CIRRELT

Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

Solution Counting Algorithms for Constraint-Centered Search Heuristics

Alessandro Zanarini
Gilles Pesant

June 2007

CIRRELT-2007-18

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
Pavillon Palasis-Prince, local 2642
Québec (Québec)
Canada G1K 7P4
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

Université Laval · UQÀM · HEC Montréal · École Polytechnique Montréal · Université de Montréal

Solution Counting Algorithms for Constraint-Centered Search Heuristics

Alessandro Zanarini[1], Gilles Pesant[1,*]

1. Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT), Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7, and École Polytechnique, Département de génie informatique, C.P. 6079, succusale Centre-Ville, Montréal, Canada H3C 3A7

Abstract. Constraints have played a central role in CP because they capture key substructures of a problem and efficiently exploit them to boost inference. This paper intends to do the same thing for search, proposing constraint-centered heuristics which guide the exploration of the search space toward areas that are likely to contain a high number of solutions. We first propose new search heuristics based on solution counting information at the level of individual constraints. We then describe efficient algorithms to evaluate the number of solutions of two important families of constraints: occurrence counting constraints, such as **all different,** and sequencing constraints, such as **regular**. In both cases we take advantage of existing filtering algorithms to speed up the evaluation. Experimental results on benchmark problems show the effectiveness of our approach.

Keywords. Counting algorithms, search heuristics, constraint programming.

Acknowledgements. This research was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant OGP0218028 and the Fonds de recherche sur la nature et les technologies (FQRNT) under grant 115724.

---

* Corresponding author: Gilles.Pesant@polymtl.ca

Dépôt légal – Bibliothèque nationale du Québec,
Bibliothèque nationale du Canada, 2007

1 Introduction

Constraint Programming (CP) is a powerful technique to solve combinatorial problems. It applies sophisticated inference to reduce the search space and a combination of variable and value selection heuristics to guide the exploration of that search space. Despite many research efforts to design generic and robust search heuristics and to analyze their behaviour, a successful CP application often requires customized, *problem-centered* search heuristics or at the very least some fine tuning of standard ones, particularly for value selection. In contrast, Mixed Integer Programming (MIP) and SAT solvers feature successful default search heuristics that basically reduce the problem at hand to a modeling issue.

Constraints have played a central role in CP because they capture key substructures of a problem and efficiently exploit them to boost inference. This paper intends to do the same thing for search, proposing *constraint-centered* heuristics. A constraint's consistency algorithm often maintains data structures in order to incrementally filter out values that are not supported by the constraint's set of valid tuples. These same data structures may be exploited to evaluate *how many* valid tuples there are. Up to now, the only visible effect of the consistency algorithms has been on the domains, projecting the set of tuples on each of the variables. Additional information about the number of solutions of a constraint can help a search heuristic to focus on critical parts of a problem or promising solution fragments. Polytime approximate or exact algorithms to count the number of solutions of several common families of constraints were given in [12]. For some families, little work was required to provide close or even exact evaluations of the number of solutions for a constraint, given the existing consistency algorithm and its data structures.

There is a large body of scientific literature on search heuristics to solve CSPs. Most of the popular dynamic variable selection heuristics favour small domain size and large degree in the constraint graph (mindom, dom/deg, dom/ddeg, dom/wdeg, Brelaz). For value selection, minimizing the number of conflicts with neighbouring variables is popular. We mention below the closest related work on search. Kask et al. [9] approximate the total number of solutions extending a partial solution to a CSP and use it in a value selection heuristic, 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 [14] proposes a generic variable selection 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 benchmark problems. The main difference between our work and these is that we focus on individual constraints whereas they consider the problem as a whole. As an interesting connection for constraint-centered heuristics, Patel and Chinneck [10] investigate several variable selection heuristics guided by the constraints that are tight at the optimal solution of the relaxation, to find feasible solutions of MIPs.

Contributions There are two main contributions in this work. First, we describe efficient algorithms to evaluate the number of solutions of two important families of constraints: occurrence counting constraints (`alldifferent`) and sequencing constraints (`regular`). With respect to [12], what is proposed for the former is a considerable improvement and for the latter it details what was only alluded to before. Second, we propose and experiment with new search heuristics based on solution counting information at the level of individual constraints.

Plan of the paper Section 2 presents some key definitions and describes the search heuristics we propose. Section 3 gives an algorithm to compute the number of solutions of `regular` constraints. Section 4 summarizes the literature on counting solutions of `alldifferent` constraints and proposes a related algorithm more suited to our purpose. Section 5 presents comparative experimental results supporting our proposal. Finally Section 6 summarizes our work and mentions some of the arising research issues.

