Counting Solutions of CSPs: A Structural Approach*

Gilles Pesant
ILOG, 1681 route des Dolines, 06560 Valbonne, France
pesant@crt.umontreal.ca

Abstract

Determining the number of solutions of a CSP has several applications in AI, in statistical physics, and in guiding backtrack search heuristics. It is a $\#\mathcal{P}$-complete problem for which some exact and approximate algorithms have been designed. Successful CSP models often use high-arity, global constraints to capture the structure of a problem. This paper exploits such structure and derives polytime evaluations of the number of solutions of individual constraints. These may be combined to approximate the total number of solutions or used to guide search heuristics. We give algorithms for several of the main families of constraints and discuss the possible uses of such solution counts.

1 Introduction

Many important combinatorial problems in Artificial Intelligence (AI), Operations Research and other disciplines can be cast as Constraint Satisfaction Problems (CSP). One is usually interested in finding a solution to a CSP if it exists (the *combinatorial existence problem*) and much scientific literature has been devoted to this subject in the past few decades. Another important question, though not as thoroughly studied, is how many solutions there are (the *combinatorial enumeration problem*). Our ability to answer this question has several applications in AI (e.g. [Orponen, 1990],[Roth, 1996],[Darwiche, 2001]), in statistical physics (e.g. [Burton and Steif, 1994], [Lebowitz and Gallavotti, 1971]), or more recently in guiding backtrack search heuristics to find solutions to CSPs [Horsch and Havens, 2000][Kask *et al.*, 2004][Refalo, 2004]. In this latter context, an estimation of the number of solutions is sought repeatedly as search progresses. The best strategy to keep the overall runtime low may be to trade accuracy for speed while computing these approximations.

The model of a CSP is centered on the constraints, which give it its structure. This paper proposes to exploit this inherent structure to derive polytime approximations to the solution count of a CSP by combining solution counts of component constraints. At the level of the individual constraint, consistency algorithms act on the domains of the variables in its scope (i.e. on which the constraint is defined) to filter out values which do not belong to any solution of this constraint, thus avoiding some useless search. For many families of constraints, all such values can be removed in polynomial time even though globally the problem is $\mathcal{NP}$-hard. The only visible effect of the consistency algorithms is on the domains, projecting the set of solutions on each of the variables. But a constraint's consistency algorithm often maintains information which may be exploited to evaluate the number of valid tuples. We intend to show that a little additional work is often sufficient to provide close or even exact solution counts for a constraint, given the existing consistency algorithm.

The global counting problem for CSPs (denoted #CSP) is $\#\mathcal{P}$-complete even if we restrict ourselves to binary constraints and is thus very likely intractable. Recent theoretical work [Bulatov and Dalmau, 2003] characterizes some tractable classes of CSPs for this problem but these are quite restrictive, especially for practical problems. The counting problem for Boolean CSPs (#SAT) is the best studied subclass (see e.g. [Birnbaum and Lozinskii, 1999]).

For binary CSPs, exponential time exact algorithms have been described (e.g. [Angelsmark and Jonsson, 2003]). For backtrack search, [Kask *et al.*, 2004] adapts Iterative Join-Graph Propagation to approximate the number of solutions extending a partial solution and uses its results in a value-ordering heuristic to solve CSPs, choosing the value whose assignment to the current variable gives the largest approximate solution count. An implementation optimized for binary constraints performs well compared to other popular strategies. [Refalo, 2004] proposes a generic variable-ordering heuristic based on the impact the assignment of a variable has on the reduction of the remaining search space, computed as the Cartesian product of the domains of the variables. It reports promising results on multi-knapsack, magic square, and Latin square completion benchmarks for (not necessarily binary) CSPs. The value- and variable-ordering heuristics described above rely on an approximation of the solution count. Both can benefit from an improvement in the quality and/or efficient computation of this approximation.

In the rest of the paper, Section 2 revisits several families of constraints to evaluate the effort necessary to provide solution counts, Section 3 explores possible uses for them, Section 4 gives some examples, and Section 5 discusses future work.

