Constraints
https://doi.org/10.1007/s10601-022-09329-w

Solution sampling with random table constraints

Mathieu Vavrille[1] · Charlotte Truchet[1] · Charles Prud'homme[2]

Accepted: 8 April 2022

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. By uniformly sampling the solutions in one cell, we can generate random solutions without revamping the model of the problem. We ensure the randomness by introducing a new family of hashing constraints: randomly generated tables, which keeps the cost of the hashing process tractable. 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. We also use our algorithm with an other, more powerful, set of hashing constraints: linear modular equalities. We experimentally show that the resulting sampling is uniform, at the cost of a longer running time.

Keywords Solutions · Sampling · Table constraint · Uniform distribution

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 [1]), user preferences [2], diverse solutions [3], robust solutions [4], etc. In this article,

Charlotte Truchet and Charles Prud'homme contributed equally to this work.

✉ Mathieu Vavrille
mathieu.vavrille@univ-nantes.fr

Charlotte Truchet
charlotte.truchet@univ-nantes.fr

Charles Prud'homme
charles.prudhomme@imt-atlantique.fr

[1] Laboratoire des Sciences du Numérique de Nantes, 2 Chemin de la Houssinière, Nantes 44322, France

[2] TASC, IMT-Atlantique, LS2N-CNRS, 4 rue Alfred Kastler, Nantes 44307, France

we propose a method to sample solutions of a constraint problem, without modifying its model. This work is motivated by many situations where the user of a constraint solver needs randomised solutions: to ease user feedback and decision making (by providing a variety of solutions, representative of the solution space), to ensure equity (to avoid patterns in consecutive solutions, for instance in planning problems), or 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). Another major drawback of this technique is that RandomVarDom replaces the strategy that may have been chosen or built for the problem, and this is likely to increase the solving time.

Our approach is inspired from UniGen [5], 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.

The chosen family of random hashing constraints have a heavy impact on the running time. In order to keep it reasonable, we choose to randomly generate table constraints [6], 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.

We implemented our proposal on top of Choco-solver [7] and compare it to RandomVarDom on a wide benchmark, built from the yearly MiniZinc competition. We show that our approach using the table constraints improves, in practice, the quality of the randomness compared to RandomVarDom, while also sampling more problems.

We also apply our algorithm with linear modular equalities [8] that are hashing constraints with stronger theoretical properties in terms of randomness, but harder to propagate. On our benchmark set, using linear modular equalities gives a better randomness quality compared to the table constraints, as it provides a uniform sampling. The downside is a longer running time.

Outline Section 2 presents the related works. Section 3 gives the notations and recalls the definitions that are needed afterwards. Section 4 presents our approach to sample solutions and section 5 is a discussion about our approach. Section 6 describes the experimental methodology, and Sections 7 and 8 present the results.

2 Related works

The question of sampling combinatorial problems is central in hardware/software verification and testing, mainly on SAT models. Some testing problems have recently been expressed with CP models, for instance because of the need for non-Boolean variables : in [9] the authors define a variability model on continuous variables. They then discretise these variables and sample solutions using RandomVarDom. In [10], the Test Suite Reduction problem is tackled with constraint optimisation problems using global constraints. In [11], array constraints are used for handling data structures.

Other works, both in SAT and in CP, have tackled the question of solution sampling or diversification in the literature. We present here an overview.

2.1 In SAT

Our work is inspired by the works on the XOR constraints. For the sampling task, these constraints were first used in [12] leading to the XORSample algorithm. They were also used in Toggle [13] to improve the coverage of generated simulations.

Meel's work on UniGen extends the XORSample algorithm into a near-uniform sampler for SAT problems. As UniGen inspired our work, we recall the algorithm here. UniGen is a two-step algorithm. The first step consists in running ApproxMC, an algorithm for SAT model counting. The search space is divided by adding randomly chosen XOR constraints to the model, until there is less than a given number of solutions. These solutions are counted within the smaller space marked by the additional constraints. This number, multiplied by a ratio between the volume of the smaller space and the volume of the real search space, gives a first approximation for the solution number. Applying this algorithm multiple times obtain a more accurate approximation. 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 [8]. Our algorithm uses the same idea of additional constraints to divide the search space, within the CP framework, for solution sampling.

There is a broad literature of SAT solution sampling. Early sampling approaches use Binary Decision Diagrams (BDD) to sample a problem [14]. After the compilation of the BDD, the random generation of a solution is a one-pass process, *i.e.* no backtrack is required. Introducing biases to the branching probability of the BDD allow to modify the probability distribution.

There are sampling methods based on random walks, such as the Markov Chain Monte Carlo algorithm. This led to the creation of WalkSat [15]. This algorithm incorporates a greedy bias in the random walk, and the authors experimentally show that it samples nearly uniformly on the problems they used. They also propose a hybrid approach interleaving simulated annealing steps and the WalkSat steps.

In [16] the same ideas of random walks are used. They combine the concepts of the Gibbs sampling, Metropolis-Hasting algorithm and WalkSat. They also extend Metropolis moves to integer variables.

We particulary want to mention SearchTreeSampler [17], which is closer to our approach. The approach has two parameters: k the maximum number of solutions recorded at each step and l the increase in the size of the pseudo-solutions. Pseudo-solutions of size m are defined as partial instantiations, where only the first m variables are instantiated, and that can be extended to solutions. The algorithm is a recursive strategy to increase the size of pseudo-solutions until m is equal to the number of variables. The algorithm starts with a set Φ containing the only pseudo-solution of size 0 (no variables are instantiated). The recursive step builds, from a set Φ of pseudo-solutions of size m, a new set Φ' of pseudo-solutions of size $m + l$. To compute this, k pseudo-solutions are sampled randomly from Φ, and extended to all the possible pseudo-solutions of size $m + l$. All these new pseudo-solutions of size $m + l$ make the set Φ'. By calling the procedure on the newly created set Φ', and so on, a random set of pseudo-solutions of size the number of variables (i.e. solutions) is created.

This approach works well because of the binary domains, but in CP the possible large domains would be an issue. To extend the pseudo-solutions to the l following variables, the algorithm may do up to 2^l calls to a SAT solver. In CP, the domains may be bigger, say, of size d. This means that one iteration may do up to $d^{l'}$ calls to a CP solver. Keeping the same order of magnitude between 2^l and $d^{l'}$ forces to choose l' smaller than l. This would force to do more iterations (*i.e.* having a small value for l'), which in the worst case would lead to an algorithm close to RandomVarDom (when $l' = 1$).

An other algorithm, QuickSampler [18], takes a completely different approach, giving different properties. The authors remark that previous approaches need a big number of solver calls to generate few solutions. They propose an algorithm with the opposite behaviour: with few solver calls it can produce a big number of solutions. To do so, they use ”atomic mutations” of solutions to generate new instantiations. They found out that combining these ”atomic mutations” often result in solutions of the problem. This way, QuickSampler is able to generate solutions orders of magnitude faster than other approaches. The downfall is the resulting solution distribution that is less precise than the other approaches such as Unigen or SearchTreeSampler.

2.2 In CP

Solution sampling in constraint programming was first studied in [19] and [20], 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. [20] is based on Iterative Join Graph Propagation [21] to compute the densities of solutions. By default this approach is exponential, but by bounding the number of variables used by i, it is possible to have an approximation of the densities bounded exponentially by i. We took the opposite view of designing a fast sampling method, that does not guarantee 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 [3] the model is re-written to add the solutions distance as constraints. In [22] 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 two related questions, but the final goals are in practice very different. On one hand, sampling multiple solutions does return diverse solutions, but it is very unlikely to find the most distant solutions. On the other hand, diversification is somehow antagonistic to random sampling, as solutions close to other ones may never be returned.

3 Preliminaries

This section provides the reader with notions and notations that we need afterwards. We first introduce our notations within the CP framework, and two previously existing methods to sample solutions in CP: the naive approach consisting in randomising the search heuristics RandomVarDom, and a series of random constraints introduced by [8], that can replace the XOR constraints for Meel’s SAT algorithm in CP with similar properties. Finally, we describe the statistical test that we will use to quantify the quality of our random method.

3.1 Constraint programming

A CSP $\mathcal{P}$ is a triple $\langle \mathcal{X}, \mathcal{C}, \mathcal{D} \rangle$ where

- $\mathcal{X} = \{X_1, \dots, X_n\}$ 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 $scp(C) = (X_{i_1}, \dots, X_{i_r})$, 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 $(x_{i_1}, \dots, x_{i_r}) \in rel(C)$. The constraints can be defined in extension (called table constraints [6]) 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.

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.

3.2 Notations

We note $\mathbb{F}_p = \{0, \dots, p-1\}$ the set of integers modulo p.

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 $\mathcal{P}$.

We note $\mathbb{P}(A)$ for the probability of the random event A. Given a random variable R, we note $\mathbb{E}(R)$ its expected value.

3.3 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 can be easily implemented in any constraint solver, without modifying the solver architecture. However, it sets the search heuristics, preventing the user from using other, more efficient, exploration strategies. Even though it provides a random search at a very low cost, the distribution of the solution may be far from the uniform distribution. We can see this on a simple example.

