Solution Sampling with Random Table Constraints

Mathieu Vavrille
Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes, France

Charlotte Truchet
Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes, France

Charles Prud'homme
TASC, IMT-Atlantique, LS2N-CNRS, F-44307 Nantes, France

Abstract

Constraint programming provides generic techniques to efficiently solve combinatorial problems. In this paper, we tackle the natural question of using constraint solvers to sample combinatorial problems in a generic way. We propose an algorithm, inspired from Meel's ApproxMC algorithm on SAT, to add hashing constraints to a CP model in order to split the search space into small cells of solutions. By sampling the solutions in the restricted search space, we can randomly generate solutions without revamping the model of the problem. We ensure the randomness by introducing a new family of hashing constraints: randomly generated tables. We implemented this solving method using the constraint solver Choco-solver. The quality of the randomness and the running time of our approach are experimentally compared to a random branching strategy. We show that our approach improves the randomness while being in the same order of magnitude in terms of running time.

2012 ACM Subject Classification Computing methodologies → Randomized search

Keywords and phrases solutions, sampling, table constraint

Digital Object Identifier 10.4230/LIPIcs.CP.2021.56

Supplementary Material *Software (Source Code)*:
`https://github.com/MathieuVavrille/tableSampling`
archived at `swh:1:dir:63a03fba176c348c1f9d698bda1b484957b6b5ce`

1 Introduction

Using constraint satisfaction as a core technique, constraint solvers have been enriched with different additional properties, such as optimisation (even with multiple objectives [8]), user preferences [18], diverse solutions [10], robust solutions [9], etc. In this article, we propose a method to sample solutions of a constraint problem, without modifying its model. This work is motivated by many situations where a user wants randomized solutions: to ease user feedback and decision making, to ensure equity (for instance in planning problems), to guarantee solution coverage (for instance in test generation problems).

Currently, a straightforward way to randomly sample solutions with a CP solver is to use RandomVarDom; that is, randomly picking a variable and a value as an enumeration strategy. However this strategy does not return uniformly drawn solutions (uniformly within the solution set), and also replaces the strategy that may have been chosen or built for the problem. Our approach is inspired from UniGen [13], a near-uniform sampling algorithm for SAT, adapted to the CP framework. The idea is to divide the search space by adding random hashing constraints, until only a small, tractable number of solutions remain. No replacement of the strategy is needed and the sampling can be done among these solutions. Our algorithm also features a dichotomic variation which accelerates the whole process.

This algorithm needs to be fed with random hashing constraints. In order to maintain the running time reasonable, we choose to randomly generate table constraints [5], which are implemented in all constraint solvers. We rely on their extensional representation of valid tuples to produce, at cheap cost, a multivariate uniform distribution.

© Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme;
licensed under Creative Commons License CC-BY 4.0

We implemented our proposal on top of Choco-solver [16] and compare it to RandomVarDom on various types of problems. We show that our approach improves, in practice, the randomness compared to RandomVarDom. In addition, the running time increase is limited in practice. This shows that adding a better level of randomization can be done at a low computation cost.

1.1 Related works

As we previously said, our work is inspired from Meel's work on UniGen [13], an algorithm to sample SAT problems. UniGen is a two-step algorithm. The first step consists in running ApproxMC, an algorithm for SAT model counting. XOR constraints are added to the model until there are less than a given number of solutions. The solutions are counted within the smaller space marked by the additional constraints, and this number, multiplied by a ratio between the volume of the smaller space and the volume of the real search space, gives a first value for the solution number. Applying this algorithm multiple times gives an approximation of the number of solutions. In the second step, this approximation is used in UniGen to sample the problem. The resulting distribution is near the uniform distribution. The idea used in ApproxMC on SAT (divide to count) has also recently been used in a CP context for model counting [15]. Our algorithm uses the same idea of additional constraints to divide the search space, within the CP framework, for solution sampling.

Among the broad literature of SAT solution sampling, we also want to mention [6], which is close to our approach. The authors sample partial solutions, and iteratively extend them by instantiating a fixed number of variables. This approach works well because of the binary domains, but in CP the possible large domains would be an issue. This would force to do more iterations, which in the worst case would lead to an algorithm close to RandomVarDom

Solution sampling in constraint programming was first studied in [4] and [7], using bayesian networks. These approaches allow to have a uniform sampling, or to choose the distribution of the solutions, but are exponential in the induced width of the constraint graph. This complexity prevents the approach from being used on big instances, and forces the use of approximations. We took the opposite view of designing a fast sampling method, knowing that we would not be able to guarantee the uniformity of the sampling.

Other approaches improve the diversity of solutions, a different task from sampling. It consists in finding solutions far from each other, for a given metric (edit distance for instance). In [10] the model is re-written to search for distant solutions. In [20] solutions are returned in an online fashion; search strategies are designed to search in spaces far from the solutions previously found. Diversification and sampling are linked but remain two very different goals. On one hand, sampling multiple solutions will necessarily return diverse solutions, but it is very unlikely to be the most distant solutions. On the other hand, diversification does not give any guarantee of randomness, as solutions close to other ones may never be returned.

1.2 Outline

Section 2 gives the notations and recalls the definitions that are needed afterwards. Section 3 presents our approach to sample solutions, and section 4 describes our experimental evaluation.