*Research conducted while the author was on sabbatical leave from École Polytechnique de Montréal.

2 Looking at Constraints

This section proceeds through a short list of usual constraints (for their origin, see e.g.[Régin, 2004]) and examines how solution counts may be derived. We assume throughout that the constraints are domain consistent, unless stated otherwise. Given a variable x, we denote its domain of possible values as $D(x)$, the smallest and largest value in $D(x)$ as $x_{\min}$ and $x_{\max}$ respectively, and the number of values in $D(x)$ as d_x. We use $\#\gamma$ to mean the number of solutions for constraint γ.

2.1 Binary Constraints

Many arc consistency algorithms have been developed for binary constraints of the form $\gamma(x, y)$. Among them, AC-4 [Mohr and Henderson, 1986] maintains a support counter for each value in $D(x)$ and $D(y)$. The total solution count is then simply the sum of those counters over the domain of one of the variables:

$$\#\gamma(x, y) \quad = \quad \sum_{v \in D(x)} sc(x, v) \tag{1}$$

where $sc(x, v)$ stands for the support counter of value v in the domain of variable x. However, other consistency algorithms only compute support witnesses as needed and hence would not have support counters available.

2.2 Arithmetic Binary Constraints

Typically, support counters are not maintained by the consistency algorithm for arithmetic binary constraints but one rather relies on the semantics of the constraint. Take for example $x < y$: knowing $x_{\min}$ and $y_{\max}$ is sufficient to remove all the inconsistent values. To compute the number of valid couples we can simply enumerate them, which done naively would require $\Omega(d_x \cdot d_y)$ time. Provided the domains are maintained in sorted order, a reasonable assumption, that enumeration can be done in $\Theta(d_x + d_y)$ time by using a running pointer in each domain as described in Algorithm 1.

function lt_card(x, y)
let t_x and t_y be tables containing the values of the domains of x and y, respectively, in increasing order;
$p := 1$; $q := 1$ $c := 0$;
while $q \leq d_y$ **do**
  while $p \leq d_x$ **and** $t_x[p] < t_y[q]$ **do**
    $p := p + 1$;
  $c := c + p - 1$;
  $q := q + 1$;
return c;

Algorithm 1: Computing the number of solutions to $x < y$.

$$\#(x < y) \quad = \quad \text{lt\_card}(x, y) \tag{2}$$

For each value v in $D(y)$, it finds out of the number of supports $sc(y, v)$ from $D(x)$ and adds them up in c. We can make the algorithm incremental by storing $sc(y, v)$ for each v in $D(y)$ and $q(w)$, the smallest q such that $w < t_y[q]$, for each w in $D(x)$: if v is removed from $D(y)$, the total number of solutions decreases by $sc(y, v)$; if w is removed from $D(x)$, the total number of solutions decreases by $d_y - q(w) + 1$ and $sc(y, v)$ is decremented by one for $q(w) \leq v \leq d_y$.

Sometimes the domains of variables involved in such constraints are maintained as intervals. This simplifies the computation, for which we derive a closed form:

$$\begin{aligned} \#(x < y) \quad &= \quad \sum_{i=y_{\min}-x_{\min}}^{y_{\max}-x_{\min}} i \quad - \quad \sum_{i=1}^{y_{\max}-x_{\max}-1} i \\ &= \quad y_{\max} \cdot (x_{\max} + 1) - x_{\min} \cdot d_y \\ &\quad -\tfrac{1}{2} \cdot (x_{\max}^2 + x_{\max} + y_{\min}^2 - y_{\min}) \end{aligned} \tag{3}$$

Clearly, what preceded equally applies to constraints $x \leq y$, $x > y$, and $x \geq y$, with small adjustments. Counting solutions for equality and disequality constraints is comparatively straightforward (remember that domain consistency has been achieved):

$$\#(x = y) \quad = \quad d_x \tag{4}$$

$$\#(x \neq y) \quad = \quad d_x \cdot d_y - |D(x) \cap D(y)| \tag{5}$$

With an appropriate representation of the intersection of $D(x)$ and $D(y)$, an incremental version of the count for $x \neq y$ requires constant time.

