Determining the number of solutions to binary CSP instances

Ola Angelsmark[1], Peter Jonsson[1], Svante Linusson[2], and Johan Thapper[2]

[1] Department of Computer and Information Science
Linköpings Universitet
S-581 83 Linköping, Sweden
{olaan,petej}@ida.liu.se
[2] Department of Mathematics
Linköpings Universitet
S-581 83 Linköping, Sweden
{linusson,jotha}@mai.liu.se

Abstract. Counting the number of solutions to CSP instances has applications in several areas, ranging from statistical physics to artificial intelligence. We give an algorithm for counting the number of solutions to binary CSPs, which works by transforming the problem into a number of 2-SAT instances, where the total number of solutions to these instances is the same as those of the original problem. The algorithm consists of two main cases, depending on whether the domain size d is even, in which case the algorithm runs in $\mathcal{O}(1.3247^n \cdot (d/2)^n)$ time, or odd, in which case it runs in $\mathcal{O}(1.3247^n \cdot ((d^2+d+2)/4)^{n/2})$ if $d = 4 \cdot k + 1$, and $\mathcal{O}(1.3247^n \cdot ((d^2+d)/4)^{n/2})$ if $d = 4 \cdot k + 3$. We also give an algorithm for counting the number of possible 3-colourings of a given graph, which runs in $\mathcal{O}(1.8171^n)$, an improvement over our general algorithm gained by using problem specific knowledge.

1 Introduction

Constraint satisfaction problems (CSPs), first described by Montanari [17], allows for natural descriptions of problems in a wide array of fields. These include such varied areas as machine vision, scheduling, temporal reasoning, graph problems, floor plan design, machine design and manufacturing, and diagnostic reasoning [13]. In particular, it has proven invaluable to artificial intelligence.

Posed with an instance of a CSP problem, one can ask two questions. At first, the question "Does there exist a solution?" might seem the most natural one to ask – and indeed it is the most common one. This is what is usually called the *decision problem.* The second question which might arise is "How many solutions are there?" – known as the *counting problem.* In this paper we will study the latter question.

For (d, l)-CSPs, *i.e.*, CSPs where the variables have domain size d and the constraints have arity l, the decision problem is known to be NP-complete [15], and this may, or may not, imply the non-existence of a polynomial-time algorithm for solving it (depending, as always, on the truth of the equation $P = NP$.)

We will avoid making any assumptions about this, since it has no impact on the results we present. The numerous applications of CSPs have, however, caused intense research, *e.g.*, identifying tractable subclasses of problems [3, 18], and constructing exact algorithms [9, 10, 20].

The corresponding counting problem belongs to a class known as #P (introduced by Valiant [21, 22]) 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 [19]. In fact, for every fixed $\varepsilon > 0$, approximating the number of solutions to a binary CSP within $2^{n^{1-\varepsilon}}$ is NP-hard. There exists, however, randomised approximation algorithms which run in polynomial time for certain restricted cases, *e.g.*, finding an estimation of the number of k-colourings of graphs with low degree [12].

Until quite recently, not much attention has been given to the problem of counting solutions. The focus has been on solving the decision problem, rather than the corresponding counting problem, and the algorithms are often probabilistic [9, 10, 20]. Lately, however, several papers have discussed the complexity of counting problems [4, 14], and a number of algorithms have been developed [2, 5–7]. One reason for this renewed interest for these kinds of problems may be due to the multitude of applications for counting counterparts of several well-studied decision problems. For example, many problems in artificial intelligence can be reduced to counting the number of models to a formula [19]. 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, and determining the number of graph homomorphisms from one graph to another has important applications in statistical physics [12, 23].

Intuitively, it seems reasonable that decision problems which are known to be NP-complete, have corresponding counting problems which are #P-complete, and indeed it has been shown that this is the case for satisfiability problems [4]. However, several decision problems for which polynomial time algorithms are known, also have associated counting problems which are #P-complete. 2-SAT belongs to this class of 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[1], and the counting problem for 3-colourability of graphs, denoted #3COL.