Example of solving using RandomVarDom

Let us define a problem with two Boolean variables X_1 and X_2, and only one constraint $X_1 + X_2 > 0$, *i.e.,* $\mathcal{P} = \langle\{X_1, X_2\}, \{X_1 \mapsto \{0,1\}, X_2 \mapsto \{0,1\}\}, \{X_1 + X_2 > 0\}\rangle$. We want to know the probability that a CP-solver using the RandomVarDom strategy will return a given solution. Let us run the solving process step by step. The initial propagation step cannot remove any value from the domain of the variables. The decision step is applied. A variable has to be chosen at random, there are two cases (each of equal probability 1/2 of being chosen):

- X_1 is chosen, then one of its value is chosen at random, there are again 2 cases (each of equal probability 1/2 of being chosen):
  - 0 is chosen. Then the only solution possible from this decision is the solution $(X_1, X_2) = (0, 1)$ (the propagation step will easily filter 0 from the domain of X_2);
  - 1 is chosen. Then the constraint is satisfied, so no more filtering can be done. A decision step is done again, picking the only uninstantiated variable X_2. Then there are two possible values:

    * 0, then the solution is $(X_1, X_2) = (1, 0)$;
    * 1, then the solution is $(X_1, X_2) = (1, 1)$;
- X_2 is chosen, then one of its value is chosen at random, there are again 2 cases (each of equal probability 1/2 of being chosen):
  - 0 is chosen. Then the only solution possible from this decision is the solution $(X_1, X_2) = (1, 0)$ (the propagation step will easily filter 0 from the domain of X_1);
  - 1 is chosen. Then the constraint is satisfied, so no more filtering can be done. A decision step is done again, picking the only uninstantiated variable X_1. Then there are two possible values:

    * 0, then the solution is $(X_1, X_2) = (0, 1)$;
    * 1, then the solution is $(X_1, X_2) = (1, 1)$.

Let us define s to be the random solution returned by the algorithm. The probability to return each solution can be computed. For example for $s = (0, 1)$, we have

$$\begin{aligned}\mathbb{P}(s = (0,1)) = &\ \mathbb{P}(X_1 \text{ is chosen first}) \cdot \mathbb{P}(0 \text{ is chosen for } X_1)\\ &+ \mathbb{P}(X_2 \text{ is chosen first}) \cdot \mathbb{P}(1 \text{ is chosen for } X_2) \cdot \mathbb{P}(0 \text{ is chosen for } X_1)\\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2}\\ &= \frac{3}{8}\end{aligned}$$

The final random distribution of the solutions when using the RandomVarDom search strategy is

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

The implications on the running time to find solutions and the quality of the randomness on different problems are discussed in section 8.

3.4 Hashing by linear modulo constraints

3.4.1 Hashing constraints

As said above, our method relies on an idea first introduced on SAT [12]. It consists in adding hashing constraints to sample the solutions. Considering that the number of solutions is unknown (and cannot be computed at a reasonable cost), the core idea is to progressively divide the search space into small buckets, until there are few solutions inside one bucket. It is then easy to randomly pick one of these few solutions.

The way the buckets are randomly chosen determines the quality of the sampling. Imagine that the hashing constraints systematically cut one domain into two parts, and only keeps the part with smaller values: then the solutions with smaller values would appear more often. In fact, any series of unary hashing constraints would introduce a bias in the sampling. For instance, if two different solutions have one value in common, they will appear in the same bucket more often than they should. This must not happen for the sampling to be uniform.

To obtain, or approximate, uniformity in the sampling, the hashing constraints must not feature any dependency inside the generated buckets. A good quality hashing can be obtained thanks to a property called 2-independence.

Definition 1 (2-independence) Let $\mathcal{P}$ be a CSP. Let $\mathcal{H}$ be a family of hashing constraints. Let s_1 and s_2 be two distinct solutions of $\mathcal{P}$. The family of hashing constraints is said to be *2-independent* if, when randomly taking a hashing constraint $h \in \mathcal{H}$,

$$\mathbb{P}(s_1 \in Sols(\mathcal{P} \wedge h) \wedge s_2 \in Sols(\mathcal{P} \wedge h)) = \mathbb{P}(s_1 \in Sols(\mathcal{P} \wedge h)) \cdot \mathbb{P}(s_2 \in Sols(\mathcal{P} \wedge h))$$

In other words, the probability that s_1 satisfies h is not affected by any information whether, or not, s_2 satisfies h. Note that the 2-independence only holds for pairs of solutions.

3.4.2 Linear modular equalities

A recent article [8] presents a propagation algorithm for a system of linear equalities modulo a prime number. This section recalls the definitions, properties and algorithms of this article. For a more in-depth analysis, and more theoretical background we refer the reader to the original article.

Family of constraints **Definition 2** (Linear Modular Equality) Let p be a prime number. Let $\tilde{a} = (a_1, \dots, a_n) \in \left(\mathbb{F}_p\right)^n$ a vector of coefficients, $b \in \mathbb{F}_p$ a constant, and $\tilde{X} = (X_1, \dots, X_n)$ a vector of variables. A *linear modular equality* is an equality

$$\sum_{i=1}^{n} a_i X_i \equiv b \pmod{p}$$

These linear modular equalities can be used as hashing constraints.

Theorem 1 ([8]) *Let $\mathcal{P} = \langle (X_1, \dots, X_n), \mathcal{D}, \mathcal{C} \rangle$ be a CSP. Let p be a prime number greater than the range of the domains (i.e. $p > \max_{1 \le i \le n}(\max \mathcal{D}(X_i) - \min \mathcal{D}(X_i))$). Then the family of hashing constraints*

$$\mathcal{H}_{(\mathrm{mod}\ p)} = \left\{ \sum_{i=1}^{n} a_i X_i \equiv b \pmod{p} \mid (a_1, \dots, a_n) \in \left(\mathbb{F}_p\right)^n, b \in \mathbb{F}_p \right\}$$

is 2-independent.

To use these hashing constraints, one has to be able to pick randomly in $\mathcal{H}_{(\mathrm{mod}\ p)}$. This can be achieved by randomly picking the coefficients a_i and the constant b in $\mathbb{F}_p$.

There are often multiple hashing constraints added to a problem. When using linear modular equalities, all the constraints can be merged into a single system of modular equalities. We now present the filtering algorithm for such a system.

Filtering algorithm The algorithm used to propagate inconsistent values is based on the Gauss-Jordan elimination. $\mathbb{F}_p$ is a field when p is a prime number, hence it is possible to apply the Gauss-Jordan elimination on a system of equations modulo a prime number.

The first step of the algorithm is to set the system in row reduced echelon form using the Gauss-Jordan elimination. This step splits the variables into two sets: the parametric variables and the non-parametric variables. The property of this form is that when all the parametric variables are instantiated, then only one value is allowed for the non-parametric variables.

No propagation is done at the beginning of the search, when only few variables are instantiated. Propagation starts when the size of the Cartesian product of the domain of the parametric variables is less than 1000 (i.e. when $\left|\prod_{X \in \mathcal{X}} \mathcal{D}(X)\right| \le 1000$). The threshold 1000 was chosen experimentally by the authors as a good tradeoff between having propagation too late (if a small threshold was chosen) and having too big of an enumeration of values (if a big threshold was chosen). When all the possible values for the parametric variables are enumerated, it is possible to compute the associated values for the non-parametric variables. This enumeration is exactly the set of instantiations allowed by the constraints (at this particular step of the search with many variables already instantiated). These instantiations serve as a support for the application of a filtering algorithm such as table constraints.

Example We show below how the propagator works on an example. We consider a problem with four variables X_1, X_2, X_3, and X_4 with domains $\mathcal{D}(X_1) = \mathcal{D}(X_3) = \{0, \dots, 4\}, \mathcal{D}(X_2) = \{0, 1, 2\}$, and $\mathcal{D}(X_4) = \{0, 1\}$. We consider a randomly generated system of linear modular equality constraints. To generate this system, the coefficients and the constant are randomly picked in $\mathbb{F}_5 = \{0, \dots, 4\}$. This yields a system such as the following:

$$\begin{cases} 3X_1 & +2X_2 & +3X_3 & +1X_4 & \equiv 4 \pmod 5 \\ 4X_1 & +1X_2 & +1X_3 & +0X_4 & \equiv 0 \pmod 5 \end{cases}$$

- The first step is to apply Gauss-Jordan elimination. The exact operations on the lines are $L_1 \leftarrow 2L_1; L_2 \leftarrow L_2 - 4L_1; L_2 \leftarrow 3L_2; L_1 \leftarrow L_1 - L_2$. This leads to the following system of equations:

$$\begin{cases} 1X_1 & +4X_2 & +0X_3 & +1X_4 & \equiv 0 \pmod 5 \\ 0X_1 & +0X_2 & +1X_3 & +1X_4 & \equiv 3 \pmod 5 \end{cases}$$

- We can now identify the parametric and the non-parametric variables. The non-parametric variables are X_1 and X_3 (used in the pivot operation of the Gauss-Jordan elimination). The parametric variables are X_2 and X_4. We can rewrite the system to clarify this:

$$\begin{cases} X_1 & \equiv 0 & -4X_2 & -1X_4 \pmod 5 \\ X_3 & \equiv 3 & -0X_2 & -1X_4 \pmod 5 \end{cases}$$

- Now, by definition of parametric variables, fixing values for X_2 and X_4 will fix the values of X_1 and X_3. By enumerating the domains of X_2 and X_4 we can enumerate all the possible instantiations satisfying the constraint:

$$\begin{aligned}(X_1, X_2, X_3, X_4) \in \{ & (0,0,3,0), \\ & (4,0,2,1), \\ & (1,1,3,0), \\ & (0,1,2,1), \\ & (2,2,3,0), \\ & (1,2,2,1)\}\end{aligned}$$

- The constraint is transformed into a table constraint using these instantiations.

