Improved Algorithms for Counting Solutions in Constraint Satisfaction Problems

Ola Angelsmark* and Peter Jonsson**

Department of Computer and Information Science
Linköpings Universitet
S-581 83 Linköping, Sweden
{olaan,petej}@ida.liu.se

Abstract. Counting the number of solutions to CSP instances has vast applications in several areas ranging from statistical physics to artificial intelligence. We provide a new algorithm for counting the number of solutions to binary CSPs which has a time complexity ranging from $\mathcal{O}\left((d/4 \cdot \alpha^4)^n\right)$ to $\mathcal{O}\left((\alpha + \alpha^5 + \lfloor d/4 - 1 \rfloor \cdot \alpha^4)^n\right)$ (where $\alpha \approx 1.2561$) depending on the domain size $d \geq 3$. This is substantially faster than previous algorithms, especially for small d. We also provide an algorithm for counting k-colourings in graphs and its running time ranges from $\mathcal{O}\left(\lfloor \log_2 k \rfloor^n\right)$ to $\mathcal{O}\left(\lfloor \log_2 k + 1 \rfloor^n\right)$ depending on $k \geq 4$. Previously, only an $\mathcal{O}\left(1.8171^n\right)$ time algorithm for counting 3-colourings were known, and we improve this upper bound to $\mathcal{O}\left(1.7879^n\right)$.

1 Introduction

Constraint satisfaction problems (CSPs), first described by Montanari [16], allows for natural descriptions of problems in a wide array of fields, cf. Kumar [14]. The most thoroughly studied problem for CSPs is that of *decidability*, i.e., 'does a given problem have a solution?' but there are a number of alternative questions one can ask, including 'Does variable x have the same value in every solution?' (the *frozen variable problem*) and 'How many solutions are there?' (the *counting problem*.) In this paper we will focus on the latter, usually denoted #CSP.

The #CSP problem has many important applications. A broad range of classical combinatorial problems such as GRAPH RELIABILITY [20] and PERMANENT [19] can be viewed as instances of the #CSP problem. This also holds for many AI problems such as approximate reasoning [17], diagnosis [18] and belief revision [8]. Solving a CSP instance is equivalent to finding a homomorphism between graphs [11], for instance, finding a k-colouring of a graph G is equivalent to finding a homomorphism from G to a complete graph with k vertices. Determining the number of graph homomorphisms from one graph to another has important applications in statistical physics [10] – e.g. computations in the *Potts*

* Partially supported by CUGS – National Graduate School in Computer Science, Sweden – and the Swedish Research Council (VR) under grant 621-2002-4126.
** Partially supported by VR under grant 221-2000-361

model and the problem of counting q-particle *Widom-Rowlinson* configurations in graphs.

Until quite recently, not much attention has been given to the computational complexity of #CSP. The counting problem belongs to a class known as #P (introduced by Valiant [19, 20]) defined as the class of counting problems computable in nondeterministic polynomial time. Computing the number of solutions to a constraint satisfaction problem is, even if we restrict ourselves to binary CSPs, complete for this class of problems [17]. Lately, however, a number of papers have addressed complexity issues in greater detail [5, 6, 15], and a number of exact algorithms have been developed [1, 3, 7, 9]. Bulatov and Dalmau [5] have formulated a plausible conjecture concerning the complexity of the counting problem: the problem can be solved in polynomial time if and only if the relations under consideration are closed under a *Mal'tsev* operation (and the problem is #P-complete otherwise). Since the set of relations closed under Mal'tsev operations is fairly limited[1], the truth of this conjecture would imply that we are forced to use exponential-time algorithms for solving most naturally arising counting problems.

In this paper we will focus on *exact, deterministic* algorithms for the following two problems: the counting problem for binary CSPs, denoted $\#(d,2)$-CSP[2], and the counting problem for k-colourability of graphs, denoted #kCOL. Algorithms for $\#(d,2)$-CSP and #3COL have been presented elsewhere [1]. The previously presented algorithm for #3COL runs in $\mathcal{O}(1.8171^n)$ time and the running time for the $\#(d,2)$-CSP algorithm is (we omit polynomial factors here and throughout the paper):

- $\mathcal{O}(\alpha^n \cdot (d/2)^n)$, if d is even, and
- $\mathcal{O}\left(\alpha^n \cdot \lceil (d^2+d)/4 \rceil^{n/2}\right)$, if d is odd

where $\mathcal{O}(\alpha^n)$ is the time complexity for solving $\#(2,2)$-CSP. Running times for a selection of domain sizes can be found in Table 1. Since the running time of this algorithm depends on the running time for solving the $\#(2,2)$-CSP problem, we provide the times both for the $\mathcal{O}(1.3247^n)$ $\#(2,2)$-CSP algorithm originally used [7] and for the improved $\mathcal{O}(1.2561^n)$ algorithm [21].

The old $\#(d,2)$-CSP algorithm has the following general outline:

1. Create $\#(d,2)$-CSP instances 'corresponding' to the original CSP instance.
2. Count the number of solutions to each of these instances.
3. Return the total number of solutions found.

The new algorithm is different in several respects. It creates *weighted* $\#(d,2)$-CSP instances instead of ordinary $\#(d,2)$-CSP instances – this enables us to use a microstructure-like [12] construction that reduces the number of instances we need to consider. When applied to odd domains, the old algorithm considers pairs

[1] For instance, if we only consider domains of size 2, there is a single polynomial class known as the *affine* subclass.