In some applications such as temporal reasoning, constants are present in the binary constraints (e.g. $x < y + t$ for some constant t). The previous algorithms and formulas are easily adapted to handle such constraints.

2.3 Linear Constraints

Many constraint models feature linear constraints of the form $\sum_{i=1}^{n} a_i x_i \circ b$ where $\circ$ stands for one of the usual relational operators and the a_i's and b are integers. Bound consistency is typically enforced, which is a fast but rather weak form of reasoning for these constraints. Working from domains filtered in this way is likely to yield very poor approximations of the number of solutions.

Example 1 *Constraint* $3x_1 + 2x_2 - x_3 - 2x_4 = 0$ *with bound consistent domains* $D(x_i) = \{0, 1, 2\},\ 1 \leq i \leq 4$ *admits only seven solutions even though the size of the Cartesian product is 81. Since the value of any one variable is totally determined by the combination of values taken by the others, the upper bound can be lowered to 27 but this is still high.*

These constraints are actually well studied under the name of linear Diophantine equations and inequations. In particular, [Ajili and Contejean, 1995] offers an algorithm adapted to constraint programming that produces a finite description of the set of solutions of a system of linear Diophantine equations and inequations over $\mathbb{N}$

When the constraints are of the form $b' \leq \sum_{i=1}^{n} a_i x_i \leq b$ with the a_i's and x_i's belonging to $\mathbb{N}$, we get knapsack constraints for which [Trick, 2003] adapts a pseudo-polynomial dynamic programming algorithm. It builds an acyclic graph $G_{\langle a_i \rangle, \langle x_i \rangle, b, b'}$ of size $\mathcal{O}(nb)$ in time $\mathcal{O}(nbd)$ (where $d = \max\{d_{x_i}\}$) whose paths from the initial node to a goal node correspond to solutions. It is pointed out that enumerating solutions amounts to enumerating paths through a depth-first search, in time linear in the size of the graph.

$$\#(b' \leq \sum_{i=1}^{n} a_i x_i \leq b) = |\{\text{paths traversing } G_{\langle a_i \rangle, \langle x_i \rangle, b, b'}\}| \tag{6}$$

Interestingly, the approach could be generalized so as to lift the nonnegativity restriction on the coefficients and variables at the expense of a larger graph of size $\mathcal{O}(nr)$ where r represents the magnitude of the range of $\sum_{i=1}^{n} a_i x_i$ over the finite domains.

2.4 Element Constraints

The ability to index an array with a finite-domain variable, commonly known as an `element` constraint, is often present in practical models. Given variables x and i and array of values a, $\texttt{element}(i, a, x)$ constrains x to be equal to the i^{th} element of a ($x = a[i]$). Note that this is not an arbitrary relation between two variables but rather a functional relationship. As a relation, it could potentially number $d_x \cdot d_i$ solutions, corresponding to the Cartesian product of the variables. As a functional relationship, there are exactly as many tuples as there are consistent values for i:

$$\#\texttt{element}(i, a, x) \quad = \quad d_i \tag{7}$$

This exact count improves on the product of the domain sizes by a factor d_x.

2.5 Regular Language Membership Constraints

Given a sequence of variables $X = \langle x_1, x_2, \ldots, x_n \rangle$ and a deterministic finite automaton $\mathcal{A}$, $\texttt{regular}(X, \mathcal{A})$ constrains any sequence of values taken by the variables of X to belong to the regular language recognized by $\mathcal{A}$ [Pesant, 2004]. The consistency algorithm for this constraint builds a directed acyclic graph $G_{X,\mathcal{A}}$ whose structure is very similar to the one used by [Trick, 2003] for knapsack constraints, as reported in Section 2.3. Here as well, each path corresponds to a solution of the constraint and counting them represents no complexity overhead with respect to the consistency algorithm.

$$\#\texttt{regular}(X, \mathcal{A}) \quad = \quad |\{\text{paths traversing } G_{X,\mathcal{A}}\}| \tag{8}$$

2.6 Among Constraints

Under constraint $\texttt{among}(c, X, V)$, variable c corresponds to the number of variables from set X taking their value from set V [Beldiceanu and Contejean, 1994]. Let $R = \{x \in X : D(x) \subseteq V\}$, the subset of variables required to take a value from V, and $P = \{x \in X \setminus R : D(x) \cap V \neq \emptyset\}$, the subset of variables possibly taking a value from V. Those two sets are usually maintained by the consistency algorithm since the relationship $|R| \leq c \leq |R| + |P|$ is useful to filter the domain of c.