2 Generic Constraint-Centered Heuristic Search Framework

Whereas most generic dynamic search heuristics in constraint programming rely on information at the fine-grained level of the individual variable (e.g. its domain size and degree), we investigate dynamic search heuristics based on coarser, but more global, information. Global constraints are successful because they encapsulate powerful specialized filtering algorithms but firstly because they bring out the underlying structure of combinatorial problems. That exposed structure can also be exploited during search. The heuristics proposed here revolve around the knowledge of the number of solutions of individual constraints, the intuition being that a constraint with few solutions corresponds to a critical part of the problem with respect to satisfiability.

Definition 1 (solution count). *Given a constraint $\gamma(x_1, \ldots, x_k)$ and respective finite domains D_i $_{1 \leq i \leq k}$, let $\#\gamma(x_1, \ldots, x_k)$ denote the number of solutions of constraint γ.*

Search heuristics following the *fail-first 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 solution.
We can go one step further with solution count information and evaluate it for each variable-value pair in an individual constraint.

Definition 2 (solution density). *Given a constraint $\gamma(x_1, \ldots, x_k)$, respective finite domains D_i $_{1 \leq i \leq k}$, a variable x_i in the scope of γ, and a value $d \in D_i$, we will call*

$$\sigma(x_i, d, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_k)}{\#\gamma(x_1, \ldots, x_k)}$$

```
1 max = 0;
2 for each constraint γ(x_1, ..., x_k) do
3     for each unbound variable x_i ∈ {x_1, ..., x_k} do
4         for each value d ∈ D_i do
5             if σ(x_i, d, γ) > max then
6                 (x*, d*) = (x_i, d);
7                 max = σ(x_i, d, γ);
8 return branching decision "x* = d*";
```

Algorithm 1: The Maximum Solution Density (MaxSD) search heuristic

the solution density *of pair* (x_i, d) *in* γ*. It measures how often a certain assignment is part of a solution.*

We can favour the highest solution density available with the hope that such a choice generally brings us closer to satisfying the whole CSP. Our choice may combine information from every constraint in the model, be restricted to a single constraint, or even to a given variable. Algorithms 1 to 3 define the search heuristics with which we will experiment in Section 5.

```
1 max = 0;
2 choose constraint γ(x_1, ..., x_k) which minimizes #γ;
3 for each unbound variable x_i ∈ {x_1, ..., x_k} do
4     for each value d ∈ D_i do
5         if σ(x_i, d, γ) > max then
6             (x*, d*) = (x_i, d);
7             max = σ(x_i, d, γ);
8 return branching decision "x* = d*";
```

Algorithm 2: The Minimum Solution Count, Maximum Solution Density (MinSC;MaxSD) search heuristic

3 Counting for Regular Constraints

The `regular`(X, Π) constraint [11] holds if the values taken by the sequence of finite domain variables $X = \langle x_1, x_2, \ldots, x_n \rangle$ spell out a word belonging to the regular language defined by the deterministic finite automaton $\Pi = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a partial transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. The filtering algorithm associated to this constraint is based on the computation of paths in a graph. The automaton is unfolded into a layered acyclic directed graph $G = (V, A)$ where vertices of a layer correspond to states of the automaton and arcs represent variable-value pairs. We

```
1 max = 0;
2 Let S = {x_i : |D_i| > 1 and minimum};
3 for each variable x_i ∈ S do
4     for each constraint γ with x_i in its scope do
5         for each value d ∈ D_i do
6             if σ(x_i, d, γ) > max then
7                 (x*, d*) = (x_i, d);
8                 max = σ(x_i, d, γ);
9 return branching decision "x* = d*";
```

Algorithm 3: The Smallest Domain, Maximum Solution Density (MinDom;MaxSD) search heuristic

denote by $v_{\ell,q}$ the vertex corresponding to state q in layer ℓ. The first layer only contains one vertex, v_{1,q_0}; the last layer only contains vertices corresponding to accepting states, $v_{n+1,q}$ with $q \in F$. This graph has the property that paths from the first layer to the last are in one-to-one correspondence with solutions of the constraint. The existence of a path through a given arc thus constitutes a support for the corresponding variable-value pair [11]. Figure 1 gives an example of a layered directed graph built for one such constraint on five variables.