[2] The necessary definitions for the CSP notions used in this paper are found in Section 2.

d	$\#(d,2)$-CSP			$\#d$COL
	Original algorithm	Original algorithm w/improvements	New algorithm	
3	2.2944^n	2.1757^n	1.9819^n	1.7879^n
4	2.6494^n	2.5122^n	2.4895^n	2^n
5	3.7468^n	3.5528^n	3.1270^n	2.7879^n
6	3.9741^n	3.7683^n	3.7456^n	2.7879^n
7	4.9566^n	4.6999^n	4.3831^n	3^n
8	5.2988^n	5.0244^n	4.9789^n	3^n
10	6.6235^n	6.2805^n	6.2350^n	3.7879^n
15	10.2611^n	9.7298^n	9.3619^n	4^n
20	13.2470^n	12.5610^n	12.4471^n	4.7879^n
25	16.9126^n	16.0369^n	15.5740^n	5^n

Table 1. Time complexities for solving $\#(d,2)$-CSP and $\#d$COL.

of variables when constructing the $\#(d,2)$-CSP instances; the technique for doing so is straightforward in principle but inelegant in practice. The new algorithm 'partitions' the domains of the variables into subdomains of sizes between 2 and 5 elements. This leads to a faster and more elegant algorithm but its analysis becomes a bit more complicated. The running time of the algorithm is:

- $\mathcal{O}\left((d/4\cdot\alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6224d)^n\right)$ if $d \equiv 0 \pmod 4$
- $\mathcal{O}\left((\alpha^5 + \lfloor d/4-1\rfloor\cdot\alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6254d)^n\right)$ if $d \equiv 1 \pmod 4$
- $\mathcal{O}\left((\alpha + \lfloor d/4\rfloor\cdot\alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6243d)^n\right)$ if $d \equiv 2 \pmod 4$
- $\mathcal{O}\left((\alpha + \alpha^5 + \lfloor d/4-1\rfloor\cdot\alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6262d)^n\right)$ if $d \equiv 3 \pmod 4$

For large domains, the term $d/4\cdot\alpha^4$ dominates the time complexity, since the number of 4-partitions increase with d. Consequently, the bounds in the list will all approach $\mathcal{O}\left((d/4\cdot\alpha^4)^n\right) \approx \mathcal{O}\left((0.6224d)^n\right)$ as d grows. The old algorithm approaches $\mathcal{O}\left((0.6280d)^n\right)$ running time and this difference may appear negligible. However, the difference is significant for many domain sizes; for instance, if $d=5$, the old algorithm runs in $\mathcal{O}\left(3.5528^n\right) = \mathcal{O}\left((0.7106d)^n\right)$ time while the new algorithm runs in $\mathcal{O}\left(3.1271^n\right) = \mathcal{O}\left((0.6255d)^n\right)$ time. Given the modularity of the algorithm, if a faster method for solving the weighted $\#(2,2)$-CSP problem is found, it is easy to 'plug it into' our algorithm, thus improving the time complexity with no extra work.

The second part of this paper contains an algorithm for $\#k$COL, which runs in $\mathcal{O}\left((c_k)^n\right)$ time where

$$c_k = \begin{cases} \lfloor \log_2 k \rfloor & \text{if } k = 2^i \\ \lfloor \log_2 k \rfloor + (\beta - 1) & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ \lfloor \log_2 k \rfloor + 1 & \text{if } 2^i + 2^{i-1} < k < 2^{i+1} \end{cases}$$

and $\beta \approx 1.7879$. Previously, only an $\mathcal{O}\left(1.8171^n\right)$ time for #3COL was known. The algorithm is once again based on a partitioning technique together with methods for efficiently combining #kCOL algorithms and an $\mathcal{O}(1.7879^n)$ time algorithm for #3COL.

The paper has the following organisation: Section 2 contains the basic definitions needed. Section 3 contains the algorithm for counting solutions to binary CSPs, while Section 4 contains the algorithm for the #kCOL problem. The new algorithm for #3COL is presented in Section 5.

2 Preliminaries

A (d,l)-*constraint satisfaction problem* ((d,l)-CSP) is a triple (V, D, C) with V a finite set of variables, D a finite domain of values, with $|D| = d$, and C a set of constraints $\{c_1, c_2, \ldots, c_q\}$. Each constraint $c_i \in C$ is a structure $R(x_{i_1}, \ldots, x_{i_j})$ where $j \leq l$, $x_{i_1}, \ldots, x_{i_j} \in V$ and $R \subseteq D^j$. A *solution* to a CSP instance is a function $f : V \to D$, such that for each constraint $R(x_{i_1}, \ldots, x_{i_j}) \in C$, $(f(x_{i_1}), \ldots, f(x_{i_j})) \in R$. Given a variable v and a set $X \subseteq D$, we let $(v; X)$ denote the unary constraint $v \in X$. Given a (d,l)-CSP instance, the basic computational problem is to decide whether an instance has a solution or not. The corresponding counting problem #(d,l)-CSP is to determine how many solutions the instance has.

We will view $(2,2)$-CSP instances as instances of *2*-SAT for increased readability. A *2*-SAT formula is a sentence consisting of the conjunction of a number of clauses, where each clause is of one of the forms $(p \vee q)$, $(\neg p \vee q)$, $(\neg p \vee \neg q)$, (p), $(\neg p)$. The *2*-SAT *problem* is to decide whether a given 2-SAT formula is satisfiable or not, and this can be done in linear time [2], whereas the #P-complete *#2*-SAT *problem* is to decide how many solutions a given formula has. We will use an extended version of the #2-SAT problem.

Definition 1. *(#2-*SAT *with weights)*
*A instance is given as a 2-*SAT *formula F over propositions $X = \{x_1, x_2, \ldots, x_n\}$, and a function $f : X \to \mathbb{N}$ assigning to each proposition an integer weight. Let M be the set of models of F and define $\Pi : M \to \mathbb{N}$ as $\Pi(m) = \sum\{f(x) \mid x \text{ true in } m\}$. Let $K = \max_{m \in M} \Pi(m)$. A* solution *to the instance is the number of models that have the total weight K.*

The currently best known algorithm for solving this problem runs in $\mathcal{O}(\alpha^n)$ time [21] where $\alpha \approx 1.2561$.

