More Robust Counting-Based Search Heuristics with Alldifferent Constraints

Alessandro Zanarini[1,2,3] and Gilles Pesant[1,2]

[1] École Polytechnique de Montréal, Montreal, Canada
[2] CIRRELT, Université de Montréal, Montreal, Canada
[3] Dynadec Europe, Belgium
{Alessandro.Zanarini,Gilles.Pesant}@cirrelt.ca

Abstract. Exploiting solution counting information from individual constraints has led to some of the most efficient search heuristics in constraint programming. However, evaluating the number of solutions for the `alldifferent` constraint still presents a challenge: even though previous approaches based on sampling were extremely effective on hard instances, they are not competitive on easy to medium difficulty instances due to their significant computational overhead. In this paper we explore a new approach based on upper bounds, trading counting accuracy for a significant speedup of the procedure. Experimental results show a marked improvement on easy instances and even some improvement on hard instances. We believe that the proposed method is a crucial step to broaden the applicability of solution counting-based search heuristics.

1 Introduction

AI problem solving relies on effective inference and search. This is true in particular for Constraint Programming where, after many years of advances on inference, there has been a more recent focus on search heuristics. The kind of search heuristics considered in this paper rely on counting the solutions to individual substructures of the problem [13]. Given a constraint γ defined on the set of variables $\{x_1, \ldots, x_k\}$ 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. Given a variable x_i in the scope of γ, and a value $d \in D_i$, we 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*[4] of pair (x_i, d) in γ. It measures how often a certain assignment is part of a solution of the constraint γ. One simple — yet very effective — solution counting-based heuristic is $maxSD$ which, after collecting the solution densities from the problem constraints, branches on the variable-value pair with the highest solution density [13].

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

For some constraints, computing solution densities can be done efficiently and even, in some cases, at (asymptotically) no extra cost given the filtering algorithm already implemented in the constraint. For the `alldifferent` constraint, computing the number of solutions is equivalent to the problem of computing the permanent of the related (0-1) adjacency matrix A that is built such that a_{ij} is equal to 1 iff $j \in D_i$. The permanent of a $n \times n$ matrix A is formally defined as [5]

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

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 assignment for the `alldifferent` constraint. Hence, the sum over all the permutations gives us the total number of `alldifferent` solutions.

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 resembles that of the determinant, there have been few advances on its exact computation. In 1979, Valiant [12] 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. A sampling approach proposed by Rasmussen was improved in [13] by adding propagation. Although providing a very good approximation, it is time consuming and suitable mainly for hard instances where the accuracy of the heuristic can balance the time spent in computing the solution densities.

In this paper we explore a different approach, trading some of the accuracy for a significant speedup in the counting procedure, in order to provide an algorithm that performs well on easy instances while keeping the lead in solving hard ones. A portfolio of heuristics could have been an alternative, first trying a computationally cheap heuristic to take care of easy instances and switching to our counting-based heuristic after a certain time limit. But as we shall see, our proposal not only improves the performance on easy instances but also on hard ones.

In the rest of this paper, Section 2 presents some known upper bounds for the permanent and their integration in solution counting-based heuristics for the `alldifferent` constraint. Section 3 evaluates our proposal on benchmark problems. Final comments are given in Section 4.

2 Bounds for `alldifferent` Solution Counting

2.1 Upper Bounds for the Permanent

In the following, we denote by A the $n \times n$ (0-1) adjacency matrix as defined in the previous section, with r_i the sum of the elements in the i_{th} row (i.e.

[5] We address the fact that the adjacency matrix may not be square in Section 2.1.

$r_i = \sum_{j=1}^{n} a_{ij}$). Note that the permanent is defined on square matrices i.e. the related bipartite graph needs to have $|V_1| = |V_2|$. In order to overcome this limitation, we can augment the graph by adding $|V_2| - |V_1|$ fake vertices to V_1 (without loss of generality $|V_1| \leq |V_2|$) each one connected to all vertices in V_2. The effect on the number of maximum matchings is stated in the following theorem.

Theorem 1. *Let $G(V_1 \cup V_2, E)$ be a bipartite graph with $|V_1| \leq |V_2|$ and the related augmented graph $G'(V_1' \cup V_2, E')$ a graph such that $V_1' = V_1 \cup V_{fake}$ with $|V_{fake}| = |V_2| - |V_1|$ and the edge set $E' = E \cup E_{fake}$ with $E_{fake} = \{(v_i, v_j) \mid v_i \in V_{fake}, \quad v_j \in V_2\}$. Let $|M_G|$ and $|M_{G'}|$ be the number of maximum matchings respectively in G and G'. Then $|M_G| = |M_{G'}|/|V_{fake}|!$.*

Proof. Given a maximum matching $m \in M_G$ of size $|V_1|$, since m covers all the vertices in V_1 then there exists exactly $|V_2| - |V_1|$ vertices in V_2 not matched. In the corresponding matching (possibly not maximum) $m' = m$ in G', the vertices in V_2 that are not matched can be matched with any of the vertices in V_{fake}. Since each vertex in V_{fake} is connected to any vertex in V_2 then there exists exactly $|V_{fake}|!$ permutations to obtain a perfect matching in G' starting from a maximum matching m in G. If there is no maximum matching of size $|V_1|$ for G then clearly there isn't any of size $|V_2|$ for G' either.

