Solution Counting Algorithms for Constraint-Centered Search Heuristics*

Alessandro Zanarini and Gilles Pesant

Department of Computer and Software Engineering
École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville
Montreal, Canada
{azanarini,pesant}@crt.umontreal.ca

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 `alldifferent`, 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.

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

* An earlier version of this paper appeared as [24].

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. Polynomial time approximate or exact algorithms to count the number of solutions of several common families of constraints were given in [16]. 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. [13] 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 [19] 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. Hsu et al. [9] apply a Belief Propagation algorithm within an Expectation Maximization framework in order to approximate variable biases (or marginals) i.e. the probability a variable takes a given value in a solution. Even though the approach is general, they derived formulas for computing and updating the variable biases only for the `alldifferent` constraint and they experimented on the Quasigroup with Holes Problem. The computation of variable biases is time-consuming but they show that the heuristic is effective even when it is employed only in the top part of the search tree.

Evaluating the number of solutions of Boolean formulas and of CSPs has received considerable attention lately. Gomes et al. [8] compute bounds on the number of solutions of SAT instances by adding to the problem so-called streamlining constraints. Gogate and Dechter [7] propose to sample solutions of a problem in order to estimate the number of solutions; they then use a sampling importance re-sampling technique that gives approximation guarantees to the results. Such works share our interest in evaluating how many solutions involve a particular variable assignment, however the main difference between our work and these is that we focus on individual constraints whereas they consider the problem as a whole. Counting on individual constraints gives more precise information (exact or approximated) but it lacks the global view of the problem; nevertheless work such as [8] and [7] require to find solutions of the problem making them useless to design search heuristics (since a solution has already been found). An approach such as [9] does not require finding solutions, the information it provides is global but not as accurate as the one we compute.

As an interesting connection for constraint-centered heuristics, Patel and Chinneck [14] investigate several variable selection heuristics guided by the individual 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 and experiment with new search heuristics based on solution counting information at the level of individual constraints. Second, we propose 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 [16], what is proposed for the former is a considerable improvement and for the latter it details what was only alluded to before.

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, part of 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 for example 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*

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

the solution density[1] *of pair (x_i, d) in γ. It measures how often a certain assignment is part of a solution.*

We can for example favour high solution densities 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 subset of variables. Algorithms 1 to 3 define the search heuristics with which we will experiment in Section 5. There are obviously many possibilities one could consider — we have investigated more than these three and we will say a few words about them at the end of Section 5.

```
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 heuristic MaxSD (Algorithm 1) simply iterates over all the variable-value pairs and chooses the one that has the highest density; assuming that the $\sigma(x_i, d, \gamma)$ are precomputed, the complexity of the algorithm is $O(qm)$ where q is the number of constraints and m is the sum of the cardinalities of the variables' domains. Interestingly, such a heuristic likely selects a variable with a small domain, in keeping with the *fail-first principle*, since its values have on average a higher density compared to a variable with many values (consider that the average density of a value is $\sigma(x_i, d, \gamma) = \frac{1}{|D_i|}$). Note that each constraint is considered individually i.e. there is no sort of combination of the information provided by different constraints for the same variable-value pair.

Heuristic MinSC;MaxSD (Algorithm 2) first selects the constraint with the lowest number of solutions (line 2) and then iterates only over the variables involved in this constraint, choosing the variable-value pair with the highest solution density. The rationale behind this heuristic is that the constraint with the fewest solutions is probably among the hardest to satisfy. Assuming again that solution counting information is precomputed, the complexity of the algorithm

[1] Also referred to as *marginal* in some of the literature.