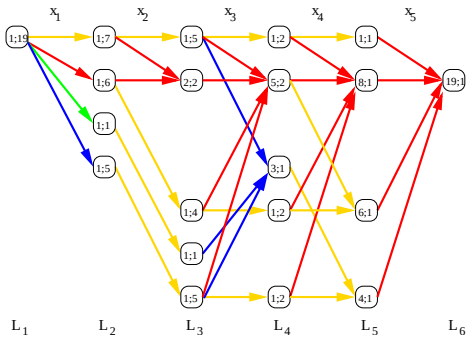

Fig. 1. The layered directed graph built for a `regular` constraint on five variables. Vertex labels represent the number of incoming and outgoing paths.

The time complexity of the filtering algorithm is linear in the size of the graph (the number of variables times the number of transitions appearing in the automaton). Essentially, one forward and one backward sweep of the graph are sufficient. An incremental version of the algorithm, which updates the graph as the computation proceeds, has a time complexity that is linear in the size of the changes to the graph.

3.1 Counting paths in the associated graph

Given the graph built by the filtering algorithm for `regular`, what is the additional computational cost of determining its number of solutions? As we already pointed out, every (complete) path in that graph corresponds to a solution. Therefore it is sufficient to count the number of such paths. We express this through a simple recurrence relation, which we can compute by dynamic programming. Let $\#op(\ell, q)$ denote the number of paths from $v_{\ell,q}$ to a vertex in the last layer. Then we have:

$$\begin{aligned} \#op(n+1, q) &= 1 \\ \#op(\ell, q) &= \sum_{(v_{\ell,q}, v_{\ell+1,q'}) \in A} \#op(\ell+1, q'), \quad 1 \leq \ell \leq n \end{aligned}$$

The total number of paths is given by

$$\#\texttt{regular}(X, \Pi) = \#op(1, q_0)$$

in time linear in the size of the graph even though there may be exponentially many of them. Therefore this is absorbed in the asymptotic complexity of the filtering algorithm.

The search heuristics we consider require not only *solution counts* of constraints but *solution densities* of variable-value pairs as well. In the graph of `regular`, such a pair (x_i, d) is represented by the arcs between layers i and $i+1$ corresponding to transitions on value d. The number of solutions in which $x_i = d$ is thus equal to the number of paths going through one of those arcs. Consider one such arc $(v_{i,q}, v_{i+1,q'})$: the number of paths through it is the product of the number of outgoing paths from $v_{i+1,q'}$ and the number of incoming paths to $v_{i,q}$. The former is $\#op(i+1, q')$ and the latter, $\#ip(i, q)$, is just as easily computed:

$$\begin{aligned} \#ip(1, q_0) &= 1 \\ \#ip(\ell+1, q') &= \sum_{(v_{\ell,q}, v_{\ell+1,q'}) \in A} \#ip(\ell, q), \quad 1 \leq \ell \leq n \end{aligned}$$

where $\#ip(\ell, q)$ denotes the number of paths from v_{1,q_0} to $v_{\ell,q}$.

In Figure 1, the left and right labels inside each vertex give the number of incoming and outgoing paths for that vertex, respectively. For example, the arc between the vertex labeled "2; 2" in layer L_3 and the vertex labeled "5; 2" in layer L_4 has $2 \times 2 = 4$ paths through it.

Let $A(i, d) \subset A$ denote the set of arcs representing variable-value pair (x_i, d). The solution density of pair (x_i, d) is thus given by:

$$\sigma(x_i, d, \texttt{regular}) = \frac{\sum_{(v_{i,q}, v_{i+1,q'}) \in A(i,d)} \#ip(i, q) \cdot \#op(i+1, q')}{\#op(1, q_0)}$$

Once these quantities are tabulated, the cost of computing the solution density of a given pair is in the worst case linear in $|Q|$, the number of states of the automaton.

3.2 An incremental version

Because a constraint's filtering algorithm is called on frequently, the graph for `regular` is not created from scratch every time but updated at every call. Given that we already maintain data structures to perform incremental filtering for `regular`, should we do the same when determining its solution count and solution densities?
For the purposes of the filtering algorithm, as one or several arcs are removed between two given layers of the graph as a consequence of a value being deleted from the domain of a variable, other arcs are considered for removal in the previous (resp. following) layers only if the out-degree (resp. in-degree) of some vertices at the endpoints of the removed arcs becomes null. Otherwise no further updates need to be propagated. Consequently even though the total amount of work in the worst case is bounded above by the size of the graph, it is often much less in practice. In the case of solution counting, the labels that we added at vertices contain finer-grained information requiring more extensive updates. Removing an arc will change the labels of its endpoints but also those of every vertex reachable downstream and of every vertex upstream which can reach that arc. Here the total amount of work in practice may be closer to the worst case. Therefore maintaining the additional data structures could prove to be too expensive.