2 Preliminaries

2.1 Constraint programming

In this article we are interested in constraint satisfaction problems (CSPs). A CSP $\mathcal{P}$ is a triple $\langle \mathcal{X}, \mathcal{C}, \mathcal{D} \rangle$ where

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables;
- $\mathcal{D}$ is a function associating a domain to every variable;
- $\mathcal{C}$ is a set of constraints, each constraint $C \in \mathcal{C}$ consists of:
  - a tuple of variables called *scope* of the constraint $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, where r is the arity of the constraint
  - a relation, i.e. a set of instantiations

$$rel(C) \subseteq \prod_{k=1}^{r} \mathcal{D}(X_{i_k})$$

A constraint is said to be satisfied if every variable $X_{i_k} \in scp(C)$ is instantiated to a value of its domain $x_{i_k} \in \mathcal{D}(X_{i_k})$, and $(x_{i_1}, \ldots, x_{i_r}) \in rel(C)$. The constraints can be defined in extension (called table constraints [5]) by giving explicitly $rel(C)$, or in intension with an expression in a higher level language. For example, the expression $X_1 + X_2 \leq 1$ for X_1, X_2 on domains $\{0, 1\}$, represents $rel(C) = \{(0, 0), (0, 1), (1, 0)\}$.

CSP solving is the search for one, some or all solutions, i.e. assignments of value to every variable such that all the constraints are satisfied. Optimisation problems (COPs) are CSPs where an objective function obj to minimise (or maximise) has been added.

Notations

Let a problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, and C a constraint, we write $\mathcal{P} \wedge C$ for the CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{C\} \rangle$. We note $Sols(\mathcal{P})$ the set of solutions of problem $\mathcal{P}$.

In the following, we only consider satisfaction problems. It is also possible to deal with optimisation problems, up to an approximation, by turning them into a satisfaction problem. Let a COP $(\mathcal{P}, obj)$ to minimise (resp. maximise), and let opt be the minimum value (resp. maximum) of the objective obj. Let $\epsilon \geq 0$, we transform the problem into a CSP $\mathcal{P} \wedge (obj \leq opt + \epsilon)$ (resp. $\mathcal{P} \wedge (obj \geq opt - \epsilon)$). As the gap ϵ increases, the solutions searched will be further from the optimal value.

2.2 Chi squared test

Evaluating the randomness of a system is a hard task because random systems can take surprising values without being biased (for example, a fair coin does, occasionally, land ten times in a row on heads). The chi squared (or χ^2) test allows to compare the result of a random experiment to an expected probability distribution. It comes from a convergence result to the χ^2 law, stated in [14] and recalled here. Let Y be a random variable on a finite set, taking the value k with probability p_k for $1 \leq k \leq d$. Let $Y_1, \ldots, Y_n$ be independent random variables of the same law as Y. Let $N_n^{(k)}$ the number of variables $Y_i, 1 \leq i \leq n$ equal to k.

▶ **Theorem 1** ([14]). *When n tends to infinity, the cumulative distribution function of the random variable*

$$Z_n = \sum_{k=1}^{d} \frac{\left(N_n^{(k)} - n \cdot p_k\right)^2}{n \cdot p_k}$$

tends to the cumulative distribution function of the law of the χ^2 with $(d-1)$ degrees of freedom (noted χ^2_{d-1}).

The χ^2 test comes down to randomly picking values by making the assumption that they follow the law of Y, compute the experimental value z_n^{exp} of Z_n, and compute the probability (called p-value)

$$\mathbb{P}(Z_n \geq z_n^{exp}) \approx \mathbb{P}(\chi^2_{d-1} \geq z_n^{exp})$$

If this probability is close to zero, then, having a more extreme result than the one obtained is very unlikely. It means that the hypothesis under which the experimental values follow the same law as Y can be confidently rejected.

2.3 Random search strategy

The search algorithm for a constraint problem alternates:

- a depth-first search, where the search space is reduced by adding constraints (called decisions), for example, an assignment of a variable to a value in its domain;
- a phase of propagation that checks the satisfiability of the constraints of the CSP.

A natural search strategy to add randomness is the strategy RandomVarDom that picks randomly (uniformly) a variable X among all the non instantiated variables, a value $x \in \mathcal{D}(X)$, and applies the decision $X = x$. This strategy has the advantage to be easily implemented in any constraint solver. However it prevents from using an other more efficient exploration strategy, and besides, the distribution of the solutions may be far from uniform. For example on the problem $\mathcal{P} = \langle \{X_1, X_2\}, \{X_1 \mapsto \{0,1\}, X_2 \mapsto \{0,1\}\}, \{X_1 + X_2 > 0\} \rangle$, let s the solution returned by a solver configured to use RandomVarDom, then we have,

$$\mathbb{P}(s = \{X_1 \mapsto 0, X_2 \mapsto 1\}) = \frac{3}{8}$$
$$\mathbb{P}(s = \{X_1 \mapsto 1, X_2 \mapsto 0\}) = \frac{3}{8}$$
$$\mathbb{P}(s = \{X_1 \mapsto 1, X_2 \mapsto 1\}) = \frac{1}{4}$$

The implications on the running time to find solutions and the quality of the randomness on different problems are discussed in section 4.

3 New sampling approach

We present here a new approach to sample solutions. This approach is twofold: first we present a way to generate random tables, and we then present an algorithm to sample solutions using these generated constraints.

Algorithm 1 Random table constraint generation algorithm.

```
1 Function RandomTable(P, v, p)
    Data: A CSP P = ⟨{X_1, ..., X_n}, D, C⟩, v > 0, 0 < p < 1
    Result: A random table constraint
2   T ← {};
3   i_1, ..., i_v ← GetIndices(P, v);
4   foreach (x_{i_1}, ..., x_{i_v}) ∈ ∏_{k=1}^{v} D(X_{i_k}) do
5       if Random() < p then
6           T.add((x_{i_1}, ..., x_{i_v}));
7   return Table((X_{i_1}, ..., X_{i_v}), T);
```

3.1 Random table constraints

The algorithm to generate random table constraints is presented in Algorithm 1. We suppose available the functions Random() that returns a random floating point number between 0 and 1, GetIndices($\mathcal{P}, v$) that returns v indices $i_1, \dots i_v$ such that $|\mathcal{D}(X_{i_k})| \neq 1, 1 \leq k \leq v$ (if there are less than v such indices, they are all returned), and Table($\mathcal{X}', T$) that creates a table constraint C such that $scp(C) = \mathcal{X}'$ and $rel(C) = T$. The two parameters of the algorithm are: v the number of variables in the table, and p the probability to add a tuple in the table. The algorithm first randomly chooses v variables among the variables whose domains are not reduced to a singleton, and then runs through all the instantiations of these v variables and adds each instantiation in the table with probability p.

