Counting Models for 2SAT and 3SAT Formulae

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 here present algorithms for counting models and max-weight models for 2SAT and 3SAT formulae. They use polynomial space and run in $O(1.2561^n)$ and $O(1.6737^n)$ time, respectively, where n is the number of variables. This is faster than the previously best algorithms for counting non-weighted models for 2SAT and 3SAT, which run in $O(1.3247^n)$ and $O(1.6894^n)$ time, respectively. In order to prove these time bounds, we develop new measures of formula complexity, allowing us to conveniently analyze the effects of certain factors with a large impact on the total running time. We also provide an algorithm for the restricted case of separable 2SAT formulae, with fast running times for well-studied input classes. For all three algorithms we present interesting applications, such as computing the permanent of sparse 0/1 matrices.

Key words: Counting Models, Satisfiability, Exponential-Time Algorithms, Exact Algorithms, Upper Bounds

1 Introduction

Most of the efforts in algorithm construction have been dedicated to algorithms for decision problems, *i.e.*, finding *a* solution to the problem instance. For instance, this can involve finding a shortest path in a graph or a satisfying assignment to a boolean formula. As a natural extension we have the

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

†Corresponding author. Tel. +46 13 282479; Fax: +46 13 264499

‡Partially supported by the *Center for Industrial Information Technology* (CENIIT) under grant 04.01, and by the *Swedish Research Council* (VR) under grants 221-2000-361 and 621-2003-3421.

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

counting problems, where one wants to not merely decide the existence of a solution, but to find the *number* of solutions. One of the first algorithms for a counting problem came in the early 1960's with Ryser's [17] $O(n^2 2^n)$ time algorithm for counting the number of perfect matchings in a bipartite graph (also known as computing the *permanent* of a 0/1 matrix). In the 1970's Valiant [19] proposed the counting complexity class #P and showed that computing the permanent is complete for #P. It is interesting to note that both NP-complete problems as well as some decision problems, known to be in P, can have a counting counterpart which is #P-complete. For instance, both #2SAT and #3SAT are #P-complete [12, 20].

Algorithms for #2SAT and #3SAT with better time bounds than the trivial $O(2^n)$ bound have been presented by Dubois [11], Zhang [21], Littman *et al.* [15] and Dahllöf, Jonsson and Wahlström [8]. In this paper, which extends the work and results in [7] and [8], we improve on the previously best running times for both these problems, with algorithms that solve the more general weighted versions. Considering weights of solutions opens the field for more applications, as seen for instance in [3] and later in this paper.

For the problem of counting the number of max weight models to a 2SAT formula, here referred to as $\#2\text{SAT}_w$, we present an algorithm with a running time in $O(1.2561^n)$, significantly improving on the previously best bound for #2SAT of $O(1.3247^n)$, achieved in [8]. There are several factors behind this improvement. One is a trick that allows us to split a constraint graph into its biconnected components. Among other things, this provides a way to remove variables which occur only once in a formula in polynomial time. Another factor is our method of analysis, where we use a special measure of formula complexity combining the number of variables and the number of clauses into a single value which is more representative of formula complexity than the standard n =#variables. By using this measure, we are able to divide our analysis into cases depending on the average degree of a formula, and capture and quantify the beneficial effects of having an algorithm which ensures that the average degree of a formula will be gradually decreasing. To say something about the corresponding decision problem, we see that it is not 2SAT, but rather a weighted variant, 2SAT_w. We are not aware of any dedicated algorithms for this problem, but to get some idea of its hardness, one can note that it contains MAXIMUM INDEPENDENT SET as a special case. MAXIMUM INDEPENDENT SET is known to be NP-complete and the so-far fastest poly-space algorithm runs in $O(1.2025^n)$ time, achieved by Robson [16].

For $\#3\text{SAT}_w$, our algorithm has a running time in $O(1.6737^n)$, and the previously best result for #3SAT is $O(1.6894^n)$, achieved in [8]. This improvement is mainly due to a more precise complexity analysis, where we use another measure of formula complexity to better capture the effects of having clauses of cardinality 2 in the formula. As for the corresponding decision problem, we are not aware of any non-trivial worst-case time bounds for 3SAT_w or any related optimization problems, but one can note that the so-far best exact poly-space algorithm for 3SAT runs in $O(1.4802^n)$ time.

We also present an algorithm that counts weighted models for 2SAT formulae having separable constraint graphs. While this class of formulae may sound exotic, we will present interesting graph applications. The separable graphs form a broad class, including many well-studied sub-classes such as the geometric graphs, graphs embeddable on surfaces of bounded genus, planar graphs, forests, grids, graphs with an excluded minor and graphs with bounded tree width. Counting in many of those classes still remains #P-complete as shown by Vadhan [18]. Our work here can be seen as an extension to the work by Díaz *et al.* presented in [10]. There the authors present an algorithm for counting homomorphisms in graphs with a bounded tree width of w. Among the applications they show how to count independent sets in linear time, while an algorithm for counting *maximum* independent sets would take exponential time using their approach. In this paper we show how our algorithm for counting in separable graphs, for the special case of graphs with fixed tree width, yields algorithms for counting maximum independent sets as well as independent sets, both with running times in $O(n^w)$.

In the following presentation we first give some preliminaries, definitions and technical tools in Section 2. Section 3 deals with a procedure to simplify a formula while keeping track of certain information. Then follow Sections 4 and 5 on the algorithms for $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$ respectively. In Section 6 we present the algorithm for separable 2SAT formulae. Section 7 deals with applications for the algorithms. Conclusions and a brief discussion about our results and possible future research directions are given in Section 8.

2 Preliminaries

In this section we review the definitions and notation used in the paper. Some more specific preliminaries will be given when the algorithms are presented.

2.1 Weighted satisfiability problems

A *propositional variable*, or *variable* for short, can have the values *true* and $false$, for brevity written as 1 and 0, respectively. A *literal* is either a propositional variable x or its negation $\neg x$. To each literal a_i a weight $w(a_i) \in N$ is associated; the vector $\mathbf{w}$ containing these weights is called a *weight vector*. A propositional formula on *conjunctive normal form* (CNF) is a conjunction of disjunctions of literals. A disjunction of literals is referred to as a *clause*. A *k-SAT formula* ($k > 0$) is a propositional formula in CNF such that each clause contains at most k literals. An *n-clause* is a clause containing n literals. $Var(F)$ denotes the variable set of F and $n(F) = |Var(F)|$. A variable x which occurs either only as x or only as $\neg x$ in a formula is called *monotone*.

We define the *degree* $d(x)$ of a variable x in the formula F as the number of clauses in F containing x. A variable x is called *singleton* if $d(x) = 1$. The maximum degree of any variable in F is denoted $d(F)$ and $n_d(F)$ is the number of variables of degree d in F. As a measure of formula complexity we introduce

$$m(F) = \sum_{x \in Var(F)} d(x)$$

A *satisfying assignment* or *model* is an assignment to every propositional variable of a formula making the entire formula true. An *extendible assignment* is an assignment to a subset of the variables, such that no clause becomes false. Note that an empty formula has one model and a formula containing the empty clause has no model.

Let F be a propositional formula, and let L be the set of all literals for all variables occurring in F. Given a weight vector $\mathbf{w}$ and a model M for F, we define the weight of M as

$$\mathcal{W}(M) = \sum_{\{l \in L \mid l \text{ is true in } M\}} w(l)$$

We use the acronym MWM for *maximum weighted model*.

Analogously to a weight vector, a *cardinality vector* $\mathbf{c}$ for F is a vector assigning an integer value $c(l) \geq 1$ to each literal $l \in L$. Cardinality vectors are used as a technical aid in Section 3. Given a cardinality vector $\mathbf{c}$ and a model M for F, we define the cardinality of M as

$$\mathcal{C}(M) = \prod_{\{l \in L \mid l \text{ is true in } M\}} c(l)$$

We can now give our problem formulations: #2SAT (#3SAT) is the problem of computing the number of satisfying assignments of 2SAT (3SAT) formulae, disregarding weight. More strictly, given a 2SAT formula F we define

$$\#2\text{SAT}(F) = |\{M \mid M \text{ is a model for } F\}|$$

Extending the definition to the problem of counting max-weight models, let $S = \{M \mid M \text{ is a MWM for } F\}$, let M' be an arbitrary MWM for F if any exist, and define

$$\#2\text{SAT}_w(F, \mathbf{c}, \mathbf{w}) = \left(\sum_{M \in S} \mathcal{C}(M), \mathcal{W}(M') \right)$$

$\sum_{M \in S} \mathcal{C}(M)$ is referred to as the *weighted model count*, and $\mathcal{W}(M')$ as the maximum model weight of F, given vectors $\mathbf{c}$ and $\mathbf{w}$. If F is unsatisfiable, then the tuple is $(0, 0)$. #3SAT and #3SAT$_w$ are defined accordingly.

2.2 Graph-related concepts

A *graph* $G = (V, E)$ is an ordered pair consisting of a finite set V of *vertices* and a set E of unordered pairs (u, v) of distinct vertices, called *edges*. A set S of vertices is *independent* if $(u, v) \notin E$ for all $u, v \in S$. A maximum independent set, denoted MIS, is an independent set such that no other independent set is larger.

We define the *constraint graph* of a formula as the graph where the vertex set is the variables and the set of edges is

$$\{(a, b) \mid a \text{ and } b \text{ occur together in at least one clause}\}.$$

This concept was introduced by Bayardo and Pehoushek [4].

The *neighbourhood* of a vertex x in a graph G, denoted $N_G(x)$, we here define as the set of vertices having an edge in common with x *together with x itself*. The neighbourhood of a vertex set X is the union of the neighbourhoods of the vertices of X. The neighbourhood of a variable in a formula F is defined to be the corresponding neighbourhood in the constraint graph. The *size* of the neighbourhood of x, $S(x)$, we will measure as $S(x) = \sum_{y \in N(x)} d(y)$ (remember that x itself is included).

A *path* $x_0 \dots x_k$ is a set of variables such that each x_i has an edge to x_{i+1} in the constraint graph. We say that a formula F is *connected* iff in the corresponding constraint graph, there is a path from each variable to every other. Otherwise, the F consists of *connected components* and within each connected component this path-condition holds. The components can be found in polynomial time.

2.3 Algorithm analysis

Our two algorithms for $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$ are recursive decomposition algorithms based on the Davis–Putnam procedure [9]. That means that we *branch* on one or more variables in the formula F, *i.e.*, we assign values to the variable(s) such that the problem for F is reduced to the problem for two or more formulas with fewer variables.

For the analysis of the $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$ algorithms, a method by Kullmann will be used [13]. Consider the branching tree that the algorithm (implicitly) constructs when applied to a problem instance. If a node v in the tree has d branches, which are labelled with real, positive numbers $t_1, \dots, t_d$ (think of these labels as measures of the reduction of complexity in the respective branch), then the *branching tuple* for v is $(t_1, \dots, t_d)$ and the *branching number* is the positive real-valued solution of

$$\sum_{i=1}^{d} x^{-t_i} = 1$$

The branching number of a branching tuple B is denoted by $\tau(B)$. A branching tuple $(t_1,\ldots,t_d)$ is said to *dominate* another branching tuple $(u_1,\ldots,u_d)$ if $t_i \le u_i$ for all $1 \le i \le d$, ensuring that $\tau(t_1,\ldots,t_d) \ge \tau(u_1,\ldots,u_d)$.