```
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

is $O(q + m)$ where m this time is restricted to the variables in the scope of the constraint selected.

```
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

Heuristic MinDom;MaxSD (Algorithm 3) considers only the variables with the smallest domain size (line 2) and among these selects the variable-value pair with the highest solution density. The complexity of the algorithm is $O(qm)$.

The above worst-case time complexity analysis is not necessarily indicative of the average behaviour of the heuristics. MaxSD requires looking at every variable-value pair for every constraint but MinDom;MaxSD only needs to process the constraints that include the variables with the smallest domain and solution density information is only required for those variables. For this reason it is particularly suited for problems that involve constraints whose solution counting or solution density procedures are particularly time consuming. MinSC;MaxSD requires the solution count of every constraint but then only needs the solution density of variable-value pairs appearing in the chosen constraint. This too is advantageous if solution densities come at an additional computational price.

3 Counting for Regular Constraints

The `regular`(X, Π) constraint [15] 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, $\delta : Q \times \Sigma \rightarrow Q$ 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 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 [15]. Figure 1 gives an example of a layered directed graph built for one such constraint on five variables; layers are vertical and denoted by $L_1, \ldots, L_6$, vertices within a layer correspond to states of the automaton. An arc joining a vertex of layer L_i to another of layer L_{i+1} represents a feasible value for variable x_i: the arc's colour stands for the value.

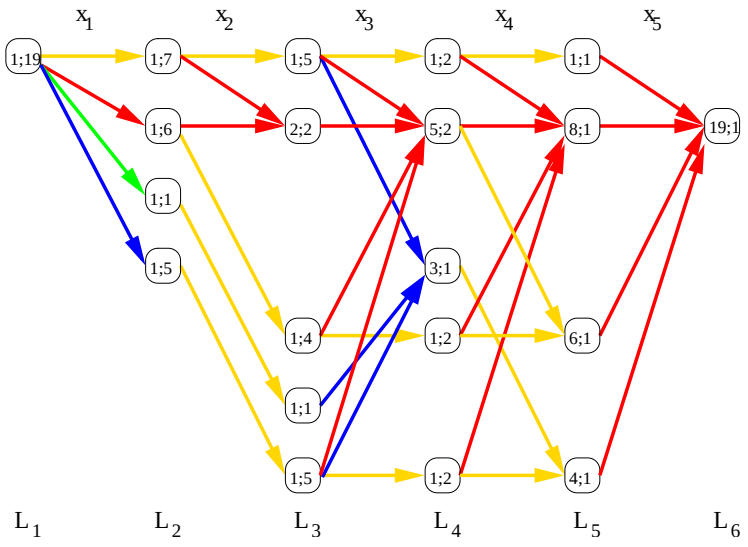

Fig. 1. The layered directed graph built for a `regular` constraint on five variables. Vertex labels represent the number of incoming and outgoing paths.

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 (respectively following) layers only if the out-degree (respectively 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 from 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 (linear on the size of the graph). Therefore maintaining the additional data structures could prove to be too expensive.

3.3 A Lazy Evaluation Version

Rather than keeping updated information on $\#op()$ and $\#ip()$ throughout the solving process (that is during both constraint propagation and branching), it is possible to compute them on an as-needed basis i.e. just before branching. The advantage is twofold: firstly we avoid computing counting information systematically for each constraint, which might end up being unnecessary; secondly we avoid updating several times the counting information (during propagation) before it is actually used (during branching). Counting is thus performed only when a request for the solution count is received. Furthermore counting information is recomputed and cached if and only if a domain event has occurred in the constraint. The request for the solution count triggers the computation of the required $\#op()$ and $\#ip()$ values. If no change in the constraint occurred

since those values were last computed, they are simply looked up in a table. Otherwise they are computed iteratively and cached before they are returned to avoid recomputing them.

On the Nonogram problem introduced in Section 5, the lazy evaluation version was 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 [20].

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 $i \in V_1$ is connected to vertex $j \in V_2$. 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.
Equivalently, the permanent can be expressed following Laplace's expansion formula:

$$per(A) = \sum_{j=1}^{n} a_{1,j}\, per(A_{1,j}) \tag{2}$$

where $A_{1,j}$ denotes the submatrix obtained from A by removing row 1 and column j (the permanent of the empty matrix is equal to 1). In the following, we will freely use both matrix and graph representations.

Note that if the `alldifferent` constraint features more values than variables, we can augment the corresponding value graph by adding $|D_X| - |X|$ vertices to X, each connected to every vertex in D_X. The number of maximum matchings in this augmented graph will be $(|D_X| - |X|)!$ times that of our original graph.

4.1 Computing the Permanent

The problem of computing the permanent 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 (Formula 2) resembles that of the determinant, there has been few advances on its exact computation. In 1961, Kasteleyn solved the problem only for a particular class of matrices (the one that represents Pfaffian graphs which is a superset of planar graphs); his algorithm runs in $O(n^3)$. In 1963, Ryser [21] proposed an exact algorithm whose running time is $\Theta(n2^n)$, thus it is practically useless for our purposes. In 1979, Valiant [23] proved that the problem is $\#P$-complete, even for 0-1 matrices, that is, under reasonable assumptions, it cannot be computed in polynomial time in the general case. 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 [18] 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 [4] (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 [3] or an approximation within an exponential factor in polytime [1].

Iterative Balancing. The work of Linial et al. [12] exploits a lower bound on the permanent of a doubly stochastic[2] $n \times n$ 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 [10] but they impose strong restrictions on the minimum vertex degree. A notable breakthrough was achieved by Jerrum et al. [11]: 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

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

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 solution densities for each variable-value pair. Moreover, we need fast algorithms that work for most 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). Algorithms based on determinants or matrix scaling are either exponential in time or give approximations that are too loose (within an exponential factor). 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 are guaranteed an ϵ-approximation. Hence the performance of a single run of the estimator and the ratio $\frac{E(X^2)}{E(X)^2}$ (called the *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. We write X_A for the random variable that estimates the permanent of matrix A; $A_{i,j}$ denotes the submatrix obtained from A by removing row i and column j. The pseudocode 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 in $O(n^2)$, hence the total complexity is in $O(n^3\omega)$. 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:

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

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 uniformly at random from W;
9          Compute X_{A_{1,j}};
10         X_A = |W| · X_{A_{1,j}};
```