For simplicity in the rest of the paper we assume $|X| = |D_X|$.

In 1963, Minc [6] conjectured that the permanent can be bounded from above by the following formula:

$$perm(A) \leq \prod_{i=1}^{n} (r_i!)^{1/r_i}. \tag{2}$$

Proved only in 1973 by Brégman [1], it was considered for decades the best upper bound for the permanent. Recently, Liang and Bai [4], inspired by Rasmussen's work, proposed a new upper bound (with $q_i = min\{\lceil \frac{r_i+1}{2} \rceil, \lceil \frac{i}{2} \rceil\}$):

$$perm(A)^2 \leq \prod_{i=1}^{n} q_i(r_i - q_i + 1). \tag{3}$$

None of the two upper bounds strictly dominates the other. In the following we denote by $UB^{BM}(A)$ the Brégman-Minc upper bound and by $UB^{LB}(A)$ the Liang-Bai upper bound. Jurkat and Ryser proposed in [3] another bound:

$$perm(A) \leq \prod_{i=1}^{n} min(r_i, i).$$

However it is considered generally weaker than $UB^{BM}(A)$ (see [11] for a comprehensive literature review). Soules proposed in [10] some general sharpening techniques that can be employed on any existent permanent upper bound in order to improve them. The basic idea is to apply an appropriate combination of functions (such as row or column permutation, matrix transposition, row or column scaling) and to recompute the upper bound on the modified matrix.

2.2 Solution Counting Algorithm for `alldifferent`

Aiming for very fast computations, we opted for a direct exploitation of UB^{BM} and UB^{LB} in order to compute an approximation of solution densities for the `alldifferent` constraint. An initial upper bound on the number of solutions of the `alldifferent`$(x_1, \ldots, x_n)$ constraint with related adjacency matrix A is simply

$$\#\texttt{alldifferent}(x_1, \ldots, x_n) \leq min\{UB^{BM}(A), UB^{LB}(A)\}$$

Note that in Formula 2 and 3, the r_i are equal to $|D_i|$; since the $|D_i|$ range from 0 to n, the factors can be precomputed and stored: in a vector $BMfactors[r] = (r!)^{1/r}, r = 0, \ldots, n$ for the first bound and similarly for the second one (with factors depending on both $|D_i|$ and i). Assuming that $|D_i|$ is returned in $O(1)$, computing the formulas takes $O(n)$ time.

Recall that matrix element $a_{ij} = 1$ iff $j \in D_i$. Assigning j to variable x_i translates to replacing the i_{th} row by the unit vector $e(j)$ (i.e. setting the i_{th} row of the matrix to 0 except for the element in column j). We write $A_{x_i=j}$ to denote matrix A except that x_i is fixed to j. We call *local probe* the assignment $x_i = j$ performed to compute $A_{x_i=j}$ i.e. a temporary assignment that does not propagate to any other constraint except the one being processed. Solution densities are then approximated as

$$\sigma(x_i, j, \texttt{alldifferent}) \approx \frac{min\{UB^{BM}(A_{x_i=j}), UB^{LB}(A_{x_i=j})\}}{\eta}$$

where η is a normalizing constant.

The local probe $x_i = j$ may trigger some local propagation according to the level of consistency we want to achieve; therefore $A_{x_i=j}$ is subject to the filtering performed on the constraint being processed. Since the two bounds in Formula 2 and 3 depend on $|D_i|$, a stronger form of consistency would likely lead to more changes in the domains and on the bounds, and presumably to more accurate solution densities. We come back to this in Section 2.3.

Once the upper bounds for all variable-value pairs have been computed, it is possible to further refine the solution count as follows:

$$\#\texttt{alldifferent}(x_1, \ldots, x_n) \leq \min_{x_i \in X} \sum_{j \in D_i} min\{UB^{BM}(A_{x_i=j}), UB^{LB}(A_{x_i=j})\}$$

This bound on the solution count depends on the consistency level enforced in the `alldifferent` constraint during the local probes. It is the one we will evaluate in Section 2.3.

If we want to compute $\sigma(x_i, j, \texttt{alldifferent})$ for all $i = 1, \ldots, n$ and for all $j \in D_i$ then a trivial implementation would compute $A_{x_i=j}$ for each variable-value pair; the total time complexity would be $O(mP + mn)$ (where m is the sum of the cardinalities of the variable domains and P the time complexity of the filtering).

Although unable to improve over the worst case complexity, in the following we propose an algorithm that performs definitely better in practice. We first introduce some additional notation: we write as D'_i the variable domains after enforcing θ-consistency[6] on that constraint alone and as $\tilde{I}$ the set of indices of the variables that were subject to a domain change due to a local probe and the ensuing filtering, that is, $i \in \tilde{I}$ iff $|D'_i| \neq |D_i|$. We describe the algorithm for the Brégman-Minc bound — it can be easily adapted for the Liang-Bai bound.