In this paper, every label t_i for a branch from a formula F to a formula F' is $\Delta f(F) = f(F) - f(F')$ for some non-negative, algorithm specific measure of complexity $f(F)$. Defining $f_{\max}(n) = \max_{n(F)=n} f(F)$, this ensures a running time of $O(poly(n) \cdot \alpha^{f_{\max}(n)})$, where $poly(n)$ is a polynomial in n and α is the highest branching number of any branching tuple that can occur in the tree. For an exhaustive discussion of this, see [13].

In general, for any measure h, Δh (*e.g.*, Δn or Δm) is understood to mean $h(F) - h(F')$ in the context of a branch from some formula F to some formula F'. If there are more than one possible F', say F_1 and F_2, then $\Delta_i h = h(F) - h(F_i)$.

3 Propagation

We will here present a function *Prop* for propagation, *i.e.*, recursive applications of the rules

1. $(1 \vee q \vee r) \longrightarrow 1$;
2. $(0 \vee q \vee r) \longrightarrow (q \vee r)$;
3. $(q) \longrightarrow q = 1$; and
4. $(q \vee r \vee s) \wedge (q \vee r) \longrightarrow (q \vee r)$.

To motivate its use, let $F\{x = 1\}$ denote substitution of 1 for x and propagation to simplify F. Then consider this example: $F = (p \vee q \vee r)$ and note that $\#3\text{SAT}(F) = 7$. It would seem that $\#3\text{SAT}(F\{q = 1\}) + \#3\text{SAT}(F\{q = 0\}) = \#3\text{SAT}(F)$ but this is not the case. We see that the expression $(p \vee q \vee r)\{q = 1\}$ is simplified to the empty formula (which has 1 model by definition) and $\#3\text{SAT}(F\{q = 1\}) + \#3\text{SAT}(F\{q = 0\}) = 4$. The problem is that the variables p and r (which can be given arbitrary values) are *eliminated* in the simplification process.

To keep track of eliminated variables – the number they contribute to the number of solutions as well as their weights – we will need four structures:

1. We will use an integer variable $c \ge 0$ for keeping track of the contribution to the number of models arising from the eliminated variables.
2. A cardinality vector $\mathbf{c}$.
3. We will use an integer variable $w \ge 0$ for keeping track of the contribution to the weight of the models arising from the eliminated variables.

4. A weight vector $\mathbf{w}$.

Let $F[x = 0]$ be the result of assigning $x = 0$ and *not* propagating the changes. That means that a clause $(x \vee y)$ becomes $(0 \vee y)$, but is not further simplified to (y). $F[x = 1]$ is defined accordingly.

The function $Prop(F, \mathbf{c}, \mathbf{w})$ performs propagation and returns the updated formula F', the weight w of the variables removed and c for the eliminated variables. It also performs the obvious simplification $(a \vee b \vee c), (a \vee b) \rightarrow (a \vee b)$. The four steps of the algorithm are performed until not applicable, then the tuple (F, c, w) is returned. Initially, let $w = 0$ and $c = 1$.

1. If F contains an empty clause then $F := (\emptyset)$, $c := 0$ and $w := 0$
2. If there is a clause $(1 \vee \ldots)$, then it is removed. If any variable a thereby gets removed then there are three cases:
   (a) if $w(a) = w(\neg a)$ then $c := c \cdot (c(a) + c(\neg a))$; $w := w + w(a)$
   (b) if $w(a) < w(\neg a)$ then $c := c \cdot c(\neg a)$; $w := w + w(\neg a)$
   (c) if $w(a) > w(\neg a)$ then $c := c \cdot c(a)$; $w := w + w(a)$
3. If there is a clause $(0 \vee \ldots)$, then 0 is removed from it.
4. If there is a clause (a), then it is removed and $c := c \cdot c(a)$, $w := w + w(a)$. If a still appears in F then $F := F[a = 1]$.
5. If there are two clauses $x = (a \vee b \vee a'), y = (a \vee b)$ then remove x. If the variable a' thereby gets removed then handle it as in case 2.

Note: dual cases are omitted, *i.e.*, the case of a clause $(\neg a)$ is considered covered in step 4.

Lemma 1 Let $(F', c, w) = Prop(F, \mathbf{c}, \mathbf{w})$ and $(c', w') = \#3\text{SAT}_w(F', \mathbf{c}, \mathbf{w})$. Then $\#3\text{SAT}_w(F, \mathbf{c}, \mathbf{w}) = (c \cdot c', w + w')$.

Proof: We have three cases; the first two are trivial: First, if $F = F'$ then *Prop* has performed no changes and the lemma holds. Second: if $F' = (\emptyset)$ then *Prop* has found that F is inconsistent and the lemma holds. Third, if $F \neq F'$ then variables have been removed. Looking at each such variable a we can justifying its removal by looking at the cases of *Prop*:

1. Already proven correct;
2. When a is removed we have three cases:

(a) If the weights for both a's literals are the same, then the models having a true may have the same weight as the models having $\neg a$ true. Hence, we multiply c by $(c(a) + c(\neg a))$ and add the weight of a to w.

(b) If the weight of $w(\neg a)$ is the largest then the MWMs will only contain $\neg a$.

(c) If the weight of $w(a)$ is the largest then the MWMs will only contain a.

3. Clearly correct.

4. If a is removed we know only a can hold in the MWMs.

5. Justified as case 2

$\square$

This lemma of course extends to $\#2\text{SAT}_w$ as well.

4 Algorithm C for $\#2\text{SAT}$

In this section we present the algorithm C for $\#2\text{SAT}_w$, prove it correct and prove an upper time bound. This section is divided into four subsections. First we give some more preliminaries specific for C, then we deal with the case $d(F) \leq 3$, the case $d(F) \leq 5$ and the general case, respectively.

4.1 Further preliminaries

This subsection deals with an important concept for reducing the input formula.

Let us start with the concept of *multiplier reduction.* Suppose that F can be partitioned into two formulas F_1 and F_2 such that $|Var(F_1) \cap Var(F_2)| = 1$, say $Var(F_1) \cap Var(F_2) = \{v\}$, and that every clause in F belongs to either F_1 or F_2. Rather than working with the whole of F, we can calculate $\#2\text{SAT}_w(F, \mathbf{c}, \mathbf{w})$ as follows:

1. Let $(c_t, w_t) = \#2\text{SAT}_w(F_1[v = 1], \mathbf{c}, \mathbf{w})$ and $(c_f, w_f) = \#2\text{SAT}_w(F_1[v = 0], \mathbf{c}, \mathbf{w})$

2. Modify $\mathbf{c}$ and $\mathbf{w}$ so that $c(v) \leftarrow c_t \cdot c(v)$, $c(\neg v) \leftarrow c_f \cdot c(\neg v)$, $w(v) \leftarrow w_t + w(v)$ and $w(\neg v) \leftarrow w_f + w(\neg v)$

3. Finally, return $\#2\text{SAT}_w(F_2, \mathbf{c}, \mathbf{w})$.

This procedure is referred to as removing F_1 by multiplier reduction, and if it is possible to partition F into F_1 and F_2 in this way, with $n(F_1), n(F_2) > 1$, then we say that multiplier reduction applies.

By a *maximally reduced* formula F we mean a formula F where none of the reductions of the algorithm we are analysing apply. Multiplier reduction counts as a reduction for this purpose.

Lemma 2 Applying multiplier reduction does not change the return value of $\#2\mathrm{SAT}_w(F, \mathbf{c}, \mathbf{w})$.

Proof: Suppose that F is partitioned into F_1 and F_2 with v as the common variable, and that F_1 is removed by multiplier reduction.

Every model M for F, with an assignment $v = b$, consists of a model M_1 for F_1 and a model M_2 for F_2, with both M_1 and M_2 assigning $v = b$. In other words, M consists of a model M_2 for F_2, assigning $v = b$, and a model $M_{1,b}$ for $F_1[v = b]$. Conversely, every model M_2 for F_2, assigning $v = b$, can be combined with a model $M_{1,b}$ for $F_1[v = b]$ into a model M for F. As $F_1[v = b]$ and F_2 have disjoint variable sets, $\mathcal{C}(M) = \mathcal{C}(M_{1,b}) \cdot \mathcal{C}(M_2)$ and $\mathcal{W}(M) = \mathcal{W}(M_{1,b}) + \mathcal{W}(M_2)$. The maximum $\mathcal{W}(M)$ that can be achieved by extending some particular M_2 assigning $v = b$ is $\mathcal{W}(M_2) + w_b$, and the weighted model count for the models for M_1, b that achieve weight w_b is c_b, for a combined weighted model count for M of $\mathcal{C}(M_2) \cdot c_b$.

After the modifications to $\mathbf{c}$ and $\mathbf{w}$ have been made by multiplier reduction, $\mathcal{C}(M_2)$ and $\mathcal{W}(M_2)$ produce exactly these numbers for each model M_2 for F_2, which means that the final return value will be the same. $\square$

Now to the algorithm itself. The algorithm C is split into three functions depending on $d(F)$ of the input formula F. The main function C, given in Section 4.4, is used whenever $d(F) > 5$, and it has two helper functions: C_5, given in Section 4.3, which is used when $4 \leq d(F) \leq 5$, and C_3, given in Section 4.2, which is used when $d(F) \leq 3$.

Because of the bookkeeping involved in using the $\mathit{Prop}(F, \mathbf{c}, \mathbf{w})$ helper function and the $\mathbf{c}$ and $\mathbf{w}$ vectors, the actual process of branching on a variable or performing an assignment and making a single recursive call is somewhat lengthy, and will not be written explicitly in the algorithms. Instead we will use the phrase *(recursively) branch on v* as a shorthand for the following:

1. Let $(F_t, c_t, w_t) = \mathit{Prop}(F[v = 1], \mathbf{c}, \mathbf{w})$ and $(F_f, c_f, w_f) = \mathit{Prop}(F[v = 0], \mathbf{c}, \mathbf{w})$.
2. Let $(c'_t, w'_t) = C(F_t, \mathbf{c}, \mathbf{w})$ and $(c'_f, w'_f) = C(F_f, \mathbf{c}, \mathbf{w})$.
3. Let $W_{true} = w(v) + w_t + w'_t$, $W_{false} = w(\neg v) + w_f + w'_f$, $C_{true} = c(v) \cdot c_t \cdot c'_t$, and $C_{false} = c(\neg v) \cdot c_f \cdot c'_f$. There are three cases.
   (a) If $W_{true} = W_{false}$, return $(C_{true} + C_{false}, W_{true})$.
   (b) If $W_{true} > W_{false}$, return (C_{true}, W_{true}).

(c) Otherwise, we have $W_{true} < W_{false}$, and return (C_{false}, W_{false}).

We need a lemma for the correctness of this construct:

Lemma 3 The result of recursively branching on the variable v in the formula F equals $\#2\text{SAT}_w(F, \mathbf{c}, \mathbf{w})$.

