A New Algorithm for Sampling CSP Solutions Uniformly at Random

Vibhav Gogate and Rina Dechter

Donald Bren School of Information and Computer Science
University of California, Irvine, CA 92697
{vgogate,dechter}@ics.uci.edu

Abstract. The paper presents a method for generating solutions of a constraint satisfaction problem (CSP) uniformly at random. Our method relies on expressing the constraint network as a uniform probability distribution over its solutions and then sampling from the distribution using state-of-the-art probabilistic sampling schemes. To speed up the rate at which random solutions are generated, we augment our sampling schemes with pruning techniques used successfully in the CSP literature such as conflict-directed back-jumping and no-good learning.

1 Introduction

The paper presents a method for generating solutions to a constraint network uniformly at random. The idea is to express the uniform distribution over the set of solutions as a probability distribution and then generate samples from this distribution using monte-carlo sampling. We develop monte-carlo sampling algorithms that extend our previous work on monte-carlo sampling algorithms for probabilistic networks [4] in which the output of generalized belief propagation is used for sampling.

Our experiments reveal that pure sampling schemes, even if quite advanced [4], may fail to output even a single solution for constraint networks that admit few solutions. So we propose to enhance sampling with search techniques that aim at finding a consistent solution fast, such as conflict directed back-jumping and no-good learning.

We demonstrate empirically the performance of our search+sampling schemes by comparing them with two previous schemes: (a) the WALKSAT algorithm [6] and (b) the mini-bucket approximation [2]. Our work is motivated by a real-world application of generating test programs in the field of functional verification (see [2] for details).

2 Preliminaries

Definition 1 (constraint network). *A constraint network (CN) is defined by a 3-tuple,* $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$*, where* $\mathbf{X}$ *is a set of variables* $\mathbf{X} = \{X_1, \ldots, X_n\}$*, associated with a set of discrete-valued domains,* $\mathbf{D} = \{\mathbf{D_1}, \ldots, \mathbf{D_n}\}$*, and a set of constraints* $\mathbf{C} = \{C_1, \ldots, C_r\}$*. Each constraint* C_i *is a pair* $(\mathbf{S_i}, \mathbf{R_i})$*, where* $\mathbf{R_i}$ *is a relation* $\mathbf{R_i} \subseteq \mathbf{D_{S_i}}$ *defined on a subset of variables* $\mathbf{S_i} \subseteq \mathbf{X}$*.* $\mathbf{R_i}$ *contains the allowed tuples of* C_i*. A solution is an assignment of values to variables* $\mathbf{x} = (X_1 = x_1, \ldots, X_n = x_n)$*,* $X_i \in \mathbf{D}_i$*, such that* $\forall\, C_i \in \mathbf{C}$*,* $\mathbf{x_{S_i}} \in \mathbf{R_i}$*.*

Definition 2 (Random Solution Generation Task). *Let* **sol** *be the set of solutions to a constraint network $\mathcal{R} = (\mathbf{X}, \mathbf{D}, \mathbf{C})$. We define a uniform probability distribution $P_u(\mathbf{x})$ relative to $\mathcal{R}$ such that for every assignment $\mathbf{x} = (X_1 = x_1, \ldots, X_n = x_n)$ to all the variables that is a solution, we have $P_u(\mathbf{x} \in \mathbf{sol}) = \frac{1}{|\mathbf{sol}|}$ while for non-solutions we have $P_u(\mathbf{x} \notin \mathbf{sol}) = 0$. The task of random solution generation is to generate positive tuples from this distribution uniformly at random.*

3 Generating Solutions Uniformly at Random

In this section, we describe how to generate random solutions using monte-carlo (MC) sampling. We first express the constraint network $\mathcal{R}(\mathbf{X}, \mathbf{D}, \mathbf{C})$ as a uniform probability distribution $\mathcal{P}$ over the space of solutions: $\mathcal{P}(\mathbf{X}) = \alpha \prod_i C_i(\mathbf{S_i} = \mathbf{s_i})$. Here, $C_i(\mathbf{s_i}) = 1$ if $\mathbf{s_i} \in R_i$ and 0 otherwise. $\alpha = 1/\sum \prod_i f_i(\mathbf{S_i})$ is the normalization constant. Clearly, any algorithm that samples tuples from $\mathcal{P}$ accomplishes the solution generation task. This allows us to use the following monte-carlo (MC) sampler to sample from $\mathcal{P}$.

Algorithm Monte-Carlo Sampling
Input: A factored distribution $\mathcal{P}$ and a time-bound ,**Output:** A collection of samples from $\mathcal{P}$.
Repeat until the time-bound expires

```
1. FOR j = 1 to n
   (a) Sample $X_j = x_j$ from $P(X_j|X_1 = x_1, \ldots, X_{j-1} = x_{j-1})$
2. End FOR
3. If $x_1, \ldots, x_n$ is a solution output it.
```

Hence forth, we will use P to denote the conditional distribution $P(X_j|X_1, \ldots, X_{j-1})$. In [2], a method is presented to compute P in time exponential in tree-width. But the tree-width is usually large for real-world networks and so we have to use approximations.

