Practically Uniform Solution Sampling in Constraint Programming

Gilles Pesant[1], Claude-Guy Quimper[2], and Hélène Verhaeghe[1]

[1] Polytechnique Montréal, Canada
gilles.pesant@polymtl.ca, helene.verhaeghe@polymtl.ca
[2] Université Laval, Canada
claude-guy.quimper@ift.ulaval.ca

Abstract. The ability to sample solutions of a constrained combinatorial space has important applications in areas such as probabilistic reasoning and hardware/software verification. A highly desirable property of such samples is that they should be drawn uniformly at random, or at least nearly so. For combinatorial spaces expressed as SAT models, approaches based on universal hashing provide probabilistic guarantees about sampling uniformity. In this short paper, we apply that same approach to CP models, for which hashing functions take the form of linear constraints in modular arithmetic. We design an algorithm to generate an appropriate combination of linear modular constraints given a desired sample size. We evaluate empirically the sampling uniformity and runtime efficiency of our approach, showing it to be near-uniform at a fraction of the time needed to draw from the complete set of solutions.

1 Introduction

Many important yet difficult problems can be expressed on a constrained combinatorial space: finding a *satisfying* element (i.e. a solution) in that space and looking for an *optimal* one according to some criterion are perhaps the most common tasks but *counting* how many solutions there are and *sampling* solutions uniformly at random also have important applications. Examples of the latter occurs in probabilistic planning [3], Bayesian inference [10], code verification [4], as well as other applications [5].

Model counters which follow an approach based on universal hashing benefit from probabilistic guarantees about the quality of the approximate count they provide. In the context of SAT models expressed through Boolean variables, such universal hashing takes the form of randomly generated parity (XOR) constraints [2]. In the wider context of finite-domain variables and CP models the latter generalize to randomly generated linear equality constraints in modular arithmetic, as recently investigated by Pesant et al. [9]. Because hashing-based model counting operates by partitioning the solution space into nearly-equal-size cells, it also offers an approach to solution *sampling*, which has been exploited for SAT [7]. Even though in principle sampling the solutions of a CP model could be achieved by first translating the model into SAT, it would be much preferable to do it directly in CP.

Vavrille et al. [11] recently proposed a solution sampler for CP inspired by that idea of splitting the search space into cells and focusing on one such cell — adding random

TABLE constraints to a CP model — but unfortunately without any theoretical guarantee about the quality of the sampling nor admittedly any generally-observed sampling uniformity in practice. In this short paper, we add certain linear modular constraints — sharing the same probabilistic guarantees as XOR constraints for SAT— in order to sample the solutions of a CP model. We show that in practice we achieve almost uniform sampling. As in [11] our approach is independent of the actual CP model being sampled and can be applied as long as the underlying CP solver supports such constraints. Additionally it is very simple to use, without any parameters to tune.

The literature is scarce on the problem of sampling CP models. Gogate and Dechter achieve uniform sampling on a CSP by first expressing it as a factored probability distribution over its solutions, which requires that each constraint be given as a set of allowed tuples [6]. Perez and Régin sample solutions according to a given probability distribution but require that the model be encoded as a single MDD constraint [8]. As previously mentioned Vavrille et al. add randomly generated TABLE constraints to a CP model thereby reducing the search space in a controlled manner prior to sampling [11]. Parameters v and p respectively control the arity of a table and the probability of a tuple being included. In practice v is chosen much smaller than the number of variables in order to keep in check the exponential growth of the tables, thus sacrificing uniformity in sampling.

In the rest of the paper Section 2 describes how a system of linear modular constraints can be used to partition the search space and offer guarantees on the quality of the sampling. We also give an algorithm to generate these constraints. Section 3 presents an empirical evaluation and a comparison to the state of the art in constraint programming.

2 Solution Sampling

The core idea of our sampling algorithm, borrowed from an approach to approximate model counting, is to employ pairwise-independent hash functions to partition the solution space into roughly equal-size cells of solutions. By aiming for a cell size corresponding to the desired number of samples, we then focus on one such cell and exhaustively enumerate its solutions. Note that while we could in principle enumerate solutions up to the desired number from a much larger cell, this would either introduce a bias in the sampling process from the branching heuristic used or require that we use a totally random branching heuristic to the detriment of better-performing ones and with poor uniformity as observed in [11]. Enumerating from an appropriately-sized cell also means a smaller search space, which could bring computational savings. Finally, because our samples come from the enumeration of a single smaller solution space, we are guaranteed there will be no duplicates.