A *partitioning* $P = \{P_1, \ldots, P_k\}$ of a domain D is a division of D into disjoint subsets s.t. $\bigcup P = D$, and a k-partition is an element of P with k elements. Given a partitioning P, let $\sigma(P, i)$ denote the number of i-partitions in P.

The multinomial coefficient

$$\binom{n}{n_1 \cdots n_k} = \frac{n!}{n_1! \cdot \ldots \cdot n_k!}$$

denotes the number of ways of partitioning a set of size n into k sets, each of size n_i (with $n = \sum_{i=1}^{k} n_i$). We will frequently use the *multinomial theorem:* for $x_1, \ldots, x_k \in \mathbb{R}$,

$$\sum_{n_1+\ldots+n_k=n} \binom{n}{n_1 \cdots n_k} \prod_{i=1}^{k} x_i^{n_k} = \left(\sum_{i=1}^{k} x_i\right)^n$$

3 Algorithm for $\#(d,2)$-Csp

The algorithm for $\#(d,2)$-Csp will be presented in three steps: (1) Theorem 1 demonstrates how partitions can be used for solving $\#(d,2)$-Csp; (2) we show how an algorithm for weighted #2-SAT can be used by Theorem 1; and (3) we show how to choose partitionings optimally in Theorem 2.

Before we prove Theorem 1, we need a straightforward lemma:

Lemma 1. *Assume there exists an $\mathcal{O}(c^n)$ time algorithm for $\#(k,l)$-Csp. Let $\mathcal{I}_k$ denote the set of $\#(d,l)$-Csp instances satisfying the following restriction: for every $(V,D,C) \in \mathcal{I}_k$ and every $v \in V$, there exists a constraint $(v;S)$ in C such that $|S| \leq k$. Then, the #Csp problem restricted to instances in $\mathcal{I}_k$ can be solved in $\mathcal{O}(c^n)$ time.*

Proof. For each variable in (V,D,C), we know that there are at most k out of d values that it can be assigned. Thus, we can modify the constraints so that every variable picks its values from the set $\{1,\ldots,k\}$. This transformation can obviously be carried out in polynomial time and the resulting instance is an instance of $\#(k,l)$-Csp which can be solved in $\mathcal{O}(c^n)$ time.

Theorem 1. *Let A be an algorithm for $\#(d,2)$-Csp running in $\mathcal{O}\left(\prod_{i=1}^{d} \alpha_i^{n_i}\right)$ time ($\alpha_i \geq 1$) when applied to an instance containing n_i i-valued variables for $1 \leq i \leq d$. Choose p and a partitioning $P = \{P_1,\ldots,P_k\}$ of $\{1,\ldots,p\}$ such that $|P_i| \leq d$ for every i. Then, there exists an algorithm for $\#(p,2)$-Csp running in $\mathcal{O}\left(\left(\sum_{i=1}^{d} \sigma(P,i)\alpha_i\right)^n\right)$ time.*

Proof. We claim that the algorithm presented in Figure 1 correctly solves the $\#(p,2)$-Csp problem. Let $I = (V,D,C)$ be an arbitrary instance of $\#(p,2)$-Csp and let M be an arbitrary model of I. The members of P are pairwise disjoint so there exists exactly one function from V to P such that $M(v) \in f(v)$ for all $v \in V$. Now, Lemma 1 implies that this model will cause an increase of variable c by one. Consequently, the algorithm will return the total number of models which proves its correctness.

The running time T of the algorithm is bounded by

$$\mathcal{O}\left(\sum_{f \in F} \prod_{v \in V} \alpha_{|f(v)|}\right)$$

where F is the set of total functions from V to P, i.e. the algorithm considers all possible combinations of elements of P and members of V. The performance

$c := 0$
for every total function $f : V \to P$ **do**
  $I := (V, D, C \cup \{(v_i; f(v_i)) \mid 1 \leq i \leq |V|\})$
  compute the number of solutions to I using Lemma 1 and add to c
end for
return c

Fig. 1. Algorithm for $\#(p, 2)$-CSP.

of algorithm A depends only on the domain sizes associated with the variables and this gives us

$$T \in \mathcal{O}\left(\sum_{k_1 + \ldots + k_{|P|} = n} \binom{n}{k_1 \cdots k_{|P|}} \prod_{i=1}^{|P|} \alpha_{|P_i|}^{k_i} \right)$$

where k_i denotes the number of variables whose domain is P_i. By applying the multinomial theorem and using the fact that there are $\sigma(P, i)$ sets of size i in P, we arrive at

$$T \in \mathcal{O}\left(\left(\sum_{i=1}^{|P|} \alpha_{|P_i|} \right)^n \right) = \mathcal{O}\left(\left(\sum_{i=1}^{d} \sigma(P, i) \alpha_i \right)^n \right).$$

□

We will now use Theorem 1 and the algorithm for weighted #2-SAT in order to construct an algorithm for $\#(d, 2)$-CSP.