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

1. Create 2-SAT 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 reason the algorithm looks this way is twofold; First of all, the fastest known algorithm for $(3, 2)$-CSP (and $(4, 2)$-CSP), suggested by Eppstein [9],

[1] The necessary definitions for the CSP notions used in this paper are found in Section 2, while the graph theoretic notions needed are located in Section 4

works by recursively breaking the problem down into 2-SAT instances. This lead us to believe that this approach would work well for the corresponding counting problem. Secondly, by moving over to 2-SAT, we gain access to the fast algorithms developed for #2-SAT [6, 7, 14, 24].

We chose to transform the problem into #2-SAT mainly because of time complexity reasons. Using #3-SAT did not lead to a speed-up, probably due to the rather high time complexity of the #3-SAT algorithm. The fastest known #2-SAT algorithm [6] runs in $\mathcal{O}(1.3247^n)$. Had we instead moved over to #3-SAT, then the fastest known algorithm would have been $\mathcal{O}(1.6894^n)$, which is significantly slower. It is possible that a faster algorithm could be achieved using the #3-SAT algorithm, but if and how this could be done remains an open question.

The running time of the algorithm (we omit polynomial factors here and throughout the paper) is as follows:

- $\mathcal{O}(1.3247^n \cdot (d/2)^n)$, if $d = 2 \cdot k, k > 0$,
- $\mathcal{O}(1.3247^n \cdot ((d^2 + d + 2)/4)^{n/2})$, if $d = 4 \cdot k + 1, k > 0$, and
- $\mathcal{O}(1.3247^n \cdot ((d^2 + d)/4)^{n/2})$, if $d = 4 \cdot k + 3, k \geq 0$.

This division into cases will be explained in detail in Section 3, but in short, the algorithm works by dividing the domain into pairs of values. For even sized domains, this is trivial, but when the domain is of odd size, this can, of course, not be as easily done. One solution would be to add a 'dummy element' to the domain, a value which would then be discarded if found in a solution, but this would give an increased complexity for all odd sized domains, thus we have focused on efficiently dividing the domain. Numerical time complexities for some domain sizes are presented in Table 1, where, for comparative reasons, the results for Eppstein's and Feder & Motwani's algorithms for the corresponding decision problem have been included. If we had chosen to add an element to domains of odd size, the complexities given for, say, 3, 5 and 7, would have been the same as those for 4, 6 and 8.

d	Eppstein [9]	Feder & Motwani [10]	Our result
3	1.3645^n	1.8171^n	2.2944^n
4	1.8072^n	2.2134^n	2.6494^n
5	2.2590^n	2.6052^n	3.7468^n
6	2.7108^n	2.9938^n	3.9741^n
7	3.1626^n	3.3800^n	4.9566^n
8	3.6144^n	3.7644^n	5.2988^n
10	4.5180^n	4.5287^n	6.6235^n
20	9.0360^n	8.3044^n	13.2470^n

Table 1. Time complexities for solving the decision problem $(d, 2)$-CSP and the corresponding counting problem $\#(d, 2)$-CSP.

Asymptotically, the algorithm approaches $\mathcal{O}((0.6624d)^n)$, where d is the domain size and n the number of variables. If we compare this to the (probabilistic) algorithm of Eppstein [9], which has a complexity of $\mathcal{O}((0.4518d)^n)$, the gap is not very large. For domain sizes of 10 elements or more, the algorithm presented by Feder & Motwani [10] is faster than Eppstein's. Furthermore, given the modularity of the algorithm, if a faster method for counting the number of solutions to a #2-SAT formula is found, it would be easy to 'plug into' our algorithm, thus improving the time complexity with no extra work.

The second part of this paper contains an algorithm for #3COL, which runs in $\mathcal{O}(1.8171^n)$ time. This is an improvement in complexity compared to our general algorithm, which is gained by using problem specific knowledge, in this case graph-theoretical properties. In particular, rather than transforming the problem into #2-SAT, we move over to #2COL, which is solvable in polynomial time.

The paper has the following organisation: Section 2 contains the basic definitions needed. Section 3 contains the algorithm for counting solutions to a binary CSP, while Section 4 an algorithm for the #3COL problem, together with the graph-theoretic notions it utilises. Section 5 summarises the discussion and suggests future work.