Such near-uniform partitioning of the solutions can be achieved by adding a system of m linear modular equalities in n integer finite-domain variables

$$A\boldsymbol{x} = \boldsymbol{b} \pmod p$$

where $\boldsymbol{x}$ is a vector of n integer finite-domain variables, A an $m \times n$ matrix whose elements belong to $[p] \triangleq \{0, 1, \ldots, p-1\}$, $\boldsymbol{b}$ a vector of m elements from $[p]$, and p the

modulus. Linear modular equalities are closely related to universal hash functions — we recall two important properties [1]. Let modulus p be a *prime* number, $\boldsymbol{x_1}, \boldsymbol{x_2} \in [p]^n$, $A, \boldsymbol{b}$ be filled uniformly at random, and $\Pr[e]$ denote the probability of event e:

uniform partitioning of solutions: $\Pr[A\boldsymbol{x_1} = \boldsymbol{b} \pmod p] = \frac{1}{p^m}$
pairwise independence: $\Pr[A\boldsymbol{x_1} = \boldsymbol{b} \pmod p \mid A\boldsymbol{x_2} = \boldsymbol{b} \pmod p] = \frac{1}{p^m}$

Another advantage of choosing p to be prime is that we can use Gauss-Jordan Elimination (GJE) to simplify and solve our system of linear equations: when p is prime every element of the finite field F_p has a multiplicative inverse, which is required in order to apply that algorithm. The system of linear equalities $A\boldsymbol{x} = \boldsymbol{b} \pmod p$ is thus rewritten in parametric form. Then we encode its set of solutions — obtained by iterating through the domains of the parametric variables — as a TABLE constraint on all of $\boldsymbol{x}$. This way we can efficiently achieve domain consistency for the system by using e.g. the compact table propagator. Details of the whole filtering algorithm can be found in [9].

Because the number of tuples for the TABLE constraint may be impractically large, we postpone adding the constraint until certain conditions are met. In this work we used the following:

- the size of the Cartesian product of the domains of the parametric variables falls below a given threshold (here, 1000);
- the likelihood that the non-parametric variables (denoted as $\boldsymbol{x''}$) support a given combination of parametric values, estimated as $\prod_{x \in \boldsymbol{x''}} |\mathrm{domain}(x)|/p$, falls below a given threshold (here, 0.5).

Whenever either one of these conditions is met during search, a TABLE constraint is added dynamically, and retracted upon backtracking.

So ultimately for filtering we use TABLE constraints but, in contrast to Vavrille et al.[11]: (i) we express them over the whole set of variables instead of some randomly-chosen small subset; (ii) our tuples originate from hash functions with theoretical guarantees instead of being randomly selected according to some probabilistic threshold; (iii) we generate TABLE constraints dynamically during search.

While linear modular constraints provide us with a way to distribute solutions fairly evenly and independently among the partition of the search space into cells, we still need to control the size of a cell so that it likely contains the desired number of solutions. Simply adding an integral number m of linear modular equalities only provides limited control on cell size especially when p is large: each cell amounts to $1/p^m$ of the search space. To correct this, linear modular *inequalities* can be added as well since they provide a finer control on cell size.

2.1 Systems of Linear Modular Inequalities

As we just saw, we need to handle systems of linear modular inequalities $A\boldsymbol{x} + \boldsymbol{b} \leq \boldsymbol{c} \pmod p$ [3] in order to produce a cell containing about as many solutions as the desired

[3] i.e. each congruent to one of $\{0, 1, \ldots, c[i]\}$. We need to add $\boldsymbol{b}$ in the inequalities because otherwise the probability that e.g. the null solution $\boldsymbol{x} = \boldsymbol{0}$ satisfies the inequality would be equal to 1.