The goal of these tables is to restrict the solution space to a smaller sub-space. The following theorem shows that in average, the number of solutions of the problem is reduced by a factor p.

▶ **Theorem 2.** *Let $\mathcal{P}$ be a CSP, and T a table constraint randomly generated with probability p. Then*

$$\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) = p|Sols(\mathcal{P})|$$

Proof. For $\sigma \in Sols(\mathcal{P})$, let γ_σ a random variable equal to 1 if and only if $\sigma \in Sols(\mathcal{P} \wedge T)$. $\mathbb{P}(\gamma_\sigma)$ is the probability that σ satisfies T. Let $X_{i_1}, \dots, X_{i_r}$ the variables chosen in T. Each instantiation of these variables has been added in the table with probability p, including the instantiation $(\sigma(X_{i_1}), \dots, \sigma(X_{i_v}))$. It means that σ satisfies the table constraint T with probability p. We then have $p = \mathbb{P}(\gamma_\sigma = 1) = \mathbb{E}(\gamma_\sigma)$. It follows:

$$\begin{aligned}\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) &= \mathbb{E}\left(\sum_{\sigma \in Sols(\mathcal{P})} \gamma_\sigma\right) \\ &= \sum_{\sigma \in Sols(\mathcal{P})} \mathbb{E}(\gamma_\sigma) \\ &= \sum_{\sigma \in Sols(\mathcal{P})} p \\ &= p|Sols(\mathcal{P})|\end{aligned}$$ ◀

The purpose of Theorem 2 is the following: by adding table constraints, we decrease the size of the solution set, and we can control how much, in average.

Algorithm 2 Sampling algorithm by adding table constraints.

```
1  Function TableSampling(P, κ, v, p)
       Data: A CSP P, κ ≥ 2, v > 0, 0 < p < 1
       Result: A solution to the problem P
2      S ← FindSolutions(P, κ);
3      if |S| = 0 then
4          return "No solution";
5      while |S| = 0 ∨ |S| = κ do
6          T ← RandomTable(P, v, p);
7          S ← FindSolutions(P ∧ T, κ);
8          if |S| ≠ 0 then
9              P ← P ∧ T;
10     return RandomElement(S);
```

3.2 Sampling algorithm

First, the auxiliary functions used in the sampling algorithm are presented. The first one is RandomElement(S) that returns a random element taken uniformly in S. The second function is FindSolutions($\mathcal{P}, s$) that enumerates the solutions of $\mathcal{P}$ until s solutions have been found, and returns them. Notice that, if this function returns s solutions, then $|Sols(\mathcal{P})| \geq s$, and if it returns less than s solutions then all the solutions have been found. The depth first search in constraint solvers makes the implementation of such a function easy.

The sampling algorithm works in the following manner: table constraints are added to the problem to reduce the number of solutions. When there are less solutions than a given pivot value, a solution is randomly returned among the remaining solutions. The algorithm is presented in details in Algorithm 2. A value κ for the pivot is chosen to bound the number of solutions enumerated in the intermediate problems, as well as the number of variables per table v and the probability p to add a tuple in the table.

The algorithm first enumerates κ solutions and immediately stops if there are no solutions, or less than κ solutions. If the problem has more than κ solutions a new table constraint is randomly generated. If the problem with this constraint still has solutions, the constraint is definitively added to the problem. The algorithm stops when there are less than κ solutions. Finally, a solution is randomly chosen from all the solutions remaining, and returned.

3.2.1 Proof of termination

When creating random algorithms, one has to be particularly careful about the termination. We show here that Algorithm 2 terminates with probability 1. A discussion about the experimental behaviour is done in section 4.3.

We fix values for $\kappa \geq 2, v > 0$ and $0 < p < 1$. The case of the initial problem not being satisfiable is caught at the beginning of the algorithm (line 3).

The following lemmas shows that there always exists a table that reduces the number of solutions of the problem without making it inconsistent, and this table is chosen with a non-zero probability. Without loss of generality, we suppose that there are always v variables in the tables. If less than v variables are not instantiated, we pick some of the already instantiated variables and use their current values to complete the instantiations.

▶ **Lemma 3.** *Let $\mathcal{P}$ be a problem with at least two solutions. In our framework, there exists a random table constraint T_0 such that*

$$0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})|$$

Proof. Let σ_1 and σ_2 two distinct solutions of the problem $\mathcal{P}$. Let i_1 such that $\sigma_1(X_{i_1}) \neq \sigma_2(X_{i_1})$. Let $i_2, \ldots, i_v$ other indices such that $|\mathcal{D}(X_{i_k})| \neq 1, 2 \leq k \leq v$. Let us define the table

$$T_0 = \textsc{Table}\left((X_{i_1}, \ldots, X_{i_v}), \{(\sigma_1(X_{i_1}), \ldots, \sigma_1(X_{i_v}))\}\right)$$