In this example, the number of enumerated tuples was less than 1000, but in practice it will not be the case at the beginning of the search. In this case, the propagator waits until this threshold is met (either other propagators or decisions during the search will reduce the domain of some parametric variables).

3.5 Chi squared test

Evaluating the randomness of a system is a difficult task. Indeed, 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 is a classical method to compare the result of a random experiment to an expected probability distribution. It comes from a convergence result of the χ^2 law, stated in [23] 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 $Y_1, \dots, Y_n$

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 2 ([23]) *When n tends to infinity, the cumulative distribution 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 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. Conversely, if the p-value stays close to one, we can confidently consider that the experimental values follow the same law as Y. Here, we are interested in the uniform distribution, i.e. $\forall k \in \{1, \dots, d\}, p_k = 1/d$.

4 New sampling approach

We present here a new approach to sample solutions in a CSP. This approach is two-fold: first we present a way to generate random tables constraints, a key ingredient for the method, and then, an algorithm to sample solutions using these generated constraints.

4.1 Random table constraints

The algorithm to generate random table constraints is presented in Algorithm 1. We suppose that the following functions are available:

- 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 $rel(C) = T$.

Algorithm 1 RANDOMTABLE(P, v, p)

```
Require: A CSP $\mathcal{P} = \langle\{X_1, \ldots, X_n\}, \mathcal{D}, \mathcal{C}\rangle$, $v > 0$, $0 < p < 1$
1: $T \leftarrow \{\}$
2: $i_1, \ldots, i_v \leftarrow$ GETINDICES($\mathcal{P}, v$)
3: for $(x_{i_1}, \ldots, x_{i_v}) \in \prod_{k=1}^{v} \mathcal{D}(X_{i_k})$ do
4:     if RANDOM() $< p$ then
5:         $T.add((x_{i_1}, \ldots, x_{i_v}))$
6:     end if
7: end for
8: return TABLE($(X_{i_1}, \ldots, X_{i_v}), T$)
```

In addition to the CSP $\mathcal{P}$, the algorithm has two parameters: v the number of variables in the table, and p the probability to add a tuple to 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 3 *Let $\mathcal{P}$ be a CSP, and T a table constraint randomly generated with probability p. Then*

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

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

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

□

The purpose of Theorem 3 is the following: by adding table constraints, we decrease the size of the solution set by a factor p on average. As p is a parameter of the algorithm, we can control how fast the reduction is operated. A low value of p has a higher chance of making the problem inconsistent, but a higher value of p reduce less the solution space.

4.2 Sampling algorithm

The sampling algorithm is presented in Algorithm 2. First, the auxiliary functions used in this 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.

The sampling algorithm can be set using three values in addition to the CSP $\mathcal{P}$. 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 (this is the purpose of the test line 8). The algorithm stops when there are less than κ solutions. Finally, a solution is randomly chosen from all the remaining solutions, and returned.

The solutions are returned one by one by our approach, similarly to UniGen and for the same reasons: once a solution has been picked, the tables used to find this solution cannot be kept to pick another one, because this would create dependencies. Thus, to generate multiple *independent* solutions, the algorithm is run from scratch multiple times. Moreover, there is no guarantee on the size of the final set, except the one ensured by line 5,

Algorithm 2 TableSampling(P, κ, v, p)

```
Require: A CSP $\mathcal{P}, \kappa \geq 2, v > 0, 0 < p < 1$
 1: $S \leftarrow$ FindSolutions($\mathcal{P}, \kappa$)
 2: if $|S| = 0$ then
 3:     return "No solution"
 4: end if
 5: while $|S| = 0 \lor |S| = \kappa$ do
 6:     $T \leftarrow$ RandomTable($\mathcal{P}, v, p$)
 7:     $S \leftarrow$ FindSolutions($\mathcal{P} \land T, \kappa$)
 8:     if $|S| \neq 0$ then
 9:         $\mathcal{P} \leftarrow \mathcal{P} \land T$
10:     end if
11: end while
12: return RandomElement($S$)
```

$0 < |S| < \kappa$. In other words, the number of solutions in the final set cannot be fixed. If a user does not mind the bias explained above, it is very easy to directly return the final set, and re-run the algorithm until the desired number of solutions is found.

4.3 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.

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 2).

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 1 *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 $i_2, \dots, i_v$ other indices such that $|\mathcal{D}(X_{i_k})| \neq 1, 2 \leq k \leq v$. Let us define the table

$$T_0 = \textsc{Table}\big((X_{i_1}, \dots, X_{i_v}), \big\{(\sigma_1(X_{i_1}), \dots, \sigma_1(X_{i_v}))\big\}\big)$$

Then $\sigma_1 \in Sols(\mathcal{P} \wedge T_0)$ so $Sols(\mathcal{P} \wedge T_0) \neq \emptyset$, 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 $Sols(\mathcal{P} \wedge T_0) \subseteq Sols(\mathcal{P})$.

In the end, we have: $Sols(\mathcal{P} \wedge T_0) \neq \emptyset, Sols(\mathcal{P} \wedge T_0) \subseteq Sols(\mathcal{P})$ and $Sols(\mathcal{P} \wedge T_0) \neq Sols(\mathcal{P})$, thus $Sols(\mathcal{P} \wedge T_0) \subset Sols(\mathcal{P})$, hence $0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})$□.

Lemma 2 *There exists a constant $\rho > 0$, depending only on the initial problem, such that, for T a randomly chosen table constraint with v variables:*

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

Proof We know from Lemma 1 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 $\textsc{RamdomTable}(v, p)$ to pick exactly T_0 (up to ordering of the scope of the constraints). Let T be a random table returned by $\textsc{RamdomTable}(v, p)$. We want to bound

$$\begin{aligned}\mathbb{P}(T = T_0) &= \mathbb{P}\big(scp(T) = scp(T_0) \wedge rel(T) = rel(T_0)\big)\\ &= \mathbb{P}\big(scp(T) = scp(T_0)\big) \cdot \mathbb{P}\big(rel(T) = rel(T_0) \mid scp(T) = scp(T_0)\big)\end{aligned}$$

There are $\binom{n}{v}$ ways of choosing the v variables appearing in the table (the ordering does not matter), so $\mathbb{P}\big(scp(T) = scp(T_0)\big) = 1/\binom{n}{v}$. 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 $\mathbb{P}\big(rel(T) = rel(T_0) \mid scp(T) = scp(T_0)\big) \geq p^k(1-p)^{d^v-k} \geq min(p, 1-p)^{d^k}$. 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 4 *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 of making the problem inconsistent or not reducing the number of solutions is less than $1 - \rho$, as stated in Lemma 2. 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.□

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 1. The solving time will be studied in practice in Section 8 .

4.4 Dichotomic table addition

From early experiments, we remarked a behaviour of the algorithm that led us to create a variant. Indeed, there is little chance that the first added tables will make the problem inconsistent. On the contrary, after many iterations, several tables have been added, and it becomes very fast to search for the κ solutions (or proving inconsistency). This is due to the fact that all the tables added previously really restrict the search space and are quickly propagated.

It is possible to modify of the algorithm by increasing the number of tables added at each step. It will reduce the number of iterations at the beginning of the algorithm. At the end, it will increase the probability of having an inconsistent problem, but as we saw, it is very fast to prove inconsistency in the last iterations. There is a tradeoff between the number of steps of the algorithm and the number of inconsistent problem created. Depending on the problem this variant may be faster than the baseline algorithm, or not.

Algorithm 3 DichotomicTableAddition(P, $nbTables$, κ, v, p)

```
Require: A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, $nbTables > 0, \kappa \geq 2, v > 0, 0 < p < 1$
 1: $\mathcal{T} \leftarrow$ array of size $nbTables$
 2: for $i = 0$ to $nbTables - 1$ do
 3:     $\mathcal{T}[i] \leftarrow$ RandomTable($\mathcal{P}, v, p$)
 4: end for
 5: $S \leftarrow$ FindSolutions($\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa$)
 6: while $|S| = 0 \wedge |\mathcal{T}| > 0$ do
 7:     $\mathcal{T} \leftarrow \mathcal{T}[0 : |\mathcal{T}|/2[$
 8:     $S \leftarrow$ FindSolutions($\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa$)
 9: end while
10: return $\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, |\mathcal{T}|$
```

The exact 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 4 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.