3.3 A lazy evaluation version

We may not be interested in the value of $\#op()$ and $\#ip()$ for every combination of arguments — for example in some search heuristics we may only want the solution densities for a particular variable. One way to avoid useless work is to lazily evaluate the $\#op()/\#ip()$ values as we require them. *Memory functions* combine the goal-oriented, top-down approach of recursive calls with the compute-once ability of dynamic programming. The request for a solution density triggers the computation of the required $\#op()/\#ip()$ values. If that value has been computed before, it is simply looked up in a table. Otherwise, it is computed recursively and tabulated before it is returned to avoid recomputing it. In some cases only a small fraction of the vertex labels are actually computed, especially if we do not require the solution count of the constraint: if we only compare variable-value pairs within a constraint, solution densities can be replaced by the number of solutions in which each pair participates, thus avoiding the computation of $\#op(1, q_0)$.
On the Nonogram problem introduced in Section 5, the lazy evaluation version was slightly faster than the version computing from scratch and up to five times faster than the version maintaining the data structures. Consequently we used the lazy evaluation version in our experiments.

4 Counting for Alldifferent Constraints

The `alldifferent` constraint restricts a set of variables to be pairwise different [15].

Definition 3 (Value Graph). *Given a set of variables* $X = \{x_1, \ldots, x_n\}$ *with respective domains* $D_1, \ldots, D_n$*, we define the* value graph *as a bipartite graph* $G = (X \cup D_X, E)$ *where* $D_X = \bigcup_{i=1,\ldots,n} D_i$ *and* $E = \{\{x_i, d\} \mid d \in D_i\}$.

There exists a bijection between a maximum matching of size $|X|$ on the value graph and a solution of the related `alldifferent` constraint. Finding the number of solutions is then equivalent to counting the number of maximum matchings on the value graph.

Maximum matching counting is also equivalent to the problem of computing the permanent of a (0-1) matrix. Given a bipartite graph $G = (V_1 \cup V_2, E)$, with $|V_1| = |V_2| = n$, the related $n \times n$ adjacency matrix A has element $a_{i,j}$ equal to 1 if and only if vertex i is connected to vertex j. The permanent of a $n \times n$ matrix A is formally defined as:

$$per(A) = \sum_{\sigma \in S_n} \prod_i a_{i,\sigma(i)} \tag{1}$$

where S_n denotes the symmetric group, i.e. the set of $n!$ permutations of $[n]$. Given a specific permutation, the product is equal to 1 if and only if all the elements are equal to 1 i.e. the permutation is a valid maximum matching in the related bipartite graph. Hence, the sum over all the permutations gives us the total number of maximum matchings. In the following, we will freely use both matrix and graph representations.

4.1 Computing the permanent

Permanent computation has been studied for the last two centuries and it is still a challenging problem to address. Even though the analytic formulation of the permanent resembles that of the determinant, there has been few advances on its exact computation. In 1979, Valiant [16] proved that the problem is $\#P$-complete, even for 0-1 matrices, that is, under reasonable assumptions, it cannot be computed in polynomial time. The focus then moved to approximating the permanent. We can identify at least four different approaches for approximating the permanent: elementary iterative algorithms, reductions to determinants, iterative balancing, and Markov Chain Monte Carlo methods.

Elementary Iterative Algorithms Rasmussen proposed in [13] a very simple recursive estimator for the permanent. This method works quite well for dense matrices but it breaks down when applied to sparse matrices; its time complexity is $O(n^3\omega)$ recently improved to $O(n^2\omega)$ by Fürer [3] (here ω is a function satisfying $\omega \to \infty$ as $n \to \infty$). Further details about these approaches will be given in the next section.

Reduction to Determinant The determinant reduction technique is based on the resemblance of the permanent and the determinant. This method randomly replaces some 1-entry elements of the matrix by uniform random elements $\{\pm 1\}$. It turns out that the determinant of the new matrix is an unbiased estimator of the permanent of the original matrix. The proposed algorithms either provide an arbitrarily close approximation in exponential time [2] or an approximation within an exponential factor in polytime [1].

Iterative Balancing The work of Linial et al. [8] exploits a lower bound on the permanent of a doubly stochastic [3] $n \times n$ matrix B: $per(B) \geq n!/n^n$. The basic idea is to use the linearity of permanents w.r.t. multiplication with constants and transform the original matrix A to an approximated doubly stochastic matrix B and then exploit the lower bound. The algorithm that they proposed runs in $O(n^5 \log^2 n)$ and gives an approximation within a factor of e^n.