Algorithm 4: Rasmussen's estimator

Fürer et al. [4] enhanced Rasmussen's algorithm with some branching strategies in order to pick up more samples in the critical parts of the matrix (Algorithm 5). 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 convergence 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:

$$\sigma(x_i, d, \gamma) \approx \frac{E(X_{A_{i,d}})}{E(X_A)} \tag{3}$$

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

```
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          if  n = s^i, i ≥ 1 then
9              K = r;
10         else
11             K = 1;
12         for ℓ = 1 to K do
13             Choose j_ℓ uniformly at random from W;
14             Compute X_{A_{1,j_ℓ}};
15         X_A = |W|(1/K Σ_{ℓ=1}^{K} X_{A_{1,j_ℓ}});
```

Algorithm 5: Fürer's estimator

maximum matching (the pseudo-code is shown in Algorithm 6). 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 pre-computation of $O(\sqrt{n}m)$ plus an additional $O(n+m)$ each time it is called [20] (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 $O(n^2 m\omega)$.

```
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 uniformly at random from W;
7      Propagation on A_{i,j};
8      Compute X_{A_{i,j}};
9      X_A = |W| · X_{A_{i,j}};
```

Algorithm 6: 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:

$$\sigma(x_i, d, \gamma) \approx \frac{|S_{x_i,d}|}{|S|} \tag{4}$$

Experimental results showed a much better approximation quality for the computation of the solution densities using samples (4) instead of using submatrix counting (3). 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 decided 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
CountSMC	1.24	1.60	2.12	2.58	4.43	4.11	1.98
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
CountSMC	1.01	1.65	2.36	2.91	3.58	3.19	2.21
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
CountSMC	3.24	5.90	10.44	12.11	15.92	10.57	3.27
CountS	2.64	2.60	2.89	3.90	5.39	6.03	2.61

Table 1. Estimators performance.

Estimator Benchmarks We compared four estimators: Rasmussen's, Fürer's, and our estimators based on submatrix counting (3) and on samples (4) (called respectively "CountSMC" and "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; we varied the percentage of values removed from the variables' domains (in the table referred to as 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. Tests with a common

time limit led to similar conclusions, with our method showing the best approximations. 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):

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

This is a very homogeneous problem, with constraints of identical type defined over m or n variables, and with each (binary) variable involved in two constraints. These puzzles typically require some amount of search, despite the fact that domain consistency is maintained on each clue. We experimented with 180 instances[3] of sizes ranging from 16×16 to 32×32.

We compared eight search heuristics: random selection for both variable and value, dom/ddeg variable selection with random value selection, IlogSolver 6.5+IBS, IlogSolver 6.5+AdvIBS, IlogCPOpt, IlogCPOpt+IBS, MaxSD, and MinSC;MaxSD. The two heuristics identified as IlogCPOpt use Ilog CP Optimizer 1.0, whereas the other ones have been developed using Ilog Solver 6.2 (when not specified) or 6.5. We experimented with Ilog Solver 6.5 since it provides Impact Based Search (IBS): although it is not explicitely documented, we believe it resembles very much the heuristic introduced in [19]. This heuristic chooses first the variable whose instantantion triggers the largest search space reduction (highest impact) that is approximated as the reduction of the cartesian product of the variables' domains. The impact is either approximated as the average reduction observed during the search or computed exactly at a given node of the