Define ν_S as the number of variables from set S taking their value from V. To derive the number of solutions, we first make the following observations:

1. $\nu_R = |R|$, a constant, so that in any solution to the constraint the value taken by $x \in R$ can be replaced by any other in its domain and it remains a solution.
2. $\nu_{X \setminus (R \cup P)} = 0$, a constant, so that in any solution to the constraint the value taken by $x \in X \setminus (R \cup P)$ can be replaced by any other in its domain and it remains a solution.
3. Any assignment of the variables of P such that $\nu_P = c - |R|$ of them take a value in V can be extended into $\prod_{x \in X \setminus P} d_x$ complete solutions.
4. There are such assignments for every value of ν_P from $c_{\min} - |R|$ to $c_{\max} - |R|$ and for every subset S of P identifying the ν_P variables in question. How many assignments, given S? Each variable $y \in S$ has $|D(y) \cap V|$ possible values and each variable $z \in P \setminus S$ has $|D(z) \setminus V|$ possible values, totaling $\#a(S) = \prod_{y \in S} |D(y) \cap V| \cdot \prod_{z \in P \setminus S} |D(z) \setminus V|$ assignments.

Putting it all together gives:

$$\#\texttt{among}(c, X, V) = \prod_{x \in X \setminus P} d_x \cdot \sum_{k=c_{\min}}^{c_{\max}} \sum_{\substack{S \subseteq P \\ |S| = k - |R|}} \#a(S) \tag{9}$$

In the worst case, the number of S sets to consider is exponential in the number of variables. It may be preferable sometimes to compute faster approximations. We can replace every $\#a(S)$ by some constant lower or upper bound. Let t_{in} be a table in which the sizes of $D(x) \cap V$ for every $x \in P$ appear in increasing order. Similarly, let t_{out} be a table in which the sizes of $D(x) \setminus V$ for every $x \in P$ appear in increasing order. We define a lower bound $\ell(a(k)) = \prod_{i=1}^{k-|R|} t_{\text{in}}[i] \cdot \prod_{i=1}^{|P|-k+|R|} t_{\text{out}}[i]$
and an upper bound
$u(a(k)) = \prod_{i=1}^{k-|R|} t_{\text{in}}[|P|-i+1] \cdot \prod_{i=1}^{|P|-k+|R|} t_{\text{out}}[|P|-i+1]$.
Then

$$\#\texttt{among}(c, X, V) \geq \prod_{x \in X \setminus P} d_x \cdot \sum_{k=c_{\min}}^{c_{\max}} \binom{|P|}{k-|R|} \cdot \ell(a(k)) \tag{10}$$

$$\#\texttt{among}(c, X, V) \leq \prod_{x \in X \setminus P} d_x \cdot \sum_{k=c_{\min}}^{c_{\max}} \binom{|P|}{k-|R|} \cdot u(a(k)) \tag{11}$$

Note that if the variables in P have identical domains then the lower and upper bounds coincide and we obtain the exact count.