Markov Chain Monte Carlo Methods Markov Chains can be a powerful tool to generate almost uniform samples. They have been used for the permanent in [6] but they impose strong restrictions on the minimum vertex degree. A notable breakthrough was achieved by Jerrum et al. [7]: they proposed the first polynomial approximation algorithm for general matrices with non-negative entries. Nonetheless this remarkable result has to face its impracticality due to a very high-computational complexity $\tilde{O}(n^{26})$ improved to $\Theta(n^{10} \log^2 n)$ later on.

Note that for our purposes we are not only interested in computing the total number of solutions but we also need that solution densities for each variable-value pair. Moreover, we need fast algorithms that work on the majority of the matrices; since the objective is to build a search heuristic based on counting information, we would prefer a fast algorithm with less precise approximation over a slower algorithm with better approximation guarantees. With that in mind, Markov Chain-based algorithms do not fit our needs (they are either too slow or they have a precondition on the minimum vertex degree). Determinant based algorithms are either exponential in time or give too loose approximations (within an exponential factor) as well as algorithms based on matrix scaling. The approach that seems to suit our needs better is elementary iterative algorithms. It combines a reasonable complexity with a good approximation. Although it gives poor results for sparse matrices, those cases are likely to appear close to the leaves of the search tree where an error by the heuristics has a limited negative impact.

4.2 Rasmussen's estimator and its extensions

Suppose we want to estimate a function Q (in our case the permanent): a traditional approach is to design an estimator that outputs a random variable X whose expected value is equal to Q. The estimator is unbiased if $E(X)$ and $E(X^2)$ are finite. A straightforward application of Chebyshev's inequality shows that if we conduct $O(\frac{E(X^2)}{E(X)^2}\epsilon^{-2})$ independent and identically distributed trials and we take the mean of the outcomes then we have guarantee of ϵ-approximation. Hence the performance of a single run of the estimator and the ratio $\frac{E(X^2)}{E(X)^2}$ (*critical ratio*) determine the efficiency of the algorithm.
In the following, we denote by $A(n,p)$ the class of random (0-1) $n \times n$ matrices in which each element has independent probability p of being 1.

[3] $\sum_i a_{i,j} = \sum_j a_{i,j} = 1$

We write X_A for the random variable that estimate the permanent of the matrix A; $A_{i,j}$ denotes the submatrix obtained from A by removing row i and column j. The pseudo-code of Rasmussen's estimator is shown in Algorithm 4; despite its simplicity compared to other techniques, the estimator is unbiased and shows good experimental behaviour. Rasmussen gave theoretical results for his algorithm applied to random matrices belonging to the class $A(n, p \geq 1/2)$. He proved that for "almost all" matrices of this class, the critical ratio is bounded by $O(n\omega)$ where ω is a function satisfying $\omega \to \infty$ as $n \to \infty$; the complexity of a single run of the estimator is $O(n^2)$, hence the total complexity is $O(n^3\omega)$. Here "almost all" means that the algorithm gives a correct approximation with probability that goes to 1 as $n \to \infty$. While this result holds for dense matrices, it breaks down for sparse matrices. Note however that there are still matrices belonging to $A(n, p = 1/2)$ for which the critical ratio is exponential. Consider for instance the upper triangular matrix:

$$\mathbf{U} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ & 1 & \dots & 1 \\ & & \ddots & \vdots \\ & & & 1 \end{pmatrix}$$

For this particular matrix Rasmussen's estimator has expected value $E(X_U) = 1$ and $E(X_U^2) = n!$, hence the approximation is likely to be very poor.

```
1  if n = 0 then
2      X_A = 1
3  else
4      W = {j : a_{1,j} = 1};
5      if W = ∅ then
6          X_A = 0;
7      else
8          Choose j u.a.r. from W;
9          Compute X_{A_{1,j}};
10         X_A = |W| · X_{A_{1,j}};
```