number of samples. While the probabilistic guarantees of such systems were given in Lemma 1 of Pesant et al. [9], namely

$$\Pr[A\boldsymbol{x} + \boldsymbol{b} \leq \boldsymbol{c}] = \Pr[A\boldsymbol{x} + \boldsymbol{b} \leq \boldsymbol{c} \mid A\boldsymbol{y} + \boldsymbol{b} \leq \boldsymbol{c}] = \frac{\prod_{i=1}^{m}(c[i]+1)}{p^m},$$

no filtering algorithm was provided for them. We now outline one that essentially recasts this in terms of equalities. For notational convenience, we first drop $\boldsymbol{b}$ by considering that it has been appended to A and a variable fixed to 1 appended to $\boldsymbol{x}$. Transform system $A\boldsymbol{x} \leq \boldsymbol{c}$, seen as a conjunction of disjunctions of equalities (one conjunct per constraint), into the equivalent disjunction of systems of equalities

$$\bigvee_{0 \leq c'[i] \leq c[i],\ 1 \leq i \leq m} A\boldsymbol{x} = \boldsymbol{c'}$$

by pushing in the conjunction. There will be $\prod(c[i]+1)$ disjuncts that we can represent compactly as $A\boldsymbol{x}$ "=" C with C being the $m \times \prod(c[i]+1)$ matrix representing the right-hand side of each disjunct. We then proceed as in the case of equalities, applying Gauss-Jordan Elimination on this augmented system and eventually enumerating tuples to be fed to a TABLE constraint.

2.2 Sampling Algorithm

Algorithm 1 describes our sampling algorithm. It takes as input the desired size of the sample expressed as a fraction λ of the number of solutions and a set of variables $\boldsymbol{x}$ spanning the search space of the CP model. The latter are typically the branching variables, which totally determine a solution, and not necessarily all the model variables. In particular auxiliary (i.e. dependent) variables need not be included. The algorithm first selects an appropriate value for p (e.g. from a precomputed table of prime numbers): p should be large enough both to achieve (i) the previously-mentioned probabilistic guarantees and (ii) the desired cell size by offering at least a few possibilities in the choice of right-hand side $\boldsymbol{c}$ for inequalities (with $p = 3$ the only option is $\boldsymbol{c} = \mathbf{1}$). It then adds to the CP model a suitable mix of m linear modular equality and inequality constraints on $\boldsymbol{x}$ so that

$$\frac{\prod_{i=1}^{m'}(c[i]+1)}{p^m} \approx \lambda, \quad m' \leq m,$$

as directed by Algorithm 2 described below. Finally it enumerates and returns all the solutions in the resulting cell (i.e. the original CP model with the added linear modular constraints). Note that it is sufficient to branch on the parametric variables of the system of equality constraints since assigning these fixes the non-parametric variables as well.

We chose to specify the sample size relative to the number of solutions because it does not require some approximation of the latter. If such an approximation is available, generating some given absolute number of samples can easily be done as well by deriving the corresponding fraction.

The success of our approach relies on two factors: an even partition of solutions into cells, provided by the linear modular constraints we use, and a relative cell size

Algorithm 1: Sampling algorithm

Input: sample fraction λ, model variables $\boldsymbol{x}$
Output: set of sampled solutions

1 $\ell \leftarrow$ largest domain value among $\boldsymbol{x}$
2 $p \leftarrow$ smallest prime $\geq \max(\ell, 5)$
3 $m, F \leftarrow$ partition(λ, p)
4 $m_{\leq} \leftarrow |F|$
5 $m_{=} \leftarrow m - m_{\leq}$
6 **if** $m_{=} > 0$ **then**
7 **for** $i \leftarrow 1$ **to** $m_{=}$ **do**
8 $\boldsymbol{b}[i] \leftarrow \mathbb{U}_{[p]}$ `// choose uniformly at random from [p]`
9 **for** $j \leftarrow 1$ **to** $|\boldsymbol{x}|$ **do** $A[i][j] \leftarrow \mathbb{U}_{[p]}$
10 post $A\boldsymbol{x} = \boldsymbol{b} \pmod p$
11 $\boldsymbol{x'} \leftarrow$ parametric variables of $A\boldsymbol{x} = \boldsymbol{b} \pmod p$ `// from GJE solved form`
12 **else** $\boldsymbol{x'} \leftarrow \boldsymbol{x}$
13 **if** $m_{\leq} > 0$ **then**
14 **for** $i \leftarrow 1$ **to** $m_{\leq}$ **do**
15 remove a factor f from F
16 $\boldsymbol{c}[i] \leftarrow f - 1$
17 $\boldsymbol{b}[i] \leftarrow \mathbb{U}_{[p]}$
18 **for** $j \leftarrow 1$ **to** $|\boldsymbol{x'}|$ **do** $A[i][j] \leftarrow \mathbb{U}_{[p]}$
19 post $A\boldsymbol{x'} + \boldsymbol{b} \leq \boldsymbol{c} \pmod p$
20 **return** all solutions of the resulting CP model, branching on $\boldsymbol{x'}$