This variant of the algorithm has comparable running times with the baseline algorithm (as seen in section 8.1). On some instances it performs better, but on others it performs worse. A user may try both variants on some instances before running the full sampling, in order to choose the best variant for her/his application.

5 Discussion

In this section, we discuss the algorithmic choices we have made in Algorithm 4.2, compare more in depth with Meel's approach, and extend our approach to other hashing constraints.

5.1 Quality of the division by the tables

In the proof of Theorem 3 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

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

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 Meel's approaches [5] 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.

5.2 Comparison with ApproxMC

In his thesis [5], Kuldeep Singh Meel presented an algorithm to count (ApproxMC), and then to sample solutions of SAT formulas (UniGen), based on the addition of XOR constraints to the problem. Their counting algorithm ApproxMC will add multiple XOR constraints to the SAT formula until there is less than a given number of solutions, and then will extrapolate the total number of solutions. Doing this function multiple times gives a result probably approximately correct (PAC), i.e. if given two parameters $0 < \epsilon$ and $0 < \delta < 1$, c is the value returned by the algorithm with parameters ϵ and δ on the formula $\mathcal{F}$, then

$$\mathbb{P}(|Sols(\mathcal{F})|/(1+\epsilon) \leq c \leq |Sols(\mathcal{F})|(1+\epsilon) \geq 1 - \delta$$

He then uses this counter to get an almost uniform sampler (the probability of sampling is close to the uniform by a factor ϵ, with ϵ that can be chosen).

Our approach is inspired from ApproxMC but differs the fact that we traded the proven uniformity to get a faster algorithm. The constraint used to reduce the solution space is not a XOR constraint (or its extension to CP, a linear modular equality constraint), but a table constraint. The table constraints with few variables will allow for a propagation closer to the root of the search tree, where the linear modular equalities will only propagate close to the bottom of the search tree. The 2-uniformity of the linear modular equalities is also their downfall: it ensures that the solutions satisfying the constraint will be well distributed in the whole solution space. Hence it is not possible to use this constraint to propagate efficiently and cut big sub-spaces of the search space.

The algorithm to get a sample has also been simplified to the strictly necessary. ApproxMC has to run multiple times the algorithm in order to get a proven model counter, and then using this model counter to sample a solution. Our algorithm is applied only once and will always return a solution (where UniGen may not return a solution).

5.3 Usage with different constraints

Algorithms 2 and 3 have been presented using the random tables, but actually they can be used with any constraint that divides the space. Indeed, we can replace the line 6 in Algorithm 2 (or line 3 in Algorithm 3) by the creation of any constraint that we want. In our experiments we also tested this sampling algorithm with randomly generated linear modular equalities (with coefficients randomly picked). This set of hashing constraints ensures the strong property of 2-independence of the presence of solutions. We call this algorithm (our approach using linear modular equalities) LinModEq and evaluate the quality of the randomness and the running time in the experiments. We expect to have a better sampling using these constraints, and eventually a uniform sampling.

5.4 Influence of the parameters

Three parameters have to be chosen to run the algorithm. We discuss here 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 7.2.

6 Experimental methodology

This section presents the methodology used to conduct the experiments. We first present the details of the implementation, and then we present the benchmarks, their characteristics, and the reason for using each one of them. The approach is independent from the constraints of the problem, so we were able to apply it on different problems without being limited by the presence or absence of a constraint.

6.1 Implementation

The code is available online[1], along with all the scripts to generate the figures presented in this article, and the benchmarks used.

6.1.1 TableSampling

The implementation has been done in Java 11 using the constraint solver `choco-solver` version 4.10.6 [7]. 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` [24] and `lastConflict` [25]).

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. This small improvement is evaluated in section 8.1.

1 https://github.com/MathieuVavrille/tableSampling

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

Management of random numbers The random number generator we use 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 [26] 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 [27]).

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.

6.1.2 LinModEq

The LinModEq approach is the Algorithm 2 where we replaced the random table constraints with random linear modular equalities. For the implementation of the propagator of the linear modular equality system, we reused the implementation by [8] that is already in Java. Their implementation is based on the `minicp` solver, modified to use belief propagation.[2] We adapted their propagator for the linear modular equalities to make it work in the `choco-solver` framework.

To ensure that our implementation is not flawed compared to the original implementation, we tested it on the same benchmark as the one used in [8]. We observe the same behaviour, even if there are differences due to the underlying CP solver. To enumerate all the solutions (without any linear modular equality), `choco-solver` is an order of magnitude faster than `minicp`. Indeed, according the official website, `minicp` "is not focused on efficiency but rather on readability to convey the concepts as clearly as possible".

Despite the gap in global performances, we observe the same behaviour as [8] when adding linear modular equalities (and by using the propagator designed for a system of linear modular equalities), i.e. that increasing the number of equalities reduces the running time.

6.2 Preliminary benchmark

We first use a small benchmark to make extensive experiments, in order to calibrate the setting of the algorithm. Its purpose is to evaluate the randomness of our approach and the impact of the parameters to extract a generic set of parameters to use as a baseline. The evaluation of the random process has to be done on small problems because of the computation cost of the χ^2 test.

This benchmark consists of three problems, detailed below. The result of the evaluation on this preliminary benchmark are presented in section 7. The running time comparison to RandomVarDom and LinModEq will be done with harder benchmarks.

6.2.1 Problems

The following problems have been chosen for their reasonable number of solutions (to apply the χ^2 test), and for their relevancy. The computation time is small enough to perform extensive experiments, and we will use the results to calibrate the parameters of our method.

[2] https://github.com/PesantGilles/MiniCPBP

***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 classical model with N variables with domain $[1, N]$, and inequality binary constraints (the same model as the one used in [8]). We use the 9-queens instance that has 352 solutions.

On Call Rostering This problem models the system of duty, notably used by healthcare workers. This instance is available in the MiniZinc benchmarks[3] 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 favour some workers at the expenses of others.

Feature Model This is a problem of software management, helping to decide on the order of implementation of software features. The instance is specified in the MiniZinc format in [28] using the data in [29]. Again, it is an optimisation problem (maximisation), the optimal value is 20,222. We add the constraint $obj \geq 17,738$ to make it a satisfaction problem with 95 solutions.

6.2.2 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 $i \in \{1, \dots, nbSols\}$ is counted. We compute the value of the variable

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

and then the p-value of the test[4] (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.

To perform 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.

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

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

As an example to understand the evolution of the p-value, let us consider a problem with two solutions, and suppose that our sampling method is biased towards the first solution, and returns it with probability 0.6. After doing 10 samples, the solution count may be (6, 4) (i.e. 6 times the first solution, and 4 times the second). Knowing we only drew 10 samples, no one can tell with certainty that the distribution is not uniform. This leads to a high p-value (here the p-value is 0.527). After doing 100 samples, the solution count may be (62, 38). At this point, it is getting unlikely that a uniform distribution can generate such a distribution, but still not impossible. The p-value is getting closer to 0 (here the p-value is 0.0164). After doing 1000 samples, the solution count may be (587, 413). Now it is almost impossible to get this solution count with a uniform distribution. The p-value will be extremely close to 0 (here the p-value is $3.7 \cdot 10^{-8}$).

What will be plotted is the evolution of the p-value depending on the number of samples. The approaches that are not uniform have a p-value that tends to 0, and the approaches giving a uniform distribution will have a p-value that tends to 1. The rate at which the p-value tends to 0 also gives information about the distance to the uniform distribution. In the previous example, if the distribution was (0.8, 0.2), then after sampling 100 solutions the solution count may be (81, 19), hence the p-value would already be extremely close to 0 when comparing to the uniform distribution (in facts the p-value is $5.6 \cdot 10^{-10}$).

Thus, the p-value allows to rank non-uniform approaches. An approach whose p-value tends slower to 0 is “more uniform” (but still not exactly uniform) than an approach whose p-value tends quickly to 0.

6.3 MiniZinc challenge benchmark

To evaluate the performances of our approach, we also perform experiments on a second, bigger, benchmark, with harder problems from the MiniZinc challenge[5]. The MiniZinc challenge is an annual solver competition on a large, diverse benchmark. To evaluate our approach, we created a benchmark based on the problems of the challenge from 2016 to 2021.