Algorithm 4: Rasmussen's estimator

Fürer et al. [3] enhanced Rasmussen's algorithm with some branching strategies in order to pick up more samples in the critical parts of the matrix. It resembles very closely the exploration of a search tree. Instead of choosing u.a.r. a single column j from W, Fürer picks up a subset $J \subseteq W$ and it iterates on each element of J. The number of times it branches is logarithmic in the size of the matrix, and for a given branching factor he showed that a single run of the estimator still takes $O(n^2)$ time. The advantage of this approach resides in the theoretical conver-

gence guarantee: the number of required samples is only $O(\omega)$ instead of Rasmussen's $O(n\omega)$, thus the overall complexity is $O(n^2\omega)$.
Both Fürer and Rasmussen estimators allow to approximately compute the total number of solution of an `alldifferent` constraint. However if we need to compute the solution density $\sigma(x_i, d, \gamma)$ we are forced to recall the estimators on the submatrix $A_{i,d}$. Hence the approximated solution density is:

$$\sigma(x_i, d, \gamma) \approx \frac{E(X_{A_{i,d}})}{E(X_A)} \tag{2}$$

Adding propagation to the estimator A simple way to improve the quality of the approximation is to add propagation to Rasmussen's estimator. After randomly choosing a row i and a column j, we can propagate on the submatrix $A_{i,j}$ in order to remove all the 1-entries (edges) that do not belong to any maximum matching (the pseudo-code is shown in Algorithm 5). This broadens the applicability of the method; in matrices such as the upper triangular matrix, the propagation can easily lead to the identity matrix for which the estimator performs exactly. However, as a drawback, the propagation takes an initial precomputation of $O(\sqrt{n}m)$ plus an additional $O(n+m)$ each time it is called [15] (here m is the number of ones of the matrix i.e. edges of the graph). A single run of the estimator requires n propagation calls, hence the time complexity is $O(nm)$; the overall time complexity is then $O(n^2m\omega)$.

```
1 if n = 0 then
2     X_A = 1
3 else
4     Choose i u.a.r. from {1 ... n};
5     W = {j : a_{i,j} = 1};
6     Choose j u.a.r. from W;
7     Propagation on A_{i,j};
8     Compute X_{A_{i,j}};
9     X_A = |W| · X_{A_{i,j}};
```

Algorithm 5: Estimator with propagation

A particularity of the new estimator is that it removes a priori all the 1-entries that do not lead to a solution. Hence it always samples feasible solutions whereas Rasmussen's ends up with infeasible solutions whenever it reaches a case in which $W = \emptyset$. This opens the door also to an alternative evaluation of the solution densities; given the set of solution samples S, we denote by $S_{x_i,d} \subseteq S$ the subset of samples in which $x_i = d$. The solution densities are approximated as:

$$\sigma(x_i, d, \gamma) \approx \frac{|S_{x_i,d}|}{|S|} \tag{3}$$

Experimental results showed a much better approximation quality for the computation of the solution densities using samples (3) instead of using

submatrix counting (2). It is worth pointing out that Fürer's provides several samples in a single run but highly biased from the decisions taken close to the root of the search tree; thus it cannot be used to compute solution densities from samples. Due to the better results obtained using samples, we decide not to apply propagation methods to Fürer's.

% Removals	0.1	0.2	0.3	0.4	0.5	0.6	0.7
Counting Error							
Rasmussen	1.32	1.76	3.66	5.78	7.19	13.80	22.65
Fürer	0.69	1.07	1.76	2.17	2.52	4.09	5.33
CountS	1.44	1.51	2.48	2.30	4.31	3.94	1.23
Average Solution Density Error							
Rasmussen	1.13	1.83	3.12	5.12	7.85	13.10	23.10
Fürer	0.58	0.92	1.55	2.49	3.74	6.25	8.06
CountS	0.73	0.76	0.80	1.01	1.33	1.81	2.03
Maximum Solution Density Error							
Rasmussen	3.91	6.57	11.60	19.86	30.32	42.53	40.51
Fürer	2.09	3.20	5.75	9.36	15.15	21.18	15.01
CountS	2.64	2.60	2.89	3.90	5.39	6.03	2.61

Table 1. Estimators performance.

Estimator benchmarks We compared three estimators: Rasmussen's, Fürer's, and ours (the version based on samples,"CountS"). Due to the very high computational time required to compute the exact number of solutions, we performed systematic experiments on `alldifferent` of size 10, 11 and 12 with varying percentage of domain value removals. Table 1 shows the error on the total number of solutions, the average and the maximum error on the solution densities (all the errors are expressed in percentage). The number of samples used is 100 times the size of the instance. The time taken for counting is slightly higher than one tenth of a second for our methods compared to one tenth for Fürer's and a few hundredths for Rasmussen's. On the other side, exact counting can take up to thousands of seconds for very loose instances to a few hundredths of a second. Due to lack of room, we do not show the tests with a common time limit: the situation is pretty much the same, with our method showing the best approximations. Note that we also tested our method with instances of bigger size (up to 30) and even with few samples (10 times the instance size): the average error remains pretty low (again on the order of 2-4%) as well as the maximum error. The current implementation of our approach makes use of Ilog Solver 6.2; we believe that a custom implementation can gain in performance, avoiding the overhead due to model extraction and to backtrack information bookkeeping.