Proof: By Lemma 1, $\#2\text{SAT}_w(F[v = 1], \mathbf{c}, \mathbf{w}) = (c_t \cdot c'_t, w_t + w'_t)$. Accounting for the contributions of $c(v)$ to the cardinality and $w(v)$ to the weight associated with setting $v = 1$, we clearly have weighted model count C_{true} and maximum model weight W_{true} for the models for F with $v = 1$. Similarly, we have weighted model count C_{false} and maximum model weight W_{false} for the models for F with $v = 0$. If $W_{true} = W_{false}$, then both the $v = 1$ and the $v = 0$ models are of maximum weight, and they must all be counted. Otherwise, either the $v = 1$ or the $v = 0$ models are all of a weight smaller than the maximum weight, and should not be counted. □

4.2 The function C_3

This subsection deals with the helper function C_3, given in Figure 1. We first give a correctness lemma for it and then prove an upper limit on its running time.

Lemma 4 $C_3(F, \mathbf{c}, \mathbf{w}) = \#2\text{SAT}_w(F, \mathbf{c}, \mathbf{w})$ assuming $d(F) \leq 3$.

Proof: We examine the cases of C_3:

1. Correct by assumption.
2. As every MWM of a component can be combined with every MWM of the other components this case is correct.
3. Correct by Lemma 2.
4. Correct by Lemma 3

□

Lemma 5 $C_3(F, \mathbf{c}, \mathbf{w})$ runs in $O(p(n) \cdot \tau(4, 4)^{n_3(F)}) \approx O(p(n) \cdot 1.1892^{n_3(F)})$ time, where $p(n)$ is a polynomial in n.

Proof: We will derive the result by using the number of variables of degree 3 in a fomula F, $n_3(F)$, as a measure. Due to the observation that if F is maximally reduced then $n_3(F) = m(F) - 2n(F)$, we can prove that $\Delta n_3 \geq 4$ along any branch by proving that $\Delta m \geq \Delta n + 4$.

Algorithm $C_3(F, \mathbf{c}, \mathbf{w})$
Case 1: If F contains no clauses, return $(1,0)$. If F contains an empty clause, return $(0,0)$.
Case 2: If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = C(F_i, \mathbf{c}, \mathbf{w})$ for the connected components $F_0, \ldots, F_j$.
Case 3: If multiplier reduction applies, apply it, removing the part with lowest $n_3(F)$ value.
Case 4: If $d(F) = 3$, pick a variable x, $d(x) = 3$, with as many neighbours of degree 3 as possible, and recursively branch on it. Otherwise, recursively branch on any variable.

Figure 1: Help function for computing $\#2\text{SAT}_w$ when $d(F) = 3$

Let F be a maximally reduced formula with $d(F) = 3$. If we branch on some variable v in F, eventually resulting in two maximally reduced formulas F_1 and F_2, the reduction in $n_3(F)$ is $n_3(F) - n_3(F_1) = m(F) - m(F_1) - 2(n(F) - n(F_1))$ for F_1 and analogously for F_2. Also note that if $n_3(F) = 0$ then F can be solved in time polynomial in n. (It is of course possible that the branching results in more than two maximally reduced formulas, if case 2 or case 3 applies, but that is a strictly easier case.)

Let V be the variables of F and V_1 the variables of F_1. A clause involving two variables u and v exists in F_1 iff the same clause exists in F and $u, v \in V_1$. We also have $V_1 \subseteq V$. Let $V' = V - V_1$. The reduction Δn in n from F to F_1 is $|V'|$, and the reduction Δm in m is

$$\sum_{v \in V'} d(v) + |\{\text{ clauses } C \text{ in } F \mid C \text{ involves variables from both } V' \text{ and } V_1\}|$$

Since $d(x) = 3$ and since there are no singletons in F, the first term is at least $2\Delta n + 1$, and since multiplier reduction does not apply, the second term is at least 2. Taking it together, and using that $m(F)$ is twice the number of edges in F and therefore even, we have $\Delta m \geq 2\Delta n + 4$, so $n_3(F) - n_3(F_1) \geq 4$. The same argument applies to F_2.

If F is not maximally reduced, simply note that $n_3(F)$ does not increase with any of the reductions. Finally, if $n_3(F) = 0$ but F is maximally recuced, then F must be 2-regular and the entire formula is taken care of in polynomial time by *Prop* and reductions.

Therefore, the algorithm $C_3(F, \mathbf{c}, \mathbf{w})$ produces $O(\tau(4,4)^{n_3(F)})$ leaves with $O(p(n))$ work in each leaf, which results in total work in $O(p(n) \cdot \tau(4,4)^{n_3(F)})$. □

4.3 The function C_5

This subsection treats the function C_5, given in Figure 2.

Algorithm $C_5(F, \mathbf{c}, \mathbf{w})$
Case 1: If F contains no clauses, return $(1, 0)$. If F contains an empty clause, return $(0, 0)$.
Case 2: If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = C(F_i, \mathbf{c}, \mathbf{w})$ for the connected components $F_0, \ldots, F_j$.
Case 3: If $d(F) < 4$, return $C_3(F, \mathbf{c}, \mathbf{w})$.
Case 4: If multiplier reduction applies, apply it, removing the part with lowest $f(F)$ value.
Case 5: Pick a variable x of maximum degree such that $S(x)$ is maximized.
(a) If $N(x)$ is connected to the rest of the graph through only 2 external vertices y, z such that $d(y) \geq d(z)$, then branch on y.
(b) Otherwise, branch on x.

Figure 2: Help function for computing $\#2\text{SAT}_w$ when $4 \leq d(F) \leq 5$. The function f is defined in the text; recall that $S(x)$, measuring the size of a neighbourhood $N(x)$, is defined by $S(x) = \sum_{y \in N(x)} d(y)$

The correctness of $C_5(F, \mathbf{c}, \mathbf{w})$ comes from an argument very similar to that concerning the correctness of $C_3(F, \mathbf{c}, \mathbf{w})$.

For the analysis of C_5, we use a piecewise linear function $f(n, m)$ as a measure of complexity, with a behaviour determined by the quotient m/n where $n = n(F)$ and $m = m(F)$. As a motivation, consider again the branching tree that the algorithm implicitly constructs when applied to a problem instance. As the calculation progresses down a path in the tree, our choice of branching variable guarantees that in every worst case situation, the quotient m/n will decrease. As we will see, there is a worst-case branching associated with each value of this quotient, and with a decreasing quotient smaller pieces of the formula are removed in each worst-case branching. If we use a classical measure of complexity, such as $n(F)$, then this means that the worst-case branching numbers are generally smaller near the top of the tree, with the highest branching numbers occurring further down. However, the estimation of the tree size (and thus the running time of the algorithm) of $O(\alpha^{f_{\max}(n)})$ is based on the assumption that the branching number is basically the same for every node of the tree, so the estimation of the running time is unnecessarily high. By using a measure of complexity that changes its behaviour with the m/n quotient, we can make sure that the worst-case branching number is equal for every value of the m/n quotient. This means that we can incorporate the effects of the decreasing m/n quotient in our upper time bound, getting closer to the actual worst-case running time.

In our analysis, we will find a sequence of worst cases as the m/n quotient increases, and with each worst case we associate a linear function $f_i(n, m) = a_i n + b_i m$, a lower limit k_i for the m/n quotient below which worse cases

i	k_i	χ_i	Running time
0	0	0	$O(1)$
1	2	0	$O(poly(n))$
2	3	1	$O(1.1892^n)$
3	3.5	1.1340	$O(1.2172^n)$
4	3.75	1.1914	$O(1.2294^n)$
5	4	1.2410	$O(1.2400^n)$
6	4+4/29	1.2536	$O(1.2427^n)$
7	4+4/9	1.2788	$O(1.2481^n)$
8	4+4/7	1.2881	$O(1.2501^n)$
9	4.8	1.3033	$O(1.2534^n)$
10	5	1.3154	$O(1.2561^n)$

Table 1: k_i, χ_i and running times

appear, and an upper limit k_{i+1} for the m/n quotient above which the case does not appear. For instance, if $m/n > 4$, then $d(x) > 4$ for the chosen variable x. Each function $f_i(n,m)$ has its parameters chosen so that the worst-case branching number in the range $k_i < m/n \leq k_{i+1}$ will be equal to $\tau(4,4)$ for every i. The range k_i–k_{i+1} is referred to as *section* i, for each i, and $f(n,m)$ is partitioned into functions $f_i(n,m)$ in the same way as the m/n quotient axis is partitioned into sections.

To simplify the presentation of the proof, we give the definitions of $f(n,m)$ and related terms here, then proceed to state some of its properties. These properties will be used in the analysis, and can be verified numerically.

Here follow the definitions of the terms involved.

$$f(n,m) = f_i(n,m) \text{ if } k_i < m/n \leq k_{i+1},\ 0 \leq i \leq 9 \tag{1}$$

$$f_i(n,m) = \chi_i n + (m - k_i n)b_i,\ 0 \leq i \leq 9 \tag{2}$$

$$\chi_0 = 0 \tag{3}$$

$$\chi_i = \chi_{i-1} + (k_i - k_{i-1})b_{i-1},\ 1 \leq i \leq 10 \tag{4}$$

$$a_i = \chi_i - k_i b_i \tag{5}$$

The exact values of k_i can be found in Table 1, along with rounded-off values for χ_i and $\tau(4,4)^{\chi_i}$, the latter being c in the $O(p(n) \cdot c^n)$ upper limit on the running time for a formula F with $m(F)/n(F) \leq k_i$.

The expressions defining the values of b_i can be found in Table 2, along with rounded-off numerical values for b_i and a_i. The expressions come from the branching numbers for the worst case in each section i.

(2) guarantees that $f(n,m)$ is continuous, as $f_{i-1}(n, k_i n) = f_i(n, k_i n) = \chi_i n$ for all i.

The properties of $f(n,m)$ that will be used in the analysis follow next.

i	b_i, definitions	b_i	a_i
0	0	0	0
1	1	1	-2
2	$\tau(1+5b_2, 5+5b_2) = \tau(4,4)$	0.2680	0.1961
3	$\tau(\chi_3+4.5b_3, 5\chi_3+4.5b_3) = \tau(4,4)$	0.2295	0.3308
4	$\tau(\chi_4+4.25b_4, 5\chi_4+5.25b_4) = \tau(4,4)$	0.1987	0.4461
5	$\tau(\chi_5+6b_5, 6\chi_5+2b_5) = \tau(4,4)$	0.0914	0.8755
6	$\tau(\chi_6+(5+25/29)b_6, 6\chi_6+(3+5/29)b_6) = \tau(4,4)$	0.0821	0.9139
7	$\tau(\chi_7+(5+5/9)b_7, 6\chi_7+(3+1/3)b_7) = \tau(4,4)$	0.0736	0.9517
8	$\tau(\chi_8+(5+3/7)b_8, 6\chi_8+(4+4/7)b_8) = \tau(4,4)$	0.0665	0.9841
9	$\tau(\chi_9+5.2b_9, 6\chi_9+5.2b_9) = \tau(4,4)$	0.0602	1.0143

Table 2: b_i and a_i parameters

1. $f(n,m) > f(n-1,m)$ if $m > 3n$
2. $f(n,m) > f(n,m-1)$ if $m > 2n$
3. $f(n,m) - f(n_1,m_1) \geq f_i(n,m) - f_i(n_1,m_1)$ if $k_i n \leq m \leq k_{i+1}n$ and $m_1/n_1 < m/n$.
4. $f(n,m) \geq f(n_1,m_1) + f(n-n_1, m-m_1)$ if $0 \leq n_1 \leq n$ and $0 \leq m_1 \leq m$.