[3] Instances taken from http://www.blindchicken.com/∼ali/games/puzzles.html

search (the exact computation provides better information but it is more time consuming). The heuristic IlogSolver 6.5+IBS is Impact Based Search where the impacts are approximated; IlogSolver 6.5+AdvIBS chooses a subset of 5 variables with the best approximated impacts and then it breaks ties based on the exact impacts; further ties are broken randomly (this heuristic is mentioned in Ilog's documentation). IlogCPOpt is a depth-first search with the default search heuristic; IlogCPOpt+IBS uses Impact Based Search in a depth-first search; Ilog CP Optimizer's IBS is not explicitely documented and we believe it is a finetuned version of the one presented in [19]. Note that 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, so in this particular problem MaxSD and MinSize;MaxSD coincide. Note also that for the same reason the min conflicts value selection does not discriminate at all.

We ran the tests on a AMD Opteron 2.4GHz with 1.5GB of RAM. The code based on Ilog CP Optimizer has been run on a AMD Opteron 2.2GHz with 1GB. We set a timeout of 10 minutes for each run. Each test has been repeated 10 times and we considered the arithmetic averages of times and backtracks.

Because of implementation compatibility issues between Ilog CP Optimizer and Ilog Solver, we could not use the same code for the propagator of the Regular constraint. We tested the difference in performance of the Ilog Solver-based propagator and the Ilog CP Optimizer-based propagator (using the same testbed and the same heuristic) and the Ilog CP Optimizer code turned out to be about 40% faster than Ilog Solver's (this difference is probably due to both the propagator codes and the solver themselves). Therefore the runtimes we report can only be compared approximately.

heuristic	avg btk	median btk	total time	% unsolved
random var/val	62662.4	193.9	20259.3	16.2%
dom/ddeg ; random value	92091.0	944.4	29606.7	24.3%
IlogSolver 6.5+IBS	2741.2	735.0	733.1	0.6%
IlogSolver 6.5+AdvIBS	6003.9	742.0	725.9	0.6%
IlogCPOpt	102376.5	233.4	18509.4	15.4%
IlogCPOpt+IBS	5024.2	8.0	702.9	0.1%
MaxSD	8385.1	6.0	5029.6	3.9%
MinSC;MaxSD	9134.8	5.0	3042.5	2.2%

Table 2. Number of backtracks, computation time (in seconds) and percentage of unsolved instances for 180 Nonogram instances.

Table 2 reports the average and median number of backtracks, the total computation time and the percentage of unsolved instances for these heuristics; note that unsolved instances are taken into account in time and backtrack averages and median. dom/ddeg is definitely ill-suited for such a problem. A purely random heuristic doesn't perform too badly here, which can be explained by the binary domains of the variables: even a random choice of value has a 50% chance

of success. The default heuristic of Ilog CP Optimizer seems to perform slightly better than the random heuristic but the situation changes if you consider the 40% performance advantage of the propagator code based on Ilog CP Optimizer. In terms of backtracks the heuristics based on counting (MaxSD, MinSC;MaxSD) and impacts (IlogCPOpt+IBS, IlogSolver 6.5+IBS, IlogSolver 6.5+AdvIBS) behave pretty similarly and they are one order of magnitude better than the other heuristics. In Figure 2, we plot the percentage of instances solved within a given time (the time is not cumulative but given to each instance).

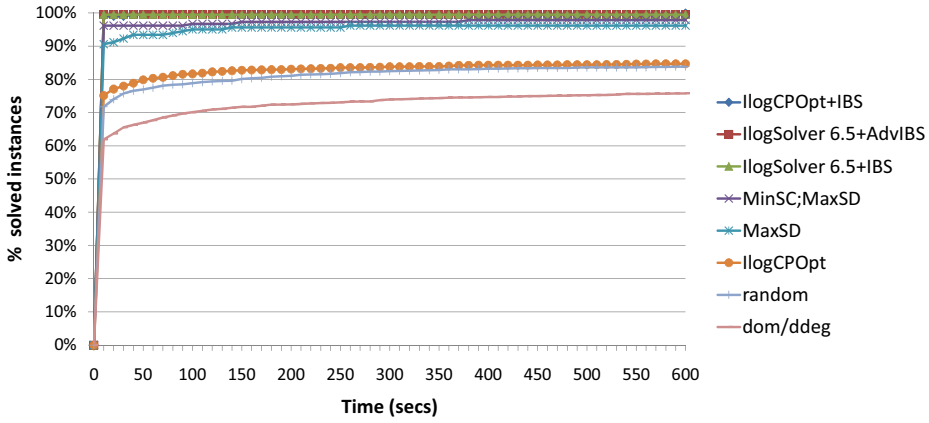

Fig. 2. Nonograms: % instances solved vs time (secs).