Then $\sigma_1 \in Sols(\mathcal{P} \wedge T_0)$ so $Sols(\mathcal{P} \wedge T_0) \neq \varnothing$, and $\sigma_2 \notin Sols(\mathcal{P} \wedge T_0)$ so $Sols(\mathcal{P} \wedge T_0) \neq Sols(\mathcal{P})$. Since we add a constraint to $\mathcal{P}$ to build $\mathcal{P} \wedge T_0$, we have $Sols(\mathcal{P} \wedge T_0) \subseteq Sols(\mathcal{P})$, thus $0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})|$. ◀

▶ **Lemma 4.** *There exists a constant $\rho > 0$, depending only on the initial problem, such that, for T a randomly chosen table constraint with v variables:*

$$\mathbb{P}\left(0 < |Sols(\mathcal{P} \wedge T)| < |Sols(\mathcal{P})|\right) \geq \rho$$

Proof. We know from Lemma 3 that there is at least one table constraint T_0 such that $0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})|$. Let d be the maximum size of the domains of the initial problem. We bound the probability of $\textsc{RandomTable}(v, p)$ to pick exactly T_0 (up to ordering of the scope of the constraints). Let T be a random table returned by $\textsc{RandomTable}(v, p)$. We want to bound

$$\begin{aligned}\mathbb{P}(T = T_0) &= \mathbb{P}\left(scp(T) = scp(T_0) \wedge rel(T) = rel(T_0)\right)\\ &= \mathbb{P}\left(scp(T) = scp(T_0)\right) \cdot \mathbb{P}\left(rel(T) = rel(T_0) | scp(T) = scp(T_0)\right)\end{aligned}$$

There is $\binom{n}{v}$ ways to choose the v variables appearing in the table (the ordering does not matter), so $\mathbb{P}\left(scp(T) = scp(T_0)\right) = 1/\binom{n}{v}$. Let k the number of tuples in T_0. There are at most d^v possible tuples in total. The probability to choose every tuple in T_0 and not the others is $p^k(1-p)^{d^v-k}$. As $k \leq d^v$ we have the lower bound $\mathbb{P}\left(rel(T) = rel(T_0) | scp(T) = scp(T_0)\right) \geq p^k(1-p)^{d^v-k} \geq min(p, 1-p)^{d^k}$. By defining $\rho = \frac{1}{\binom{n}{v}} min(p, 1-p)^{d^k}$ we have the desired bound, and $\rho > 0$ because $0 < p < 1$. ◀

We proved that during an iteration of the loop, there is a probability strictly greater than 0 to remove solutions without making the problem inconsistent. We can now prove that the algorithm terminates with probability 1. The proof is similar to the one showing that tossing a fair coin, until tails comes up, ends with probability 1.

▶ **Theorem 5.** *Algorithm 2 terminates with probability 1.*

Proof. For some $k > |Sols(\mathcal{P})| - \kappa$, we want to find an upper bound of the probability that the algorithm has not stopped after k iterations. In some cases, an iteration reduces the number of solutions to the problem without making it inconsistent. There can be at most $|Sols(\mathcal{P})| - \kappa$ such iterations, because the algorithm stops if there is less than κ solutions (condition of the while line 5). For the other iterations, the condition of the while loop ensures that: either the (most recently added) table made the problem inconsistent, or it did not reduce the number of solutions. The probability for this to happen is less than $1 - \rho$, as stated in Lemma 4. Thus, the probability that, after k iterations, the algorithm did not stop, is less than $(1-\rho)^{k-|Sols(\mathcal{P})|+\kappa}$. This probability tends to zero when k tends to infinity. This proves that the algorithm stops with probability 1. ◀

Algorithm 3 Algorithm of dichotomic addition of tables.

```
1 Function DichotomicTableAddition(P, nbTables, κ, v, p)
    Data: A CSP P = ⟨X, D, C⟩ , nbTables > 0, κ ≥ 2, v > 0, 0 < p < 1
    Result: P with the new table constraints, and the number of added tables
2   T ← array of size nbTables;
3   for i = 0 to nbTables − 1 do
4     T[i] ← RandomTable(P, v, p);
5   S ← FindSolutions(P ∧ ⋀_{t∈T} t, κ);
6   while |S| = 0 ∧ |T| > 0 do
7     T ← T[0 : |T|/2[;
8     S ← FindSolutions(P ∧ ⋀_{t∈T} t, κ);
9   return P ∧ ⋀_{t∈T} t, |T|;
```

This proof is built with an upper bound, and considers the worst case (when solutions are eliminated slowly), but in practice there is more than one table satisfying Lemma 3. The solving time will be studied in practice in Section 4.5.

3.3 Dichotomic table addition

It is possible to improve the efficiency of the algorithm by increasing the number of tables added at each step. At the beginning of the search, a table has a small probability to make the problem inconsistent, so it is wiser to add more constraints to reduce the number of calls to the solver. This algorithm is inspired from the unbounded dichotomic search: first, find i such that the value we want to guess is between 2^i and 2^{i+1}, and then, run a usual dichotomic search between 2^i and 2^{i+1}.

The algorithm of dichotomic table addition is presented in Algorithm 3, and should replace lines 6 to 9 of Algorithm 2. Let τ be the number of tables added at the previous step, we choose $nbTables = 1$ if $\tau = 0$ or $nbTables = 2\tau$ otherwise, and $nbTables$ tables are generated and stored in an array $\mathcal{T}$. The algorithm then enumerates κ solutions to the problem where the tables in $\mathcal{T}$ have been added. If there is no solutions, it deletes half of the constraints in $\mathcal{T}$. The procedure stops when the problem is satisfiable or $|\mathcal{T}| = 0$.

Theorem 5 can be extended to the case of the dichotomic table addition, because line 6 in Algorithm 3 ensures that the problem does not become inconsistent.

3.4 Discussion

In this section, we discuss the algorithmic choices we have made in Algorithm 2.