The basic idea is to compute the bound for the matrix A and reuse it to speed up the computation of the bounds for $A_{x_i=j}$ for all $i = 1, \ldots, n$ and $j \in D_i$. Let

$$\gamma_k = \begin{cases} \frac{BMfactors[1]}{BMfactors[|D_k|]} & \text{if } k = i \\ \frac{BMfactors[|D'_k|]}{BMfactors[|D_k|]} & \text{if } k \in \tilde{I} \setminus \{i\} \\ 1 & \text{otherwise} \end{cases}$$

$$\begin{aligned} UB^{BM}(A_{x_i=j}) &= \prod_{k=1}^{n} BMfactors[|D'_k|] = \prod_{k=1}^{n} \gamma_k \; BMfactors[|D_k|] \\ &= UB^{BM}(A) \prod_{k=1}^{n} \gamma_k \end{aligned}$$

Note that γ_k with $k = i$ (i.e. we are computing $UB^{BM}(A_{x_i=j})$) does not depend on j; however $\tilde{I}$ does depend on j because of the domain filtering.

Algorithm 1: Solution Densities

```
1  UB = BMbound(A) ;
2  for i = 1,...,n do
3      varUB = UB * BMfactors[1] / BMfactors[|D_i|] ;
4      total = 0;
5      forall j ∈ D_i do
6          set x_i = j;
7          enforce θ-consistency;
8          VarValUB[i][j] = varUB;
9          forall k ∈ Ĩ \ {i} do
10             VarValUB[i][j] = VarValUB[i][j] * BMfactors[|D'_k|] / BMfactors[|D_k|];
11         total = total + VarValUB[i][j];
12         rollback x_i = j;
13     forall j ∈ D_i do
14         SD[i][j] = VarValUB[i][j]/total;
15 return SD;
```

[6] any form of consistency

Algorithm 1 shows the pseudo code for computing $UB^{BM}(A_{x_i=j})$ for all $i = 1, \ldots, n$ and $j \in D_i$. Initially, it computes the bound for matrix A (line 1); then, for a given i, it computes γ_i and the upper bound is modified accordingly (line 3). Afterwards, for each $j \in D_i$, θ-consistency is enforced (line 7) and it iterates over the set of modified variables (line 9-10) to compute all the γ_k that are different from 1. We store the upper bound for variable i and value j in the structure $VarValUB[i][j]$. Before computing the bound for the other variables-values the assignment $x_i = j$ needs to be undone (line 12). Finally, we normalize the upper bounds in order to correctly return solution densities (line 13-14). The time complexity is $O(mP + m\tilde{I})$.

If the matrix A is dense we expect $|\tilde{I}| \simeq n$, therefore most of the γ_k are different from 1 and need to be computed. As soon as the matrix becomes sparse enough then $|\tilde{I}| \ll n$ and only a small fraction of γ_k needs to be computed, and that is where Algorithm 1 has an edge. In preliminary tests conducted over the benchmark problems presented in the Section 3, Algorithm 1 with arc consistency performed on average 25% better than the trivial implementation.

2.3 Counting Accuracy Analysis

But how accurate is the counting information we compute from these bounds? We compared the algorithm based on upper bounds with the previous approaches: Rasmussen's algorithm, Furer's algorithm and the sampling algorithm proposed in [13]. We generated `alldifferent` instances of size n ranging from 10 to 20 variables; variable domains were partially shrunk with a percentage of removal of values p varying from 20% to 80% in steps of 10%. We computed the exact number of solutions and removed those instances that were infeasible or for which enumeration took more than 2 days (leaving about one thousand instances). As a reference, the average solution count for the `alldifferent` instances with 20% to 60% of values removed is close to one billion solutions (and up to 10 billions), with 70% of removals it decreases to a few millions and with 80% of removals to a few thousands.

Randomized algorithms were run 10 times and we report the average of the results. In order to verify the performance with varying sampling time, we set a timeout of respectively 1, 0.1, 0.01, and 0.001 second. The running time of the counting algorithm based on upper bounds is bounded by the completion of Algorithm 1. The measures used for the analysis are the following:

counting error: relative error on the solution count of the constraint (computed as the absolute difference between the exact solution count and the estimated one and then divided by the exact solution count)

maximum solution density error: maximum absolute error on the solution densities (computed as the maximum of the absolute differences between the exact solution densities and the approximated ones)

average solution density error: average absolute error on the solution densities (computed as the average of the absolute differences between the exact solution densities and the approximated ones)

Note that we computed absolute errors for the solution densities because counting-based heuristics usually compare the absolute value of the solution densities.

Plot 1 shows the counting error for the sampling algorithm, Rasmussen's and Furer's with varying timeout. Different shades of gray indicate different percentages of removals; series represent different algorithms and they are grouped based on the varying timeouts.