Given a $\#(d, 2)$-CSP instance $\Theta = (V, D, C)$ containing k_1 1-valued, k_2 2-valued, k_3 3-valued, etc., variables, we can transform it into a weighted #2-SAT instance as follows:

- If $x \in V$ takes its value from a singelton set $\{1\}$, we can remove x in polynomial time by assigning 1 to x and propagate the variable.
- If a variable $x \in V$ takes its value from a two-valued set $\{1, 2\}$, we introduce a proposition x_1 with the interpretation that $x = 1$ if x_1 is true, and $x = 2$ otherwise.
- For a k-valued variable x, we create k propositional variables $x_1, \ldots, x_k$, with the interpretations that $x = i$ if x_i is true, and $x \neq i$ if x_i is false. To ensure that at most one of these can be true in a satisfying assignment, we add clauses

$$\bigwedge_{i \leq j,\ i, j \in \{1, 2, \ldots, k\}} (\neg x_i \vee \neg x_j).$$

– Constraints involving only variables with more than 2 possible values can be transformed in a straightforward manner; Given a constraint $xRy \in C$, with x having domain D_x and y having domain D_y, we add the clauses

$$\bigwedge_{a \in D_x, b \in D_y, (a,b) \notin R} (\neg x_a \vee \neg y_b).$$

However, if one of the variables is 2-valued, we need to take into account that its negation also corresponds to an assignment, hence, if $xRy \in C$, $x \in \{1,2\}$ and $y \in D_y$, we add the clauses

$$\bigwedge_{b \in D_y, (0,b) \notin R} (\neg x_0 \vee \neg y_b) \quad \wedge \bigwedge_{b \in D_y, (1,b) \notin R} (x_0 \vee \neg y_b).$$

The case when two variables are 2-valued can be transformed analogously.

Now, we note that there is a slight difference between the two kinds of propositions: A proposition stemming from a 2-valued variable will, regardless of truth-value, give an assignment to the original variable, but this does not hold for the k-valued variables with $k > 2$. To remedy this, we exploit the possibility to give weights to the propositions: we give the weight 0 to each proposition corresponding to a 2-valued variable, and the remaining propositions are assigned the weight 1. A model of the instance will then have the value $\sum_{i \geq 3} k_i$, i.e., the number of k-valued variables, $k > 2$.

After this transformation, we get a 2-SAT formula F with $k_2 + 3k_3 + \ldots$ variables, and applying the algorithm for weighted #2-SAT from [21] will give a running time of $\mathcal{O}\left(1^{k_1} \cdot \alpha^{k_2+3k_3+\cdots}\right) = \mathcal{O}\left(1^{k_1} \cdot \alpha^{k_2} \cdot \alpha^{3k_3} \cdot \ldots\right)$, since each 2-valued variable introduces one propositional variable, and each k-valued variable, $k \geq 3$, introduces k propositional variables.

Given a partitioning P of a domain D containing d elements, it follows (by combining the algorithm above and Theorem 1) that we can solve #$(d,2)$-CSP in $\mathcal{O}\left(T(P)^n\right)$ time where

$$T(P) = \sigma(P,1) + \sigma(P,2)\alpha + \sum_{i=3}^{d} \sigma(P,i)\alpha^i.$$

For instance, we can solve #$(d,2)$-CSP in $\mathcal{O}\left(\alpha^{d \cdot n}\right)$ time by using the trivial partitioning $P = \{\{1,\ldots,d\}\}$ but this can be improved by using other partitionings. For example, assume $d = 6$. Using the previous argument, this would give a time complexity of $\mathcal{O}\left(\alpha^{6n}\right) \approx \mathcal{O}\left(3.9278^n\right)$, but if we instead use the partitioning $\{\{1,2\},\{3,4,5,6\}\}$, we arrive at a running time of $\mathcal{O}\left((\alpha+\alpha^4)^n\right) \approx \mathcal{O}\left(3.7455^n\right)$, which is quite an improvement. The obvious question is how to partition the domain to get the optimal running time.

Let $P = \{P_1,\ldots,P_k\}$ be an arbitrary partitioning of a domain D. Since the actual names of the elements in a partition is not important (by Lemma 1), we let the multi-set $[|P_1|, \dots, |P_k|]$ represent the partitioning P. We will use the following fact repeatedly in the proof: if $T([a_1, \dots, a_k]) < T([b_1, \dots, b_m])$, then

$$T([a_1, \dots, a_k, c_1, \dots, c_p]) < T([b_1, \dots, b_m, c_1, \dots, c_p])$$

for all choices of $c_1, \dots, c_p$.

Theorem 2. *Let D be a domain of size $d \geq 2$. If $d < 6$, then the partitioning $P = [d]$ is optimal. Otherwise, the following partitionings are optimal:*

- *if $d = 4k$, $P = [4, 4, 4, \dots, 4]$ (k partitions of size 4)*
- *if $d = 4k + 1$, $P = [5, 4, 4, \dots, 4]$ ($k - 1$ of size 4)*
- *if $d = 4k + 2$, $P = [2, 4, 4, \dots, 4]$ (k of size 4)*
- *if $d = 4k + 3$, $P = [2, 5, 4, 4, \dots, 4]$ ($k - 1$ of size 4)*

Proof. Let P be a partition from the list and assume it is suboptimal, i.e., there is a strictly better partition P^* which is optimal. We show that such a P^* does not exist and, consequently, that P is optimal.

Step 1: P^* contains no 1-partitions.
If $d = 2$, then it is obvious that $P^* = [2]$ and P^* contains no 1-partition. We can consequently suppose that $d > 2$. Assume first that $P^* = [1, \dots, 1]$ (d 1-partitions) and let $P' = [2, 1, \dots, 1]$ ($d - 1$ 1-partitions). It is easy to see that $T(P') < T(P^*)$ so P^* contains at least one a-partition such that $a > 1$. If a=2, i.e. $P^* = [1, 2, p_1, \dots, p_k]$, we let $P' = [3, p_1, \dots, p_k]$ and note that $2.2561 \approx T([1, 2]) > T([3]) \approx 1.9819$ and P^* is not optimal.

Finally, we assume that $P^* = [1, a, p_1, \dots, p_k]$ for some $a > 2$. We define $P' = [2, a - 1, p_1, \dots, p_k]$ and show that $T(P') < T(P^*)$ by induction over a. This contradicts the optimality of P^* and proves that P^* does not contain any 1-partitions.

Basis: If $a = 3$, then $2.9819 \approx T([1, 3]) > T([2, 2]) \approx 2.5122$.