Our heuristics in most of the instances behaved like the impact based search; only few instances required more computing time than IlogCPOpt+IBS yielding a significant increase of the total running time shown in Table 2. It's interesting to observe that the two Ilog Solver 6.5 heuristics behave almost identically meaning that, in this particular problem, the exact computation of the impacts or the approximation of them does not really make a difference. If we compare the number of backtracks (Figure 3, note that the x-axis is logarithmic), MinSC;MaxSD typically required fewer backtracks than IlogCPOpt+IBS to solve the majority of the instances. Note that in Figure 3 the curves stop either because they hit 100% of solved instances or because they were not able to perform more backtracks within the time limit. IlogCPOpt, dom/ddeg and the random heuristics were able to explore a larger portion of the search tree without necessarily solving more instances. Interestingly, the random and IlogCPOpt heuristics behave very similarly.

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

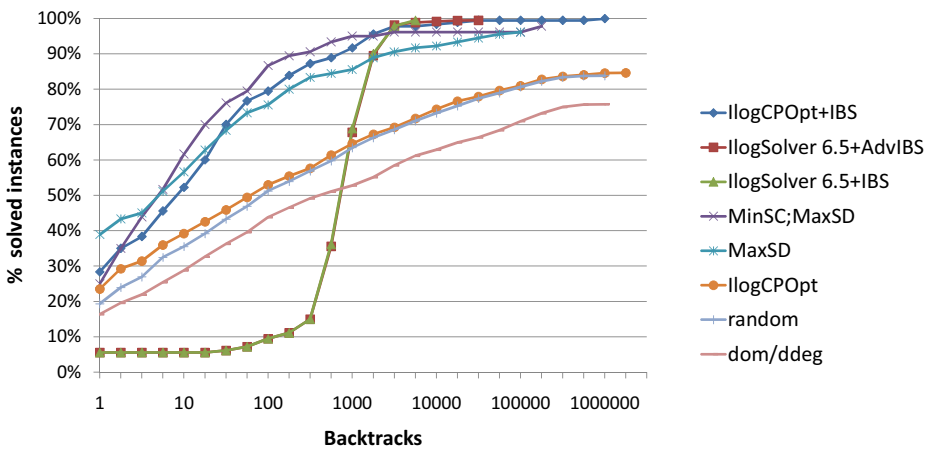

Fig. 3. Nonograms: % instances solved vs log backtracks.

gives a partially-filled Latin Square instance and asks to complete it. It is easily modeled as:

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

In this problem, each constraint is defined on n variables and is of the same type; each variable is involved in two constraints and has the same domain (disregarding the clues). This is also a very homogeneous problem.

We tested eight search heuristics: dom/ddeg variable selection with min conflicts value selection (one of the most robust heuristics for QWH), IlogSolver 6.5+IBS, IlogSolver 6.5+AdvIBS, IlogCPOpt, IlogCPOpt+IBS, EMBP-a, MinDom;MaxSD, and MaxSD. The IlogCPOpt and IlogCPOpt+IBS, IlogSolver 6.5+IBS, IlogSolver 6.5+AdvIBS heuristics are the same as the ones used for the nonogram problem. The EMBP-a is our own implementation of the algorithm presented in [9]: we chose the version in which variable biases are updated through an approximated formula that is shown to have the best tradeoff between speed and accuracy. Note however that our implementation differs from Hsu et al.'s since we apply their heuristic throughout the search tree whereas they applied it only in the first 5% variables' instantiations (afterwards they use the quicker and simpler dom/wdeg). We got mixed results for MinSC;MaxSD and did not report them here. For counting, we used an exact algorithm for 0.25 seconds and, in case of timeout, we ran CountS for another 0.25 seconds. The advantage of the exact counting algorithm is twofold: if it does not timeout it provides exact information and, most important, when the search gets close to the leaves of the search tree the solution count for each constraint is relatively low, so an exact enumeration is extremely fast (faster

than sampling). As a drawback, however, at the beginning of the search tree the exact algorithm possibly times out. Note that the counting is performed only if a relevant domain changed.

