4: There exists some 2-clause in F. Pick a variable x in the following way:

1. If some variable v is a member of a 2-clause $(v \vee w)$ and $d(v) = 1$, pick w as x.
2. If some variable v is a member of a 2-clause $(v \vee w)$ and $d(v) = 2$, pick w as x.
3. Otherwise, pick as x a variable which is a member of as many 2-clauses as possible.

Return $D(\Psi(F, R, \{x\})) + D(\Psi(F, R, \{\neg x\}))$.

Case 5: Pick a variable x in the following way:

1. If $d(v) = 1$ for some variable v, pick a neighbour of maximum degree as x.
2. If $d(v) = 2$ for some variable v, pick a neighbour of maximum degree as x.
3. Otherwise, pick as x a variable of maximum degree.

Return $D(\Psi(F, R, \{x\})) + D(\Psi(F, R, \{\neg x\})$.

Fig. 4. Function for computing #3-SAT

where λ is the largest eigenvalue of T. This can be seen by using generating functions for solving recurrence equations [6]. In this example:

$$T = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

and $\lambda = 1.8019$.

Some terminology is needed. A *k-clause* is a clause that contains k literals. The degree of x in F ($d_F(x)$), for a variable x, denotes the number of clauses in F that x appears in. Usually, we will use $d(x)$ when the formula F is clear from the context. Note that this is different from $\delta_F(x)$ as defined in Section 2. A formula where every variable has degree k is *k-regular*. $d_2(x)$ denotes the number of 2-clauses that x is a member of.

The algorithm is presented in Figure 4. The idea behind it is that if some variable v has a low degree ($d(v) = 1$ or 2), then we can pick its neighbours as branching variables to reach a situation where $d(v) = 0$ and eliminate v in this way. If no such variable exists, we simply pick a variable of high degree.

Lemma 2. *If F is a connected k-regular formula with only 3-clauses, then no k-regular formula $F' \neq F$ with only 3-clauses can be reached by assigning truth values to variables in F.*

Proof. Let F be a connected, k-regular formula which only contains 3-clauses. Assume that Γ is a set of assignments such that $F|_\Gamma$ is k-regular and only contains 3-clauses, and that $F|_\Gamma \neq F$. Note that since $F|_\Gamma$ only contains 3-clauses, $F|_\Gamma \subseteq F$. Let $F_1 = F|_\Gamma$ and $F_2 = F - F|_\Gamma$.

Since F is connected, there is some variable x which occurs in both F_1 and F_2. Since x occurs in F_1, it is not assigned a value by Γ. However, since $d_{F_1}(x) = d_F(x) - d_{F_2}(x) < k$, F_1 cannot be k-regular.

Theorem 2. *Algorithm D runs in $O(1.6894^n)$ time.*

Proof. We will analyze the time complexity of D case by case.
Cases 0, 1 and 3: These cases take $O(1)$ time.
Case 2: This case does not increase the time needed.
Case 4.1: When $w = true$, $d(v) = 0$ and v is removed from F; when $w = false$, $v = true$. $T(n) = 2T(n-2)$ so $T(n) \in O(1.4142^n)$.
Case 4.2: When $w = true$, 5.1 is met (or case 4.1, or some earlier case); when $w = false, v = true$. $T(n) = T(n-2)+T(n-3)+2T(n-4)$ so $T(n) \in O(1.5664^n)$.
Case 4.3: Even if the chosen variabel is only a member of one 2-clause, we have $T(n) = T(n-1) + T(n-2)$ with solution $T(n) \in O(1.6180^n)$. If it is a member of several 2-clauses, then the limit will be lower.
Case 5.1: This case takes $O(1.5214^n)$ time. Suppose $d(v) = 1$, $(u \vee v \vee w) \in F$ and that u is the chosen variable. When $u = true$, $d(v) = 0$ and v is removed from F; when $u = false$, case 4.1 is met. In total, the recurrence is $T(n) = T(n-2) + 2T(n-3)$ with solution $O(1.5214^n)$.
Case 5.2: This case takes $O(1.6560^n)$ time. Suppose $d(v) = 2$, $(u \vee v \vee w) \in F$ and that u is the chosen variable. When $u = true$, case 5.1 is met (or some earlier case); when $u = false$, case 4.2 is met (or some earlier case). The worst case is when no earlier case is met, and the recurrence for this case is $T(n) = 2T(n-3) + 3T(n-4) + 2T(n-5)$ with solution $O(1.6560^n)$.
Case 5.3: This case requires a more careful analysis. We shall formulate a set of recurrence equations that describes the behaviour of the algorithm in this case. Let x be the chosen variable. When we reach this case, F contains no 2-clauses. Setting $x = true$ creates k_1 2-clauses, and setting $x = false$ creates k_2 2-clauses, where $k_1 + k_2 = d(x)$. For the moment, assume that $d(x) = 4$.