These will be referred to in the analysis as Property 1–4. Properties 1 and 2 follow from Table 2, but Property 3 requires some investigation.

Note that we now have two equivalent ways of expressing $f_i(n,m)$: $\chi_i n + (m - k_i n)b_i$ and $a_i n + b_i m$.

We will focus on the transition of a single barrier k_i, *i.e.*, that m/n belongs in section i while m_1/n_1 belongs in section $i-1$. If the property holds for all such barriers, then it holds globally.

Assume that $k_i n \leq m \leq k_{i+1}n$, $k_{i-1}n_1 \leq m_1 \leq k_i n_1$, $n_1 < n$ and $m_1 < m$. We want to check that $f_i(n_1,m_1) \geq f_{i-1}(n_1,m_1)$, *i.e.* $f_i(n_1,m_1) - f_{i-1}(n_1,m_1) = (a_i - a_{i-1})n_1 + (b_i - b_{i-1})m_1 \geq 0$. Through standard formula manipulation, the definitions of a_i and χ_i give us that $a_i - a_{i-1} = k_i(b_{i-1} - b_i)$. Inserting this into the previous inequality, we get $(b_{i-1} - b_i)(k_i n_1 - m_1) \geq 0$, and as $k_i n_1 > m_1$ by assumption, we see that Property 3 follows from the observation that b_i is decreasing with increasing i.

Regarding Property 4, let a, c be constants so that $n_1 = an$ and $m_1 = am + c$. Let $k_i \leq k = m/n \leq k_{i+1}$, $\chi = f(n,m)/n$ and assume w.l.o.g. that $c \geq 0$. We have

$$\begin{aligned} f(\alpha n, k\alpha n) &= \chi \alpha n \text{ for any } \alpha && (6) \\ f(an, kan + c) &= \chi an + cb_{up} \text{ for some } b_{up} && (7) \\ f((1-a)n, k(1-a)n - c) &= \chi(1-a)n - cb_{down} \text{ for some } b_{down} && (8) \end{aligned}$$

The part $b_{up}c$ is shorthand for some summation $c_1 b_i + c_2 b_{i+1} + \ldots$ with $c_1 + c_2 + \ldots = c$ so that $b_{up}c \leq b_i c$, and similarly $b_{down}c \geq b_i c$. Thus,

$$
\begin{aligned}
f(n_1, m_1) + f(n - n_1, m - m_1) &= & (9)\\
\chi n + (b_{up} - b_{down})c &\leq & (10)\\
\chi n &= f(n, m) & (11)
\end{aligned}
$$

for some b_{up}, b_{down}, as b_i is decreasing.

The bottom-most non-zero function $f_1(n, m)$ corresponds to the algorithm C_3, and is applicable for m/n values from $k_1 = 2$ to $k_2 = 3$, as this is the range of m/n where C_3 is the worst case, as we shall see. The rest of the functions $f_i(n, m)$ correspond to worst cases for C_5. $f(n, m)$ is undefined for $m > 5n$, but as a curiosity one can note that if a function $f_{10}(n, m)$ is introduced with $b_{10} = 0$ and $a_{10} = \chi_{10}$, then the worst-case branching for the case $d(x) > 5$ (which we will find in the next section) gets a branching number of $\tau(\chi_{10}, 7\chi_{10}) < \tau(4, 4)$.

We need a lemma that allows us to make a connection between the value of $m(F)/n(F)$ and worst-case branchings.

Lemma 6 Let F be a non-empty formula such that $m(F)/n(F) = k$, and define $\alpha(x)$ and $\beta(x)$ such that

$$
\begin{aligned}
\alpha(x) &= d(x) + |\{y \in N(x) \mid d(y) < k\}| & (12)\\
\beta(x) &= 1 + \sum_{\{y \in N(x) \mid d(y) < k\}} 1/d(y) & (13)
\end{aligned}
$$

There exists some variable $x \in Var(F)$ with $d(x) \geq k$ such that $\alpha(x)/\beta(x) \geq k$.

Proof: Consider the following sums:

$$
\begin{aligned}
A &= \sum_{\{x \in Var(F) \mid d(x) \geq k\}} \alpha(x) & (14)\\
B &= \sum_{\{x \in Var(F) \mid d(x) \geq k\}} \beta(x) & (15)
\end{aligned}
$$

We may view every variable x with $d(x) \geq k$ as contributing exactly $d(x)$ to A and 1 to B, and each variable y with $d(y) < k$, i to A and $i/d(y)$ to B, for some integer $i \leq d(y)$ (which can be viewed as contributing a fraction $i/d(y)$ of the full contributions of a variable of degree $d(y)$). We find that there are numbers $n'_i(F)$ with $n'_i(F) \leq n_i$ for $i < k$ and $n'_i(F) = n_i(F)$ for $i \geq k$ such that the following holds:

$$
\begin{aligned}
A &= \sum_i i n'_i(F) = m(F) - \sum_{i<k} i(n_i(F) - n'_i(F)) & (16)\\
B &= \sum_i n'_i(F) = n(F) - \sum_{i<k} (n_i(F) - n'_i(F)) & (17)
\end{aligned}
$$

Here, we used $\sum_i i \cdot n_i(F) = m(F)$ and $\sum_i n_i(F) = n(F)$. As $m(F) = k \cdot n(F)$, we have:

$$A \geq k \cdot B \tag{18}$$

The set $\{x \in Var(F) \mid d(x) \geq k\}$ is clearly not empty. Hence, if we had $\alpha(x) < k\beta(x)$ for all x with $d(x) \geq k$, inequality (18) could not hold. Therefore there is an x with $d(x) \geq k$ such that $\alpha(x) \geq k\beta(x)$. □

Now, we can proceed by proving an upper bound for the running time of C_5. The proof will be divided into lemmas according to the value of $m(F)/n(F)$. We will need to prove that the worst-case branching number in each section is $\tau(4, 4)$.

As a final thing before the proof, a note on case **5a** of the algorithm.

If case **5a** of the algorithm is applied, we remove y by assignment and $N(x)$ by multiplier reduction. In total, we reduce n by at least $d(x) + 2$ and m by at least $S(x) + 4$ in both branches. We will see that when using case **5b**, the largest worst-case reduction is $\Delta n = d(x) + 1$ and $\Delta m = S(x) + 4$. When $m > 3n$, so that Property 1 is in effect, this is clearly a harder case than case **5a**.

Lemma 7 For a maximally reduced formula F with $m \leq 2n$, $C_5(F, \mathbf{c}, \mathbf{w})$ runs in time polynomial in n.

Proof: The existence of a singleton variable x in F implies one of two things: $n(F) = 2$ or multiplier reduction applies. The former is an uninteresting case, so we assume that $n(F) > 2$ and consequently $d(x) = 2$ for every $x \in Vars(F)$, *i.e.*, the constraint graph of F is a cycle.

Assume that F' is a maximally reduced formula which is reached somewhere in the branching tree starting with $C_5(F, \mathbf{c}, \mathbf{w})$, other than the root. Again, since F' is maximally reduced, either $n(F') \leq 2$ or the constraint graph of F' forms another, smaller cycle. However, since our algorithm never adds or modifies clauses, if F' forms a cycle then this cycle must be contained in the larger cycle of the constraint graph of F, which is impossible. Instead, we have $n(F') \leq 2$.

Finally, each reduction used in our algorithms either removes some variable directly or splits the formula into two formulas with disjoint variable sets. We see that F, after a polynomial amount of work, is reduced to at most n formulas of a constant size, which gives us a polynomial amount of total work. □

Lemma 8 For a maximally reduced formula F with $m \leq 3n$, $C_5(F, \mathbf{c}, \mathbf{w})$ runs in time $O(\tau(4, 4)^n)$.

Proof: In this lemma, we will use $f_1(n, m)$ as a measure, with $b_1 = 1$ and $a_1 = -2$. For the special case that $d(F) \leq 3$, this measure is identical to the one used in the analysis of $C_3(F, \mathbf{c}, \mathbf{w})$.

For a branching from a maximally reduced F to maximally reduced F_1 and F_2, f is reduced by at least 4. By the argument from Lemma 5 in Section 4.2, we have $\Delta m \geq 2\Delta n + 4$ for both case 5a and case 5b of the algorithm. If F_1 belongs to section 1 (*i.e.*, $m(F_1) \leq 3n(F_1)$), this argument is enough to prove the $\tau(4,4)$ limit.

Otherwise, if $3n(F_1) \leq m(F_1) \leq 3.5n(F_1)$, assume that $m(F) = 3n(F) - a$ and $m(F_1) = 3n(F_1) + b \leq 3.5n(F_1)$, so that $f(F) = n - a$ and $f(F_1) = \chi_2 n(F_1) + b \cdot b_2 = n - \Delta n + b \cdot b_2$. We have

$$\Delta m = 3\Delta n - a - b \geq 2\Delta n + 4$$

and so

$$f(F) - f(F_1) = \Delta n - a - b \cdot b_2 > \Delta n - a - b \geq 4.$$

Finally, if $m(F_1) > 3.5n(F_1)$, let $m = m(F), n = n(F), \Delta m = m(F) - m(F_1)$ and $\Delta n = n(F) - n(F_1)$. Assume that $m/n = a$ and $(m - \Delta m)/(n - \Delta n) = b > 3.5$. We have $m - \Delta m = an - \Delta m = (n - \Delta n)b$ and $\Delta m \geq 2\Delta n$, *i.e.*,

$$\Delta m = an - (n - \Delta n)b = \Delta n b - (b - a)n > 2\Delta n, \text{ so } n - \Delta n < (a-2)n/(b-2).$$

Thus

$$f(F_1) < 1.4(a-2)n/(b-2) < (1.4/1.5)(a-2)n$$

and

$$f(F) - f(F_1) > (0.1/1.5)n(a-2).$$

For a sufficiently large n, this is clearly greater than 4.

If any of the reductions in cases 2 and 4 have been applied, then the above applies to every maximally reduced F_i, and by Property 4 the total amount of work is smaller. This argument applies in Lemmas 9 and 10 as well.

If case 3 is used, then by Lemma 5 the running time is $O(p(n) \cdot \tau(4,4)^{f(F)})$.

In all cases, $f(F) - f(F_1) \geq 4$ for sufficiently large n. The same goes analogously for $f(F) - f(F_2)$. As $f(n, m) \leq n$ when $m \leq 3n$, the running time of this case is in $O(\tau(4,4)^n)$. □

Lemma 9 For a maximally reduced formula F with $m \leq 4n$, $C_5(F, \mathbf{c}, \mathbf{w})$ runs in time $O(1.2400^n)$.

Proof: This lemma uses $f_2(n, m)$ through $f_4(n, m)$ as measures, with parameters as previously defined.

First, we consider the case that $d(F) = 5$. In this case, there are no guarantees concerning $S(x)$, so any case might occur. There are three worst cases, each of which gets a branching number less than $\tau(4,4)$.

1. $\Delta_1 n = \Delta_2 n = 6$, $\Delta_1 m = \Delta_2 m = 20$, resulting in a branching dominated by $\tau(4,4)$ for all $f_i(n,m)$, even if $m - \Delta m > 4(n - \Delta n)$.