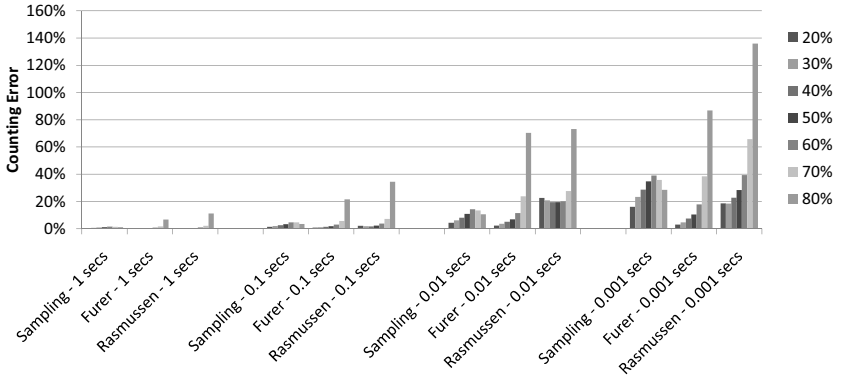

Fig. 1. Counting Error for one thousand alldifferent instances with varying variable domain sizes.

The relative counting error is maintained reasonably low for 1 and 0.1 second of sampling, however it increases considerably if we further decrease the timeout. Note that at 0.001 the sampling algorithm reaches its limit being able to sample only a few dozens solutions (both Rasmussen's and Furer's are in the order of the hundreds of samples). We left out the results of the algorithm based on upper bounds to avoid a scaling problem: the counting error varies from about 40% up to 2300% when enforcing domain consistency in Algorithm 1 (UB-DC) and up to 3600% with arc consistency (UB-AC) or 4800% with forward checking (UB-FC). Despite being tight upper bounds, they are obviously not suitable to approximate the solution count. Note nonetheless their remarkable running times: UB-DC takes about one millisecond whereas UB-AC and UB-FC about a tenth of a millisecond (with UB-FC being slightly faster).

Despite the poor performance in approximating the solution count, they provide a very good tradeoff in approximation accuracy and computation time when deriving solution densities.

Figure 2 and 3 show respectively the maximum and average solution density errors (note that the maximum value in the y-axis is different in the two plots). Again the sampling algorithm shows a better accuracy w.r.t. Rasmussen's and Furer's. Solution density errors are very well contained when using the upper bound approach: they are the best one when compared to the algorithms with an equivalent timeout and on average comparable to the results obtained by the

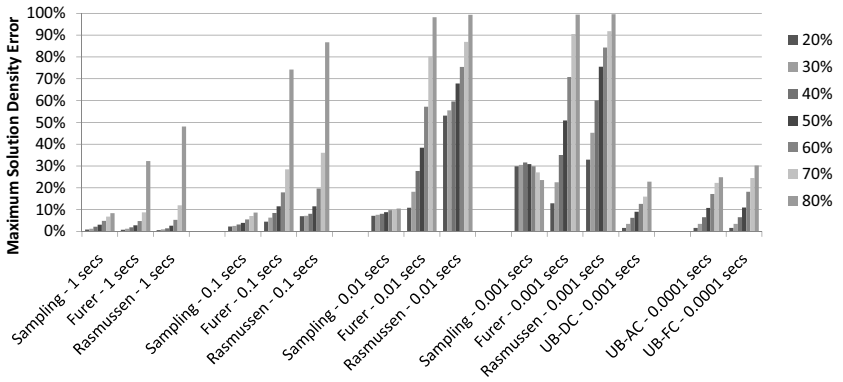

Fig. 2. Maximum Solution Density Error for one thousand alldifferent instances with varying variable domain sizes.

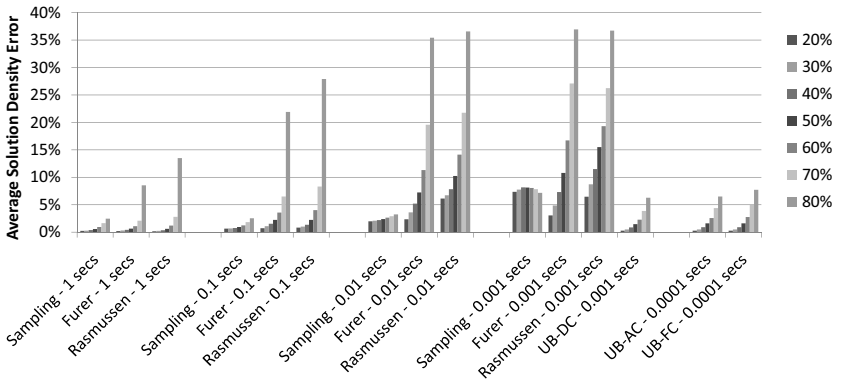

Fig. 3. Average Solution Density Error for one thousand alldifferent instances with varying variable domain sizes.

sampling algorithm with a timeout of 0.01 seconds. Therefore, upper bounds offer a good accuracy despite employing just a tenth (UB-DC) or a hundredth (UB-AC, UB-FC) of the time of the sampling algorithm with comparable accuracy. Furthermore, errors for the upper bound algorithm are quite low when the domains are dense (low removal percentage) and on par with the sampling algorithm with a timeout of 0.1 or even 1 second. Note that in the context of search heuristics dense domains are more likely to happen closer to the root of the search tree hence when it is important to have a good heuristic guidance. Finally, as expected, enforcing a higher level of consistency during the local probes brings more accuracy, however the difference between UB-DC, UB-AC and UB-FC is not striking.