2 Preliminaries

A *binary constraint satisfaction problem ((d,2)-CSP)*, is a triple $\langle V, D, C\rangle$, with V a finite set of variables, D a 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 triple xRy, where $x, y \in V$, and $R \in D \times D$. To simplify the discussion, we will assume $|V|$ to be an even number, denoted n. A *solution* to a CSP instance is a function $f : V \rightarrow D$, such that for each constraint xRy, $(f(x), f(y)) \in R$. Given a CSP instance, the computational problem is to decide whether the instance has a solution, or not. The corresponding *counting problem* is to determine how many solutions the instance has.

A *2-SAT* formula is a sentence consisting of the conjunction of a number of clauses, where each clause contains at most two literals and 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 [1], whereas the *#2-SAT problem* is to decide *how many* solutions a given formula has. The currently best known algorithm runs in $\mathcal{O}(1.3247^n)$ time [6].

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

The main points of the algorithm, which were mentioned in Section 1, will now be discussed in more detail. To begin with, we give a simplified description, which would give a much worse time complexity than what we eventually will get. This speedup is gained by reducing the number of propositional variables from $2 \cdot n$ to n in the final step.

Assume we have a binary CSP instance P, with domain size d and n variables. The problem can be transformed into a number of instances of 2-SAT, $I_0, I_1, \ldots, I_m$, where m depends on the size of the domain and the number of variables, such that the number of solutions to P equals the total number of solutions to all these instances. In each instance I_k, the propositional variables correspond to an assignment of a value to a variable in the original problem. For a given variable $x \in V$, and a domain value $e \in D$, the propositional variable x_e is true iff x is assigned the value e.

A 2-SAT instance I_k consists of two parts: First, we have clauses corresponding to the constraints in the original problem. For example, say we have the constraint $x \neq y$. Since x and y will never have the same value in a solution, we get the clauses $(\neg x_e \vee \neg y_e)$ for all $e \in D$.

The remaining clauses are constructed by dividing the domain into pairs. For domains of even size, this is straightforward, and we get $d/2$ pairs of values, where each value appears exactly once. Since a solution to the instance implies a certain value for each variable, we know that this value will be represented in one pair. For example, say we have a pair (e_1, e_2), and a variable x. We then add the clauses $(x_{e_1} \vee x_{e_2})$, $(\neg x_{e_1} \vee \neg x_{e_2})$ and, for all $e \in D$ with $e \neq e_1, e \neq e_2$, we add the clause $(\neg x_e)$. This reads as: x can assume exactly one of the values e_1 or e_2, and no other. For each variable, we get $d/2$ possible assignments, or propositional variables, thus we need $(d/2)^n$ instances to cover all of the original problem.

	x_1, x_2	x_1, x_2	x_1, x_2	x_1, x_3	x_1, x_3	x_1, x_3	x_2, x_3	x_2, x_3	x_2, x_3
x y	y_1, y_2	y_1, y_3	y_2, y_3	y_1, y_2	y_1, y_3	y_2, y_3	y_1, y_2	y_1, y_3	y_2, y_3
1 1	X	X	-	X	X	-	-	-	-
1 2	X	-	X	X	-	X	-	-	-
1 3	-	X	X	-	X	X	-	-	-
2 1	X	X	-	-	-	-	X	X	-
2 2	X	-	X	-	-	-	X	-	X
2 3	-	X	X	-	-	-	-	X	X
3 1	-	-	-	X	X	-	X	X	-
3 2	-	-	-	X	-	X	X	-	X
3 3	-	-	-	-	X	X	-	X	X

Table 2. The case of $D = \{1, 2, 3\}, V = \{x, y\}$. ($x_i, x_j$ is short for $x_i \vee x_j$)