2. $\Delta_1 n = 2$, $\Delta_1 m = 12$, $\Delta_2 n = 6$, $\Delta_2 m = 22$, resulting in a branching dominated by $\tau(3,6) < \tau(4,4)$ for all $f_i(n,m)$, even if $m - \Delta_2 m > 4(n - \Delta_2 n)$.

3. $\Delta_1 n = 1$, $\Delta_1 m = 10$, $\Delta_2 n = 6$, $\Delta_2 m = 24$. This branching has $\Delta_i m/\Delta_i n \geq 4$ for $i = 1, 2$, meaning that $m(F_i)/n(F_i) \leq 4$ for $i = 1, 2$, and for all f_i up to $k = 4$, this case has a branching number less than $\tau(4,4)$.

Next, the actual worst cases, beginning with section 2.

Section 2: $\boldsymbol{m/n \in [3, 3.5]}$, $\boldsymbol{d(F) = 4}$ Lemma 6 guarantees that there will be some variable x with $\alpha(x)/\beta(x) > 3$, and the minimum $S(x)$ for variables with this property is 15, occurring if the degrees of the neighbours are 2, 3, 3, and 3. Note that this only tells us that $S(x) \geq 15$; we have no guarantees that $\alpha(x)/\beta(x) > 3$ for the chosen variable x.

By Properties 1 and 2, while looking for a worst case for $f_i(n,m)$ when $i > 1$ we may assume that $Prop$ removes a minimum number of variables. Under this assumption, we find that the minimum value of $\Delta m/\Delta n$ is 3.5, occurring when an assignment removes neighbours with degrees 2, 3, and 3, or 2, 2, and 3, so we can assume that $(m - \Delta m)/(n - \Delta n) \leq 3.5$ for every worst case recursion.

There are two candidates for worst case recursion to examine in this case.

1. With $S(x) = 15$, at least one neighbour has degree 2. The worst variant of this case is when the neighbours have degrees 2, 3, 3, and 3 and the recursion is maximally unbalanced, with a branching number of $\tau(2a_2 + 10b_2, 5a_2 + 18b_2) = \tau(2 + 4b_2, 5 + 3b_2) < \tau(4,4)$.

2. The worst possible case without neighbours of degree 2 is when every neighbour has degree 3, with $S(x) = 16$, and when the branching is maximally unbalanced, with branching number $\tau(a_2 + 8b_2, 5a_2 + 20b_2) = \tau(1 + 5b_2, 5 + 5b_2) = \tau(4,4)$ by definition of b_2. This case also occurs with $S(x) = 17$, with neighbours of degrees 3, 3, 3, and 4.

We find that the second case is harder than the first, and that both are at most $\tau(4,4)$. As given in the tables, we have $b_2 \approx 0.2680$, $a_2 \approx 0.1961$, and $\chi_3 = 1 + 0.5b_2 \approx 1.1340$.

Cases with $S(x) = 17$ can occur when $m(F)/n(F) \leq 3.5$, so the next section begins when $m(F)/n(F) > 3.5$.

Section 3: $\boldsymbol{m/n \in [3.5, 3.75]}$, $\boldsymbol{d(F) = 4}$ The minimum value for $S(x)$ for variables with $\alpha(x)/\beta(x) > 3.5$ is 18, and $S(x) = 18$ and $S(x) = 19$ both provide worst-case branchings with branching number $\tau(a_3 + 8b_3, 5a_3 + 22b_3) = \tau(\chi_3 + 4.5b_3, 5\chi_3 + 4.5b_3) = \tau(4,4)$, by definition of b_3. We have

$b_3 \approx 0.2295$, $a_3 \approx 0.3308$ and $\chi_4 = \chi_3 + 0.25b_3 \approx 1.1914$. We also note that $\Delta m/\Delta n \geq 4$ for any worst case branching when $S(x) > 16$.

Cases with $S(x) = 19$ can occur up to $m(F)/n(F) = 3.75$, so the next section begins at 3.75.

Section 4: $m/n \in [3.75, 4]$, $d(F) = 4$ The only possible value for $S(x)$ is 20, resulting in a worst case branching number of $\tau(a_4 + 8b_4, 5a_4 + 24b_4) = \tau(\chi_4 + 4.25b_4, 5\chi_4 + 5.25b_4) = \tau(4,4)$ by definition of b_4. We have $b_4 \approx 0.1987$, $a_4 \approx 0.4461$ and $\chi_5 = \chi_4 + 0.25b_4 \approx 1.2410$.

As we see, the worst-case branching number is $\tau(4,4)$ for sections 2, 3 and 4. With $m \leq 4n$, we have $f(n,m) \leq \chi_5 n$ and a worst-case running time for $C_5(F, \mathbf{c}, \mathbf{w})$ of $O(\tau(4,4)^{\chi_5 n}) \subset O(1.2400^n)$. $\square$

Lemma 10 For a maximally reduced formula F with $m \leq 5n$, $C_5(F, \mathbf{c}, \mathbf{w})$ runs in time $O(1.2561^n)$.

Proof: This lemma uses $f_5(n,m)$ through $f_9(n,m)$ as measures, with parameters as previously defined.

We know that $d(F) = 5$, so we proceed immediately with section 5. We also know that x is monotone, so the maximally unbalanced cases are the only possible cases. This makes checking that $\Delta m/\Delta n \geq k_i$ trivial.

Section 5: $m/n \in [4, 4 + 4/29]$ The minimum value for $S(x)$ for variables with $\alpha(x)/\beta(x) > 4$ is 23, with a worst-case branching with branching number $\tau(a_5 + 10b_5, 6a_5 + 26b_5) = \tau(\chi_5 + 6b_5, 6\chi_5 + 2b_5) = \tau(4,4)$ by definition of b_5, and $\Delta_2 m/\Delta_2 n = 4 + 1/3 > 4 + 4/29$. We have $b_5 \approx 0.0914$, $a_5 \approx 0.8755$ and $\chi_6 = \chi_5 + (4/29)b_5 \approx 1.2536$.

$S(x) = 23$ can occur up to $m(F)/n(F) = 4 + 4/29 \approx 4.1379$, so the next section begins at that point.

Section 6: $m/n \in [4+4/29, 4+4/9]$ The minimum value for $S(x)$ for variables with $\alpha(x)/\beta(x) > 4+4/29$ is 24, and $S(x) = 24$ and $S(x) = 25$ both have worst-case branchings with a branching number of $\tau(a_6 + 10b_6, 6a_6 + 28b_6) = \tau(\chi_6 + (5 + 25/29)b_6, 6\chi_6 + (3 + 5/29)b_6) = \tau(4,4)$ by definition of b_6, and $\Delta_2 m/\Delta_2 n = 4 + 2/3 > 4 + 4/9$. We have $b_6 \approx 0.0821$, $a_6 \approx 0.9139$ and $\chi_7 = \chi_6 + (4/9 - 4/29)b_6 \approx 1.2788$.

$S(x) = 25$ can occur up to $m(F)/n(F) = 4 + 4/9 \approx 4.4444$, so the next section begins at that point.

Section 7: $m/n \in [4+4/9, 4+4/7]$ The minimum value for $S(x)$ for variables with $\alpha(x)/\beta(x) > 4+4/9$ is 26, and $S(x) = 26$ and $S(x) = 27$ both have worst-case branchings with a branching number of $\tau(a_7 + 10b_7, 6a_7 + 30b_7) = \tau(\chi_7 + (5+5/9)b_7, 6\chi_7 + (3+1/3)b_7) - = \tau(4,4)$ by definition of b_7, and $\Delta_2 m/\Delta_2 n = 5$, which means that $\Delta m/\Delta n \geq 5$ in all remaining sections. We have $b_7 \approx 0.0736$, $a_7 \approx 0.9517$ and $\chi_8 = \chi_7 + (4/7 - 4/9)b_7 \approx 1.2881$.

$S(x) = 27$ can occur up to $m(F)/n(F) = 4 + 4/7 \approx 4.5714$, so the next section begins at that point.

Algorithm $C(F, \mathbf{c}, \mathbf{w})$
Case 1: If F contains no clauses, return $(1, 0)$. If F contains an empty clause, return $(0, 0)$.
Case 2: If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = C(F_i, \mathbf{c}, \mathbf{w})$ for the connected components $F_0, \ldots, F_j$.
Case 3: If there exists a non-monotone variable x with $d(x) \geq 5$, branch on x.
Case 4: If $d(F) < 6$, return $C_5(F, \mathbf{c}, \mathbf{w})$.
Case 5: Pick a variable x of maximum degree and branch on it.

Figure 3: Main function for computing #2SAT

Section 8: $\boldsymbol{m/n \in [4 + 4/7, 4.8]}$ The minimum value for $S(x)$ for variables with $\alpha(x)/\beta(x) > 4+4/7$ is 28, and $S(x) = 28$ and $S(x) = 29$ both have worst-case branchings with a branching number of $\tau(a_8 + 10b_8, 6a_8 + 32b_8) = \tau(\chi_8 + (5 + 3/7)b_8, 6\chi_8 + (4 + 4/7)b_8) = \tau(4, 4)$ by definition of b_8. We have $b_8 \approx 0.0665$, $a_8 \approx 0.9841$ and $\chi_9 = \chi_8 + (0.8 - 4/7)b_8 \approx 1.3033$.

$S(x) = 29$ can occur up to $m(F)/n(F) = 4.8$, so the last section begins at that point.

Section 9: $\boldsymbol{m/n \in [4.8, 5]}$ The only possible value for $S(x)$ is 30, with a worst case branching number of $\tau(a_9 + 10b_9, 6a_9 + 34b_9) = \tau(\chi_9 + 5.2b_9, 6\chi_9 + 5.2b_9) = \tau(4, 4)$ by definition of b_9. We have $b_9 \approx 0.0602$, $a_9 \approx 1.0143$ and $\chi_{10} = \chi_9 + 0.2b_9 \approx 1.3154$.

The total running time of the algorithm is $O(\tau(4, 4)^{\chi_{10} n}) \subset O(1.2561^n)$, which is the bound we wanted to prove. $\square$

4.4 The main function C

Now, we can finally give the last part of our algorithm: The function C, applicable to a general formula F, is given in Figure 3.

Theorem 11 $C(F, \mathbf{c}, \mathbf{w})$ runs in $O(1.2561^n)$ time.