3.4.1 Quality of the division by the tables

In the proof of Theorem 2 the random variables $(\gamma_\sigma)_{\sigma \in Sols(\mathcal{P})}$ are not independent. For example, let σ_1 and σ_2 two solutions to the problem that only differ on one variable X, then

$$\mathbb{P}(\gamma_{\sigma_2} = 1 | \gamma_{\sigma_1} = 1) = \mathbb{P}(X \in scp(T)) \cdot p + \mathbb{P}(X \notin scp(T)) \quad (1)$$

Indeed, if the variable X appears in T, then σ_2 will be kept with probability p, but if X is out of the scope of T, then σ_2 will always be kept. If the table does not have all the variables in its scope, then it may not split the clusters of solutions which take the same values on

multiple variables. This notion of independence is central in the approaches of Kuldeep Meel [13] to show the uniformity of the sampling. Contrarily to this approach, our sampling is not uniform. We choose to have tables of a controlled size for sake of efficiency.

Formula 1 showing the non independence also shows that increasing the number of variables in the table makes the random variables γ_σ more independent, hence the whole sampling process closer to uniformity. Tables containing all the variables of the problem would make the random variables γ_σ fully independent, since in this case $\mathbb{P}(X \notin scp(T)) = 0$. This would give a theoretical guarantee on the sampling, but is impossible to generate in practice.

3.4.2 Influence of the parameters

Three parameters have to be chosen to run the algorithm. We can already estimate the impact of the parameters on the running time and on the quality of the randomness.

- As seen in the previous subsection, increasing the number of variables in the tables should improve the randomness, but will also exponentially increase the number of tuples in the table, with a negative impact on the running time.
- Reducing the probability of adding a tuple in a table should improve the running time because the tables will be smaller, so the propagation will be faster, and the number of added tables will be lower because the problem will be more quickly reduced.
- The impact of the pivot on the running time is unclear. Having a higher pivot means that more solutions have to be enumerated at each step, but it also means that the algorithm will stop after adding fewer constraints.

These hypotheses will be experimentally verified in section 4.

4 Experiments

This section presents the experiments done to test our approach. First, we evaluate the behaviour in term of randomness. Then, we compare the running time of our approach to the strategy RandomVarDom. The code is available online [1], along with all the scripts to generate the figures presented in this article.

4.1 Implementation

The implementation has been done in Java 11 using the constraint solver `choco-solver` version 4.10.6 [16]. It is possible to create a model directly in Java using the `choco-solver` library, or by giving a file in the FlatZinc format (generated from the MiniZinc format). Unless the FlatZinc file defines a strategy, the solver default strategy is used (`dom/Wdeg` [3] and `lastConflict` [12]).

A technical improvement has been done, by adding a propagation step before the generation of a table (before line 6 of algorithm 2). This avoids enumerating some tuples that would be immediately deleted by propagation.

In the following, the algorithm used is TableSampling with DichotomicTableAddition.

[1] `https://github.com/MathieuVavrille/tableSampling`

Management of randomness

The random number generator used is the default one in Java : `java.util.Random`. This generator uses a formula of linear congruence to modify a seed on 48 bits given as input. The Java documentations points to [11], see section 3.2.1 for more information. This randomness generator has flaws (notably a period of 2^{48}), but is sufficient to our needs (as shown in [2]).

The implementation uses a single instance of the random number generator, passed as argument to every function needing it. This avoids a non independence behaviour due to a bad generation of random seeds.

4.2 Problems

The approach is independent from the constraints of the problem, so we were able to apply it on four different problems, including three real life problems. We present here the models and their characteristics.

N-queens

The first problem is the N-queens problem, which consists of placing N queens on an $N \times N$ chessboard such that no queen attacks an other one (queens attack in every 8 directions, as far as possible). We implemented it with the usual model with N variables with domain $[1, N]$, an `all_different` constraint and inequality binary constraints (for diagonal attacks).

Renault Mégane Configuration

This is the problem of configurations of the Renault Mégane introduced in [1] and already used in [10] for the search of diverse solutions. There are 101 variables with domains containing up to 43 values, and the 113 constraints are modeled by table constraints, the majority of them are non binary. This problem is loosely constrained, hence having more than $1.4 \cdot 10^{12}$ solutions.

On Call Rostering

This problem models the system of duty, notably used by healthcare workers. This problem is available in the MiniZinc benchmarks [2] and contains different constraint types, such as linear constraints, global constraints `count`, absolute values, implications and table constraints. Many datasets are available but only the smallest (`4s-10d.dzn`) has been used here. It is an optimisation problem (minimization), so it was necessary to transform this problem into a satisfaction problem by bounding the objective function. The optimal value is 1:

- There are 136 solutions with $obj \leq 1$
- There are 2,099 solutions with $obj \leq 2$
- There are more than 10,000 solutions with $obj \leq 3$

By randomly sampling the solutions, the solver can be used as a tool to help people creating plannings to decide on (giving them multiple plannings to compare), and brings a form of equity between the workers. Indeed, oriented search methods could favor some workers at the expenses of others.

[2] `https://github.com/MiniZinc/minizinc-benchmarks/tree/master/on-call-rostering`

Feature Models

These are problems of software management, helping to decide on the order of implementation of software features. The problem is specified in the MiniZinc format in [17] using the data in [19]. Again, it is an optimisation problem (maximization), the optimal value is 20,222. We add the constraint $obj \geq 17,738$ to make it a satisfaction problem with 95 solutions.

4.3 Experimental behaviour

We discuss here the experimental behaviour of TableSampling. In practice, we see that most of the computations are done at the beginning of the algorithm. At the beginning, most of the tables added do not make the problem inconsistent. We really benefit from the dichotomic addition of tables. Most of the time is spent finding κ solutions, because as only few tables are added to the problem, the search space is not reduced much, so searching for solutions is not sped up.