close to sample fraction λ. For the latter, we seek an expression whose denominator is determined by m (given p) and whose numerator can be decomposed into at most m factors all less than p (Algorithm 3), each giving rise to a corresponding inequality. Algorithm 2 computes a suitably accurate combination of linear modular equalities and inequalities to restrict the search space to a cell of the desired size. It repeatedly attempts to reach the target accuracy (given by parameter ϵ, set to 0.01 in our experiments) while keeping the numerator of our approximation to λ below threshold $\nu^{\max}$ (set to 100) because it corresponds to the number of disjuncts in the translation of the system of inequalities we will generate (Section 2.1), doubling parameter ϵ after each attempt until we succeed. In practice it only requires a few attempts. In each attempt we consider, for increasing values of m, a potential factorization of integral numerators $\lfloor \nu \rfloor$ and $\lceil \nu \rceil$, where ν is the exact rational numerator such that $\nu/p^m = \lambda$, provided that the resulting approximation of λ would be accurate enough. For example *partition*$(\lambda = 0.02, p = 11)$ requires two attempts, settling on $m = 3$ and $F = \langle 9, 3 \rangle$ with a relative error below 0.015.

Algorithm 2 always terminates. Indeed, the inner loop increases ν on line 7 and eventually exits the loop on line 9 if ν becomes too large. The outer loop increases ϵ on line 12 and terminates on line 8 if it becomes too large.

Algorithm 3 decomposes integer a into the fewest factors (and no more than d), none of them larger than f. Failing this, it returns an empty list. It is an adaptation of

Algorithm 2: partition(λ, p)

Input: sample fraction λ, modulus p
Output: number of constraints m and list of factors F

$F \leftarrow \langle\rangle$
repeat
  $m \leftarrow 0$
  $\nu \leftarrow \lambda$
  while $F = \langle\rangle$ **do**
    $m \leftarrow m + 1$
    $\nu \leftarrow \nu \times p$
    if $\frac{|\nu - 1|}{\nu} \leq \epsilon$ **then return** $m, \langle\rangle$ `// equalities are sufficient`
    if $\nu > \nu^{\max}$ **then break** `// numerator too large; try bigger` ϵ
    if $\frac{\nu - \lfloor\nu\rfloor}{\nu} \leq \epsilon$ **then** $F \leftarrow$ factorize$(\lfloor\nu\rfloor, p-1, m)$
    if $F = \langle\rangle \wedge \frac{\lceil\nu\rceil - \nu}{\nu} \leq \epsilon$ **then** $F \leftarrow$ factorize$(\lceil\nu\rceil, p-1, m)$
  $\epsilon \leftarrow 2\epsilon$
until $\nu \leq \nu^{\max}$
return m, F

Algorithm 3: factorize(a, f, d)

Input: integer a to factorize, largest possible factor f, maximum number of factors d
Output: list F of factors

$F \leftarrow \langle\rangle$
while $a > 1 \wedge f > 1$ **do**
  while $a \mod f = 0$ **do**
    add f to F
    $a \leftarrow a \div f$
  $f \leftarrow f - 1$
if $a = 1 \wedge |F| \leq d$ **then return** F
else return $\langle\rangle$

the simple *Trial division* algorithm [4] in which we instead try factors in *decreasing* order to minimize the number of factors and thus maximize the number of equality constraints we will use, which contributes to reduce the number of branching variables $\boldsymbol{x'}$. Because $a \leq \nu^{\max}$ and $f < p$ using a more efficient algorithm is not worthwhile.

3 Experiments

In this section we evaluate the sampling uniformity and computational efficiency of our approach using benchmark instances from the literature. The `java.util.Random` random number generator is used in all our experiments. The random table approach [11]

[4] https://en.wikipedia.org/wiki/Trial_division

is evaluated using the authors' own code. The code for our approach is available as well.[5]

3.1 Benchmark Problems

For all our experiments, we based ourselves on four problems. The N-Queens problem (used in [11]), the Feature Models problem (software configuration problem, used in [11]), a Synthetic_n_d problem (n variables of domain size d, composed of two sub-problems, the total number of solutions is computable analytically, used and defined in [9]), and the Myciel problem (graph coloring, used in [9]). For the instances with a low number of solutions (Feature Models, 9-Queens, Synthetic_10_5) we aim to generate as many samples as 30 times the number of solutions. For our approach we use fraction $\lambda \in \{0.01, 0.05, 0.25\}$ (it is hard to go below 1% with that few solutions). For the other instances (Myciel 4, 15-Queens, Synthetic_10_10) we perform 100 runs with $\lambda = 10^{-5}$. For the random table approach we use the best combination of parameters reported for instances appearing in [11] and their generally-recommended values ($\kappa = 16, p = 1/32$) otherwise.