5 Experimental Results

We evaluate the proposed constraint-centered search heuristics on two benchmark problems modeled with the `alldifferent` and `regular` constraints.

Nonogram A Nonogram (problem 12 of CSPLib) is built on a rectangular $n \times m$ grid and requires filling in some of the squares in the unique feasible way according to some clues given on each row and column. As a reward, one gets a pretty monochromatic picture. Each individual clue indicates how many sequences of consecutive filled-in squares there are in the row (column), with their respective size in order of appearance. Each sequence is separated from the others by at least one blank square but we know little about their actual position in the row (column). Such clues can be modeled with `regular` constraints (the actual automata $\mathcal{A}_i^r, \mathcal{A}_j^c$ are not difficult to derive but lie outside the scope of this paper):

$$\begin{array}{ll} \texttt{regular}((x_{ij})_{1\leq j\leq m}, \mathcal{A}_i^r) & 1 \leq i \leq n \\ \texttt{regular}((x_{ij})_{1\leq i\leq n}, \mathcal{A}_j^c) & 1 \leq j \leq m \\ x_{ij} \in \{0,1\} & 1 \leq i \leq n,\ 1 \leq j \leq m \end{array}$$

These puzzles typically require some amount of search, despite the fact that domain consistency is maintained on each clue. We experimented with 75 instances of sizes ranging from 16×16 to 24×24.
We compared four search heuristics: random selection for both variable and value, dom/ddeg variable selection with min conflicts value selection, MaxSD, and MinSC;MaxSD. A variable selection heuristic based solely on domain size is not useful for this problem since every unbound variable has an identical domain of size 2. Note also that for the same reason the min conflicts value selection does not discriminate at all.

heuristic	avg btk	median btk	total time
random var/val	348.0	16	40.7
dom/ddeg ; min conflicts	33640.4	146	4405.2
MaxSD	236.0	2	57.0
MinSC;MaxSD	48.5	3	8.9

Table 2. Number of backtracks and computation time (in seconds) for 75 Nonogram instances.

Table 2 reports the average and median number of backtracks and the total computation time for these heuristics. dom/ddeg is definitely ill-suited for such a problem: the statistics reported should even be higher since ten instances where interrupted after five minutes of computation. A purely random heuristic performs fairly well here, which can be explained by the binary domains of the variables: even a random choice of value has a 50% chance of success. MaxSD performs better than the random heuristic in terms of backtracks but not enough to offset its higher computational cost, yielding a slightly higher computation time. MinSC;MaxSD is the best of the four, with a significantly lower average

number of backtracks and the best computation time. The difference in performance between our two heuristics is actually strongly influenced by a few instances for which MaxSD behaved poorly: if we look at the median number of backtracks, the two are very close and markedly lower than for the random heuristic.

Quasigroup with Holes A Latin Square of order n is defined on a $n \times n$ grid whose squares each contain an integer from 1 to n such that each integer appears exactly once per row and column. The Quasigroup with Holes (QWH) problem gives a partially-filled Latin Square instance and asks to complete it. It is easily modeled as:

$$\begin{array}{ll} \texttt{alldifferent}((x_{ij})_{1\leq j\leq n}) & 1 \leq i \leq n \\ \texttt{alldifferent}((x_{ij})_{1\leq i\leq n}) & 1 \leq j \leq n \\ x_{ij} = d & (i,j,d) \in S \\ x_{ij} \in \{1,2,\ldots,n\} & 1 \leq i,j \leq n \end{array}$$

We tested four search heuristics: dom/ddeg variable selection with min conflicts value selection (one of the most robust heuristics for QWH), MinDom;MaxSD, MaxSD, and a lazy version of MaxSD. For counting, we used an exact algorithm for 0.1 seconds and, in case of timeout, we ran CountS for another 0.1 seconds. Note that the counting is done only if a domain event occurs, that is, the counting algorithm is woken up in a way that is similar to constraint propagation. The lazy version of maximum solution density recounts at each event when the search is close to the tree root (whenever less than 20% of variables are assigned), every 2 events when the unbound variables are between 20% and 50% and every 3 events thereafter. The four heuristics were tested on 40 balanced QWH instances with about 41% of holes, randomly generated following [4]. We set the time limit to 1200 seconds. Table 3 shows the results. The

heuristic	avg btk	median btk	total time	unsolved
dom/ddeg ; min conflicts	788887.1	365230.5	19070.7	10
MinDom;MaxSD	17626.3	10001.5	25983.8	19
MaxSD	5634.0	2534.2	11371.3	1
LazyMaxSD	7479.6	2243.7	10258.0	2