4 Approximating P using Iterative Join Graph Propagation

Because exact methods for computing the conditional probabilities P are impractical when the tree-width is large, we consider a generalized belief propagation algorithm called Iterative Join Graph Propagation (IJGP) [3] to compute an approximation to P. IJGP is a belief-propagation algorithm that takes a factored probability distribution $\mathcal{P}$ and a partial assignment $\mathbf{E} = \mathbf{e}$ as input. It then performs message passing on a special structure called the join-graph. The output of IJGP is a collection of functions which can be used to approximate $\mathcal{P}(X_j|\mathbf{e})$ for each variable X_j of $\mathcal{P}$. If the number of variables in each cluster is bounded by i (called the i-bound), we refer to IJGP as IJGP(i). The time and space complexity of one iteration of IJGP(i) is bounded exponentially by i.

IJGP(i) can be used to compute an approximation Q of P by executing it with $\mathcal{P}$ and the partial assignment $X_1 = x_1, \ldots, X_{j-1} = x_{j-1}$ as input and then using Q instead of P in step 1(a) of algorithm monte-carlo sampling. In this case, IJGP(i) should be executed n times, one for each instantiation of variable X_j to generate one full sample. This process may be slow because the complexity of generating N samples in this way is $O(N * n *$

$exp(i))$. To speed-up the sampling process, in [4] we pre-computed the approximation of P by executing IJGP(i) just once, yielding a complexity of $O(N * n + exp(i))$.

Therefore, in order to be able to have a flexible control between the two extremes of using IJGP(i) just once, prior to sampling, versus using IJGP(i) at each variable instantiation, we introduce a control parameter p, measured as a percentage, which allows executing IJGP(i) every p % of the possible n variable instantiation. We call the resulting technique IJGP(i,p)-sampling.

4.1 Rejection of Samples

It is important to note that when all P's are exact in the algorithm monte-carlo sampling, all samples generated are guaranteed to be solutions to the constraint network. However, when we approximate P using IJGP such guarantees do not exist and our scheme will attempt to generate samples that are not consistent and therefore need to be rejected. Since the amount of rejection in IJGP(i,p)-sampling can be quite high, we equipped the basic IJGP(i,p)-sampling scheme with pruning algorithms common in search schemes for solving constraint problems. We describe these schemes in the next section.

5 Backjumping and No-good learning to improve IJGP(i,p)-sampling

Traditional sampling algorithms start sampling anew from the first variable in the ordering when an inconsistent assignment (sample) is generated. Instead, the algorithms can backtrack to the previous variable and sample a new value for the previous variable as is common in search algorithms. In other words, we could perform backtracking search instead of pure sampling. Before we sample a new value for the previous variable, we can update our sampling probability to reflect the discovery of the rejected sample. Also, instead of using naive backtracking we can use a more advanced approach such as conflict-directed backjumping and no-good learning. In conflict-directed backjumping, the algorithm backtracks a few levels back, to a variable that can be relevant to the current variables, instead of the recent previous variable [1]. In no-good learning each time an inconsistent assignment (sample) is discovered, the algorithm adds the assignment as a constraint (no-good) to the original constraint network so that in subsequent calls to the search procedure, the same assignment is not sampled. We learn only those no-goods which are bounded by i (the i-bound of IJGP(i)) to maintain constant space.

Once a no-good bounded by i is discovered, we check if the scope of the no-good is included in a cluster of the join-graph. If it is, then we insert the no-good in the cluster and subsequent runs of IJGP utilize this no-good; thereby potentially improving its approximation. We refer to the algorithm resulting from adding back-jumping search and no-good learning to IJGP(i,p)-sampling as IJGP(i,p)-SampleSearch.

6 Experimental Evaluation

We experimented with 5 algorithms (a) IJGP(i,p)-sampling which does not perform search, (b) MBE(i)-sampling which uses mini-bucket-elimination instead of IJGP to

Problems (N,K,C,T)	Time	IJGP(3,p)-SampleSearch No learning				IJGP(3,p)-SampleSearch learning				MBE(3)-SampleSearch
		p=0	p=10	p=50	p=100	p=0	p=10	p=50	p=100	p=0
		KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S
100,4,350,4	1000s	0.0346 0.0074 82290	0.0319 0.0061 42398	0.0108 0.0028 19032	0.011 0.0017 11792	0.0403 0.0086 103690	0.0172 0.0048 37923	0.013 0.0026 25631	0.0053 0.0008 9872	0.134 0.073 93823
100,4,370,4	1000s	0.0249 0.0089 18894	0.0235 0.0062 17883	0.0267 0.0084 2983	0.0156 0.0037 1092	0.0167 0.0058 28346	0.0188 0.0061 14894	0.0143 0.0049 3329	0.0106 0.0019 1981	0.107 0.0332 33895

Table 1. Performance of IJGP(3,p)-sampling and MBE(3)-sampling on random binary CSPs.