A domain of odd size can, naturally, not be divided evenly into pairs. One possible solution to this problem is to simply add a 'dummy element' to the domain, a value we ignore in the subsequent results. However, this would increase the complexity of the algorithm unnecessarily. For instance, it would be equally hard to solve problems with domain size 3 as 4. Instead, we use Proposition 1 to help us with the division of the domain. Note that, unlike the case with even sized domains, where we only considered pairs of propositional variables, we now consider two variables at a time, each with its associated pair of assignments, and we assume we have an even number of variables in the problem. Proposition 1 tells us that we need $(d^2 + d + 2)/4$ such cases to cover all possible assignments if the domain size d is on the form $4 \cdot k + 1$, and $(d^2 + d)/4$, if $d = 4 \cdot k + 3$. Table 2 shows how the situation looks for domains of size 3. The boxed columns in the table clearly cover all possible assignments, but, as can be seen, there are overlaps; some assignments occur in more than one pair of variables. The propositional variables x_1, y_1 occur in columns 2 and 4, x_2, y_3 in columns 2 and 9, and x_3, y_2 are found in both columns 4 and 9. We need to make sure that these assignments are not part of more than one solution, thus to avoid counting these twice, we add the clause $\neg(x_1 \wedge y_1)$, which is equivalent to $(\neg x_1 \vee \neg y_1)$, to one of the instances containing these assignments. In the general case, when an overlap containing the propositional variables x_i and y_j is found, we add the clause $(\neg x_i \vee \neg y_j)$ to all but one of the instances containing this overlap. Had we considered more than two variables at a time, we would have been forced to use more than two propositional variables per clause, thus leaving the 2-SAT setting, and the overall algorithm would have had a much higher time complexity.

For even sized domains, we now have n sets, each containing $d/2$ pairs, while for odd sized domains, we have $n/2$ sets of 4-tuples, with $(d^2 + d + 2)/4$ or $(d^2 + d)/4$ elements, depending on whether $d = 4 \cdot k + 1$ or $d = 4 \cdot k + 3$. By combining the clauses given by one element from each of these sets and the clauses coming from the constraints, we get a set of 2-SAT instances corresponding to the original problem, and each of these instances contains $n \cdot d$ propositional variables. Note that, for even sized domains, each pair give rise to $d - 2$ clauses on the form $\neg x_c$, hence we can in each instance, using unit propagation, remove $n \cdot (d - 2)$ propositional variables, and get $2 \cdot n$ in each instance. For odd sized domains, the situation is similar. Each 4-tuple give rise to $2 \cdot d$ propositional variables, out of which $2 \cdot (d - 2)$ can be removed through unit propagation, leaving 4. Since we have $n/2$ sets to combine elements from, we get $2 \cdot n$ propositional variables per instance for this case too. In all instances, among these $2 \cdot n$ variables, there are n pairs where both cannot be true in a solution, since a variable in the original problem cannot be assigned two different values. For each such pair x_{e_1}, x_{e_2}, take a fresh variable ξ_{x_e}, with the interpretation that ξ_{x_e} is true iff x_{e_1} is true and x_{e_2} is false, and vice versa, and replace all occurrences of x_{e_1} with ξ_{x_e} and all occurrences of x_{e_2} with $\neg\xi_{x_e}$. Through this, we get n propositional variables in each instance, both for even and odd sized domains.

Proposition 1. *Let x, y be a pair of variables taking their values from the domain D, with $|D| = d$. For each odd d, define the set C_d recursively as:*

$$\begin{aligned}
C_1 &= \{\ (x_1 \vee x_1) \wedge (y_1 \vee y_1)\} \\
C_3 &= \{\ (x_1 \vee x_2) \wedge (y_1 \vee y_3), \\
&\qquad (x_1 \vee x_3) \wedge (y_1 \vee y_2), \\
&\qquad (x_2 \vee x_3) \wedge (y_2 \vee y_3)\}
\end{aligned}$$

and for $d \geq 5$,

$$C_d = C_{d-4} \cup A_{1,d} \cup A_{2,d} \cup A_{3,d}$$

where

$$A_{1,d} = \bigcup_{i=1}^{(d-3)/2} \{(x_{d-3} \vee x_{d-2}) \wedge (y_{2i-1} \vee y_{2i}), (x_{d-1} \vee x_d) \wedge (y_{2i-1} \vee y_{2i})\}$$

$$A_{2,d} = \bigcup_{j=1}^{(d-3)/2} \{(x_{2j-1} \vee x_{2j}) \wedge (y_{d-3} \vee y_{d-2}), (x_{2j-1} \vee x_{2j}) \wedge (y_{d-1} \vee y_d)\}$$

$$\begin{aligned} A_{3,d} = \{ &(x_{d-2} \vee x_{d-1}) \wedge (y_{d-2} \vee y_d), \\ &(x_{d-2} \vee x_d) \wedge (y_{d-2} \vee y_{d-1}), \\ &(x_{d-1} \vee x_d) \wedge (y_{d-1} \vee y_d)\} \end{aligned}$$