Proof: Cases 1 and 2 are reductions, or take only constant time. For case 3, the worst case is $T(n) = T(n - 2) + T(n - 5)$ with solutions in $O(\tau(2, 5)^n) \subset O(1.2366^n)$. Case 4 takes $O(1.2561^n)$ time, by Lemma 10. For case 5, the worst case is $T(n) = T(n - 1) + T(n - 7)$ with solutions in $O(\tau(1, 7)^n) \subset O(1.2555^n)$. All these cases are contained in $O(1.2561^n)$. $\square$

5 Algorithm D for #3SAT

In this section we will present the algorithm D for #3SAT$_w$ and provide an upper bound on its running time. The complexity analysis is somewhat

Algorithm $D(F, \mathbf{c}, \mathbf{w})$
Case 1: If $n(F) < 10$, return $D_E(F, \mathbf{c}, \mathbf{w})$
Case 2: If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = D(F_i, \mathbf{c}, \mathbf{w})$ for the connected components $F_0, \ldots, F_j$.
Case 3: If multiplier reduction applies, apply it, removing the part with lowest $n(F)$ value.
Case 4: If there exists a variable v such that $d(v) = d_3(v) = 1$, let a be a neighbour of maximum degree and recursively branch on a
Case 5: If there exists a variable v such that $d(v) = 2$ and $d_2(v) > 0$, let a be a neighbour that shares a 3-clause with v, if possible, or else a neighbour of maximum $d_2(a)$, and recursively branch on a
Case 6: If there exists at least one 2-clause in F, let v be a variable with maximum $d(v)$ among all variables with maximum $d_2(v)$, and recursively branch on v
Case 7: If there exists a variable v such that $d(v) = d_3(v) = 2$ then, assuming that one 3-clause containing v is $(v \vee a \vee b)$, recursively branch on $b = 1$, $b = 0 \wedge a = 1$ and $b = 0 \wedge a = 0 \wedge v = 1$
Similarly for other 3-clauses containing v or $\neg v$.
Case 8: Pick a variable v of maximum degree and recursively branch on it.

Figure 4: Function for computing $\#3\text{SAT}_w$

delicate and requires numerical calculations to obtain a solution to an optimization problem as we shall see.

There is a helper function $D_E(F, \mathbf{c}, \mathbf{w})$ that will use exhaustive search to calculate $\#3\text{SAT}_w(F, \mathbf{c}, \mathbf{w})$. We will only apply it to instances of constant size, and thus we consider its running time to be in $O(1)$.

The algorithm can be seen in Figure 5. When starting, assume that F cannot be further simplified by $Prop$.

To establish the correctness of D one needs only to apply some minor modifications to the correctness proof of C. We therefore move on to the time complexity analysis.

In the time complexity analysis, we will measure the formula complexity using the function $f(F) = n - \Psi(k)$, where k is the number of 2-clauses in F. $\Psi(k)$ is a real-valued function such that $0 = \Psi(0) < \Psi(1) < \ldots < \Psi(4) < 1$, $\Psi(k) = \Psi(4)$ for all $k > 4$, and $\Psi(k+1) - \Psi(k) \geq \Psi(k+2) - \Psi(k+1)$ for all k, where $\Psi(1)$ through $\Psi(4)$ will be given exact values later in this section, using numerical optimization. In other words, we use a more fine-grained measure than just $n(F)$, with four sub-divisions between n and $n + 1$. Note that since $\Psi(k) < 1$ we have $n - 1 < f(F) \leq n$ and $f(F) \leq 0$ only if $n = 0$, and since $k = 0$ whenever $n = 0$ we have $f(F) \geq 0$, as required by our method of analysis. Note also that $f_{\max}(n) = n$.

The intuitive reason for using this particular $f(F)$ is that if the worst-

k	$\Psi(k)$	$\Psi(k) - \Psi(k-1)$
1	0.24478	0.24478
2	0.45956	0.21478
3	0.62457	0.16501
4	0.76707	0.14250

Table 3: $\Psi(k)$ values

case running time of the algorithm is modelled using a two-variable recursion $T(n,k)$, then the values of $T(n,k)$ settle into a pattern similar to $c^{n-\Psi(k)}$ for some function $\Psi(k)$ and constant c. It is important to note, however, that the correctness of the bounds given in this section does not rely upon either the optimality of $\Psi(k)$ or any similarities between $f(F)$, $T(n,k)$ and real-world worst-case running times. For any $\Psi(k)$ obeying the given rules you get a worst-case branching number c such that the running time of $D(F)$ is $O(c^n)$, but if you use a non-optimal $\Psi(k)$ then the constant c will be higher than necessary.

To simplify the presentation of the proof, the optimized values of $\Psi(k)$ are given in Table 3. Having them available when the proof is presented makes it easier to verify the claims.

We will now proceed to prove the time complexity of $D(F)$.

Theorem 12 $D(F)$ runs in $O(1.6737^n)$ time.

Proof: Let k be the number of 2-clauses in F. We will give the branching numbers for the various cases of the algorithm, using the measure $f(F) = n - \Psi(k)$ with the values for $\Psi(k)$ given above.

Case 1 takes $O(1)$ time. Cases 2 and 3 do not increase the time complexity.

Case 4: In both branches, at least the variables v and a are removed, as v will be removed by multiplier reduction if $d(v) = d_2(v) = 1$. If $d_2(a) > 0$, then at least one more variable is removed in some branch, otherwise we have $\Delta k \geq 0$ in both branches. In both cases, the branching is dominated by $\tau(2 - \Psi(4), 2) < 1.6181$.

Case 5: If $d_2(v) = 2$, assume w.l.o.g. that there exists a 2-clause $(v \vee a)$ in F. In both branches, at least the variables v and a are removed, as well as at least two 2-clauses. If a is involved in some 2-clause not containing v or $\neg v$, then at least one more variable is removed in some branch, leading to a branching tuple dominated by $(1, 2)$, otherwise $d_3(a) > 0$ and we have a worst-case branching number of $\tau(2 - \Psi(2), 2 - \Psi(1)) < 1.5239$.

If $d_2(v) = 1$, assume w.l.o.g. that there exists a 3-clause $(v \vee a \vee b)$ in F and that we are branching on a. In the $a = 1$ branch v is removed, and in the $a = 0$ branch either v is removed by some reduction or a new 2-clause $(v \vee b)$ is

Ref.	Tuple	Number
1	$(1-\Psi(1), 2+(\Psi(2)-\Psi(1)))$	1.6701
2	$(1-(\Psi(2)-\Psi(1)), 2+(\Psi(3)-\Psi(2)))$	1.6674
3	$(1-(\Psi(3)-\Psi(2)), 2+(\Psi(4)-\Psi(3)))$	1.6504
4	$(1-(\Psi(4)-\Psi(3)), 2)$	1.6737
5	$(1-\Psi(2), 3-(\Psi(2)-\Psi(1)))$	1.6664
6	$(1-(\Psi(2)-\Psi(1)), 3-\Psi(2))$	1.5933
7	$(2-(\Psi(2)-\Psi(1)), 2-\Psi(2))$	1.5184
8	$(1-(\Psi(3)-\Psi(1)), 3-(\Psi(3)-\Psi(1)))$	1.6533
9	$(1-(\Psi(3)-\Psi(2)), 3-\Psi(3))$	1.6034
10	$(2-(\Psi(3)-\Psi(1)), 2-\Psi(3))$	1.5902
11	$(1-(\Psi(4)-\Psi(2)), 3-(\Psi(4)-\Psi(1)))$	1.6448
12	$(1-(\Psi(4)-\Psi(3)), 3-\Psi(4))$	1.6222
13	$(2-(\Psi(4)-\Psi(2)), 2-(\Psi(4)-\Psi(1)))$	1.5496
14	$(1-\Psi(3), 4-\Psi(3))$	1.6737
15	$(1-(\Psi(4)-\Psi(1)), 4-\Psi(4))$	1.6268
16	$(1-\Psi(4), 5-\Psi(4))$	1.6737

Table 4: Branching tuples and branching numbers for case 6

created. If v is removed by some reduction, the branching tuple is dominated by $(2-\Psi(2), 2-\Psi(1))$ or $(1, 2)$. If $d_2(a) = 0$, then there are two worst-case branching tuples, both dominated by $(2-\Psi(1), 1+\Psi(1))$, with a branching number of 1.5950. Otherwise, at least one more variable is removed in some branch. Every resulting branching is dominated by $\tau(1,2) < 1.6181$, $\tau(2-\Psi(2), 2-\Psi(1)) < 1.5239$ or $\tau(1-\Psi(2), 4-\Psi(4)) < 1.5923$.

Case 6: We will give the possible worst case branchings for each value of $d_2(v)$. We let k denote the number of 2-clauses in F. Note that the worst case branching for a particular value of $d_2(v)$ will always have a minimum $d_3(v)$: If v is a literal in a 3-clause, then this 3-clause contributes nothing when $v = 1$ and increases k by 1 when $v = 0$.

Since there are so many hard worst case branchings for this case, the branchings and the branching numbers are given in Table 4 for overview. The branching numbers are all at most 1.6737.

If $d_2(v) = 1$, the worst case is when $d(v) = 3$, and supposing that the 2-clause is $(v \vee a)$, we know that $d_2(a) = 1$, so only one 2-clause is removed in both branches. Also, $d_3(v) = 2$, resulting in two newly created 2-clauses. The worst case, because of the behaviour of $\Psi(k)$, is the case where both 3-clauses include v, so that the branching is $\tau(1-(\Psi(k)-\Psi(k-1)), 2+(\Psi(k+1)-\Psi(k)))$. The branching tuples and branching numbers for these cases are lines 1–4 of Table 4. Cases with $k > 4$ result in $\tau(1,2) < 1.6181$.

If $d_2(v) = 2$, we similarly have $d(v) = 3$ and, if the neighbours of v in the 2-clauses are a and b, $d_2(a) \leq 2$ and $d_2(b) \leq 2$. For the non-monotone case $(v \vee a), (\neg v \vee b)$ we have, disregarding the 3-clause, 2 variables and 2 or 3 2-clauses removed both when $v = 1$ and $v = 0$. For the monotone case $(v \vee a), (v \vee b)$, we have 1 variable and 2 2-clauses removed when $v = 1$, and 3 variables and 2–4 2-clauses removed when $v = 0$. In both cases, one 2-clause will be created in one of the branches, guaranteeing that $k \geq 1$ in that branch. Because of this guarantee, the results are different depending on the value of k. Lines 5–7 of Table 4 contain the cases for $k = 2$, beginning with the monotone case. Lines 8–10 contain the cases for $k = 3$ and lines 11–13 contain the cases for $k = 4$, both beginning with the monotone case. $k > 4$ will not bring about any new worst cases, as $\Psi(k)$ flattens out and $\Psi(k) - \Psi(k_1)$ decreases.

If $d_2(v) \geq 3$, the worst case is when $d_3(v) = 0$. When $k = 3$, we have $d_2(v) = 3$ resulting in $\tau(1 - \Psi(3), 4 - \Psi(3))$ or $\tau(2 - \Psi(3), 3 - \Psi(3))$, where the former is clearly the worst case. When $k = 4$, we have $d_2(v) = 3$ resulting in $\tau(1 - (\Psi(4) - \Psi(1)), 4 - \Psi(4))$, which is a candidate for the worst case, or $\tau(2 - \Psi(4), 3 - \Psi(4)) < \tau(1, 2) < 1.6181$, and $d_2(v) = 4$ resulting in $\tau(1 - \Psi(4), 5 - \Psi(4))$, $\tau(2 - \Psi(4), 4 - \Psi(4))$ and $\tau(3 - \Psi(4), 3 - \Psi(4))$, where the first case is clearly the worst case. The worst of these cases are lines 14—16 of Table 4.

Case 7: Since $k = 0$ when this case is met, we need only count the number of variables removed. In the first two branches, either v is removed by some reduction, or case 4 is met. The possible branching numbers $\tau(2, 3, 3)$, $\tau(3, 3, 3, 3)$, $\tau(2, 3, 4, 4)$ and $\tau(3, 3, 3, 4, 4)$ are all smaller than 1.6181.