At the end of the algorithm, when there are only few solutions remaining (but still more than κ), there is a higher probability to make the problem inconsistent by adding a table. Actually, this is not an issue, because it becomes really fast to find the solutions (or to prove that the problem is inconsistent). This is due to the fact that all the tables added previously really restrict the search space and are quickly propagated.

4.4 Quality of the randomness

The first goal of the experiments is to evaluate the quality of the randomness, i.e. knowing if the solutions are sampled randomly and uniformly. The following results show that even if the solutions are not sampled uniformly, the approach using table constraints is *more uniform* than the strategy RandomVarDom.

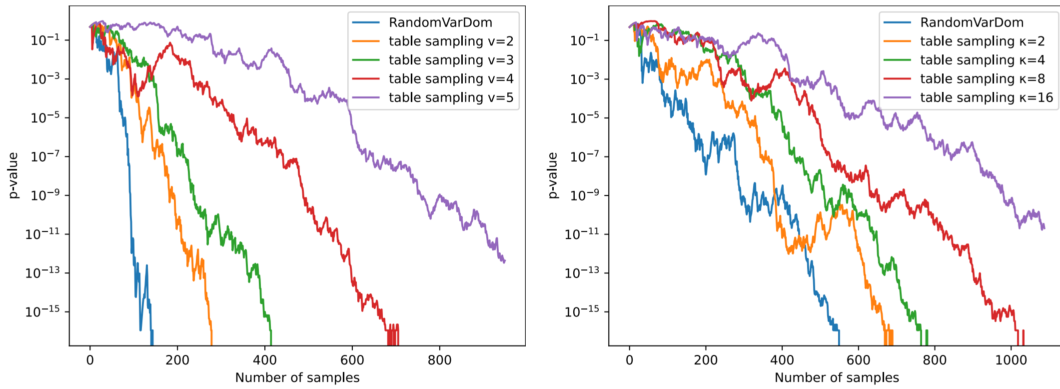
4.4.1 Evaluation of the uniformity

To have a numerical measure of the uniformity of the sampling, we used the χ^2 test. Knowing the number $nbSols$ of solutions of a problem (and numbering these solutions), $nbSamples$ samples are drawn and the number of occurences $nbOcc_i$ of each solution $i \in \{1, \ldots, nbSols\}$ is counted. We compute the value of the variable

$$z_{exp} = \sum_{k=1}^{nbSols} \frac{(nbOcc_k - nbSamples/nbSols)^2}{nSamples/nbSols}$$

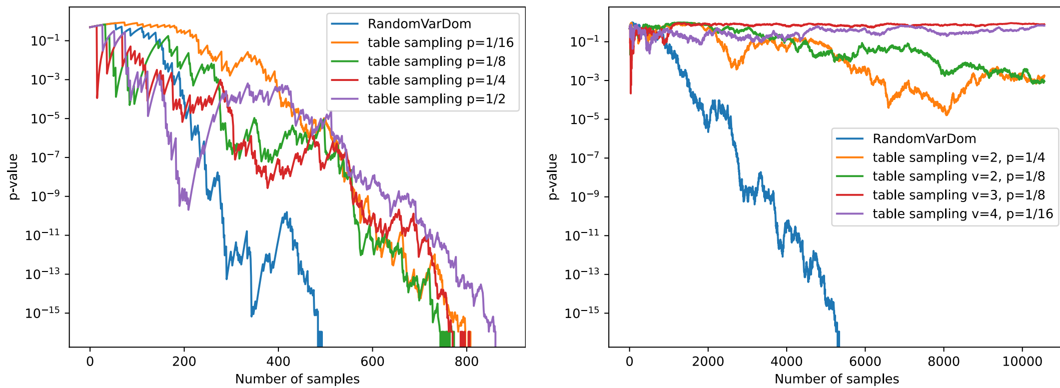
and then the p-value of the test[3] (i.e. the probability that the χ^2 law takes a more extreme value than z_{exp}). This p-value gives a numerical value of the quality of the randomness. More specifically, a large number of samples are drawn (more than the number of solutions) and the evolution of the p-value depending on the number of samples is plotted. In our case, as the sampling is not uniform, the p-value will tend to 0 when the number of samples increases, but we remark that the sampling using tables has a p-value which tends slower to 0 than the default sampling using RandomVarDom.

[3] We use the library "Apache Commons Mathematics Library" (`https://commons.apache.org/proper/commons-math/`) for the probability computations



(a) Feature Models problem with $\kappa = 2$ and $p = 1/2$.

(b) On Call Rostering problem with the constraint $obj \leq 1$, $v = 4$ and $p = 1/8$.



(c) On Call Rostering problem with the constraint $obj \leq 2$, $\kappa = 16$ and $v = 5$.

(d) 9-queens problem with $\kappa = 8$, with different values for v and p.

Figure 1 Evolution of the p-value on different problems, with different parameters. Plotting of graphs using the table sampling and RandomVarDom.

To do this test we need to know the number of solutions *nbSols* and to sample multiple times *nbSols* solutions, so the evaluation of the randomness can only be done on small instances. These instances are the 9-queens (352 solutions), the Feature Models with the constraint $obj \geq 17,738$ (95 solutions) and the On Call Rostering problem with the constraints $obj \leq 1$ and $obj \leq 2$ (136 and 2,099 solutions).

We want to evaluate the impact of the evolution of a parameter (number of variables, pivot, or probability) on the randomness of the algorithm of sampling by tables. To do so, in the figures that follow, we plot the evolution of the p-value for the strategy RandomVarDom, as well as for different values of parameters for the sampling by tables, by changing one parameter at a time. The legend gives the parameters associated to each execution (v for the number of variables, κ for the pivot and p for the probability).