Example 2 *Consider* $\texttt{among}(c, \{x_1, x_2, x_3, x_4, x_5\}, \{1, 2\})$ *with* $D(c) = \{3, 4\}$, $D(x_1) = \{1, 3, 4\}$, $D(x_2) = \{1, 2\}$, $D(x_3) = \{3, 4\}$, $D(x_4) = \{2\}$, *and* $D(x_5) = \{1, 2, 3\}$. *We obtain* $R = \{x_2, x_4\}$, $P = \{x_1, x_5\}$, *and* $\#\texttt{among}(c, \{x_1, x_2, x_3, x_4, x_5\}, \{1, 2\}) = d_{x_2} \cdot d_{x_3} \cdot d_{x_4} \cdot (|D(x_1) \cap \{1, 2\}| \cdot |D(x_5) \setminus \{1, 2\}| + |D(x_5) \cap \{1, 2\}| \cdot |D(x_1) \setminus \{1, 2\}| + |D(x_1) \cap \{1, 2\}| \cdot |D(x_5) \cap \{1, 2\}|) = 2 \cdot 2 \cdot 1 \cdot (1 \cdot 1 + 2 \cdot 2 + 1 \cdot 2) = 28$, *by (9). Using (10),(11) we get* $16 \leq \#\texttt{among}(c, \{x_1, x_2, x_3, x_4, x_5\}, \{1, 2\}) \leq 40$. *In comparison, the size of the Cartesian product is* 72.

2.7 Mutual Exclusion and Global Cardinality Constraints

Besides `among`, other constraints are concerned with the number of repetitions of values. Constraint $\texttt{alldiff}(X)$ forces the variables of set X to take different values. Constraint $\texttt{gcc}(Y, V, X)$, for a sequence of variables $Y = \langle y_1, y_2, \ldots, y_m \rangle$ and a sequence of values $V = \langle v_1, v_2, \ldots, v_m \rangle$, makes each y_i equal to the number of times

a variable of X takes value v_i. It is a generalization of the former.

Already for the `alldiff` constraint, the counting problem includes as a special case the number of perfect matchings in a bipartite graph, itself equivalent to the $\mathcal{NP}$-hard problem of computing the permanent of the adjacency matrix representation of the graph. So currently we do not know of an efficient way to compute $\#\texttt{alldiff}(X)$ or $\#\texttt{gcc}(Y, V, X)$ exactly. A polytime randomized approximation algorithm for the permanent was recently proposed in [Jerrum *et al.*, 2004] but its time complexity, in $\Omega(n^{10})$, remains prohibitive. Therefore even getting a reasonable approximate figure is challenging. We nevertheless propose some bounds.

Upper Bound

If $D(x) \subseteq D(y)$ for some variables $x, y \in X$ then regardless of the value v taken by x we know that at most $d_y - 1$ possibilities remain for y since $v \in D(y)$. We generalize this simple observation by considering $\mathcal{D} = \{D(x) : x \in X\}$, the distinct domains of X. For each $D \in \mathcal{D}$, define $E_D = \{x \in X : D(x) = D\}$, the set of variables with domain D, and $S_D = \{x \in X : D(x) \subset D\}$, the set of variables whose domain is properly contained in D. Then

$$\#\texttt{alldiff}(X) \leq \prod_{D \in \mathcal{D}} \prod_{i=1}^{|E_D|} (|D| - |S_D| - i + 1) \qquad (12)$$

Computing it requires taking the product of n terms, each of which can be computed easily in $\mathcal{O}(nm)$ time and probably much faster with an appropriate data structure maintaining the E_D and S_D sets. This upper bound is interesting because for the special case where all domains are equal, say $\{v_1, v_2, \ldots, v_m\}$, it simplifies to the exact solution count,

$$\prod_{i=1}^{n} (m - i + 1) = m!/(m - n)!$$

Lower Bound

As hinted before, the consistency algorithm for `alldiff` computes a maximum matching in a bipartite graph. It is known that every maximum matching of a graph can be obtained from a given maximum matching by successive transformations through even-length alternating paths starting at an unmatched vertex or through alternating cycles.[1] Each transformation reverses the status of the edges on the path or cycle in question: if it belonged to the matching it no longer does, and vice versa. The result is necessarily a matching of the same size. Finding every maximum matching in this way (or any other) would be too costly, as we established before, but finding those that are just one transformation away is fast and provides a lower bound on the number of maximum matchings, or equivalently on the solution count of `alldiff`.