Then C_d covers all possible assignments of x and y and this can not be done with fewer than $|C_d|$ cases, where

$$|C_d| = \begin{cases} (d^2 + d + 2)/4 & \text{if } d = 4k + 1 \\ (d^2 + d)/4 & \text{if } d = 4k + 3. \end{cases}$$

Proof. C_1 and C_3 are easily verified. Figure 1 shows how the recursive definition of C_d is constructed. Note that in this figure a case '$(x_i \vee x_{i+1}) \wedge (y_j \vee y_{j+1})$' is represented by a 2×2 square covering the values in $[i, i+1] \times [j, j+1]$. $A_{1,d}$ and $A_{2,d}$ then cover the rectangles $[d-3, d] \times [1, d-3]$ and $[1, d-3] \times [d-3, d]$ respectively. $A_{3,d}$ is the same as C_3, but translated to cover the values in $[d-2, d] \times [d-2, d]$. Combined with C_{d-4} this proves that all possible assignments are covered.

For $d = 4k + 1$ we have

$$\begin{aligned} |C_d| &= |C_{d-4}| + |A_{1,d}| + |A_{2,d}| + |A_{3,d}| = \\ &= |C_{d-4}| + 2(d-3)/2 + 2(d-3)/2 + 3 = |C_{d-4}| + 2d - 3 = \\ &= \ldots = |C_1| + \sum_{i=1}^{k} (2(4i+1) - 3) = (d^2 + d + 2)/4 \end{aligned}$$

and for $d = 4k + 3$ we have

$$|C_d| = |C_{d-4}| + 2d - 3 = \ldots = |C_3| + \sum_{i=1}^{k} (2(4i+3) - 3) = (d^2 + d)/4.$$

We finally prove that there can be no smaller set of cases covering all assignments. Every case covers at most 4 different assignments and there are d^2 possible assignments. Since d is odd, we note that for each value of x there must be a value of y such that this assignment is covered by at least two different cases. Therefore there must be at least d assignments which are covered more than once. This shows that $\lceil (d^2 + d)/4 \rceil$ is a lower bound on the number of cases needed to cover all possible assignments. □

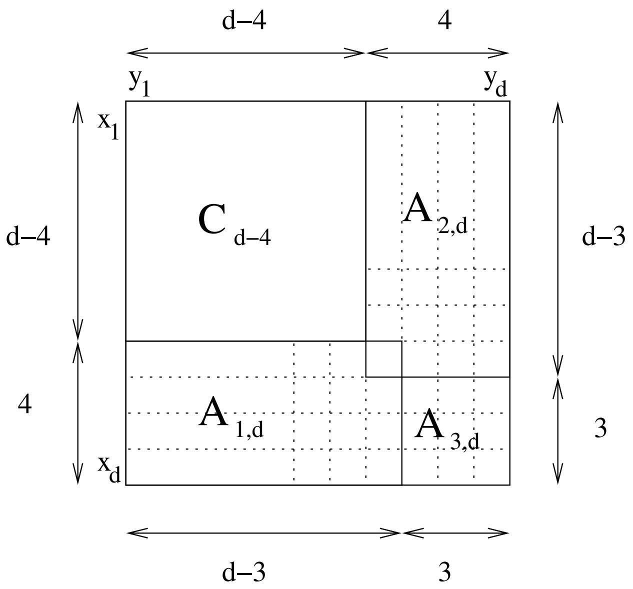

Fig. 1. The construction of C_d

Theorem 1. *Given a binary CSP $P = \langle V, D, C\rangle$, with $|V| = n$ and $|D| = d$, there exists an algorithm for determining the number of solutions in*

- $\mathcal{O}\left(1.3247^n \left(\frac{d}{2}\right)^n\right)$ *time, if* $d = 2 \cdot k, k \geq 1$
- $\mathcal{O}\left(1.3247^n \left(\frac{d^2+d+2}{4}\right)^{\frac{n}{2}}\right)$ *time, if* $d = 4 \cdot k + 1$, *and*
- $\mathcal{O}\left(1.3247^n \left(\frac{d^2+d}{4}\right)^{\frac{n}{2}}\right)$ *time, if* $d = 4 \cdot k + 3$.