Induction: Assume the claim holds for $a = p$. We show that it also holds for $a = p + 1$ by proving that $T([1, p + 1]) > T([2, p])$, i.e. $1 + \alpha^{p+1} > \alpha + \alpha^p$. First observe that $1.3217 \approx \alpha^2 - \alpha + 1 > \alpha \approx 1.2561$. Now, $1 + \alpha^{p+1} = (1 + \alpha^p) \cdot \alpha - (\alpha - 1) >$ /ind. hyp./ $> (\alpha + \alpha^{p-1}) \cdot \alpha - \alpha + 1 = \alpha^2 + \alpha^p - \alpha + 1 >$ /observation/ $> \alpha^p + \alpha$.

Step 2: P is optimal when $1 < d \leq 5$.
We note that domains of sizes 2 and 3 should not be partitioned since this implies the introduction of a 1-partition. Furthermore, domains of sizes 4 and 5 should not be partitioned either since $T([4]) < T([2, 2])$ and $T([5]) < T([3, 2])$.

Step 3: P^* contains only a-partitions, $a \in \{2, 4, 5\}$, when $d > 5$.
Assume P^* contains an a-partition and $a > 5$. We begin by inductively proving that $T([a, p_1, \dots, p_k]) > T([a - 2, 2, p_1, \dots, p_k])$, i.e. $\alpha^{an} > (\alpha^{a-2} + \alpha^2)^n$. Obviously, this holds when $a = 6$. Assume the assertion holds for every $a < p$. Now, $\alpha^p = \alpha \cdot \alpha^{p-1} >$ /ind. hyp./ $> \alpha \cdot (\alpha^{p-3} + \alpha^2) = \alpha^{p-2} + \alpha^3 > \alpha^{p-2} + \alpha^2$. Thus

we will always benefit from partitioning a domain if $d > 5$ which implies that the optimal partition P^* will only consist of partitions of sizes 2, 3, 4 and 5.

Assume $P^* = [3, ...]$ and note that since $d > 5$, $P^* = [3, a, ...]$ where $a \in \{2, 3, 4, 5\}$. It is easy to check that $T([3, 2]) > T([5])$, $T([3, 3]) > T([4, 2])$, $T([3, 4]) > T([5, 2])$ and $T([3, 5]) > T([4, 4])$ so P^* contains no 3-partition.

Step 4: P is optimal when $d > 5$.
We note that P^* cannot contain more than one 2-partition since $T([4]) < T([2, 2])$ and it cannot contain more than one 5-partition since $T([2, 4, 4]) < T([5, 5])$. Finally, P^* cannot contain fewer 2 and 5 partitions than P, since otherwise, it would not be a partition of an arbitrary domain. The only alternative is that P^* has the same number of 2- and 5-partitions as P, so $P = P^*$ and P is optimal. $\square$

Consequently, the resulting algorithm has the following time complexity:

- $\mathcal{O}\left((d/4 \cdot \alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6224d)^n\right)$ if $d \equiv 0 \pmod 4$
- $\mathcal{O}\left((\alpha^5 + \lfloor d/4 - 1 \rfloor \cdot \alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6254d)^n\right)$ if $d \equiv 1 \pmod 4$
- $\mathcal{O}\left((\alpha + \lfloor d/4 \rfloor \cdot \alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6243d)^n\right)$ if $d \equiv 2 \pmod 4$
- $\mathcal{O}\left((\alpha + \alpha^5 + \lfloor d/4 - 1 \rfloor \cdot \alpha^4)^n\right) \subseteq \mathcal{O}\left((0.6262d)^n\right)$ if $d \equiv 3 \pmod 4$

For large domains, the term $d/4 \cdot \alpha^4$ dominates the time complexity, since the number of 4-partitions increase with d. Consequently, the bounds in the list will all approach $\mathcal{O}\left((d/4 \cdot \alpha^4)^n\right) \approx \mathcal{O}\left((0.6222d)^n\right)$ as d grows.

4 Algorithm for $\#kCOL$

We will now present an algorithm for the $\#kCOL$ problem. We prefer to present this algorithm using graph-theoretic notation instead of Csp notation since this enables us to use familiar concepts such as induced subgraphs and independent sets. We start with the necessary graph-theoretic preliminaries.

A *graph* G consists of a set $V(G)$ of *vertices*, and a set $E(G)$ of *edges*, where each element of E is an unordered pair of vertices. The *size* of a graph G, denoted $|G|$, is the number of vertices. A k*-colouring* of G is a function $f : V(G) \to \{1, \ldots, k\}$ such that for all $v, w \in V(G)$, if $C(v) = C(w)$ then $(v, w) \notin E(G)$; that is, no adjacent vertices have the same colour. If G is a graph and $S \subseteq V(G)$, the graph $G|S$ has vertex set S and $E(G|S) = \{(u, v) \in E(G) \mid u, v \in S\}$, is called the *subgraph of* G *induced by* S. We write $G - S$ to denote the graph $G|(V(G) - S)$.

Theorem 3. *Let $A_1, \ldots, A_m$ be algorithms for $\#k_1COL, \ldots, \#k_mCOL$, respectively, such that algorithm A_i runs in $\mathcal{O}\left(\alpha_i^n\right)$ time. Choose p and a partitioning $P = \{P_1, \ldots, P_k\}$ of $\{1, \ldots, p\}$ such that $|P_i| \in \{k_1, \ldots, k_m\}$ for all i. Then, there exists an algorithm for $\#pCOL$ with a time complexity of $\mathcal{O}\left(((|P| - 1) + \max_{P_i \in P} \alpha_{|P_i|})^n\right)$.*

Proof. We claim that the algorithm presented in Figure 2 correctly solves the $\#pCOL$ problem. To prove this, arbitrarily choose an instance G of $\#pCOL$, let F be the set of all total functions $V(G) \to P$ and choose one $f \in F$. We begin by proving that lines 4–8 correctly computes the number of p-colourings satisfying the following condition: variable v can only be coloured by the colours in $f(v)$.