3.2 Quality of Sampling

As in [11], we evaluate the statistical quality of the sampling by computing the p-value of Pearson's χ^2 statistic. Given the number of solutions to the problem $nSol$, the number of samples $nSample$, the number of occurrences $nOcc_k$ of a solution k in the samples and P the expected number of occurrences of a given solution assuming a uniform distribution ($P = \frac{nSample}{nSol}$), we compute

$$z_{exp} = \sum_{k=1}^{nSol} \frac{(nOcc_k - P)^2}{P}.$$

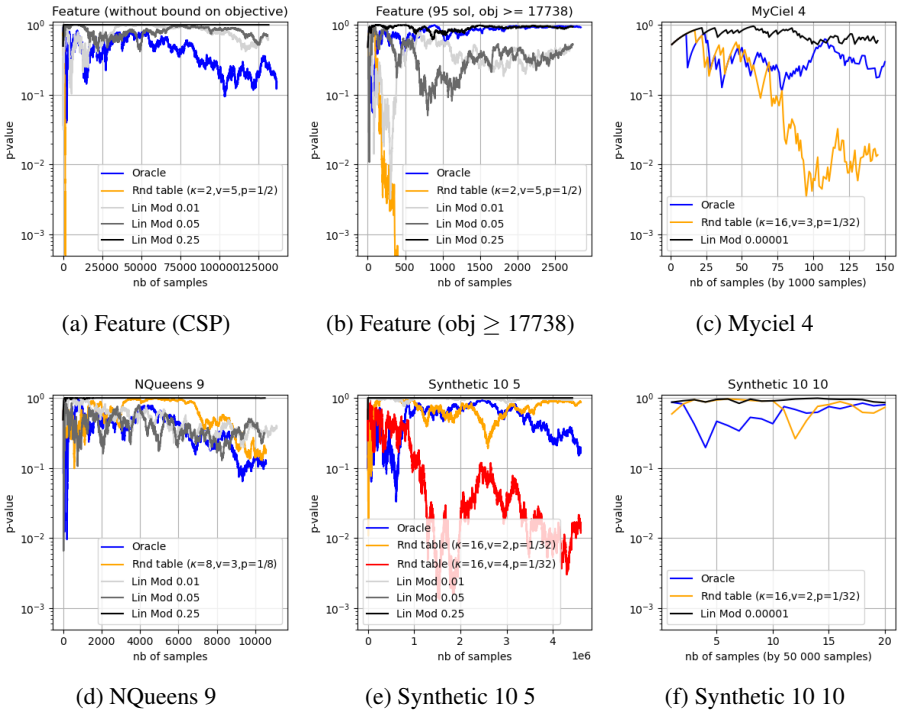
The p-value is computed based on this expression.[6] The closer to 1 the p-value is, the closer to a uniform distribution the sampling is.

Figure 1 shows the evolution of this p-value as samples are drawn. We compare our approach (*Lin Mod*) to that of [11] (*Rnd table*) as well as to an oracle (*Oracle*) that simulates random sampling from the pool of all solutions by generating a sequence of random integers between 1 and $nSol$. As pointed out in [11], this generator is not perfect. Our study of the evolution of the p-value confirms it.

For *9-Queens* the random table approach had achieved very good uniformity for some combinations of its parameters [11]. We achieve even nearer uniformity for all three values of λ. On *Synthetic*, the random tables over 2 variables perform very well (however, not as well over 4 variables) but our approach is practically perfect. *Feature Models* is an instance for which it was reported that the random table approach did not

[5] The sampling method is implemented in the MiniCPBP solver https://github.com/PesantGilles/MiniCPBP and our examples on how to use the method are available https://github.com/363734/UniformSampling

[6] The statistical library used is "Apache Commons Mathematics Library" (https://commons.apache.org/proper/commons-math/).