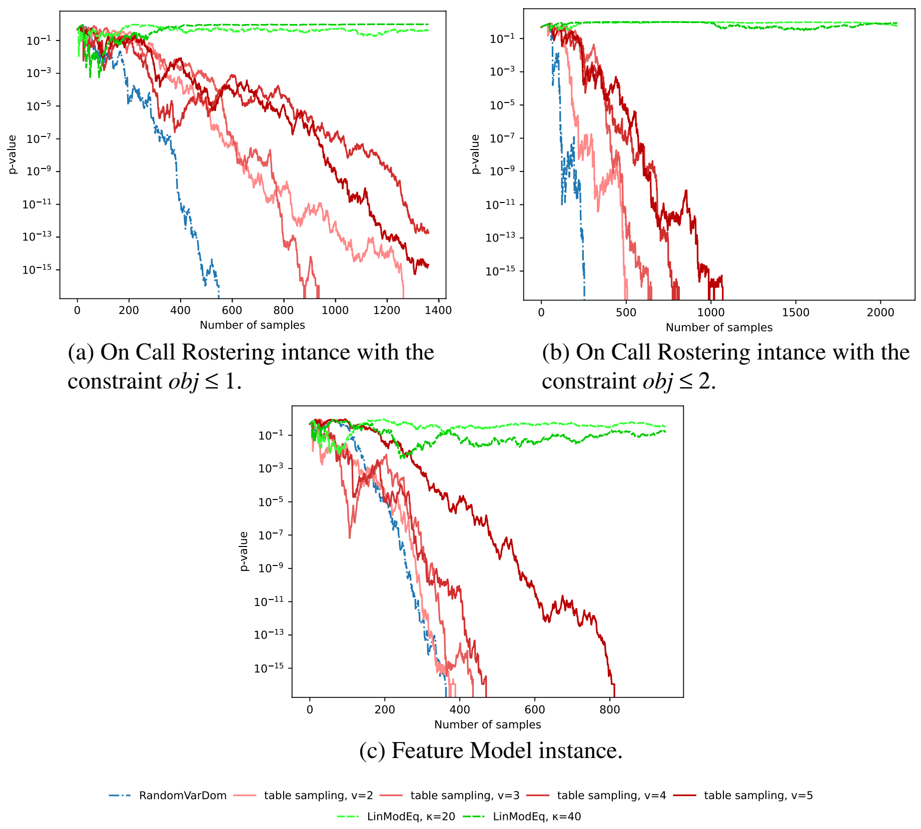
The challenge contains very hard instances, and sets a time limit of 20 minutes. We chose to keep this timeout of 20 minutes for our computations. On many optimisation problems, no solvers were able to prove that the solutions found (if any) were optimal. We restricted the benchmark to problems where `choco-solver` was able to find the optimal value and prove it. We transformed all the optimisation problems into satisfaction problems by fixing the objective function to its optimal value.

We recall that if we choose a parameter $\kappa > |Sols(\mathcal{P})|$, the TableSampling approach boils down to enumerating all the solutions and returning one at random. This is not interesting at all to test our approach as it will never add any constraint (either table constraint or linear modular equality). In the experiments, we chose $\kappa = 16$ (see section 7.2), hence we restricted the benchmark to problems where we were able to enumerate more than 15 solutions before the timeout of 20 minutes.

The approaches are random, hence the running time is impacted by this randomness. To limit these random factors, we run each approach 10 times on every instance and average the total time. If an approach times out on one of the runs, we record it as a timeout of the approach on this instance (we are not able to average the time anymore).

To sum up, this is how the benchmark has been built:

[5] https://www.minizinc.org/challenge.html



(a) On Call Rostering intance with the constraint $obj \leq 1$.

(b) On Call Rostering intance with the constraint $obj \leq 2$.

(c) Feature Model instance.

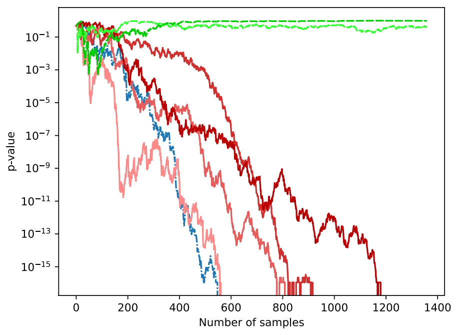
Fig. 1 Evolution of the p-value on different problems, with $\kappa = 16, p = 1/8$, and different values for v

- we start with all the instances from the MiniZinc challenge from 2016 to 2021;
- we remove the optimisation problems where we were not able to find the optimal value in less than 20 minutes. We then transform them in satisfaction problems by fixing the objective to the optimal value;
- we remove all the problems that have less than 16 solutions (enumerated in less than 20 minutes);
- we run each approach 10 times on each instance and collect the average running time.

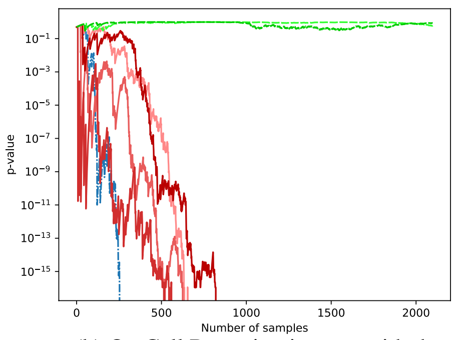
The final benchmark contains 82 instances.

7 Preliminary experiments

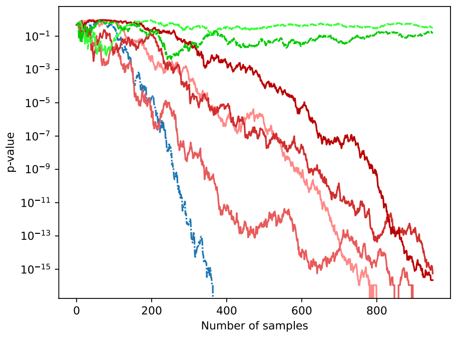
This section presents the results of the preliminary experiments done to evaluate the impact of the parameters. RandomVarDom, TableSampling and LinModEq have been run to sample the problems multiple times. Different sets of parameters (for κ, v, and p) have been



(a) On Call Rostering intance with the constraint $obj \leq 1$.



(b) On Call Rostering intance with the constraint $obj \leq 2$.



(c) Feature Model instance.

Fig. 2 Evolution of the p-value on different problems with $v = 5$, $p = 1/2$, and different values for κ

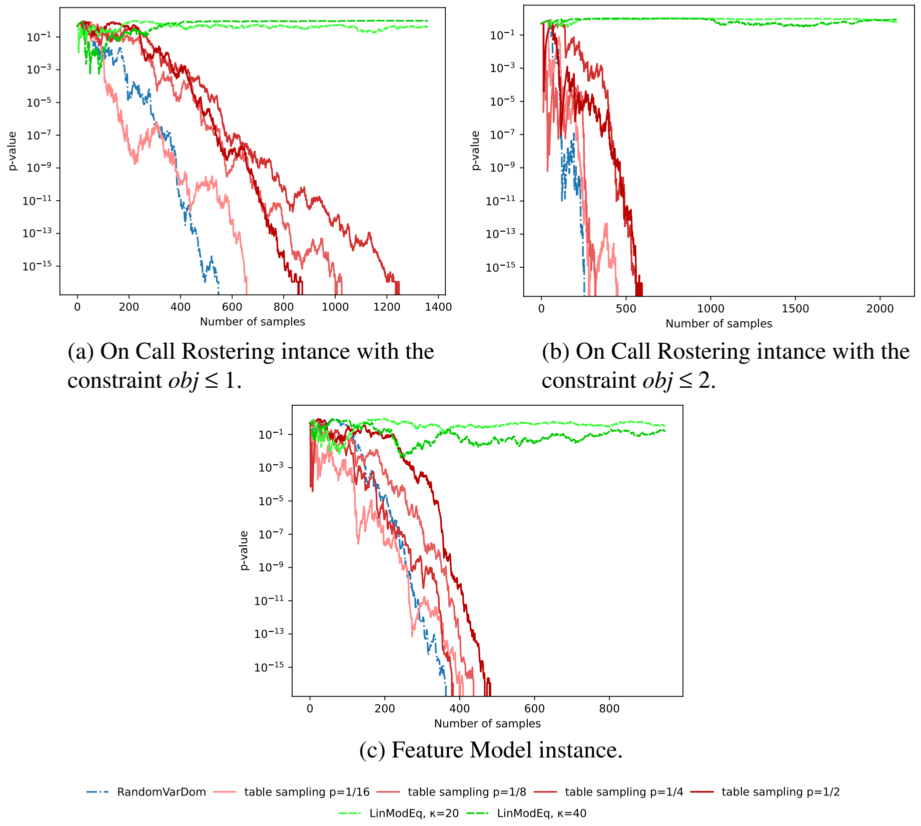
used for TableSampling and LinModEq. Figures 1, 2, 3 and 5 show some results that highlight the behaviour of the approaches.

Remark 1 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.

7.1 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.

The graphs for different problems are plotted in figures 1, 2, and 3. On every graph, there is one line (in dash-dotted blue) for the p-value of RandomVarDom, multiple plain lines for TableSampling with different parameters (in shades of red), and multiple dashed lines for LinModEq with different values for κ (in shades of green). A p-value that tends



(a) On Call Rostering intance with the constraint $obj \leq 1$.

(b) On Call Rostering intance with the constraint $obj \leq 2$.

(c) Feature Model instance.

Fig. 3 Evolution of the p-value on different problems, with $\kappa = 8$, $v = 3$, and different values for p

to 0 means that the sampling is not uniform, and a p-value that tends to 1 means that the sampling is uniform.

The first remarkable fact is that LinModEq, i.e. our approach using the linear modular equalities instead of the table constraints, is uniform. For the two values of κ that are plotted, the p-value always tend to 1. This was expected as discussed in section 5.3.

The second remark, also expected, is that RandomVarDom is never uniform. We saw on a very simple example that this search strategy does not sample uniformly, and on these problems with more structure, it is even more apparent.