The eight heuristics were tested on 40 balanced QWH instances with about 41% holes, randomly generated following [6]. For the heuristics that involve some sort of randomization, we ran the test 10 times and report the average. We set the time limit to 20 minutes. The machine specifications are the same as in the nonogram tests. Table 3 shows the results.

heuristic	avg btk	median btk	total time	% unsolved
dom/ddeg ; min conflicts	788887.1	365230.5	19070.7	25.00%
IlogSolver 6.5+IBS	1276892.0	2040017.0	28395.9	55.00%
IlogSolver 6.5+AdvIBS	1028093.8	351063.4	12046.1	14.50%
IlogCPOpt	965308.0	998744.6	27024.5	43.75%
IlogCPOpt+IBS	436252.0	91294.0	12289.4	20.00%
EMBP-a	18398.5	21467.3	31108.2	53.25%
MinDom;MaxSD - 0.25 secs	19776.3	5881.5	11967.2	12.50%
MaxSD - 0.25 secs	3503.7	634.5	4939.1	0.00%

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

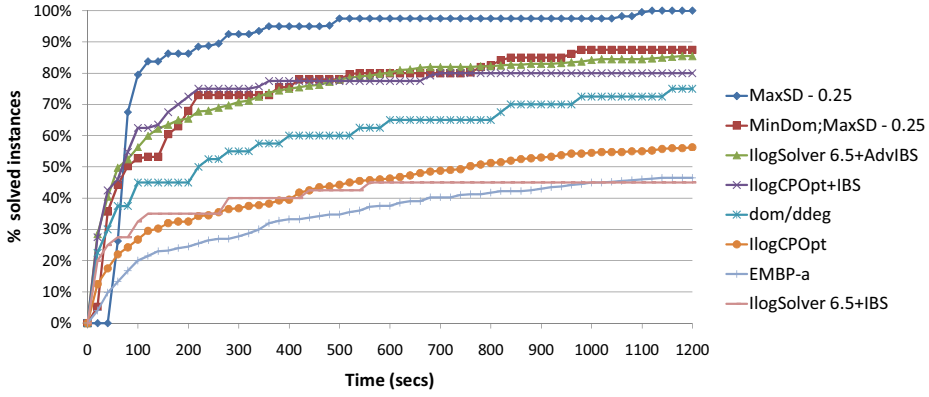

Fig. 4. QWH: % instances solved vs time (secs).

The 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. Note that IlogSolver 6.5+AdvIBS and IlogSolver 6.5+IBS – the worst performer along with EMBP-a – behave very differently: in the former the computation of the exact impacts (even for only a small subset

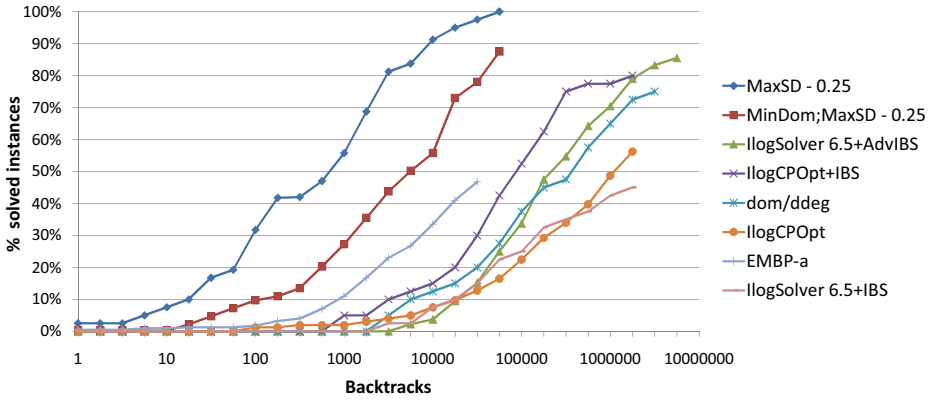

Fig. 5. QWH: % instances solved vs log backtracks.