Assume without loss of generality that $P_1 = \{1, \ldots, k\}$ for some $k > 0$. Let $G_1 = G|\{v \in V(G) \mid f(v) = P_1\}$ and $G_{\geq 2} = G|\{v \in V(G) \mid f(v) \neq P_1\}$. We show that $\#pCOL(G) = \#|P_1|COL(G_1) \cdot \#(p - |P_1|)COL(G_2)$. By using this fact inductively, it follows that c' contains the number of solutions relative to f after completion of lines 4-8. To prove this, arbitrarily choose one $|P_1|$-colouring M_1 of G_1 and one $p - |P_1|$-colouring M_2 of G_2. Now, consider the function

$$M(v) = \begin{cases} M_1(v) & \text{if } v \in G_1 \\ M_2(v) + k & \text{otherwise} \end{cases}$$

We prove that M is a p-colouring of G. Arbitrarily choose an edge $(x, y) \in E(G)$. If $x \in G_1$ and $y \notin G_1$, then $M(x) \neq M(y)$ due to the construction of M. Otherwise, x and y are both in G_1 or $G_{\geq 2}$ and the choice of M_1, M_2 guarantees that $M(x) \neq M(y)$. This leads to the following conclusion: any $|P_1|$-colouring of G_1 can be combined with any $p - |P_1|$-colouring of G_2 and $\#pCOL(G) = \#|P_1|COL(G_1) \cdot \#(p - |P_1|)COL(G_2)$.

Finally, it is easy to see that it is correct to add the numbers together as is done in line 9: Let M be an arbitrary p-colouring of G. The fact that P is a partitioning guarantees that there exists exactly one function $f \in F$ such that $M(v) \in f(v)$ for all $v \in V(G)$. Consequently, each model will be counted exactly once (due to the correctness of algorithms $A_1, \ldots, A_m$) and the algorithm is correct.

The running time of the algorithm is bounded by

$$\mathcal{O}\left(\sum_{f \in F} \sum_{P_i \in P} \alpha_{|P_i|}^{|\{v \in V(G); f(v) = P_i\}|}\right) = \text{/reformulation/}$$

$$\mathcal{O}\left(\sum_{n_1 + \ldots + n_{|P|} = n} \binom{n}{n_1 \cdots n_{|P|}} \cdot \left(\sum_{P_i \in P} \alpha_{|P_i|}^{n_i}\right)\right) = \text{/alg. manipulation/}$$

$$\mathcal{O}\left(\sum_{n_1 + \ldots + n_{|P|} = n} \sum_{P_i \in P} \binom{n}{n_1 \cdots n_{|P|}} \cdot \alpha_{|P_i|}^{n_i}\right) = \text{/alg. manipulation/}$$

$$\mathcal{O}\left(\sum_{P_i \in P} \sum_{n_1 + \ldots + n_{|P|} = n} \binom{n}{n_1 \cdots n_{|P|}} \cdot \alpha_{|P_i|}^{n_i}\right) = \text{/multiplication by 1/}$$

```
input: undirected graph G
c := 0
for every total function f : V(G) → P  do
  c' := 1
  for P_i ∈ P  do
    G' := G|{v ∈ V(G) | f(v) = P_i}
    c' := c' · #|P_i|COL(G')
  end for
  c := c + c'
end for
return c
```

Fig. 2. Algorithm for $\#pCOL$.

$$\mathcal{O}\left(\sum_{P_i \in P} \sum_{n_1+\ldots+n_{|P|}=n} \binom{n}{n_1 \cdots n_{|P|}} \cdot \alpha_{|P_i|}^{n_i} \cdot \prod_{\substack{j=1 \\ j \neq i}}^{|P|} 1^{n_j}\right) = \text{/multinomial th./}$$

$$\mathcal{O}\left(\sum_{P_i \in P} ((|P|-1) + \alpha_{|P_i|})^n\right) = \mathcal{O}\left((|P| - 1 + \max_{P_i \in P} \alpha_{|P_i|})^n\right)$$

□

We note that #2COL can be solved in polynomial time since the number of 2-colourings of a 2-colourable graph G equals 2^c where c is the number of connected components in G. In the next section, we show that #3COL can be solved in $\mathcal{O}(\beta^n)$ time where $\beta \approx 1.7879$.

Now, from Theorem 3 it follows that as k grows, the number of partitions dominate the time complexity of the algorithm. For example, for $k = 7$, we could use the partition $[3, 2, 2]$ which gives a running time of $\mathcal{O}(3.7879^n)$. If we instead use the partitioning $[3, 4]$ where the #4COL problem is solved using the partition $[2, 2]$, we would get an $\mathcal{O}(((2-1) + (2-1+1))^n) = \mathcal{O}(3^n)$ time algorithm which is significantly faster. Combining the idea of minimising the number of partitions with Theorem 3 and the algorithms for $\#k$COL, $2 \leq k \leq 3$, yields the following result.

Theorem 4. *There is an algorithm for solving the $\#k$COL problem in $O((c_k)^n)$ time, where, for some $i \in \mathbb{N}$,*

$$c_k = \begin{cases} \lfloor \log_2 k \rfloor & \text{if } k = 2^i \\ \lfloor \log_2 k \rfloor + (\beta - 1) & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ \lfloor \log_2 k \rfloor + 1 & \text{if } 2^i + 2^{i-1} < k < 2^{i+1} \end{cases}$$

Proof. We recursively use the partitioning $[\lfloor\frac{k}{2}\rfloor, \lceil\frac{k}{2}\rceil]$ with the algorithms for #2COL and #3COL as base cases. By Theorem 3, the resulting algorithm runs in $\mathcal{O}(c_k^n)$ time where c_k is defined by the recursion $c_2 = 1, c_3 = \beta$ and $c_k = 1+c_{\lceil k/2\rceil}$ Using $\beta \approx 1.7879$ from Section 5 and solving this gives the result. It remains to show that this recurrence has the solution stated in the theorem:

First, let $k = 2^i$. By iterating the recursion, we get $c_k = 1+c_{k/2} = 2+c_{k/4} = \ldots = (i-1)+c_{k/2^{i-1}} = (i-1)+c_2 = i = \log_2 k$.