3 Experimental Results

In addition to counting accuracy, we measured the performance of search heuristics using such information to solve combinatorial problems by running experiments on two benchmark problems. We compared the maxSD heuristic [13] to Impact Based Search (IBS) and to the dom/ddeg variable selection heuristic coupled with the minconflicts or the lexicographic value selection heuristic. IBS selects the variable whose instantiation triggers the largest search space reduction (highest impact) that is approximated as the reduction of the Cartesian product of the variables' domains (see [8]). For the maxSD heuristic, we used the counting algorithm proposed in [13] and alternatively the approach based on UB^{BM} and UB^{LB}; for Algorithm 1 we tested three consistency levels (forward checking – UB-FC, arc consistency – UB-AC, and domain consistency – UB-DC). In order to get the best out of the heuristic presented in [13] throughout the instance sets, we decided to modify slightly the counting algorithm. The original counting algorithm first tries with an exact enumeration of the solutions for 0.2 second and in case of timeout proceeds with the sampling for the same amount of time. Here, we decided to do the same (exact+sampl) except that the sampling phase does not have a timeout but is instead bounded by the sample size, that is set dynamically. We got good results by setting the number of samples for each constraint to ten times the maximum domain size of the variables in the scope of the constraint. We also report on sampling used alone (sampl) and on maintaining either arc or domain consistency during sampling. Again for heuristics that have some sort of randomization, we took the average over 10 runs. We used Ilog Solver 6.6 on a AMD Opteron 2.4 GHz with 1GB of RAM. For each instance the timeout was set to 20 minutes.

3.1 Quasigroup with Holes Problem

We first tested our algorithm on the Quasigroup with Holes Problem (QWH). It 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 most common model uses a matrix of integer variables and an `alldifferent` constraint for each row and each column. We tested on the 40 hard instances used in [13] that have $n = 30$ and 42% of holes and we generated 60 additional instances outside the phase transition respectively with 45%, 47% and 50% of holes, using [2].

Results are shown in Table 1 (timeout instances are included in the averages). Figures 4 and 5 show the percentage of solved instances within a given time for the instance sets with respectively 42% and 45% of holes (time is not cumulative). Every heuristic solved at least 95% of instances in each set except for the instances with 42% of holes, where dom/ddeg solved 73% of them, IBS solved 85%, maxSD sampl-DC solved 80%, and maxSD sampl-AC solved 85% (see Figure 4). Note that all the heuristics based on maxSD solved every instance with 45% of holes or more. maxSD sampl brings an impressive reduction of the number of backtracks compared to the first two heuristics but without a significant computational advantage. In order to speed it up, we can add exact counting to

heuristic	time	median	bckts	time	median	bckts
	42% holes			45% holes		
dom/ddeg; minconflicts	497.0	243.8	752883	6.6	1.0	11035
IBS	344.9	72.3	914849	94.1	19.9	247556
maxSD sampl-DC	398.8	358.3	15497	20.0	16.2	619
maxSD sampl-AC	339.7	285.4	15139	29.0	14.3	1349
maxSD exact+sampl-DC	132.0	72.2	4289	115.2	110.0	517
maxSD exact+sampl-AC	142.9	67.9	5013	125.7	110.8	1092
maxSD UB-DC	110.5	13.6	31999	1.3	1.0	164
maxSD UB-AC	82.4	3.7	68597	0.7	0.2	582
maxSD UB-FC	105.5	7.8	104496	0.5	0.3	447
heuristic	time	median	bckts	time	median	bckts
	47% holes			50% holes		
dom/ddeg; minconflicts	60.3	0.1	118089	0.1	0.1	36
IBS	5.1	2.2	16126	2.8	2.2	10012
maxSD sampl-DC	22.8	10.0	657	19.4	13.3	355
maxSD sampl-AC	6.3	6.1	34	7.7	7.7	8
maxSD exact+sampl-DC	187.3	187.8	8	269.0	270.0	29
maxSD exact+sampl-AC	191.0	187.0	450	262.0	263.5	2
maxSD UB-DC	1.5	1.5	20	2.4	2.3	3
maxSD UB-AC	0.3	0.3	30	0.3	0.3	2
maxSD UB-FC	0.3	0.3	56	0.3	0.3	6

Table 1. Average solving time (in seconds), median solving time and average number of backtracks for 100 QWH instances of order 30.