approximate P and does not perform backjumping and no-good learning, (c) IJGP(i,p)-SampleSearch as described in section 5, (d) MBE(i)-SampleSearch which incorporates backjumping in MBE(i)-sampling as described in section 5 and (e) WALKSAT (which only works on SAT instances). We experimented with randomly generated binary constraint networks and SAT benchmarks available from satlib.org. Detailed experiments are presented in the extended version of the paper [5]. Here, we describe results on 100-variable random CSP instances, on logistics benchmarks and on verification benchmarks. For each network, we compute the fraction of solutions that each variable-value pair participates in i.e. $P_e(X_i = x_i)$. Our sampling algorithms output a set of solution samples **S** from which we compute the approximate marginal distribution: $P_a(X_i = x_i) = \frac{N_{\mathbf{S}(x_i)}}{|\mathbf{S}|}$ where $N_{\mathbf{S}(x_i)}$ is the number of solutions in the set **S** with X_i assigned the value x_i. We then compare the exact distribution with the approximate distribution using two error measures (accuracy): (a) *Mean Square error* - the square of the difference between the approximate and the exact and (b) *KL distance* - $P_e(x_i) * log(P_e(x_i)/P_a(x_i))$ averaged over all values, all variables and all problems. We also report the number of solutions generated by each sampling technique.

100-variable random CSPs: We experimented with randomly generated 100-variable CSP instances with domain size and tightness of 4. Here, we had to stay with relatively small problems in order to compute the exact marginal for comparison. The time-bound used is indicated by the column *Time* in Table 1. The results are averaged over 100 instances. We used an i-bound of 3 in all experiments. Here, pure IJGP(i,p)-sampling and pure MBE(i)-sampling ([2]) which do not perform search did not generate any consistent samples (solutions) and so we report results on IJGP(i,p)-SampleSearch and MBE(i)-SampleSearch in Table 1. We can see that the accuracy of IJGP(i,p)-SampleSearch increases and the number of solutions generated decrease as we increase p (see Table 1). Thus, we clearly have a trade-off between accuracy and the number of solutions generated as we change p. It is clear from Table 1 that our new scheme IJGP(i,0)-SampleSearch is better than MBE(i) based solution sampler both in terms of accuracy and the number of solutions generated. Also, no-good learning improves the accuracy of IJGP(i,p)-SampleSearch in most cases.

SAT benchmarks: We experimented with logistics and verification SAT benchmarks available from satlib.org. On all the these benchmarks instances, we had to reduce the number of solutions that each problem admits by adding unary clauses in order to apply our exact algorithms. Here, we only experimented with our best performing algorithm

	Logistics.a			Logistics.d			Verification1			Verification2		
	N=828,Time=1000s			N=4713,Time=1000s			N=2654,Time=10000s			N=4713,Time=10000s		
	IJGP(3,10)		WALK	IJGP(3,10)		WALK	IJGP(3,10)		WALK	IJGP(3,10)		WALK
	No Learn	Learn		No Learn	Learn		No Learn	Learn		No Learn	Learn	
KL	0.00978	0.00193	0.01233	0.0009	0.0003	0.0008	0.0044	0.0037	0.003	0.0199	0.0154	0.01
MSE	0.001167	0.00033	0.00622	0.00073	0.00041	0.0002	0.0035	0.0021	0.0012	0.009	0.0088	0.0073
#S	23763	32893	882	10949	19203	28440	1394	945	11342	1893	1038	8390

Table 2. KLD, Mean-squared Error and #Solutions for SAT benchmarks

IJGP(i,p)-SampleSearch with i=3 and p=10. From Table 2 we can see that on the logistics benchmarks, IJGP(3,10)-SampleSearch is slightly better than WALKSAT in terms of accuracy while on the verification benchmarks WALKSAT is slightly better than IJGP(3,10)-SampleSearch. WALKSAT however dominates IJGP(3,10)-SampleSearch in terms of the number of solutions generated (except Logistics.a).

7 Summary and Conclusion

The paper presents a new class of algorithms for generating random, uniformly distributed solutions for constraint satisfaction problems. The algorithms that we develop fall under the class of monte-carlo sampling algorithms that sample from the output of a generalized belief propagation algorithm and extend our previous work [4]. We show how to improve upon conventional monte-carlo sampling methods by integrating sampling with back-jumping search and no-good learning. This has the potential of improving the performance of monte-carlo sampling methods used in the belief network literature [4], especially on networks having large number of zero probabilities. Our best-performing schemes are competitive with the state-of-the-art SAT solution samplers [6] in terms of accuracy and thus present a Monte-carlo (MC) style alternative to random walk solution samplers like WALKSAT [6].

Acknowledgements

This work was supported by the NSF grants IIS-0331707 and IIS-0412854.

References

1. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
2. Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *AAAI*, 2002.
3. Rina Dechter, Kalev Kask, and Robert Mateescu. Iterative join graph propagation. In *UAI '02*, pages 128–136. Morgan Kaufmann, August 2002.
4. Vibhav Gogate and Rina Dechter. Approximate inference algorithms for hybrid bayesian networks with discrete constraints. *UAI-2005*, 2005.
5. Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. Technical report, University of California, Irvine, 2006.
6. Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, 2004.