Case 8: Since $d(v) \geq 3$ for all variables in F, the only situation where $d(v) = 3$ for the chosen variable v is when F is 3-regular. As every modification of the formula that our algorithm performs is either a deletion of a variable of clause, or a shortening of a clause, this situation only occurs once along each path down the branching tree, so in the general case we may assume that $d(v) \geq 4$. If there are a 3-clauses that contain v, and b 3-clauses that contain $\neg v$, then the branching number for this case is $\tau(1 + \Psi(a), 1 + \Psi(b))$. Because of the properties of $\Psi(k)$ and τ, the worst case is $\tau(1, 1 + \Psi(4)) < 1.6737$. $\square$

We will now give a description of how the values for $\Psi(k)$ were found. The branchings that can not be proven to have a branching number of at most 1.6737 when $\Psi(k)$ is unknown are two cases from case 5, fourteen cases from case 6 and one case from case 8. Let $B_1(x), \ldots, B_{17}(x)$ be the value of each of these branching numbers given a vector $x = (\Psi(1), \ldots, \Psi(4))$. Then the running time of $D(F)$ is $O(m(x)^n)$ where $m(x) = \max_{1 \leq i \leq 17} B_i(x)$, and the optimization problem we have to solve is to find the x which minimizes $m(x)$ under the conditions that $0 < \Psi(1) < \ldots < \Psi(4) < 1$. Note that since each $B_i(x)$ is continuous as long as these conditions hold, $m(x)$ is also continuous, which is a sufficient condition for standard algorithms to achieve at least a local optimum. The result of this optimization is the values given in Table 3.

6 Algorithm C_{sep}

In this section we will present an algorithm for $\#2\text{SAT}_w$ for a special class of formulae, namely those with a separable constraint graph. Due to the kinship between 2SAT formulae and graphs this class of formulae enjoys interesting properties as we shall see. Before that, however, we need some additional preliminaries.

A graph is *complete* if every pair of distinct vertices is joined by an edge. The complete graph with n vertices is denoted K_n. $H = (V_H, E_H)$ is a *subgraph* of $G = (V_G, E_G)$ iff $V_H \subseteq V_G$ and $E_H \subseteq E_G$. For an edge $e = (u, v)$, u and v are called its *endpoints*. If S is a set of edges of G, the operation of deleting S from G and identifying the endpoints is called *contracting* S. A *minor* of a graph G is any graph H obtained from G by a series of vertex deletions, edge deletions and edge contractions.

Let S be a class of graphs closed under the subgraph relation and $f(n)$ be a non-negative function. In [14], Lipton and Tarjan define an $f(n)$*-separator theorem* for S as follows:

Definition 13 (Separator theorem) There exist constants $\alpha < 1$ and $\beta > 0$ such that if G is any n-vertex graph in S, then the vertices of G can be divided into three sets A, B and C such that no edge joins a vertex in A with a vertex in B, $max(|A|, |B|) \leq \alpha n$ and $|C| \leq \beta f(n)$.

A *separator algorithm* is a polynomial time algorithm Sep that takes a graph G as input and returns the tuple (A, B, C) of Definition 13. The constants α, β and f differ between different separator algorithms, but we require that $f(n) \in o(n)$. If there is an $f(n)$-separator theorem for S we say that S is a *separable* graph class and that $G \in S$ is a separable graph. The class of separable 2SAT formulae we define as the class of formulae having separable constraint graphs.

Let F be a separable 2SAT formula with the constraint graph G_F, such that $G_F = (V, E)$, where $|V| = n$, and let Sep be a separator algorithm for the class that G_F belongs to. The algorithm $C_{sep}(F, Sep)$ will return a tuple (m, w), such that m is the number of maximum weighted models for F and w is the weight of any such model. The algorithm recursively breaks down its input until a constant size, say b, of the input is reached. The constant sized sub-problems are solved by calling C_E which will exhaustively search for the number and weight of the MWMs. To break down the input, Sep will be used to obtain A, B and C. If A and B were both disjoint and their union would equal G_F we would need just two recursive calls, but that is not possible in general. Instead, we are required to do a recursive call for

Algorithm $C_{sep}(F, Sep)$
1. If $n(F) < b$, return $C_E(F)$
2. $c \leftarrow 0; w \leftarrow 0$
3. Let $(A, B, C) = Sep(G_F)$
4. For each extendible assignment κ_C to the variables of C:
$(c, w) = comb(w(\kappa_C), C_{sep}(F_A, Sep), C_{sep}(F_B, Sep), (c, w))$
5. Return (c, w)

Figure 5: Algorithm solving $\#2\mathrm{SAT}_w$ for separable 2SAT formulae

every assignment of the variables of C, or rather every assignment of C that is extendible. The idea is that we check every possible configuration of C, deciding which variables should be set to 1, and which should be set to 0. Let F_A (F_B) denote the formula obtained from collecting just the clauses where variables in A (B) participate. The weight of a partial assignment κ_C is the sum of the weights of the literals which are true under κ_C.

There is a helper function $(c', w') = comb(a, (c_1, w_1), (c_2, w_2), (c, w))$ with the following definition:

$$(c', w') = \begin{cases} (c, w) & \text{if } a + w_1 + w_2 < w \\ (c + (c_1 \cdot c_2), w) & \text{if } a + w_1 + w_2 = w \\ (c_1 \cdot c_2, a + w_1 + w_2) & \text{if } a + w_1 + w_2 > w \end{cases}$$

We will use *comb* to combine the results so far with the current recursive calls. Thus the algorithm reads as shown in Figure 6.

The following theorem will establish the correctness of $C_{sep}(G_F, Sep)$:

Theorem 14 $C_{sep}(G_F, Sep)$ returns the tuple (c, w), where c is the number of MWM in F and w the weight of any such MWM.

Proof: We note that if $n(F) < b$ then the output is correct by assumption. Else, we calculate A, B and C, and since every extendible assignment of C is checked, the algorithm is sound and complete. Note that if no assignment is extendible, $(0, 0)$ will be returned. $\square$

We now give an upper time bound on the running time of C_{sep}:

Theorem 15 The running time $T(n)$ of C_{sep} is in $O(poly(n) \cdot 2^{\beta f(n)z \log n})$, where $z = -\frac{1}{\log \alpha}$.

Proof: Any tree of depth k having maximum degree t has at most $\frac{t^{k+1}-1}{t-1}$ nodes. Let us first establish an upper bound on k:

Looking at a particular node y in the recursion tree, it processes the subgraph $G_F^y \subseteq G_F$. We know that $|G_F| = n$, and that in each child y' of y it holds that $|G_F^{y'}| \leq \alpha |G_F^y|$. Following one path downwards the recursion tree, at the last level the node processes a subgraph of constant size, say 1. Hence, $\alpha^k n \leq 1 \Leftrightarrow \log \alpha^k n \leq \log 1 \Leftrightarrow k \log \alpha + \log n \leq 0 \Leftrightarrow k \leq -\frac{\log n}{\log \alpha}$

As for the degree, we know that $t = 2^{\beta f(n)}$, and hence the number of nodes is

$$\frac{(2^{\beta f(n)})^{k+1} - 1}{2^{\beta f(n)} - 1}$$

In each node the work is $O(poly(n))$ and so we have

$$T(n) \in O\left(poly(n) \cdot \frac{(2^{\beta f(n)})^{(z \log n)+1}}{2^{\beta f(n)} - 1}\right) = O(poly(n) \cdot 2^{\beta f(n) z \log n})$$

□

We next look at some particular instance classes to see how restrictions on the constraint graph can be exploited.

Graphs with an excluded minor H: Alon *et al.* [2] have presented a separator algorithm that guarantees $|C| \leq h^{3/2} n^{1/2}$ and $\alpha \leq 2/3$, where $h = |H|$. Hence, for this class, $\#2\text{SAT}_w$ can be solved in time $O(2^{h^{3/2}\sqrt{n}z' \log_2 n})$, where $z' \approx 1.7$. For two special subclasses we have even faster running times:

1. **Graphs of bounded genus:** Aleksandrov and Djidjev [1] have presented a separator algorithm for splitting a graph $G = (V, E)$ that is embeddable on a surface of bounded genus g. It runs in $O(n+g)$ time and guarantees that $|C| \leq 4\sqrt{gn + n/\varepsilon}, \forall \varepsilon \in (0,1)$ and $|F| \leq \varepsilon n$ for every connected component F obtained from $G - C$. We choose $\varepsilon = 1/2$ and get $\alpha \leq 2/3$ and so we have that $T(n) \in O(2^{4\sqrt{gn+2n}z' \log_2 n})$.

2. **Planar graphs:** The classical separator theorem for planar graphs by Lipton and Tarjan [14] has $\alpha \leq 2/3$ and $|C| \leq 2\sqrt{2n}$. This gives a running time in $O(2^{2\sqrt{2n}z' \log_2 n})$.

A complexity theoretical note is appropriate here: Vadhan [18] has proved that #MAXIMUM INDEPENDENT SET remains #P-complete even for planar bipartite graphs of degree ≤ 3. As we shall see, $\#2\text{SAT}_w$ is at least as hard as #MAXIMUM INDEPENDENT SET, and so one can say the following about the complexity of the graphs in the previous paragraph: $\#2\text{SAT}_w$ for the last class is obviously #P-complete, and the same holds for the second class. As for the first class, there are subclasses for which $\#2\text{SAT}_w$ is polynomial time solvable (*e.g.*, an excluded K_2 minor). However, for many interesting subclasses *e.g.*, graphs with no K_h minor (and $h > 2$), the problem is #P-complete.

We now look at a polynomial time solvable case:

Graphs with bounded tree width: For a graph G with a bounded tree width w, one can in $O(n^w)$ time separate G such that $|C| \leq k$ and $\alpha = 2/3$ (see for instance [5]). Using Theorem 15 we would get an upper time bound in $O(n^{2w})$, but we can do better. The running time $T(n)$ can be described by the recurrence

$$T(n) \leq 2T(2n/3) + n^w.$$

Applying the Master Theorem for divide and conquer recurrences (see for instance [6]), we get a running time in $O(n^w)$.

The matter of counting in this class has been previously dealt with by Díaz *et al.* [10]. Their main result is an algorithm that counts homomorphisms in linear time for fixed w. This algorithm can be used for counting, among other things, the number of independent sets in a bounded tree width graph in linear time. Their results are however not fully comparable with ours: using their algorithm for counting *maximum* independent sets would take time exponential in the number of vertices of the graph (see Corollary 5.10 in [10]), whereas our reduction (to be shown later in this paper) implies an $O(n^w)$ time algorithm. On the other hand, modifying our reduction for maximum independent sets to allow all independent sets still yields an $O(n^w)$ time algorithm.

7 Applications

We here give some interesting applications for the counting of MWM for 2SAT and 3SAT formulae. We start with an application for D. This reduction first appeared in an article by Valiant [20].

#CIRCUIT SATISFIABILITY can be solved in $O(1.6737^{n+m})$ time, where n is the number of inputs and m the number of gates.
INSTANCE: A one-output boolean combinatorial circuit consisting of AND, OR and NOT gates.
QUESTION: How many (different) inputs make the output 1?
REDUCTION: Each gate has (at most) two inputs and one output. Hence it can be mimicked using (at most) three variables. So, at the first level of the circuit each input translates to a variable and then each gate yields a variable.