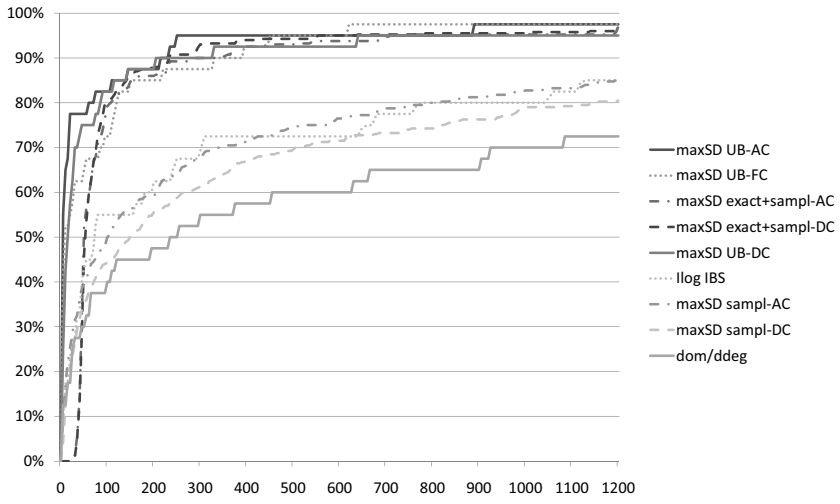

Fig. 4. Percentage of solved instances vs time for QWH instances with 42% of holes

produce more accurate information. maxSD exact+sampl is the heuristic with the lowest number of backtracks on the hard instances together with a significantly lower runtime; however, as expected, it runs longer on the easy instances: this

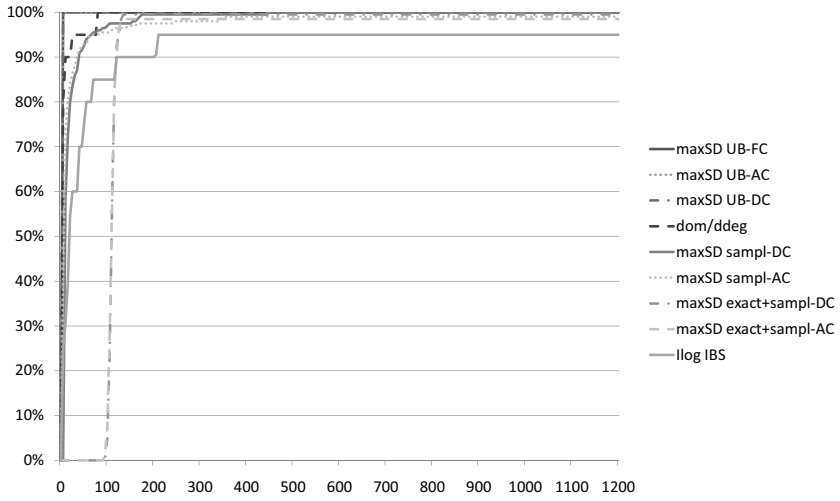

Fig. 5. Percentage of solved instances vs time for QWH instances with 45% of holes

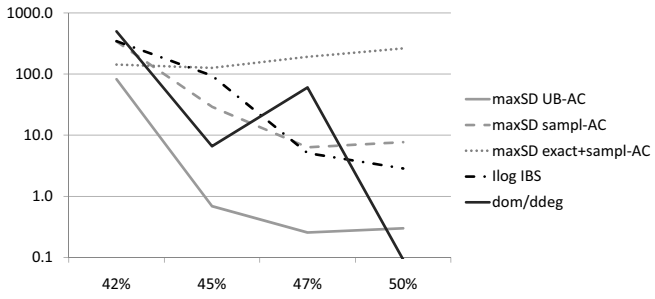

Fig. 6. Total average time vs % holes

can be explained by the fact that the easy instances have more loose constraints therefore the initial exact enumeration is more likely to time out. In Figure 4 we can see that the sampling algorithm alone is able to solve some instances within few seconds whereas maxSD exact+sampl-DC does not solve any instance within 40 seconds because of the high overhead due to exact enumeration. Sampling alone struggles more with the hard instances and it ends up solving just 85% of the instances whereas maxSD exact+sampl-DC solves 97% of the instances.

The previous heuristics were significantly outperformed in all the instance sets by the heuristics based on upper bounds. As shown in Figure 6, maxSD UB-DC, maxSD UB-AC, maxSD UB-FC are very quick in solving easy instances and yet they are capable of solving the same number of instances as maxSD exact+sampl-DC. The latter heuristic shows its limit already in the set of instances with 45% of holes where no instance is solved within a hundred seconds, whilst maxSD based on upper bounds almost instantaneously solves all the instances. maxSD UB-AC was overall the best of the set on all the instances with

up to a two orders of magnitude advantage over IBS in terms of solving time and up to four orders of magnitude for the number of backtracks. Enforcing a higher level of consistency leads to better approximated solution densities and to a lower number of backtracks, but it is more time consuming than simple arc consistency. A weaker level of consistency like forward checking can pay off on easy instances but it falls short compared to UB-AC on the hard ones. Note also that maxSD UB-DC increases the solving time, despite lowering the backtracks, when the instances have more holes (apart from the 42% holes instances): in those cases m increases and the overhead of propagation becomes important (see Section 2.2). However we could not reuse the maximum matching and the strongly connected components (see [9]) computed for the propagation (there is no access to the underlying propagation code) — a more coupled integration of the counting algorithm with the propagation algorithm could lead to a performance gain. We did not consider attempting exact counting before computing the upper bound (exact+UB) because this would have caused timeouts on the easier instances, as observed for exact+sampl, and unnecessarily increased the runtimes on such instances.