▶ Remark 6. The figures show the p-value in a logarithmic scale, because it tends to 0. Moreover, as the computations are done using floating point representation, a p-value smaller than 10^{-16} will be considered to be equal to 0.

4.4.2 Impact of the number of variables

We first vary the number of variables used in the generated tables. Fig. 1a shows the evolution of the p-value on the Feature Models problem with parameters $\kappa = 2$ and $p = 1/2$. We remark that increasing the number of variables in the table makes the p-value tend to zero slower, meaning that the sampling is closer to uniformity. As we remarked earlier, this is due to a better independence in the probability that two solutions will satisfy the table constraints (see section 3.4.1).

4.4.3 Impact of the pivot

On Fig. 1b, we vary the pivot on the problem On Call Rostering, with the constraint $obj \leq 1$ and parameters $v = 4$ and $p = 1/8$. Here we observe that increasing the pivot improves the randomness. Indeed, when the pivot is high, at each step a lot of solutions are enumerated, and in the end a random solution will be picked among a lot of other solutions, leading to a better randomness. The extreme case is the perfect (but costly) sampling process, when the pivot is higher than the number of solutions: the algorithm is then simply an enumeration of all the solutions and returns a random solution.

4.4.4 Impact of the probability

Fig. 1c shows the p-value for different values of the table probability p, on the On Call Rostering problem, with the constraint $obj \leq 2$ and the parameters $\kappa = 16$ and $v = 5$. There is no clear influence of the probability to add tuples in the tables to the quality of the randomness. This allows, when choosing the probability, to focus on the running time (as we will see in section 4.5.1).

4.4.5 Quality of the randomness

The last test was done on the 9-queens problem. Fig. 1d shows the evolution of the p-value with $\kappa = 8$ and different values for v and p. On this particular problem (and on the N-queens problem for any N), the sampling using the table constrains is actually uniform in practice (the p-value tends to 1). As we already said, we have no theoretical guarantee with our approach, but it seems that, on some problems, we may achieve uniformity in practice. We believe that it is due to the structure of the solution space, because the N-queens problem is a very structured problem with many symmetries. Thus, it is likely that the solutions are properly spread on the search space.

4.4.6 Comparison with RandomVarDom

On every graph, we also plotted the evolution of the p-value for the sampling using the search strategy RandomVarDom. We see on the three first graphs that our approach tends to zero after significantly more samples than RandomVarDom. Using TableSampling makes the sampling more uniform. For example, after sampling 50 solutions, the p-value of RandomVarDom is 0.004, but the p-value of TableSampling (with parameters $\kappa = 2, v = 2, p = 1/2$) is 0.1. On the N-queens problem, RandomVarDom is not uniform but our approach is, meaning that it really improved the quality of the randomness.

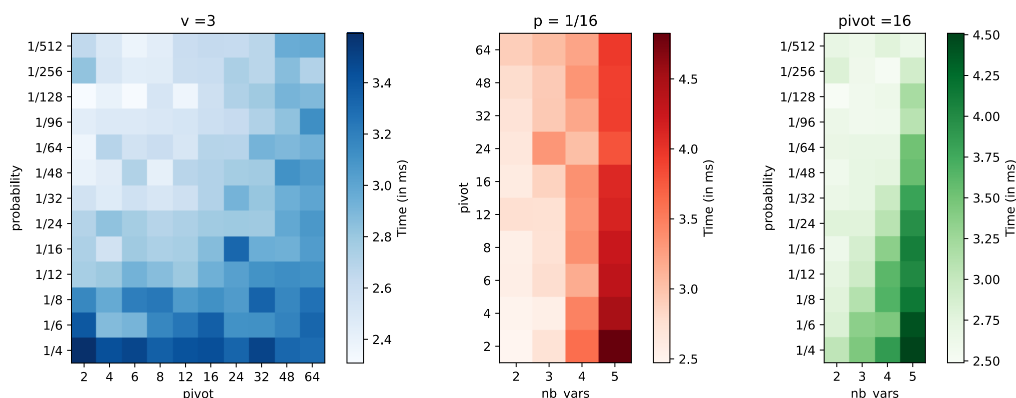


Figure 2 Heat maps of the time to sample one solution, by fixing different parameters, on the On Call Rostering problem with the constraint $obj \leq 3$.

4.5 Running time

The evaluation of the running time is done in two parts. The parameters may have an important impact on the running time: we experimentally investigate in Section 4.5.1 which parameters have a positive impact. Then, we compare the running time of our method and that of using RandomVarDom as a randomization strategy.

In the section, we use problems with more than 10,000 solutions. This eliminates the Feature Models problem, which has too few solutions to provide a meaningful statistical test. For each method, 50 samples are done, in order to to average the running time.

4.5.1 Impact of the parameters

To show the impact of the parameters on the time to sample one solution, we fix one parameter, and vary the two other parameters, to plot a heat map of the time, in function of the two parameters. Fig. 2 shows the three heat maps obtained by fixing the number of variables, the pivot or the probability. The hypotheses done in section 3.4.2 are verified here experimentally:

- decreasing the number of variables in the tables, decreases the running time;
- decreasing the probability of adding a tuple in the table decreases the running time.

As we previously saw, increasing the number of variables improves the randomness. There is a compromise to do between the running time (having few variables in the tables) and the quality of the randomness (having many variables in the tables).

We also saw that the probability to add a tuple in the tables does not have a clear impact on the quality of the randomness, so it is best to choose small probabilities to have smaller tables, hence giving a faster propagation, as well as fewer tables added during the computations.