Let $T(n,k)$ represent a situation where F contains n variables and k 2-clauses. The initial situation is $T(n,0)$. Now, we enumerate a number of possible recurrences where x is the chosen variable, and the 2-clauses it appears in are $(x \vee v_i)$ or $(\neg x \vee v_i)$. Some cases that are clearly dominated by some listed case are ignored.

$$T(n,0) = T(n-1,k_1) + T(n-1,k_2) \tag{6}$$
$$T(n,k) = T(n-1,k-1) + T(n-2,k) \text{ if } d_2(x) = 1 \tag{7}$$
$$T(n,k) = T(n-2,k-1) + T(n-1,k) \text{ if } d_2(x) = 1 \tag{8}$$
$$\begin{aligned} T(n,k) = {} & T(n-1,k-2) + \\ & T(n-3,k-d_2(v_1)-d_2(v_2)) \text{ if } d_2(x) = 2 \end{aligned} \tag{9}$$
$$\begin{aligned} T(n,k) = {} & T(n-2,k-1-d_2(v_1)) + \\ & T(n-2,k-1-d_2(v_2)) \text{ if } d_2(x) = 2 \end{aligned} \tag{10}$$

The recurrences for $d_2(x) > 2$ are similar to (9)–(10).

The idea behind these recurrences is that when $x = true$ then every 2-clause $(x \vee v_i)$ is removed and for every 2-clause $(\neg x \vee v_i)$, $v_i = true$ and $d_2(v_i)$ 2-clauses are removed. In (7)–(8), we know that $d_2(v_1) = 1$; in (9)–(10) and similar recurrences, we know that $1 \leq d_2(v_i) \leq d_2(x)$ for all v_i involved.

In (7)–(8), $d(x) \geq 2$ is assumed, so that one new 2-clause will be created, either when $x = true$ or $x = false$. $d(x) = 1$ only occurs when $(x \vee v_1)$ is disjoint from the rest of F, so this assumption is allowed.

If we go through all possible combinations of rules like these, we find that the worst case is when only recurrences (6) and (7) are used, and $k_1 = 4, k_2 = 0$. The solution to this system is $T(n, 0) \in O(1.6894^n)$.

If $d(x) > 4$ then more 2-clauses are created and either more recurrences with low $d_2(x)$ or recurrences with a higher $d_2(x)$ are used. These cases are both easier than $d(x) = 4$.

The remaining case is when F is 3-regular. Then, as Lemma 2 shows, for the rest of the recursive calculations no 3-regular formula will appear, so there will always exist variables of degree 1 or 2. If $T(n)$ is the time required for a 3-regular formula and $T'(n)$ is the time required if variables of degree 1 or 2 are guaranteed to exist, we have $T(n) = 2T'(n-1)$ where $T'(n) \in O(1.6560^n)$, so $T(n) \in O(2 \cdot 1.6560^{n-1}) = O(1.6560^n)$.

In total, this shows that algorithm D runs in $O(1.6894^n)$ time.

References

1. R. J. Bayardo, Jr. and J. Pehoushek. Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 157–162. AAAI Press, 2000.
2. E. Birnbaum and E. L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.
3. V. Dahllöf and P. Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2002)*, pages 292–298, 2002.
4. O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64, 1991.
5. L. Goldberg and M. Jerrum. Counting unlabelled subtrees of a tree is #P-complete. *LMS Journal of Computation and Mathematics*, 3:117–124, 2000.
6. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, USA, second edition, 1994.
7. C. S. Greenhill. The complexity of counting colourings and independent sets in sparse graphs and hypergraphs. *Computational Complexity*, 9(1):52–72, 2000.
8. H. B. Hunt III, M. V. Marathe, V. Radhakrishnan, and R. E. Stearns. The complexity of planar counting problems. *SIAM Journal on Computing*, 27(4):1142–1167, 1998.
9. D. Kozen. *The design and analysis of algorithms*. Springer-Verlag, 1992.
10. M. L. Littman, T. Pitassi, and R. Impagliazzo. On the complexity of counting satisfying assignments. Unpublished manuscript.
11. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

12. S. P. Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM Journal on Computing*, 31(2):398–427, 2001.
13. L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
14. W. Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155(1):277–288, 1996.