We end this section by mentioning a few unsuccessful trials. We tried computing the permanent bound of Algorithm 1 (line 1) incrementally during search. We also tried to apply one technique inspired by [10]. Soules observed that $per(A) = per(A^T)$ however $UB^{BM}(A)$ is not necessarily equal to $UB^{BM}(A^T)$ (the same for the Liang-Bai bound). We implemented the code to compute the bounds on both the matrix and its transposed and we kept the minimum of such upper bounds. Neither of these two attempts led to any significant improvement.

Adding Randomized Restarts The QWH problem exhibits heavy-tail behavior in runtime distributions when the instances are generated close to the phase transition. Nonetheless, heavy-tails can be largely avoided by adding randomized restarts to the search procedure. We tried a subset of the heuristics tested above with randomized restart techniques. All the heuristics have been randomized such that one variable-value pair is chosen at random with equal probability between the best two provided by the heuristic. We implemented Walsh's universal strategy to generate the restart cutoff sequence (that is $\alpha r^0, \alpha r^1, \alpha r^2, \ldots$ with $r = 2$ and α equal to 5% the number of variables). The heuristics were not able to gain from the randomized restarts on the easier instances but only on the ones with 42% of holes. maxSD UB-FC improved by 35% the average running time whereas IBS and dom/ddeg; minconflicts degraded their performance by respectively about 45% and 32%.

3.2 Travelling Tournament Problem with Predefined Venues

The Travelling Tournament Problem with Predefined Venues (TTPPV) was introduced in [5] and consists of finding an optimal single round robin schedule for a sport event. Given a set of n teams, each team has to play against each other team. In each game, a team is supposed to play either at home or away, however no team can play more than three consecutive times at home or away.

The particularity of this problem resides on the venues of each game, that are predefined, i.e. if team a plays against b we already know whether the game is going to be held at a's home or at b's home. A TTPPV instance is said to be balanced if the number of home and away games differ by at most one for each team; otherwise it is referred to as unbalanced or random.

The TTPPV was originally introduced as an optimization problem where the sum of the travelling distance of each team has to be minimized, however [5] shows that it is particularly difficult to find even a feasible solution using traditional integer linear programming methods (ILP). Balanced instances of size 18 and 20 (the size is the number of teams) were taking from roughly 20 to 60 seconds to find a first feasible solution with ILP; unbalanced instances could take up to 5 minutes (or even time out after 2 hours of computation). Hence, the feasibility version of this problem already represents a challenge. Therefore we attempted to tackle it with Constraint Programming and solution counting heuristics.

We modelled the problem in the following way:

$$\texttt{alldifferent}((x_{ij})_{1\leq j\leq n-1}) \qquad 1 \leq i \leq n \tag{4}$$

$$\texttt{regular}((x_{ij})_{1\leq j\leq n-1}, PV_i) \qquad 1 \leq i \leq n \tag{5}$$

$$\texttt{alldifferent}((x_{ij})_{1\leq i\leq n}) \qquad 1 \leq j \leq n-1 \tag{6}$$

$$x_{ij} = k \iff x_{kj} = i \qquad 1 \leq i \leq n, 1 \leq j \leq n-1 \tag{7}$$

$$x_{ij} \in \{1, \ldots, n\} \qquad 1 \leq i \leq n, 1 \leq j \leq n-1 \tag{8}$$

A variable $x_{ij} = k$ means that team i plays against team k at round j. Constraint (7) enforces that if team a plays against b then b plays against a in the same round; constraint (4) enforces that each team plays against every other team; the home-away pattern associated to the predefined venues of team i (PV_i) is defined through a regular constraint (5). Finally constraint (6) is redundant and used to achieve additional filtering.

We tested 40 balanced and 40 unbalanced instances borrowed from [5] with sizes ranging from 14 to 20. For the `regular` constraint we used the counting algorithm proposed in [13]. Results are reported in Table 2 for balanced and unbalanced instances (timeout instances are included in the averages). Figure 7 shows the percentage of unbalanced instances solved within a given time limit (time is not cumulative).

Balanced instances do not present a challenge for any of the heuristics tested: the lightweight heuristic dom/ddeg; lexico is the one performing better together with maxSD based on upper bounds. The sampling algorithm here shows its main drawbacks i.e. it is not competitive in solving easy instances: the number of backtracks is low indeed but the time spent in sampling is simply a waste of time on easy instances though crucial on difficult ones. Exact enumeration adds another constant overhead to the counting procedure with the results of being three orders of magnitude slower than upper bounds based on arc consistency or forward checking.

	balanced			unbalanced		
heuristic	time	bckts	%solved	time	bckts	% solved
dom/ddeg; lexico	0.1	27	100%	901.2	2829721	25%
IBS	10.7	8250	100%	631.7	1081565	50%
maxSD sampl-DC	25.9	2	100%	140.7	3577	91%
maxSD exact+sampl-DC	120.4	1	100%	216.9	1210	91%
maxSD UB-DC	6.7	1	100%	36.8	245	98%
maxSD UB-AC	0.6	1	100%	30.6	2733	98%
maxSD UB-FC	0.5	1	100%	30.5	2906	98%

Table 2. Average solving time (in seconds), number of backtracks, and percentage of instances solved for 80 TTPPV instances.