The behaviour of our approach is in the middle of these two remarks. On most of the problems, the sampling is not uniform. However, it is always closer to the uniform distribution than RandomVarDom. This can be seen by the fact that the p-value tends to 0 slower for TableSampling (no matter what the set of parameters used) than RandomVarDom. On the 9-queens problem (in figure 4) the p-value even tends to 1, which means that our approach samples uniformly solutions, even if RandomVarDom is still not uniform. 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.

These experiments show that our approach is really beneficial for the quality of the randomness compared to the basic random strategy. By using table constraints as hashing

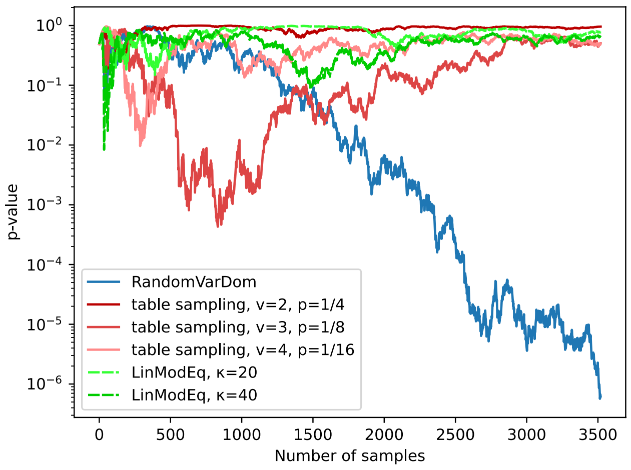


Fig. 4 Evolution of the p-value on the 9-queens problem with $\kappa = 8$ and different parameters for v and p

constraints, one can improve the randomness, however the sampling may not be uniform. By using linear modular equalities, it is possible to sample uniformly from the solution set, the downside being the increased running time to do so, as linear modular equalities are harder to propagate, and they propagate slower than table constraints.

7.2 Impact of the parameters

Our approach has the advantage of being parametric. It allows the user to choose the parameters most suited for his/her application. But this requires a good knowledge of the impact of the parameters on the running time and the quality of randomness. We verify here experimentally the hypotheses done in section 5.4.

We ran experiments by changing the parameters, and evaluating the evolution of the p-value and the running time with these different sets of parameters. We use the previous evaluation of the p-value, and for the running time figure 5 shows heat maps of running times. In these heat maps, one parameter is fixed, and we vary the two others. The darker the colour, the longer it took to sample one solution.

7.2.1 Impact of the number of variables

We first vary the number of variables used in the generated tables. Fig. 1 shows the evolution of the p-value on the instances with parameters $\kappa = 16$ and $p = 1/8$. We remark that increasing the number of variables in the table makes the p-value tend to zero more slowly, 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 5.1).

The evolution of the running time can be seen in the heat maps Fig. 5b, c. We clearly see that increasing the number of variables increases the running time. This behaviour is easily explained, because increasing the number of variables exponentially increases the number of tuples in the tables.

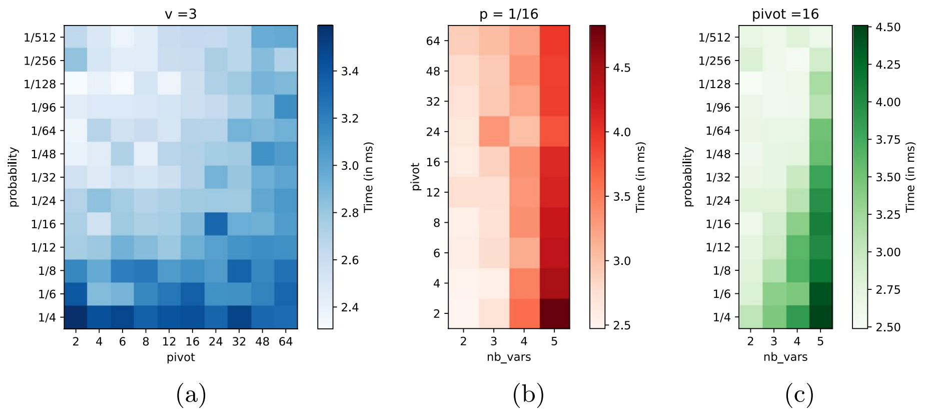


Fig. 5 Heat maps of the time to sample one solution, by fixing different parameters, on the On Call Rostering instance with the constraint

7.2.2 Impact of the pivot

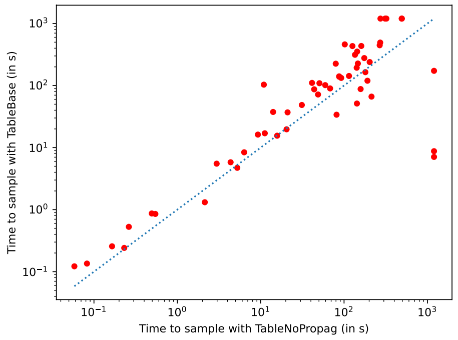
On Fig. 2, we vary the pivot on the different instances, with the parameters $v = 5$ and $p = 1/2$. Here we observe that increasing the pivot improves the randomness. Indeed, when the pivot is high, at each step a lot of solutions are enumerated, and in the end a random solution will be picked among 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 then simply performs an enumeration of all the solutions, and returns a (perfectly uniform) random solution.

The evolution of the running time can be seen in the heat maps Fig. 5a, b. From these results there is no clear impact of the pivot on the running time.

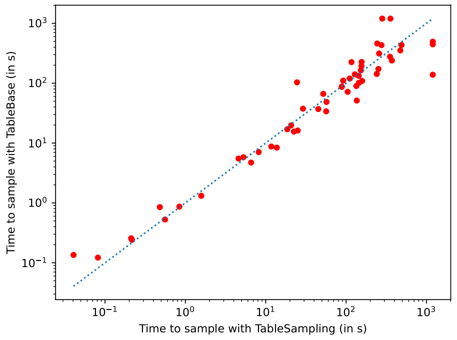
7.2.3 Impact of the probability

Figure 3 shows the p-value for different values of the table probability p, and with the parameters $\kappa = 8$ and $v = 3$. There is no clear influence of the probability to add tuples in the tables on the quality of the randomness.

However the probability has an impact on the running time. We see on heat maps Fig. 5a, c that decreasing the probability decreases the running time. Having a small probability allows to have smaller tables, hence the solution space is reduced faster. Thus, the algorithm converges faster to a small set of solution (smaller than κ). There is no point in reducing the probability too much, because at some point the average table will be empty. One has to find a trade-off for the probability p, and it depends on the average size of the variables domains. For variables with big domains, the probability has to be small in order to keep the tables tractable. On the other hand, tables holding only on Boolean variables need a higher probability, to avoid empty tables.



(a) Scatter plot of the time to sample one solution for the base algorithm versus the same algorithm without the propagation step. Each dot is an instance.



(b) Scatter plot of the time to sample one solution for the base algorithm versus the dichotomic variation. Each dot is an instance.

Fig. 6 Plots of the running time of each variation of the algorithm

7.2.4 Base set of parameters

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 also depend on the application: instances with big domains may require smaller v to avoid too big tables (for example, $v = 4$ for domains of size 100 means enumerating 10^8 tuples). From our experiments, we suggest the default parameter values: $\kappa = 16$ and $p = 1/16$. We choose to have $p = 1/\kappa$, because it reduces the chances of having inconsistencies after adding a table: we know the problem has more than κ solutions. Thus, after adding a table with probability $1/\kappa$, there will be more than one solution on average.

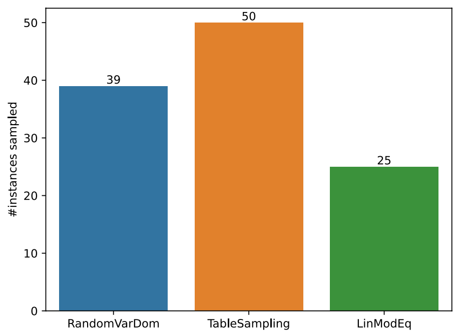
8 MiniZinc challenge experiments

We present here the running time results on the MiniZinc challenge benchmark, as presented section 6.3. The raw results are given in Appendix 1. The experiments have been done using the parameters $\kappa = 16, v = 2$, and $p = 16$.

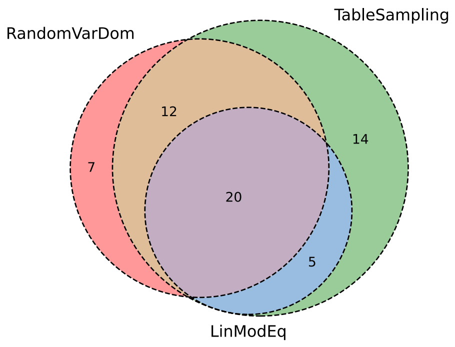
8.1 Difference between the variants

We first evaluate the impact of the variants of our approach. We compare the computation time with/without the dichotomic variation, and with/without the early propagation step (introduced in section 6.1.1).

Figure 6 shows the scatter plots of the sampling time for each variation. We consider that the baseline of the algorithm (called TableBase) is its version with propagation, but without the dichotomic variation. We compare this baseline algorithm with TableNoPropag (same algorithm without the propagation step) and TableSampling (same algorithm with the dichotomic variation).