(a) Feature (CSP) (b) Feature (obj $\geq$ 17738) (c) Myciel 4

(d) NQueens 9 (e) Synthetic 10 5 (f) Synthetic 10 10

Fig. 1: Evolution of the p-value for competing sampling approaches on six benchmark instances.

perform well. On the contrary we see that our approach yields very good uniformity, even outperforming the oracle on the version with an unbounded objective. There is also a marked difference in quality on *Myciel*, whose model features 21 variables.

So on these benchmarks our sampling approach generally leads to better uniformity than the random table approach and even the oracle. Uniformity generally increases with λ, which is expected since the number of samples drawn at a time increases as well and these are necessarily distinct by design (we enumerate solutions in a more constrained space) whereas the oracle sampler may generate the same solution multiple times. A typical λ would be closer to 0.01 than to 0.25.

3.3 Runtime Efficency of our Sampling Approach

In the previous section we gave empirical evidence that our approach newly contributes near-uniform sampling. But it remains to show that it is less time-consuming than the brute-force approach of enumerating all solutions and then sampling uniformly at random from them (i.e. the oracle). When the number of solutions is small the latter may indeed be sufficient but in a realistic setting we sample from a large pool of solutions.

Instance	#solutions	$\frac{\#\text{solutions}}{\lvert\text{search space}\rvert}$	Runtime ratio			
			Oracle	Random table		
				$v=2$	$v=3$	$v=4$
15-Queens	2.28e+6	5.2e-12	5.4	2e-3	3e-3	1e-2
Myciel 4	1.42e+8	3.0e-07	12.6	0.9	1.3	4.8
Synthetic_10_5	1.53e+5	1.6e-02	105.1	4.7	5.3	5.3
Synthetic_10_10	9.92e+8	1.0e-01	1183.9	2.7	13.3	137.9

Table 1: Runtime ratio between the other two approaches considered and ours to sample about 0.001% of solutions (for random table we use $\kappa = 16$, $p = 1/32$).

Table 1 reports for the larger instances the runtime ratio between enumerating the solutions in the whole search space and in a cell approximately 10^{-5} of its size (our approach). In each case we observe computational savings, even by a few orders of magnitude, without being close to an ideal 10^5 ratio from the corresponding reduction in search space because there is an overhead in handling linear modular constraints and recall that active domain filtering through TABLE constraints occurs lower in the search tree once they are posted. Better ratios appear to have more to do with the higher density of solutions than the actual number of solutions (see Table 1, second and third column). We also report the runtime ratio with the random table approach even though its sampling is not nearly as uniform. Except for 15-Queens, it is not faster either.

4 Conclusion

We described a novel approach to sample the set of solutions of any CP model. It is based on adding linear modular constraints to the model, thereby reducing its search space while preserving the proportion of solutions, and then exhaustively exploring that reduced search space. The amount of reduction being applied corresponds to the fraction of solutions one wishes to sample. Experiments on benchmark problems provided empirical evidence that our approach offers computationally-efficient near-uniform sampling.

Acknowledgements

We thank the anonymous reviewers for their constructive criticism which helped us improve the original version of the paper. Financial support for this research was provided in part by NSERC Discovery Grants 218028/2017 and 05953/2016.

References

1. Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3218–3224. AAAI Press, 2016.

2. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2013.
3. Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30:565–620, 2007.
4. Rafael Tupynambá Dutra. *Efficient Sampling of SAT and SMT Solutions for Testing and Verification*. PhD thesis, University of California, Berkeley, 2019.
5. Johannes K Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *Journal of Experimental Algorithmics (JEA)*, 26:1–26, 2021.
6. Vibhav Gogate and Rina Dechter. A new algorithm for sampling CSP solutions uniformly at random. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 711–715. Springer, 2006.
7. Kuldeep S. Meel, Moshe Y. Vardi, Supratik Chakraborty, Daniel J. Fremont, Sanjit A. Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. Constrained sampling and counting: Universal hashing meets SAT solving. In Adnan Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016.
8. Guillaume Perez and Jean-Charles Régin. Mdds: Sampling and probability constraints. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2017.
9. Gilles Pesant, Kuldeep S. Meel, and Mahshid Mohammadalitajrishi. On the usefulness of linear modular arithmetic in constraint programming. In Peter J. Stuckey, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5-8, 2021, Proceedings*, volume 12735 of *Lecture Notes in Computer Science*, pages 248–265. Springer, 2021.
10. Tian Sang, Paul Beame, and Henry A Kautz. Performing bayesian inference by weighted model counting. In *AAAI*, volume 5, pages 475–481, 2005.
11. Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution sampling with random table constraints. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 56:1–56:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.