Now let $2^i < k \leq 2^i + 2^{i-1}$. For $i = 2$, this amounts to $4 < k \leq 6$, and $\lceil k/2\rceil = 3$, and, since $c_3 = \beta$ by definition, $c_k = \beta + 1 = \lfloor\log_2 k\rfloor + \beta - 1$. For $i > 2$, $c_{\lceil k/2\rceil} = \lfloor\log_2 k/2\rfloor + \beta - 1 = \lfloor\log_2 k\rfloor + \beta - 2$ so $c_k = 1 + c_{\lceil k/2\rceil} = 1 + \lfloor\log_2 k\rfloor + \beta - 2 = \lfloor\log_2 k\rfloor + \beta - 1$.

Finally, we have $2^i + 2^{i-1} < k < 2^{i+1}$. For $i = 2$, $k = 7$, and $\lceil 7/2\rceil = 4$, which gives $c_7 = c_4 + 1 = 2 + 1 = \lfloor\log_2 7\rfloor + 1$. If $2^i + 2^{i-1} < k < 2^{i+1}$, $2^{i-1} + 2^{i-2} < k/2 \leq 2^i$, and $c_{\lceil k/2\rceil} = \lfloor\log_2 k/2\rfloor + 1 = \lfloor\log_2 k\rfloor$ if $\lceil k/2\rceil < 2^i$. If $\lceil k/2\rceil = 2^i$, then, as was shown earlier, $c_{\lceil k/2\rceil} = \lfloor\log_2 k\rfloor$. Consequently, $c_k = 1 + c_{\lceil k/2\rceil} = 1 + \lfloor\log_2 k\rfloor$. □

5 Algorithm for #3*COL*

We now present an $\mathcal{O}(1.7879^n)$ time algorithm for counting the number of 3-colourings of a graph. We denote the three possible colours R, G and B. An *independent set* S of G is a subset of $V(G)$, such that for every pair $v, w \in S \rightarrow (v, w) \notin E(G)$. The *neighbourhood* of a vertex v is the set of all adjacent vertices, $\{w \mid (v, w) \in E(G)\}$, denoted $Nbd(v)$

Given a graph G, we associate a variable with each vertex for keeping its colour and we let $G[x := X]$ denote the graph G with the colour of vertex x changed to X.

We define an $R\{G/B\}$ *assignment* of the graph G as a total function $f : V(G) \rightarrow \{R, GB\}$. We say that an $R\{G/B\}$ assignment f is *refineable* to a 3-colouring of G iff for each of the vertices v having colour GB, we can assign $v := G$ or $v := B$ in such a manner that we obtain a 3-colouring of G. We note that having an $R\{G/B\}$ assignment for G which is refineable to a 3-colouring of G, is equivalent to the assignment having the following properties:

P1. the vertices with colour R form an independent set;

P2. the induced subgraph of vertices with colour GB is 2-colourable.

Obviously, these conditions can be checked in polynomial time. We can also count the number of possible refinements of an $R\{G/B\}$ assignment: consider the graph $G' = G|\{v \in V(G) \mid f(v) = GB\}$ and note that the number of refinements equals 2^c where c is the number of connected components in G'. Given an $R\{G/B\}$ assignment f, let $Count_2(G, f)$ denote this number (which is easily computable in polynomial time).

1 **algorithm** #3C-1(G)
2 **if** all $v \in V(G)$ are $R\{G/B\}$-coloured **then**
3   **return** $Count_2(G)$
4 **elsif** there exists an uncoloured vertex x with an uncoloured neighbour y **then**
5   **return** #3C-1($G[x := R, y := GB]$)+#3C-1($G[x := GB]$)
6 **else** cycle through all R/GB assignments of the uncoloured vertices, apply $Count_2$ on each graph and return the total number of 3-colourings.
7 **end if**

Fig. 3. Algorithm #3C-1

1 **algorithm** #3C-2(G)
2 $c := 0$
3 **for** every 3-colouring f of $G - I$ **do**
4   $c := c + \prod_{v \in I}(3 - |\{f(w) \mid w \in Nbd(v)\}|)$
5 **end for**
6 **return** c

Fig. 4. Algorithm #3C-2

We are now ready to present the algorithm. Let $\phi = (1 + \sqrt{5})/2$ and let $C \approx 0.4711$ be the unique real positive root of the equation $\phi^{1-C} \cdot 2^C = 3^{1-C}$. Begin by identifying an independent set I in G of maximum size using, for instance, Beigel's algorithm [4]. If $|I| \leq C \cdot |G|$, then apply the algorithm in Figure 3. Otherwise, apply the algorithm in Figure 4.

To see that algorithm #3C-1 is correct, we note that the algorithm considers all $R\{G/B\}$-assignments that can be refined to a 3-colouring. In line 3, an uncoloured vertex x with an uncoloured neighbour y is chosen. In the first recursive branch, x is assigned the colour R which implies that y must be coloured GB. In the other branch, x is coloured GB and this choice does not restrict the possible colourings of y. In line 4, the algorithm exhaustively considers all $R\{G/B\}$-assignments of the uncoloured vertices.

The correctness of #3C-2 can proved as follows: For each 3-colouring f of $G - I$, we claim that

$$\prod_{v \in I}(3 - |\{f(w) \mid w \in Nbd(v)\}|)$$

is the number of ways f can be extended to a 3-colouring of G. Assume for instance that $v \in I$ has three neighbours x, y, z that are coloured with R, G and B, respectively. Then, $3 - |\{f(w) \mid w \in Nbd(v)\}|)$ equals 0 which is correct since f cannot be extended in this case. It is easy to realise that the expression gives the right number of possible colours in all other cases, too. Since I is an independent set, we can simply multiply the numbers of allowed colours in order to count the number of possible extensions of f.