Proof. As was described in the previous discussion, we start with creating the clauses corresponding to the constraints:

$$C_{xRy} = \bigwedge_{i,j \in D, (i,j) \notin R} (\neg x_i \vee \neg y_j)$$

For the value restricting clauses we have two cases.

Case 1: $d = 2 \cdot k, k > 0$

Divide the domain into $d/2$ pairs of values. For each variable x from the original problem instance, and each pair of values $i, i+1$, create the pair x_i, x_{i+1}, and the clauses

$$(x_i \vee x_{i+1}) \wedge (\neg x_i \vee \neg x_{i+1}) \bigwedge_{c \in D, c \neq i, c \neq i+1} (\neg x_c)$$

Combining these, as was described in the previous discussion, we get $(d/2)^n$ instances, each with $d \cdot n$ propositional variables. In each instance, we have $n \cdot (d-2)$ clauses on the form $\neg x_c$, hence using unit propagation, we arrive at $2 \cdot n$ propositional variables in each instance. Now let $m = 2 \cdot k, k \geq 0$, and, for each pair of propositional variables x_m, x_{m+1}, introduce a fresh variable ξ_{x_m}, with the interpretation that ξ_{x_m} is true iff x_m is true and x_{m+1} is false, and ξ_{x_m} is false iff x_m is false and x_{m+1} is true. Replace all occurrences of x_m and x_{m+1} with the new variable ξ_{x_m}, and $\neg \xi_{x_m}$, respectively, thereby reducing the number of propositional variables in each instance to n.

Case 2: $d = 2 \cdot k + 1, k > 0$
Given a pair of variables from the original problem, each with an associated pair of assignments, Proposition 1 shows that, if $d = 4 \cdot k + 1$, we need $(d^2 + d + 2)/4$, and if $d = 4 \cdot k + 3$, we need $(d^2 + d)/4$ such pairs to cover all possible assignments of values to the variables. To ensure the uniqueness of each solution we count, if there are overlaps between two assignments x_i and y_j, *i.e.*, these assignments occur in more than one pair of variables, we add the clause $(\neg x_i \vee \neg y_j)$ to all but one instance containing these assignments. If we now perform a similar simplification as was done in the previous case, we get n propositional variables in each of the resulting 2-SAT instances.

Summary
Depending on the size of the domain, we now have

- $\left(\frac{d}{2}\right)^n$ instances if $d = 2 \cdot k$,
- $\left(\frac{d^2+d+2}{4}\right)^{\frac{n}{2}}$ instances if $d = 4 \cdot k + 3$, and
- $\left(\frac{d^2+d}{4}\right)^{\frac{n}{2}}$ instances if $d = 4 \cdot k + 1$.

Using the algorithm for #2-SAT presented in Dahllöf *et al.* [6], we can count the number of solutions in each instance in $\mathcal{O}(1.3247^n)$ time, and the result follows. □

4 Algorithm for $\#3COL$

We now present an algorithm for counting the number of 3-colourings of a graph. If we were to use the previously described algorithm, we would get a time complexity of $\mathcal{O}(2.2944^n)$, but as will be seen, this can be improved to $\mathcal{O}(1.8171^n)$ by exploiting problem specific knowledge.

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. The *neighbourhood* of a vertex v is the set of all adjacent vertices, $\{w \mid (v, w) \in E(G)\}$, denoted $Nbd(v)$ An *independent set* S of G is a subset of $V(G)$, such that for every pair $v, w \in S \to (v, w) \notin E(G)$. A *3-colouring* of G is a function $C : V(G) \to \{R, G, B\}$ 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. To *3-assign* $v \in G$ means assigning $colour(v)$ a value from the set $\{R, G, B\}$. If G is a graph and $S \subseteq V(G)$, the graph $G|S$ has the 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 $G|(V - S)$.