of variables) makes a clear difference with respect to the latter which uses only approximated impacts. EMBP-a is computationally expensive so within the time limit is not able to perform many backtracks; so despite a fairly low number of fails it has a very high total solving time. In Figure 4, we plot the percentage of instances solved for a given time limit. In MaxSD, counting is particularly time consuming at the root node where all the constraints need to be processed whereas during the search it is slightly more light-weight since it is performed only on the constraints that experienced some variables' domain modifications. This overhead makes MaxSD relatively slow to solve the easier instances; in these instances MinDom;MaxSD is faster since for each search node it has to count only a subset of the constraints (the ones that involve the variables with minimum domain size). For this reason, we believe that MinDom;MaxSD is better suited for easy problems. However MaxSD catches up quickly and, with a time limit of 80 seconds, is able to solve more instances than any other heuristic tested. If we analyze the number of backtracks (Figure 5) the situation remains unchanged as counting-based heuristics perform better than Ilog's and dom/ddeg — they were able to solve more instances with a significantly lower number of backtracks. Again, as a drawback, the other heuristics were able to explore a larger portion (even though less promising) of the search tree, performing a higher number of backtracks within the time limit. Note that EMBP-a solves with a fairly low number of backtracks (lower than impact based heuristics but higher than counting based heuristics) about half of the instances but lacks in robustness leaving more than half of the instances unsolved.

We ran some tests on easier instances outside the phase transition: the other heuristics 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.

We also tested the robustness of MaxSD while varying the counting time dedicated to each constraint. We varied it from 0.05 to 0.30 seconds (with steps of 0.05 secs) and we compared it to the other heuristics. In Figure 6, we plotted the sum of the minimum and maximum solving times over the 10 runs as well as the average time for the different heuristics (we kept a 20 minute time limit). We also tested how our heuristic would behave in case of exact information: in MaxSD - Exact we removed the sampling phase and we simply left the exact counting algorithm run without a time limit.

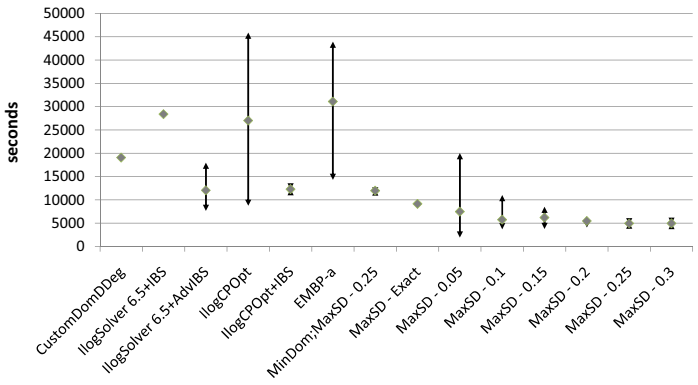

Fig. 6. Minimum, maximum and average total time with varying counting time.

heuristic	avg btk	avg total time	% Exact Counting
MaxSD - 0.05 secs	8228.4	7464.6	95.71%
MaxSD - 0.10 secs	5211.4	5745.0	97.35%
MaxSD - 0.15 secs	4567.1	6198.1	97.65%
MaxSD - 0.20 secs	4611.0	5458.7	97.80%
MaxSD - 0.25 secs	3503.7	4939.0	98.33%
MaxSD - 0.30 secs	3410.2	4952.4	98.67%
MaxSD - Exact	3826.4	9137.6	100%

Table 4. Average number of backtracks, total computation time (in seconds) and percentage of `alldifferent` constraints for which the exact counting algorithm did not time out

Using MaxSD, the difference between minimum and maximum solving time narrows while increasing the counting time. The reason is twofold: the precision of the sampling increases with the counting time; during the search the percentage of `alldifferent` constraints for which the exact counting does not time out increases as shown in the third column of Table 4. Even though exact counting provided fairly satisfying results, doing so does not scale well as the arity of the

`alldifferent` constraints increases or when they are less tight.

Other heuristics We also ran the same experiments on other heuristics mainly with the aim of finding a good way of aggregating solution counting information extracted from different constraints. Among the different ones we tried, we mention two aggregation functions: the average of the solution densities and the minimum between the solution densities. The heuristic selects the variable-value pair whose aggregated information is maximum. The two heuristics performed definitely worse compared to the MaxSD, leading to solving times from 2 to 4 times worse and number of backtracks from 3 to 7 times worse.

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.