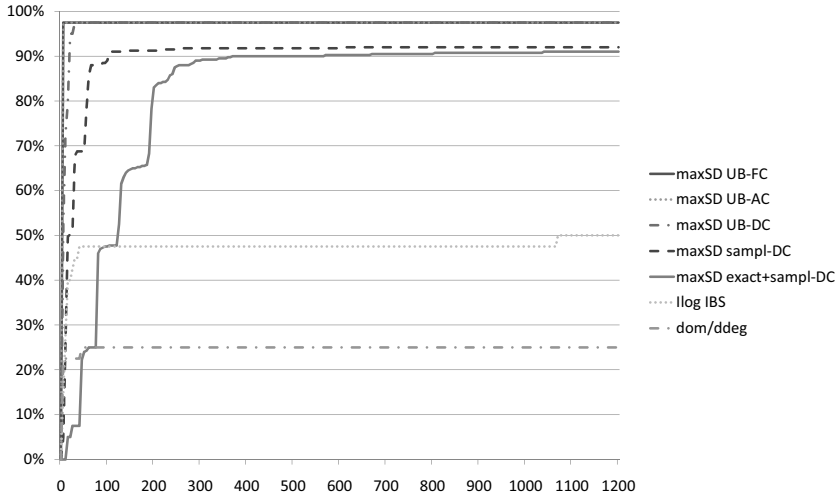

Fig. 7. Percentage of solved instances vs time for non balanced instances of the TTPPV

Unbalanced instances are harder to solve and none of the heuristics were able to solve all 40 instances within the time limit — note that in this set of instances six out of forty are infeasible. maxSD is significantly faster than any other heuristic: counting based on upper bounds also allowed to cut computing time by almost 80% w.r.t. the sampling algorithm and by 85% w.r.t. exact enumeration and sampling. 90% of the instances are solved in 100 seconds by maxSD sampl-DC whereas maxSD UB-AC and maxSD UB-FC take less than 2 seconds to solve 97.5% of the instances (maxSD UB-FC takes slightly more).

Remarkably, maxSD with upper bounds proved the infeasibility of five of the six instances, and with small search trees. None of the other heuristics tested were able to prove the infeasibility of any of the six instances. Gains are remarkable also in the number of backtracks (three orders of magnitude better than the other heuristics). maxSD with upper bound-based counting turned out to be the most consistent heuristic, performing very well both on hard and easy instances with an average solving time up to 20 times better than IBS.

4 Conclusion and Future Work

Solution counting heuristics are to date among the best generic heuristics to solve CSPs. We believe that the basic idea and some of the algorithms are relevant to general AI, not just within the CSP framework, and to OR as well. As an indication, this line of work recently inspired new branching direction heuristics for mixed integer programs [7].

In this paper, we propose an algorithm to compute solution densities for `alldifferent` constraints that actually broaden the applicability of such heuristics. The new approach is suitable both for easy and hard problems and it proves the competitiveness of solution counting based heuristics w.r.t. other state-of-the-art heuristics. In the future, we would like to try the more sophisticated upper bounds proposed in [10] and [11] to see whether they can bring an actual benefit to our heuristics. An interesting combination is to use upper bounds to identify a small subset of promising variable-value pairs and then apply an algorithm with a better approximation accuracy (such as sampling) on the selected subset.

References

1. L.M. Bregman. Some Properties of Nonnegative Matrices and their Permanents. *Soviet Mathematics Doklady*, 14(4):945 – 949, 1973.
2. Carla Gomes and David Shmoys. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *COLOR 2002: Proceedings of Computational Symposium on Graph Coloring and Generalizations*, pages 22–39, 2002.
3. W.B. Jurkat and Herbert John Ryser. Matrix Factorizations of Determinants and Permanents. *Journal of Algebra*, 3:1 – 27, 1966.
4. Heng Liang and Fengshan Bai. An Upper Bound for the Permanent of (0,1)-Matrices. *Linear Algebra and its Applications*, 377:291 – 295, 2004.
5. R.A. Melo, S. Urrutia, and C.C. Ribeiro. The traveling tournament problem with predefined venues. *Journal of Scheduling*, 12(6):607–622, 2009.
6. Henryk Minc. Upper Bounds for Permanents of (0, 1)-matrices. *Bulletin of the American Mathematical Society*, 69:789 – 791, 1963.
7. J. Pryor. Branching Variable Direction Selection in Mixed Integer Programming. Master's thesis, Carleton University, 2009.
8. Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.
9. Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *AAAI '94: Proceedings of the Twelfth National Conference on Artificial Intelligence (vol. 1)*, pages 362–367. American Association for Artificial Intelligence, 1994.
10. George W. Soules. New Permanental Upper Bounds for Nonnegative Matrices. *Linear and Multilinear Algebra*, 51(4):319 – 337, 2003.
11. George W. Soules. Permanental Bounds for Nonnegative Matrices via Decomposition. *Linear Algebra and its Applications*, 394:73 – 89, 2005.
12. Leslie Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.
13. Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. *Constraints*, 14(3):392–413, 2009.