A *matching* in a graph is a set M of vertices such that each vertex $v \in M$ has an edge to one and only one other vertex in M. The maximum matching (wrt. to the number of matched vertices) of a graph is computable in polynomial time [16]. Let $Match(G)$ be a function that computes a maximum matching of G and returns a pair (G_u, G_m) where $G_u \subseteq V(G)$ contains the unmatched vertices and $G_m \subseteq E(G)$ contains the matched pairs. We say that G is *perfectly matched* if $G_u = \emptyset$.

We consider an arbitrary graph G with n vertices and assume, without loss of generality, that it is connected.

A summary of the algorithm #3C for computing $\#3COL$ can be found in Figure 2. The algorithm has two main branches, where the choice of which branch to follow depends on the number of unmatched vertices. Section 4.1 contains the case where the unmatched vertices are less than a third of the total number of vertices in the graph, and Section 4.2 deals with the opposite case. The complexity analysis is done in Section 4.3.

algorithm #3C
$(G_u, G_m) = Match(G)$
$c := 0$
if $|G_u| < |G|/3$ **then**
  for every R{G/B} assignment f of G **do**
    if f is an R{G/B} colouring **then** $c := c + Count_2(G, f)$
    end if
  end for
  return c
else
  for every 3-colouring f of G_m **do**
    $c := c + \prod_{v \in G_u} (3 - |\{f(w) \mid w \in Nbd(v)\}|)$
  end for
  return c
end if

Fig. 2. Algorithm #3C

4.1 Case 1: $|G_u| < |G|/3$

We define a *R{G/B} assignment* of the graph G as a total function $f : V(G) \to \{R, GB\}$ satisfying the following requirement:

if $v - w$ is a pair in G_m, then $f(v) \neq R$ or $f(w) \neq R$.

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 call such an assignment an R{GB}-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).

Now, let us take a closer look at the algorithm. All R{G/B} assignments of G can be efficiently enumerated and there are exactly $2^{|G_u|} \cdot 3^{|G_m|/2}$ such assignments to consider. For each assignment that can be refined to a colouring, the algorithm counts the number of corresponding 3-colourings and adds this number to the variable c. Obviously, c contains the total number of 3-colourings after all assignments have been checked.

4.2 Case 2: $|G_u| \geq |G|/3$

Let $G' = G|\{v \in V; v \text{ appears in } G_m\}$. We begin by noting that G' is perfectly matched and each matched pair $p-q$ can obtain at most six different assignments. Hence, we need to consider at most $6^{|G_m|/2}$ assignments and, in the worst case, $6^{|G_m|/2}$ 3-colourings.

For each 3-colouring f of G, we claim that

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

is the number of ways f can be extended to a 3-colouring of G. Assume for instance that $v \in G_u$ 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 G_u is an independent set, we can simply multiply the numbers of allowed colours in order to count the number of possible extensions of f.

4.3 Complexity analysis

Assume n is the number of vertices in the graph and C satisfies $|G_u| = C \cdot n$.

Case 1: $|G_u| < n/3$ and $C < 1/3$. The number of assignments we need to consider are

$$3^{(n-|G_u|)/2} \cdot 2^{|G_u|} = (3^{(1-C)/2} \cdot 2^C)^n.$$

Since the function $f(C) = 3^{(1-C)/2} \cdot 2^C$ is strictly increasing when $C > 0$, the largest number of assignments we need to consider appears when C is close to $1/3$, *i.e.*, $|G_u|$ is close to $n/3$. In this case, the algorithm runs in $\mathcal{O}((3^{1/3} \cdot 2^{1/3})^n) = \mathcal{O}(6^{n/3}) \approx \mathcal{O}(1.8171^n)$ time.