By not applying transformations in succession, we also avoid having to check whether two matchings obtained by distinct sequences of transformations are in fact identical, which could happen. However, successive transformations from distinct connected components is safe: each resulting matching is distinct. We may therefore take the product of the number of maximum (sub)matchings in each connected component of the bipartite graph. Identifying the connected components C takes time linear in the size of the graph. Enumerating all the appropriate cycles and paths of each connected component can be done in $\mathcal{O}(nm^2)$ time. Let their number be ν_g for each connected component $g \in C$. Then

$$\#\texttt{alldiff}(X) \geq \prod_{g \in C} (\nu_g + 1) \qquad (13)$$

For the `gcc` constraint, we may be able to adapt the ideas which gave rise to the previous lower bound. The consistency algorithm for this constraint is usually based on network flows. Given a network and a flow through it, a residual graph can be defined and the circuits in this graph lead to equivalent flows. Another idea is to decompose the constraint into `among` constraints on singleton sets of values and use (9)-(11).

3 Using Solution Counts of Constraints

This section examines the possible uses of solution counts of constraints.

3.1 Approximate the Solution Count of a CSP

They can be used to approximate the number of solutions to a CSP. Given a model for a CSP, consider its variables as a set S and its constraints as a collection C of subsets of S defined by the scope of each constraint. Add to C a singleton for every variable in S. A *set partition* of S is a subset of C whose elements are pairwise disjoint and whose union equals S (see Section 4 for some examples). To each element of the partition corresponds a constraint (or a variable in case of a singleton) for which an upper bound on the number of solutions can be computed (or taken as the cardinality of the domain in case of a singleton). The product of these upper bounds gives an upper bound on the total number of solutions. In general there are several such partitions possible and we can find the smallest product over them. If it is used in the course of a computation as variables become fixed and can be excluded from the partition, new possible partitions emerge and may improve the solution count.

3.2 Guide Search Heuristics

As we saw in the introduction, these counts are also useful to develop robust search heuristics. They may allow a finer evaluation of impact in the generic search heuristic of [Refalo, 2004] since their approximation of the size of the search space is no worse than the Cartesian product of the domains. Value-ordering heuristics such as [Kask *et al.*, 2004] are also good candidates. Other search heuristics following the *first-fail principle* (detect failure as early as possible) and centered on constraints can be guided by a count of the number of solutions left for each constraint. We might focus the search on the constraint currently having the smallest number of solutions, recognizing that failure necessarily occurs through a constraint admitting no more solutions (see Section 4).

[1] An *alternating* path or cycle alternates between edges belonging to a given matching and those not belonging to it.

3.3 Evaluate Projection Tightness

We can also compute the ratio of the (approximate) solution count of a constraint to the size of the Cartesian product of the appropriate domains, in a way measuring the tightness of the projection of the constraint onto the individual variables. A low ratio stresses the poor quality of the information propagated to the other constraints (i.e. the filtered domains) and identifies constraints whose consistency algorithm may be improved. Tightness can also serve as another search heuristic, focusing on the constraint currently exhibiting the worst tightness (i.e. the smallest ratio). An ideal ratio of one corresponds to a constraint perfectly captured by the current domains of its variables.

4 Some Examples

The purpose of this section is to illustrate solution counts and their possible uses through a few simple models, easier to analyse.

4.1 Map Coloring

Consider the well-known map coloring problem for six European countries: Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands. Any two countries sharing a border cannot be of the same color. To each country we associate a variable (named after its first letter) that represents its color. Constraints

$$f \neq b, f \neq \ell, f \neq g, \ell \neq g, \ell \neq b, b \neq n, g \neq n, g \neq d, g \neq b$$

model the problem. If we allow five colors, there are 1440 legal colorings. The cardinality of the Cartesian product of the domains of the variables initially overestimates that number to be 15625 (5^6). A set partition such as $\{f \neq \ell, b \neq n, g \neq d\}$ provides a slightly better estimate, using (5):

$$(d_x \cdot d_y - |D(x) \cap D(y)|)^3 = (5 \cdot 5 - 5)^3 = 8000.$$

Noticing that four of these countries are all pairwise adjacent, an alternate model is

$$\texttt{alldiff}(\{b, f, g, \ell\}), b \neq n, g \neq n, g \neq d$$

yielding a still better upper bound on the solution count, $5! \cdot 5^2 = 3000$, using (12) with set partition $\{\texttt{alldiff}(\{b, f, g, \ell\}), n, d\}$. We see here that the solution counts of the constraints in a CSP model, particularly those of high arity, can quickly provide good approximations of the number of solutions of CSPs. The projection tightness of $b \neq n$, $g \neq n$, and $g \neq d$ is $20/25 = 0.8$; that of $\texttt{alldiff}(\{b, f, g, \ell\})$ is $5!/5^4 = 0.192$. This could be taken as an indication that search should focus on the latter.