(a) Histogram of the number of sampled instances per algorithm.



(b) Venn diagram of the number of sampled problems. For example the blue subset contains 5 instances solved only by TABLESAMPLING and LINMODEQ, but not RANDOMVARDOM.

Fig. 7 Plots of the number of instances sampled by each approach

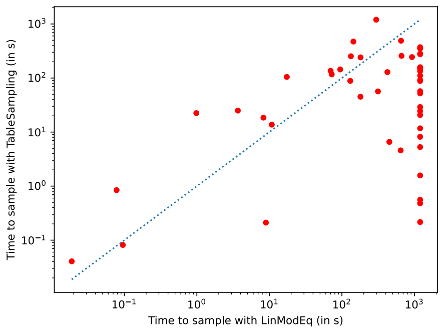
We see from Fig. 6a that the propagation improves the running time (more points are above the $x = y$ line than under).

The impact of the dichotomic variant is less clear. According to Fig. 6b, both algorithms TableBase and TableSampling seem to be equivalent in terms of solving time on the whole benchmark set. However, we observe that some instances are solved significantly faster by one approach, or by the other. This choice is thus left to the user.

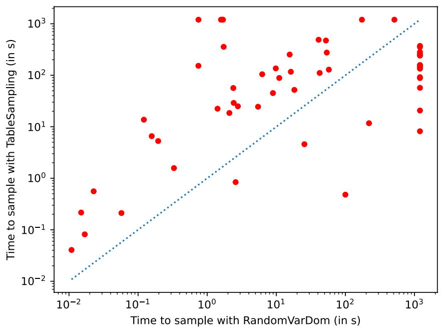
8.2 Timeouts

The first interesting measure of the running time of an approach is the number of instances solved. Here, we look at the number of instances on which the approaches were able to sample in less than 20 minutes, out of the 82 total instances. Figure 7 shows two plots about the number of instances sampled by each approach. Figure 7a simply shows the histogram of the number of sampled instances. Our approach, TableSampling was able to sample the most instances with 50 instances sampled, RandomVarDom was able to sample 39 instances, and LinModEq is behind with only 25 sampled instances. Recall that the main goal of our approach was to ensure a good randomness at a cheap cost, and we experimentally showed TableSampling improves the randomness over RandomVarDom. We actually see that we are able to sample 11 more instances, hence we are also competitive in terms of running time.

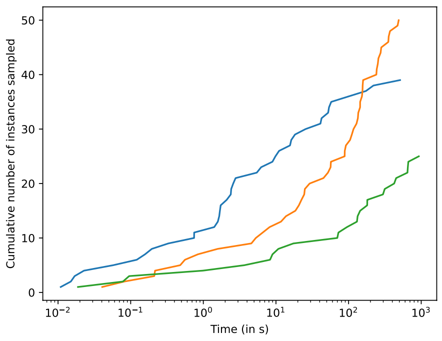
To understand more precisely the differences between the approaches, a Venn diagram of how many instances are solved by each approach is plotted in Fig. 7b. The three circles represent the three approaches, and the different intersections between these circles represent the number of instances sampled by one or multiple approaches. For example, the subset containing the 5 (in blue) represents that exactly 5 instances were sampled by only TableSampling and LinModEq. There are different takeaways from this Venn diagram:



(a) Scatter plot of the time to sample one solution for LINDOMEQ verusTABLESAMPLING. Each dot is an instance.



(b) Scatter plot of the time to sample one solution for RANDOMVARDOM verus TABLESAMPLING. Each dot is an instance.



(c) Plot of the comulative number of instances sampled in less than a given time.

Fig. 8 Plots of the running time of each approach

- TableSampling can sample all the instances that LinModEq samples (the circle of LinModEq is included in the circle of TableSampling). This was expected because the linear modular equalities constraints used in LinModEq are more expensive to propagate compared to table constraints. This is the price to pay for a better randomness.
- 19 instances were sampled by TableSampling but not RandomVarDom, where only 7 instances were sampled by RandomVarDom but not TableSampling. We believe this is due to the possibility, for TableSampling, to keep good search heuristics.

8.3 Running Time

The timeouts give a first insight on the comparison of the approaches, but we can also look at the running time on the instances that were sampled without timeout. Figures 8a, b are scatter plots of the sampling time for each instance. On both figures TableSampling is on the y-axis, so if an instance is above the diagonal, it means that TableSampling was slower to sample it compared to the other approach. Figure 8c is another representation of the running times where for each approach we plot the number of instances sampled faster than a given time (the x-axis).

On the comparison between LinModEq and TableSampling, we first see that most of the instances were sampled faster by TableSampling than LinModEq. This was expected, because the base algorithm is the same but linear modular equalities are harder to propagate, and propagate less (and later) than table constraints.

The comparison between RandomVarDom and TableSampling is very interesting. On the instances that were sampled by both approaches, RandomVarDom is most of the time faster than TableSampling. However, as said above, a lot of instances are only sampled by TableSampling (the 19 instances where RandomVarDom took 1200s to sample, i.e. timed out), and these instances appear to be among the harder instances (TableSampling takes from one second to few minutes to sample them). This behaviour can be explained by the fact that RandomVarDom is a search strategy, meaning it cannot coexist with other, efficient heuristics. Using a random search strategy on hard instances is likely to make bad branching decisions, leading to a lot of time spent in unsatisfiable subtrees of the search tree. On the contrary, our approach does not require a modification of the heuristics, should there exist efficient ones. Thus, TableSampling can benefit from all the dedicated (or black box) search strategies designed in solvers.

On the other hand, the 7 instances solved by RandomVarDom and not TableSampling give a limit of our algorithm. These instances have variables with big domains. For example in the `zephyrus` instances, some variables have a domain of size 4097. With $v = 2$ in our experiments, this may lead to an enumeration of more than 8 million tuples. These big domains are a limit of our approach because the creation and the propagation of the random tables will be costly. In this situation, the design of other algorithms to generate random tables could be interesting. For example, it is possible to generate tables with a fixed number of tuples. It would also be possible to first use the variables with a small domain, or to have a v changing during the resolution, depending on the size of the domains.

9 Conclusion

We presented an algorithm using table constraints to randomly sample solutions of a problem. Experiments show that our algorithm offers a reasonably good quality of randomness, while keeping the computation time tractable. The most important feature of our method is that it does not require modifications of the solver settings, nor of the model.

Our approach is lightweight, because it uses the solver as a black box. Besides, the table constraint offer a wide range of possibilities to tweak the solving process, depending on the user's needs. For example, by playing on the probabilities for certain tuples to be selected, one can orient the sampling in certain subspaces, depending on the user's need. This allows us to tackle randomisation with any given distribution, not necessarily uniform. On the same idea, reducing the probability of tuples included in previously found solutions would induce a diversified search.

Exploiting the random reduction of the search space yields other promising ideas. For instance, portfolio algorithms runs several solving processes in parallel, which ideally all search in different subspaces. Feeding the processes with random reduced search spaces would force them to explore different subspaces without any biases.

Raw results of the MiniZinc benchmark

This appendix presents in Table 1 the computation results.

Table 1 First table of raw results

Year	Problem	Instance	RANDOM-VARDOM	TABLESAMPLING		LIN-MODEQ
				Base	Dicho	
2016	rcpsp-wet	j30_27_5-wet	⊥	⊥	⊥	⊥
		j30_44_8-wet	52.3s	352s	472s	144s
	carpet-cutting	mzn_rnd_test.11	⊥	⊥	281s	⊥
	java-auto-gen	binpack_11	⊥	19.7s	20.7s	⊥
		plusexample_6	⊥	313s	257s	664s
	elitserien	handball20	⊥	48.6s	57.1s	⊥
		handball3	⊥	89.9s	134s	⊥
		handball7	⊥	109s	158s	⊥
		handball5	⊥	277s	352s	⊥
	filters	dct_1_3	747ms	⊥	⊥	⊥
		fir_1_3	15.0ms	241ms	216ms	⊥
		fir_1_4	22.8ms	527ms	558ms	⊥
		ar_1_3	2.42s	37.5s	29.1s	⊥
		ewf_1_2	99.9s	851ms	480ms	⊥
	zephyrus	12__6__8__3	1.58s	⊥	⊥	⊥
		12__8__6__3	1.69s	⊥	⊥	⊥
		14__6__8__3	1.73s	⊥	356s	⊥
		14__8__6__3	746ms	164s	152s	⊥
	depot-placement	rat99_6	511s	⊥	⊥	⊥
		rat99_5	8.95s	37.0s	45.1s	180s
		ulysses22_5	42.5s	119s	110s	⊥
		st70_5	53.8s	433s	275s	⊥
	mrcpsp	j30_17_10	330ms	1.32s	1.58s	⊥
		j30_15_5	18.3s	66.3s	52.0s	⊥
2017	community-detection	Sampson.s10.k3	⊥	⊥	⊥	⊥
	steelmillslab	bench_14_1	⊥	⊥	⊥	⊥
		bench_16_10	⊥	⊥	⊥	⊥
	opd	small_bibd_08_28_14	⊥	⊥	⊥	⊥
		small_bibd_06_50_25	⊥	⊥	⊥	⊥
	tc-graph-color	k10_31	⊥	493s	⊥	297s
		k5_05	⊥	101s	144s	94.9s
	routing-flexible	routing_GCM_0022	173s	⊥	⊥	⊥
	groupsplitter	u5g1pref0	1.41s	15.5s	22.4s	983ms
		u12g1pref1	11.0s	87.8s	88.6s	130s
		u12g1pref0	15.6s	172s	252s	133s
2018	soccer-computational	xIGData_22_12_22_5	⊥	⊥	⊥	⊥
	test-scheduling	t30m10r10-5	⊥	⊥	⊥	⊥
		t100m10r3-2	⊥	⊥	⊥	⊥
	racp	j30_26_2_1.0	⊥	⊥	⊥	⊥
	rotating-workforce	Example103	⊥	⊥	⊥	⊥
		Example1479	⊥	⊥	⊥	⊥
	oocsp_racks	oocsp_racks_100_r1_cc	⊥	⊥	⊥	⊥
		oocsp_racks_050_r1	⊥	⊥	⊥	⊥
	elitserien	handball6	⊥	87.0s	88.3s	⊥
		handball18	⊥	110s	91.9s	⊥
		handball15	⊥	132s	144s	⊥
		handball1	⊥	226s	156s	⊥
		handball16	⊥	239s	371s	⊥
	concert-hall-cap	concert-cap.mznc2018.02	2.10s	17.0s	18.5s	8.29s
		concert-cap.mznc2018.148	57.7s	140s	128s	424s
	steiner-tree	es10fst03.stp	2.78s	16.2s	25.0s	3.67s
		es10fst10.stp	6.25s	71.7s	104s	17.4s
	neighbours	neighbours1	16.2s	225s	116s	72.4s