Case 2: $|G_u| \geq n/3$ and $C \geq 1/3$. The number of assignments we need to consider are

$$6^{(n-|G_u|)/2} = (6^{(1-C)/2})^n.$$

Since the function $f(C) = 6^{(1-C)/2}$ is strictly decreasing when $C > 0$, the largest number of assignments we need to consider appears when $C = 1/3$, *i.e.*, $|G_u| = n/3$. In this case, the algorithm runs in $\mathcal{O}(6^{n/3}) \approx \mathcal{O}(1.8171^n)$ time.

5 Conclusion

We have presented an algorithm for counting the number of solutions to binary constraint satisfaction problem instances. It works by, given a CSP, creating a set of 2-SAT instances, where each instance corresponds to a set of assignments of values to variables. A method for efficiently dividing domains of odd size into pairs of values was given, which makes it possible to avoid an unnecessary increase in time complexity for odd sized domains. The modularity of the algorithm makes it possible to improve the time complexity of the algorithm with no additional work whenever an improved algorithm for the #2-SAT problem is found. Furthermore, as was shown in the proof of Proposition 1, the construction we use cannot be done using fewer instances, thus in order to improve the algorithm, we need to consider a different construction altogether.

We have also shown that using problem specific knowledge, we can improve the complexity of the algorithm, and we give an algorithm for determining the number of possible 3-colourings of a graph.

Several open questions remain, however. We have provided an algorithm for *binary* CSPs, but it is not always the case that an n-ary relation can be represented by binary constraints using n variables [17], thus the problem of counting the number of solutions to a general CSP instance remains. How to deal with the general $\#k$-colouring problem also remains to be investigated. Can ideas similar to those utilised in our algorithm for #3-colouring be found for the general case?

Numerous tractable subclasses of CSPs have been found [18]. Jeavons *et al.* [11] showed the equivalence between finding graph homomorphisms and solving constraint satisfaction problems. Counting graph homomorphisms is #P-complete in most cases [8], so the existence of polynomial time algorithms for the counting problem for the tractable subclasses is unlikely, but this has not yet been investigated.

6 Acknowledgments

Ola Angelsmark is supported in part by CUGS – National Graduate School in Computer Science, Sweden. Peter Jonsson is partially supported by the *Swedish Research Council* (VR), under grant 221-2000-361. Johan Thapper is supported by the *Programme for Interdisciplinary Mathematics* at the Department of Mathematics, Linköpings Universitet.

References

1. 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*, 8:121–123, 1979.
2. R. Bayardo 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-2000)*, pages 157–162, 2000.
3. M. Cooper, D. A. Cohen, and P. G. Jeavons. Characterising tractable constraints. *Artificial Intelligence*, 65(2):347–361, 1994.
4. N. Creignou and M. Hermann. Complexity of generalized satisfiability counting problems. *Information and Computation*, 125:1–12, 1996.
5. V. Dahllöf and P. Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2002)*, pages 292–298, 2002.
6. 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*, Aug. 2002. To appear.
7. O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64, 1991.
8. M. Dyer and C. Greenhill. The complexity of counting graph homomorphisms. *Random Structures and Algorithms*, 17:260–289, 2000.
9. D. Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms (SODA-2001)*, pages 329–337, 2001.
10. T. Feder and R. Motwani. Worst-case time bounds for coloring and satisfiability problems. Unpublished manuscript.

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. M. Jerrum. A very simple algorithm for estimating the number of k-colourings of a low-degree graph. *Random Structures and Algorithms*, 7:157–165, 1995.
13. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–44, Spring, 1992.
14. M. L. Littman, T. Pitassi, and R. Impagliazzo. On the complexity of counting satisfying assignments. Unpublished manuscript, 2001.
15. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
16. S. Micali and V. V. Vazirani. An $O(\sqrt{|v|} \cdot |e|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (FOCS-1980)*, pages 10–16. IEEE Computer Society, 1980.
17. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
18. J. K. Pearson and P. G. Jeavons. A survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, Royal Holloway, University of London, July 1997.
19. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.
20. U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science (FOCS-1999)*, pages 410–414. IEEE Computer Society, 1999.
21. L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.
22. L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
23. E. Vigoda. Improved bounds for sampling colorings. In *40th Annual Symposium on Foundations of Computer Science (FOCS-1999)*, pages 51–59. IEEE Computer Society, 1999.
24. W. Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155(1):277–288, 1996.