Let us now analyse the effect of coloring one of the countries red. Regardless of the country we choose, the number of legal colorings will decrease five fold to 288 since colors are interchangeable. The true impact of fixing a variable, measured as 1 minus the ratio of the remaining search space after and before ([Refalo, 2004]), is thus 0.8. Table 4.1 reports after each country is colored red the cardinality of the Cartesian product of the domains (column 2), the upper bound on the solution count from each of the two set partitions (columns 4 and 6), and their corresponding impacts (columns 3, 5, and 7).

	domains		1st partition		2nd partition	
var	solns	impact	solns	impact	solns	impact
ℓ	1600	.898	1024	.872	600	.800
f	1600	.898	1024	.872	600	.800
b	1280	.918	768	.904	480	.840
g	1024	.934	576	.928	384	.872
n	2000	.872	1280	.840	360	.880
d	2500	.840	1600	.800	480	.840
avg	–	.893	–	.869	–	.839

Table 1: Comparing different approximations of the number of solutions of a small map coloring problem.

Again we note a significant improvement in the approximation of the number of solutions over the Cartesian product, particularly from the second set partition based on the model using an `alldiff` constraint. The latter also provides a closer approximation of the true impact: the average computed impact is less than 5% away whereas it is more than 11% away for the average computed impact using Cartesian products.

4.2 Rostering

Consider next a simple rostering problem: a number of daily jobs must be carried out by employees while respecting some labor regulations. Let E be the set of employees, D the days of the planning horizon, and J the set of possible jobs, including a day off. Define variables $j_{de} \in J$ $(d \in D, e \in E)$ to represent the job carried out by employee e on day d. To ensure that every job is performed on any given day, we can state the following constraints:

$$\texttt{gcc}(\langle 1, 1, \ldots, 1\rangle, J \setminus \{\text{day off}\}, (j_{de})_{e \in E}) \qquad d \in D. \quad (14)$$

Some jobs are considered more demanding than others and we wish to balance the workload among employees. We introduce variables w_{de} representing the workload of employee e on day d and link them to the main variables in the following way:

$$\texttt{element}(j_{de}, (\ell_j)_{j \in J}, w_{de}) \qquad d \in D,\ e \in E \quad (15)$$

where ℓ_j corresponds to the load of job j. We then add constraints enforcing a maximum workload k:

$$\textstyle\sum_{d \in D} w_{de} \leq k \qquad e \in E. \quad (16)$$

Finally work patterns may be imposed on individual rosters:

$$\texttt{regular}((j_{de})_{d \in D}, \mathcal{A}) \qquad e \in E \quad (17)$$

with the appropriate automaton $\mathcal{A}$.

To approximate the number of solutions, there are a few possibilities for a set partition of the variables $(j_{de})_{d \in D, e \in E}$ and $(w_{de})_{d \in D, e \in E}$: constraints (14)(16) are one, constraints (17)(16) are another, and so are constraints (15). In deciding which one to use, its size (the number of parts), the quality of the bounds, and the projection tightness of each constraint may help. Partition (15) has size $|D| \cdot |E|$ compared to $|D| + |E|$ and $2 \cdot |E|$ for the other two: a smaller size

means larger parts and probably a better overall approximation since each solution count has a more global view. We presented exact counts for every constraint used here except (14), making partition (14)(16) less attractive. A set partition that includes constraints with low tightness (which is difficult to assess here without precise numbers) is more likely to do significantly better than the size of the Cartesian product of the individual domains, which can be considered a baseline.