The number of variables in the tables should be chosen as a trade-off between the desired quality of randomness and the running time. It will depend on the application: instances with big domains may require smaller v not to have too big tables (for example, $v = 4$ for domains of size 100 would have to enumerate 10^8 tuples). From our experiments, we suggest as a baseline to use the parameters $\kappa = 16$ and $p = 1/32$.

Table 1 Comparison of the time to sample a solution between RandomVarDom and table sampling.

Problem	RandomVarDom	Table sampling				Ratio
		v	κ	p	Time	
Renault Mégane Configuration	32 ms	2	16	1/16	55 ms	1.7
				1/32	59 ms	1.8
			32	1/16	89 ms	2.8
				1/32	86 ms	2.7
		3	16	1/16	74 ms	2.3
				1/32	67 ms	2.1
			32	1/16	83 ms	2.6
				1/32	65 ms	2.0
On Call Rostering	38 ms	2	16	1/16	14 ms	0.36
				1/32	14 ms	0.38
			32	1/16	15 ms	0.4
				1/32	18 ms	0.46
		3	16	1/16	18 ms	0.47
				1/32	15 ms	0.4
			32	1/16	19 ms	0.5
				1/32	17 ms	0.44
12-queens	2 ms	2	16	1/16	4 ms	2.4
				1/32	4 ms	2.3
			32	1/16	7 ms	4.1
				1/32	7 ms	3.9
		3	16	1/16	10 ms	5.6
				1/32	8 ms	4.3
			32	1/16	12 ms	6.9
				1/32	11 ms	6.0

4.5.2 Comparison to RandomVarDom

In this Section, we compare the running time to RandomVarDom. Table 1 shows the running time to sample one solution for 8 different sets of parameters of table sampling. To take into account the variability of the solving time, we measure it on 50 samplings, and report the average time to get one sample. The ratio between the time of TableSampling and the time of RandomVarDom is also given.

We showed that the quality of the randomness of TableSampling is better than the one of RandomVarDom, and we would expect to pay a price in running time in return. But, in practice, the running times are still within the same order of magnitude. On the On Call Rostering problem, it is even two to three time faster to use TableSampling instead of RandomVarDom On the other two problems, RandomVarDom is faster. This behaviour can be explained quite easily: the On Call Rostering problem is very sparse, and there are many values in the domains of the variables that do not lead to solutions. Thus, the RandomVarDom strategy provokes a lot of fails during the search, because it often picks values that do not appear in solutions. In comparison, on the problems of Renault Mégane Configuration and N-queens, a lot of values in the domains of the variables may lead to solutions, so the probability of failing because of a bad choice is low. Our approach also allows to use a different and more powerful strategy, since it can be combined with any search strategy. We can thus take advantage of all the progress made in the design of search strategies.

5 Conclusion

We presented an algorithm using table constraints to randomly sample solutions of a problem. We improved this algorithm by increasing the number of tables added at each step. We experimented our approach on four different problems, involving different types of constraints and different domains of variables. We showed that the sampling is closer to uniformity than a sampling using the search strategy RandomVarDom. Even with this improved randomness, the running time remains comparable to RandomVarDom. Our approach uses the solver as a black box, hence can be applied to a wide range of problems.

In the future, we plan to study structured problems, and investigate how to improve the sampling using the structure of the problem. Optimisations problems are also an other research direction. Instead of artificially turning them into satisfaction problems, we plan to use the samples previously found to directly search for close to optimal solutions.

References

1. Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps—application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.
2. Eric Bach. Realistic analysis of some randomized algorithms. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 453–461, 1987.
3. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.
4. Rina Dechter, Kalev Kask, Eyal Bin, Roy Emek, et al. Generating random solutions for constraint satisfaction problems. In *AAAI/IAAI*, pages 15–21, 2002.
5. Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.
6. Stefano Ermon, Carla P Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. *arXiv preprint*, 2012. `arXiv:1210.4861`.
7. Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In *International Conference on Principles and Practice of Constraint Programming*, pages 711–715. Springer, 2006.
8. Renaud Hartert and Pierre Schaus. A support-based algorithm for the bi-objective pareto constraint. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2674–2679. AAAI Press, 2014. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8337`.
9. E. Hebrard. *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. PhD thesis, University of New South Wales, 2006.
10. Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *AAAI*, volume 5, pages 372–377, 2005.
11. Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.
12. Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Last conflict based reasoning. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 133–137. IOS Press, 2006. URL: `http://www.booksonline.iospress.nl/Content/View.aspx?piid=1661`.

13 Kuldeep S Meel. Constrained counting and sampling: bridging the gap between theory and practice. *arXiv preprint*, 2018. arXiv:1806.02239.

14 Karl Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

15 Gilles Pesant, Kuldeep S. Meel, and Mahshid Mohammadalitajrishi. On the usefulness of linear modular arithmetic in constraint programming. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 18th International Conference, CPAIOR 2021, Vienna, Austria, September 21-24, 2021, Proceedings*, Lecture Notes in Computer Science. Springer, 2021.

16 Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL: http://www.choco-solver.org.

17 Björn Regnell and Krzysztof Kuchcinski. Exploring software product management decision problems with constraint solving-opportunities for prioritization and release planning. In *2011 Fifth International Workshop on Software Product Management (IWSPM)*, pages 47–56. IEEE, 2011.

18 Francesca Rossi, Kristen Brent Venable, and Toby Walsh. Preferences in constraint satisfaction and optimization. *AI Mag.*, 29(4):58–68, 2008. doi:10.1609/aimag.v29i4.2202.

19 Günther Ruhe and Moshood Omolade Saliu. The art and science of software release planning. *IEEE software*, 22(6):47–53, 2005.

20 Yevgeny Schreiber. Value-ordering heuristics: Search performance vs. solution diversity. In *International Conference on Principles and Practice of Constraint Programming*, pages 429–444. Springer, 2010.