Techniques such as randomized restart strategies [5] and some following works inspired by them (for instance [2]) and learning (possibly applied to randomized restarts [22]) present orthogonal aspects to what we propose here and we believe that the future integration of such techniques with counting-based heuristics can bring a further boost in search efficacy.

Finally there are many more families of constraints for which efficient solution counting algorithms must be found (see e.g. [17]).

Acknowledgements

The authors would like to thank the anonymous referees for constructive comments and Michael Mac-Vicar, Claude-Guy Quimper, and Ronan Le Bras for

helping in the experiments. Financial support for this research was provided in part by the Natural Sciences and Engineering Research Council of Canada and the Fonds québécois de la recherche sur la nature et les technologies.

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. Beck, J.C. 2007. Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. *Journal of Artificial Intelligence Research*, 29, 49–77.
3. Chien, S.; Rasmussen, L.; and Sinclair, A. 2002. Clifford Algebras and Approximating the Permanent. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC 2002)*, ACM Press, 222–231.
4. Fürer, M and Kasiviswanathan, S. P. 2004. An Almost Linear Time Approximation Algorithm for the Permanent of a Random (0-1) Matrix. In *Proceedings of the Twenty-Fourth International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, Springer-Verlag LNCS 3258, 54–61.
5. Gomes, C. 2003. Complete Randomized Backtrack Search. In *Constraint and Integer Programming: Toward a Unified Methodology*, Milano, M., ed. Kluwer, 233–283.
6. Gomes, C. and Shmoys, D. 2002. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations (COLOR 2002)*, 22–39.
7. Gogate, V. and Dechter, R. 2007. Approximate Counting by Sampling the Backtrack-free Search Space. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI 2007)*. AAAI Press, 198–203.
8. Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006)*, 54–61.
9. Hsu, E.I.; Kitching, M.; Bacchus, F. and McIlraith, S.A. 2007. Using EM to Find Likely Assignments for Solving CSP's. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI 2007)*. AAAI Press, 224–230.
10. Huber, M. 2006. Exact Sampling from Perfect Matchings of Dense Regular Bipartite Graphs. *Algorithmica*, 44(3), 183–193.
11. Jerrum, M., Sinclair, A.; and Vigoda, E. 2001. A Polynomial-time Approximation Algorithm for the Permanent of a Matrix with Non-Negative entries. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC 2001)*, ACM Press, 712–721.
12. Linial, N.; Samorodnitsky, A.; and Wigderson, A. 2000. A Deterministic Strongly Polynomial Algorithm for Matrix Scaling and Approximate Permanents, *Combinatorica*, 20(4), 545–568.
13. Kask, K.; Dechter, R.; and Gogate, V. 2004. Counting-Based Look-Ahead Schemes for Constraint Satisfaction. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Springer-Verlag LNCS 3258, 317–331.
14. Patel, J., and Chinneck, J. W. 2006. Active-Constraint Variable Ordering for Faster Feasibility of Mixed Integer Linear Programs. *Mathematical Programming*, 110(3), 445–474.

15. Pesant, G. 2004. A regular language membership constraint for finite sequences of variables. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Springer-Verlag LNCS 3258, 482–495.
16. Pesant, G. 2005. Counting Solutions of CSPs: A Structural Approach. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 260–265.
17. Pesant, G., and Quimper, C.-G. 2008. Counting Solutions of Knapsack Constraints. In *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2008)*, 203–217
18. Rasmussen, L. E. 1994. Approximating the permanent: a simple approach. *Random Structures and Algorithms*, 5(2), 349–361.
19. Refalo, P. 2004. Impact-Based Search Strategies for Constraint Programming. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Springer-Verlag LNCS 3258, 557–571.
20. Régin, J.-C. 1994. A Filtering Algorithm for Constraints of Difference in CSPs. In *Procedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994)*. AAAI Press. 362–367.
21. Ryser, H.J. 1963. Combinatorial Mathematics. Carus Mathematical Monographs No. 14. Wiley.
22. Sellmann, M. and Ansotegui C. 2006. Disco - Novo - GoGo: Integrating Local Search and Complete Search with Restarts. In *Twenty-First National Conference on Artificial Intelligence (AAAI 2006)*. AAAI Press, 1051–1056.
23. Valiant, L. G. 1979. The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8(2), 189–201.
24. Zanarini, A. and Pesant, G. 2007. Solution Counting Algorithms for Constraint-Centered Search Heuristics. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2007)*, Springer-Verlag LNCS 4741, 743–757.