Table 1 (continued)

Year	Problem	Instance	RANDOM-VARDOM	TABLESAMPLING Base	TABLESAMPLING Dicho	LIN-MODEQ
2019	steelmillslab	bench_20_8	⊥	⊥	⊥	⊥
		bench_19_6	⊥	⊥	⊥	⊥
	median-string	p2_10_8-0	⊥	142s	241s	180s
	zephyrus	12__8__6__3	1.66s	⊥	⊥	⊥
		14__6__6__3	157ms	4.72s	6.57s	451s
		12__6__6__3	195ms	5.80s	5.28s	⊥
	stack-cuttingstock	d3	121ms	8.40s	13.7s	10.8s
	groupsplitter	u6g1pref1	2.39s	33.9s	56.5s	314s
		u9g1pref1	9.84s	51.3s	136s	69.9s
	stochastic-vrp	vrp-s4-v2-c3_svrp-v2-c3_det	2.58s	867ms	841ms	78.2ms
2020	soccer-computational	xIGData_22_12_22_5	⊥	⊥	⊥	⊥
	collaborative-construction	46	⊥	⊥	⊥	⊥
	skill-allocation	skill_allocation_mzn_1m_1	⊥	⊥	⊥	⊥
	pentominoes	07	⊥	⊥	⊥	⊥
		05	⊥	⊥	⊥	⊥
		04	⊥	⊥	⊥	⊥
		06	⊥	⊥	⊥	⊥
		02	5.45s	103s	24.5s	⊥
	minimal-decision-sets	breast-cancer_train4	⊥	138s	⊥	⊥
	racp	j30_26_2_1.0	⊥	446s	⊥	⊥
	is	A3PZaPjnUz	⊥	7.07s	8.18s	⊥
		v1HjuSBQMb	219s	8.75s	11.7s	⊥
	p1f-pjs	10	⊥	193s	155s	⊥
	bnn-planner	cellda_y_10s	⊥	460s	244s	926s
2021	opt-cryptoanalysis	r1	10.9ms	135ms	40.6ms	18.8ms
		r2	16.9ms	121ms	81.5ms	95.3ms
		r3	57.4ms	256ms	211ms	8.96s
		r4	25.5s	5.51s	4.58s	646s
	peacable_queens	8	41.0s	434s	489s	654s

Acknowledgements This article is an extended version of a conference paper: *Solution sampling with random table constraints*, Mathieu Vavrille, Charlotte Truchet, Charles Prud'homme, Proceedings of CP 2021, 56:1-56:17. This work was presented at the conference on October 2021: https://cp2021.lirmm.fr/submissions/74. Compared to the conference article, this version has been extended with the implementation of another series of hashing constraints, proposed in [8], inside the framework, more extensive experiments on a wider benchmark, and a more thorough analysis of the results and algorithm behaviour.

Funding Partial financial support was received from the French National Research Agency as part of the DeCrypt project (ANR-18-CE39-0007) for two of the authors (Prud'homme and Truchet).

Declarations

Competing interest Apart from the project mentioned above, the authors have no other competing interests to declare that are relevant to the content of this article.

References

1. Hartert, R., & Schaus, P. (2014). A support-based algorithm for the bi-objective pareto constraint. In C. E. Brodley, & P. Stone (Eds.), *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada* (pp. 2674–2679). AAAI Press.
2. Rossi, F., Venable, K. B., & Walsh, T. (2008). Preferences in constraint satisfaction and optimization. AI Mag., 29, 58–68.
3. Hebrard, E., Hnich, B., O'Sullivan, B., & Walsh, T. (2005). Finding diverse and similar solutions in constraint programming. In *AAAI* (pp. 372–377). volume 5.
4. Hebrard, E. (2006). *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. Ph.D. thesis University of New South Wales.
5. Meel, K. S. (2018). Constrained counting and sampling: bridging the gap between theory and practice. arXiv preprint arXiv:1806.02239, .
6. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.-C., & Schaus, P. (2016). Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming* (pp. 207–223). Springer.
7. Prud'homme, C., Fages, J.-G., & Lorca, X. (2016). *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.
8. Pesant, G., Meel, K. S., & Mohammadalitajrishi, M. (2021). 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.
9. Acher, M., Temple, P., Jézéquel, J.-M., Galindo, J. A., Martinez, J., & Ziadi, T. (2018). Varylatex: Learning paper variants that meet constraints. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems* (pp. 83–88).
10. Gotlieb, A., & Marijan, D. (2017). Using global constraints to automate regression testing. AI Magazine, 38, 73–87.
11. Plazar, Q., Acher, M., Bardin, S., & Gotlieb, A. (2017). Efficient and complete fd-solving for extended array constraints. In IJCAI 2017.
12. Gomes, C. P., van Hoeve, W. J., Sabharwal, A., & Selman, B. (2007). Counting csp solutions using generalized xor constraints. In *AAAI* (pp. 204–209).
13. Plaza, S. M., Markov, I. L., & Bertacco, V. (2008). Random stimulus generation using entropy and xor constraints. In *2008 Design, Automation and Test in Europe* (pp. 664–669). IEEE.
14. Yuan, J., Shultz, K., Pixley, C., Miller, H., & Aziz, A. (1999). Modeling design constraints and biasing in simulation using bdds. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051)* (pp. 584–589). IEEE.
15. Wei, W., Erenrich, J., & Selman, B. (2004). Towards efficient sampling: Exploiting random walk strategies. In *AAAI* (pp. 670–676). volume 4.
16. Kitchen, N., & Kuehlmann, A. (2007). Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design* (pp. 258–265). IEEE.
17. Ermon, S., Gomes, C. P., & Selman, B. (2012). Uniform solution sampling using a constraint solver as an oracle. arXiv preprint arXiv:1210.4861, .
18. Dutra, R., Laeufer, K., Bachrach, J., & Sen, K. (2018). Efficient sampling of sat solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (pp. 549–559). IEEE.
19. Dechter, R., Kask, K., Bin, E., Emek, R. et al. (2002a). Generating random solutions for constraint satisfaction problems. In *AAAI/IAAI* (pp. 15–21).
20. Gogate, V., & Dechter, R. (2006). A new algorithm for sampling csp solutions uniformly at random. In *International Conference on Principles and Practice of Constraint Programming* (pp. 711–715). Springer.
21. Dechter, R., Kask, K., & Mateescu, R. (2002b). Iterative join-graph propagation, .
22. Schreiber, Y. (2010). Value-ordering heuristics: Search performance vs. solution diversity. In *International Conference on Principles and Practice of Constraint Programming* (pp. 429–444). Springer.
23. Pearson, K. (1900). 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, 157–175.
24. Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *ECAI* (p. 146). volume 16.
25. Lecoutre, C., Sais, L., Tabary, S., & Vidal, V. (2006). Last conflict based reasoning. In G. Brewka, S. Coradeschi, A. Perini, & P. Traverso (Eds.), *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 (pp. 133–137). IOS Press volume 141 of *Frontiers in Artificial Intelligence and Applications*.
26. Knuth, D. E. (2014). *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional.
27. Bach, E. (1987). Realistic analysis of some randomized algorithms. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (pp. 453–461).
28. Regnell, B., & Kuchcinski, K. (2011). 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)* (pp. 47–56). IEEE.
29. Ruhe, G., & Saliu, M. O. (2005). The art and science of software release planning. IEEE software, 22, 47–53.