Table 3. Number of backtracks, computation time (in seconds) and the number of unsolved instances for 40 hard QWH instances of order 30.

heuristics based on maximum density were the ones performing better in term of backtracks (two orders of magnitude of difference), total time and number of instances solved. We also ran some tests on easier instances outside the phase transition: the dom/ddeg heuristic did better than our heuristics in terms of running time but not in terms of number of backtracks. It is worth mentioning that the number of backtracks by our heuristics only diminished slightly on these easier instances, so the heuristics appear fairly robust throughout the range.

6 Conclusion and Open Issues

This paper advocated using constraints not only for inference but also for search. The key idea is to use solution counting information at the level of individual constraints. We showed that for some widely-used constraints such information could be computed efficiently, especially given the support already in place for domain filtering. We also proposed novel search heuristics based on solution counting and showed their effectiveness through experiments.

From the point of view of CP systems, we are really introducing a new functionality for constraints alongside satisfiability testing, consistency and domain filtering, entailment, etc. As we argued, providing this support does not necessarily require a lot of extra work. It would, however, benefit from some thinking about how best to offer access to solution counts and solution densities, from a programming language design perspective.

We believe there are still several open issues regarding this work. Even though we have had some success with the search heuristics we proposed, little has been tried so far about combining the information originating from the different constraints, which should increase robustness in cases where the constraints give hugely conflicting information. We saw already that some compromises were attempted for the `alldifferent` constraint to cut down its computation time — a more in-depth investigation is required, including finding out a way to make it more incremental. Finally there are many more families of constraints for which efficient solution counting algorithms must be found.

References

1. Barvinok, A. 1999. Polynomial time algorithms to approximate permanents and mixed discriminants within a simply exponential factor. *Random Structures and Algorithms* 14, 29–61.
2. Chien, S.; Rasmussen, L.; and Sinclair, A. 2002. Clifford Algebras and Approximating the Permanent. In *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*, ACM Press, 222–231.
3. Fürer, M and Kasiviswanathan, S. P. 2004. An Almost Linear Time Approximation Algorithm for the Permanent of a Random (0-1) Matrix. In FSTTCS 2004. Springer-Verlag LNCS 3258. 54–61.
4. Gomez, C. P. and Shmoys, D. 2002. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *Proc. Computational Symposium on Graph Coloring and Generalizations*.
5. Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proc. AAAI'06*. 54–61.
6. Huber, M. 2006. Exact Sampling from Perfect Matchings of Dense Regular Bipartite Graphs. *Algorithmica* 44, 183–193.
7. Jerrum, M.; Sinclair, A.; and Vigoda, E. 2001. A Polynomial-time Approximation Algorithm for the Permanent of a Matrix with Non-Negative entries. In *Proc. 33th Annual ACM Symposium on Theory of Computing (STOC)*, ACM Press, 712–721.

8. Linial, N.; Samorodnitsky, A.; and Wigderson, A. 200. A deterministic strongly polynomial algorithm for matrix scaling and approximate permanents, *Combinatorica* 20, 545–568.
9. Kask, K.; Dechter, R.; and Gogate, V. 2004. Counting-Based Look-Ahead Schemes for Constraint Satisfaction. In *Proc. CP'04*. Springer-Verlag LNCS 3258. 317–331.
10. Patel, J., and Chinneck, J. W. 2006. Active-Constraint Variable Ordering for Faster Feasibility of Mixed Integer Linear Programs. *Mathematical Programming*.
11. Pesant, G. 2004. A regular language membership constraint for finite sequences of variables. In *Proc. CP'04*, Springer-Verlag LNCS 3258, 482–495
12. Pesant, G. 2005. Counting Solutions of CSPs: A Structural Approach. In *Proc. IJCAI'05*. 260–265.
13. Rasmussen, L. E. 1994. Approximating the permanent: a simple approach. *Random Structures and Algorithms* 5, 349-361.
14. Refalo, P. 2004. Impact-Based Search Strategies for Constraint Programming. In *Proc. CP'04*. Springer-Verlag LNCS 3258. 557–571.
15. Régin, J.-C. 1994. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proc. Twelfth National Conference on Artificial Intelligence (AAAI)*, volume 1. AAAI Press. 362–367.
16. Valiant, L. G. 1979. The complexity of computing the permanent, *Theoretical Computer Science* 8, 189–201.