5 Discussion

This paper argued that looking at the number of solutions of individual constraints in CSPs is interesting and useful. It can approximate the number of solutions as a whole or help guide search heuristics. Efficient ways of counting solutions were given for several families of constraints. The concepts of set partition over the variables and of projection tightness were defined in order to combine solution counts and measure the efficacy of consistency algorithms, respectively.

But this is a first step and many questions remain. Some of the main families of constraints have been investigated but others were left out: for example, those useful in scheduling or packing problems such as `cumulative` and `diffn`, or those for routing problems such as `cycle`. How close can we get to the solution count for them and at what computational cost? For the simplest constraints that we investigated, we mentioned how the algorithms computing solution counts could be made incremental. This is an important issue to achieve efficiency when such counts are computed repeatedly as in backtrack search. What can be done for the other constraints? The whole question of the usefulness of such an approach hasn't been settled either. The answer is likely to come, at least in part, from empirical evidence. Computational experiments will need to be run on larger and more realistic models.

Acknowledgements

Philippe Refalo's work on impact-based search sparked the idea for this paper. I thank Jean-Charles Régin for discussions and the anonymous referees for their constructive comments. This work was partially supported by the Canadian Natural Sciences and Engineering Research Council under grant OGP0218028.

References

[Ajili and Contejean, 1995] F. Ajili and E. Contejean. Complete Solving of Linear Diophantine Equations and Inequations without Adding Variables. In *Proc. CP'05*, pages 1–17. Springer-Verlag LNCS 976, 1995.

[Angelsmark and Jonsson, 2003] O. Angelsmark and P. Jonsson. Improved Algorithms for Counting Solutions in Constraint Satisfaction Problems. In *Proc. CP'03*, pages 81–95. Springer-Verlag LNCS 2833, 2003.

[Beldiceanu and Contejean, 1994] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20:97–123, 1994.

[Birnbaum and Lozinskii, 1999] E. Birnbaum and E. L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.

[Bulatov and Dalmau, 2003] A.A. Bulatov and V. Dalmau. Towards a Dichotomy Theorem for the Counting Constraint Satisfaction Problem. In *Proc. FOCS'03*, pages 562–573. IEEE Computer Society, 2003.

[Burton and Steif, 1994] R. Burton and J. Steif. Nonuniqueness of Measures of Maximal Entropy for Subshifts of Finite Type. *Ergodic Theory and Dynamical Systems*, 14:213–236, 1994.

[Darwiche, 2001] A. Darwiche. On the Tractable Counting of Theory Models and its Applications to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logic*, 11:11–34, 2001.

[Horsch and Havens, 2000] M. Horsch and B. Havens. Probabilistic Arc Consistency: A Connection Between Constraint Reasoning and Probabilistic Reasoning. In *UAI-2000*, pages 282–290, 2000.

[Jerrum *et al.*, 2004] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries. *Journal of the ACM*, 51:671–697, 2004.

[Kask *et al.*, 2004] K. Kask, R. Dechter, and V. Gogate. Counting-Based Look-Ahead Schemes for Constraint Satisfaction. In *Proc. CP'04*, pages 317–331. Springer-Verlag LNCS 3258, 2004.

[Lebowitz and Gallavotti, 1971] J. Lebowitz and G. Gallavotti. Phase Transitions in Binary Lattice Gases. *Journal of Mathematical Physics*, 12:1129–1133, 1971.

[Mohr and Henderson, 1986] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Orponen, 1990] P. Orponen. Dempster's Rule of Combination is #-Complete. *Artificial Intelligence*, 44:245–253, 1990.

[Pesant, 2004] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proc. CP'04*, pages 482–495. Springer-Verlag LNCS 3258, 2004.

[Refalo, 2004] P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proc. CP'04*, pages 557–571. Springer-Verlag LNCS 3258, 2004.

[Régin, 2004] J.-C. Régin. Global Constraints and Filtering Algorithms. In M. Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer, 2004.

[Roth, 1996] D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence*, 82:273–302, 1996.

[Trick, 2003] M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.