There is a kinship between #2SAT and many graph problems which we will exploit in the following. As a first example we give a reduction from #PERFECT MATCHINGS. Note that #PERFECT MATCHING for general graphs contains the computation of the permanent for a 0/1-matrix – the classic #P-complete problem. As previously mentioned, the so-far best algorithm to that end runs in $O(n^2 2^n)$ time. For general graphs with no restriction on the degree, our algorithm runs in $O(1.2561^{|E|})$ and as $|E| \in O(|V|^2)$ our algorithm cannot compete with Ryser's. However, for sparse graphs, having degree at most 6, we get a running time in $O(1.9819^n)$. To the best of our knowledge, this is the first algorithm to this end for sparse graphs.

#PERFECT MATCHINGS and #MATCHINGS can be solved in $O(1.2561^{|E|})$ time, where $|E|$ is the number of edges.
INSTANCE: A graph $G = (V, E)$.
QUESTION: A matching is a subset $E' \subseteq E$ such that a number of vertices (not necessarily all) is covered once by an edge in E'. A perfect matching is a matching that covers all vertices. What is the number of matchings/perfect matchings?
REDUCTION: For the #PERFECT MATCHINGS problem we create an instance of #2SAT in the following way: Let each edge of G form a boolean variable e_i such that $w(e_i) = 1$ and $w(\neg e_i) = 0$. For each vertex x of G there is a set $e_1 \dots e_j$ of edges having x as an endpoint. For every such set, form the clauses $(\neg e_1 \vee \neg e_2) \dots (\neg e_1 \vee \neg e_j) \dots (\neg e_2 \vee \neg e_3) \dots (\neg e_2 \vee \neg e_j) \dots (\neg e_{j-1} \vee \neg e_j)$. As one can see, this implies that at most one of the variables in the set can be *true*. Applying an algorithm for counting the MWMs, if the weight equals $|G|/2$ then the number of MWMs is the number of perfect matchings of G, otherwise the number is 0. Obviously, the #MATCHINGS problem can also be solved using a slight modification of the reduction: simply assign $w(e_i) = w(\neg e_i) = 1$.

We next look at the #MAXIMUM INDEPENDENT SET and #INDEPENDENT SET problems. The so-far best algorithm for the former problem by Dahllöf and Jonsson [7] runs in $O(1.3247^n)$ time.

#MAXIMUM INDEPENDENT SET and #INDEPENDENT SET for general graphs is solvable in $O(1.2561^n)$ time; for graphs with an excluded H minor in $O(2^{|H|^{3/2}\sqrt{n}1.7\log_2 n})$ time, for graphs of bounded genus g in $O(2^{4\sqrt{gn+2n}1.7\log_2 n})$ time, for planar graphs in $O(2^{2\sqrt{2n}1.7\log_2 n})$ time and for graphs with a bounded tree width of w in $O(n^w)$ time.
INSTANCE: A graph $G = (V, E)$, with a weight $w(x)$ for each vertex $x \in V$.
QUESTION: What is the number of independent sets/maximum independent sets?
REDUCTION: For the #MAXIMUM INDEPENDENT SET problem, let each vertex x_i of G form a boolean variable x_i and each edge (x_j, x_k) give rise to a clause $(\neg x_j \vee \neg x_k)$ and each $z \in V$ that has no edge the clause (z) (since z must be in every maximum independent set). The weights associated with each vertex x_i of G is transferred to the positive literal x_i and $w(\neg x_i) = 0$. Clearly, the number of MWMs equals #MAXIMUM INDEPENDENT SET. For the #INDEPENDENT SET problem one simply assigns $w(x) = w(\neg x_i) = 1$

and each $z \in V$ that has no edge gives rise to the clause $(z \vee \neg z)$.

As we now have algorithms for the #MAXIMUM INDEPENDENT SET problem, we can employ the reductions of Dahllöf and Jonsson [7] and get the following upper time bounds:

#EXACT COVER for general graphs can be solved in $O(1.2561^n)$ time (and for separable graphs time bounds are given above).
INSTANCE: A finite set U and a collection C of subsets $c_1, \dots, c_n$ of U.
QUESTION: If C contains an exact cover for U – *i.e.*, a subcollection $C' \subseteq C$ such that every element of U occurs in exactly one member of C' – what is the number of exact covers?
REDUCTION: Let (U, C) be an arbitrary instance of the #EXACT COVER problem. Construct a weighted graph $W = (V, E)$ as follows: let each $c_i \in C$ give rise to a vertex $v_i \in V$ whose weight is $|c_i|$. Add an edge between v_i and v_j if and only if $c_i \cap c_j \neq \varnothing$. Clearly, no independent set in W can have a weight greater than $|U|$. Furthermore, the independent sets of weight $|U|$ in W corresponds to solutions of (U, C).

#EXACT HITTING SET can be solved in $O(1.2561^n)$ time.
INSTANCE: A finite set U and a collection C of subsets $c_1, \dots, c_n \subseteq U$ such that $\bigcup_{c_i} = U$
QUESTION: A solution is a minimum size subset $H \subseteq U$ hitting each c_i exactly once, *i.e.*, $|c_i \cap H| = 1$. What is the number of solutions?
REDUCTION/MODIFICATION: To ensure the minimum size property we will have to add a slight modification to the algorithms such that whenever a heavier solution is preferred over lighter ones, there is also a preferrement of having as few true positive literals as possible. Then, let (U, C) be an arbitrary instance of the #EXACT HITTING SET problem. Construct a weighted graph $W = (V, E)$ as follows: Let each element $x_j \in U$ form a vertex. Add an edge between each pair of elements that are contained within the same c_i. The weight $w(x_j)$ is the number of subsets x_j appears in. Obviously, a maximum independent set mis such that the weight of mis equals $|C|$ covers all subsets.

#WEIGHTED SET PACKING for general graphs can be solved in $O(1.2561^n)$ time.
INSTANCE: A finite set U and a collection C of subsets $c_1, \dots, c_n \in U$ and for each c_i there is an associated weight $w(c_i)$.
QUESTION: A solution is a collection $C' \subseteq C$ of disjoint sets of maximum weight. What is the number of such solutions?
REDUCTION: Let (U, C) be an arbitrary instance of the #WEIGHTED SET PACKING problem. Construct a weighted graph $W = (V, E)$ as follows: introduce one vertex v_i for each $c_i \in C$ and assign it weight $w(c_i)$. Add an

edge between v_i and v_j if and only if $c_i \cap c_j \neq \varnothing$. Obviously, a maximum weighted, independent set found in W constitutes a solution.

8 Discussion and Conclusions

We have presented three algorithms for counting models for restricted boolean formulae. The algorithms C and D improve the upper time bounds for two well-studied problems. The algorithm C_{sep} exploits the possibilities of separation to count fast in 2SAT formulae. The applications for our algorithms include one of the most studied counting problems, namely #PERFECT MATCHING.

The method of analysis used in this paper, combined with the appropriate measures, provides a convenient way to capture and quantify the effects of properties that might otherwise be difficult to analyse, such as the decreasing average degree in C and the existence of 2-clauses in some branches in D. In algorithms with similar properties, it is likely that these measures can be reused.

When it comes to ways of even further improving algorithms for #2SAT and #3SAT, one alternative might be to perform a more careful analysis of the neighbourhood configurations, especially for #2SAT. Another option, if large memory usage is acceptable, might be to employ various dynamic programming or caching tricks, trading a decreased runtime for possibly exponential memory usage.

The ideas in D might well be extended into an algorithm for #kSAT for any fix k, with a running time dependent on the parameter k but better than $O(2^n)$. We have examined this issue briefly, and while the results seem promising, performing an analysis for the general case is challenging.

In the context of decision problems, adding weights seems to increase the complexity considerably. For instance, 2SAT is polynomial time solvable whereas the 2SAT_w problem is NP-hard (it contains the MAXIMUM INDEPENDENT SET problem). It is thus interesting to note that our algorithms for #2SAT and #3SAT were easily extended into algorithms for #2SAT_w and #3SAT_w. They are also, to the best of our knowledge, the only algorithms for solving the 2SAT_w and 3SAT_w problems published.

References

[1] L. Aleksandrov, H. Djidjev, Linear algorithms for partitioning embedded graphs of bounded genus, SIAM J. Discr. Math. 9 (1996) 129–150.

[2] N. Alon, P. Seymour, R. Thomas, A separator theorem for graphs with an excluded minor and its applications, Proc. 22nd ACM Symp. Theory of Computing, 1990, pp. 293–299.

[3] O. Angelsmark, P. Jonsson, Improved algorithms for counting solutions in constraint satisfaction problems, Proc. 9th Conf. on Principles and Practice of Constraint Programming, 2003, pp. 81–95.

[4] R. Bayardo, J. D. Pehoushek, Counting models using connected components, Proc. 17th National Conf. AI and 12th Conf. on Innovative Applications of Artificial Intelligence, 2000, pp. 157–162.

[5] H. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, Theoret. Comp. Sci. 209 (1998) 1–45.

[6] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, The MIT press, Cambridge, Massachusetts, 1990.

[7] V. Dahllöf, P. Jonsson, An algorithm for counting maximum weighted independent sets and its applications, Proc. 13th ACM-SIAM Symp. on Discrete Algorithms, 2002, pp. 292–298.

[8] V. Dahllöf, P. Jonsson, M. Wahlström, Counting satisfying assignments in 2-SAT and 3-SAT, Proc. 8th Int. Computing and Combinatorics Conf., 2002, pp. 535–543.

[9] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.

[10] J. Díaz, M. J. Serna, D. M. Thilikos, Counting H-colorings of partial k-trees, Theoret. Comput. Sci. 281 (2002) 291–309.

[11] O. Dubois, Counting the number of solutions for instances of satisfiability, Theoret. Comput. Sci. 81 (1991) 49–64.

[12] D. Kozen, The design and analysis of algorithms, Springer-Verlag, New York, 1992.

[13] O. Kullmann, New methods for 3-SAT decision and worst-case analysis, Theoret. Comput. Sci. 223 (1999) 1–72.

[14] R. Lipton, R. E. Tarjan, A separator theorem for planar graphs, SIAM J. Appl. Math. 36 (1979) 177–189.

[15] M. L. Littman, T. Pitassi, R. Impagliazzo, On the complexity of counting satisfying assignments, In the working notes of the LICS 2001 workshop on Satisfiability.

[16] M. Robson, Finding a maximum independent set in time $O(2^{n/4})$, Tech. Report, LaBRI, Université Bordeaux I, 2001.

[17] H. J. Ryser, Combinatorial Mathematics, The Mathematical Association of America, Washington, 1963.

[18] S. P. Vadhan, The complexity of counting in sparse, regular, and planar graphs, SIAM J. Comput. 31 (2001) 398–427.

[19] L. Valiant, The complexity of computing the permanent, Theoret. Comput. Sci. 8 (1979) 189–201.

[20] L. Valiant, The complexity of enumeration and reliability problems, SIAM J. Comput. 8 (1979) 410–421.

[21] W. Zhang, Number of models and satisfiability of sets of clauses, Theoret. Comput. Sci. 155 (1996) 277–288.