Finally, we consider the time complexity of our algorithm for #3COL. Assume n is the number of vertices in the input graph G. Beigel's [4] algorithm for finding independent sets of maximum size runs in $\mathcal{O}(1.2226^n)$ time. We show below that the worst-case running times of algorithms #3C-1 and #3C-2 are in $\mathcal{O}(1.7879^n)$ which clearly dominates Beigel's algorithm.

Algorithm #3C-1: In the analysis of this algorithm we will encounter recurrences of the form $T(n) \leq \sum_{i=1}^{k} T(n-r_i) + \text{poly}(n)$. They satisfy $T(n) \in \mathcal{O}(\tau(r_1,\ldots,r_k)^n)$ where $\tau(r_1,\ldots,r_k)$ is the largest, real-valued root of the equation $1 - \sum_{i=1}^{k} x^{-r_i} = 0$, see Kullman [13]. This bound does not depend on the polynomial factor $\text{poly}(n)$ or the boundary conditions $T(1) = b_1, \ldots, T(k) = b_k$.

If line 4 of the algorithm is not reached, it is straightforward to calculate its running time: lines 2 and 3 satisfy the recursive equation $T(n) \leq T(n-1) + T(n-2) + \text{poly}(n)$ and $T(n) \in \mathcal{O}(\phi^n)$ where $\phi = (\sqrt{5}+1)/2$ is the largest, real-valued root of the equation $1 - x - x^2 = 0$.

We continue by studying line 4 in the algorithm. If this case is reached, the uncoloured vertices form an independent set I' in G. Consequently, $T(n) \in \mathcal{O}\left((2^p \cdot \phi^{(1-p)})^n\right)$ where $p = |I'|/n$. Since I is a maximum independent set in G, it follows that $|I'| \leq |I| \leq C \cdot n$. Consequently, the worst case of the algorithm appears when $g(p) = 2^p \cdot \phi^{(1-p)}$ is maximised under the constraint $p \leq C$. Since $g(p)$ is strictly increasing on $[0, C]$, $g(p)$ is maximised when $p = C$. In this case, the algorithm runs in $\mathcal{O}\left((2^C \cdot \phi^{(1-C)})^n\right) \approx \mathcal{O}(1.7879^n)$ time.

Algorithm #3C-2: We know that $|I| \geq C \cdot n$. Let p satisfy $|I| = p \cdot n$. The number of assignments considered is $3^{n-|I|} = (3^{1-p})^n$. Since the function $g(p) = 3^{(1-p)}$ is strictly decreasing when $p > C$, the largest number of assignments we need to consider appears when p is close to C. In this case, the algorithm runs in $\mathcal{O}\left((3^{1-C})^n\right) \approx \mathcal{O}(1.7879^n)$ time.

6 Acknowledgments

The authors would like to thank Johan Thapper for providing valuable comments on this paper. Ola Angelsmark is supported in part by CUGS – the National Graduate School in Computer Science, Sweden – and in part by the *Swedish Research Council* (VR) under grant 621-2002-4126. Peter Jonsson is partially supported by VR under grant 221-2000-361.

References

1. O. Angelsmark, P. Jonsson, S. Linusson, and J. Thapper. Determining the number of solutions to a binary CSP instance. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-2002)*, Sept. 2002.
2. B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123, Mar. 1979.

3. R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence AAAI/IAAI*, pages 157–162, 2000.
4. R. Beigel. Finding maximum independent sets in sparse and general graphs. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-1999)*, pages 856–857, 1999.
5. A. Bulatov and V. Dalmau. Towards a dichotomy theorem for the counting constraint satisfaction problem. Technical report, Computing Laboratory, Oxford University, 2003. Available from `web.comlab.ox.ac.uk/oucl/work/andrei.bulatov/counting.ps`.
6. N. Creignou and M. Hermann. Complexity of generalized satisfiability counting problems. *Information and Computation*, 125:1–12, 1996.
7. V. Dahllöf, P. Jonsson, and M. Wahlström. Counting satisfying assignments in 2-SAT and 3-SAT. In *Proceedings of the 8th Annual International Computing and Combinatorics Conference (COCOON-2002), Singapore*, pages 535–543, Aug. 2002.
8. A. Darwiche. On the tractable counting of theory models and its applications to truth maintenance and belief revision. *Journal of Applied Non-Classical Logic*, 11(1–2):11–34, 2001.
9. O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64, 1991.
10. M. Dyer and C. Greenhill. The complexity of counting graph homomorphisms. *Random Structures and Algorithms*, 17:260–289, 2000.
11. P. G. Jeavons, D. A. Cohen, and J. K. Pearson. Constraints and universal algebra. *Annals of Mathematics and Artificial Intelligence*, 24:51–67, 1998.
12. P. Jégou. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In *Proceedings of the 11th (US) National Conference on Artificial Intelligence (AAAI-93)*, pages 731–736, Washington DC, USA, July 1993. American Association for Artificial Intelligence.
13. O. Kullman. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999.
14. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, Spring 1992.
15. M. L. Littman, T. Pitassi, and R. Impagliazzo. On the complexity of counting satisfying assignments. Unpublished manuscript, 2001.
16. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
17. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.
18. T. K. Satish Kumar. A model counting characterization of diagnoses. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis*, pages 70–76, 2002.
19. L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.
20. L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
21. M. Wahlström. An algorithm for #2SAT. Technical report, Department of Computer Science, Linköping, Sweden, 2003. In preparation. Preliminary version available from `http://www.ida.liu.se/~magwa/research/num2sat.ps`.