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. The main idea is to express the CSP as a factored probability distribution over its solutions and then generate samples from this distribution using probabilistic sampling schemes. We suggest parameterized sampling schemes that sample solutions from the output of a generalized belief propagation algorithm. To speed up the rate at which random solutions are generated, we augment our sampling algorithms with techniques used successfully in the CSP literature to improve search such as conflict-directed back-jumping and no-good learning. The motivation for this tasks comes from hardware verification. Random test program generation for hardware verification can be modeled and performed through CSP techniques, and is an application in which uniform random solution sampling is required.

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 generating samples from this distribution using monte-carlo sampling. We develop novel monte-carlo sampling algorithms that extend our previous work on monte-carlo sampling algorithms for probabilistic networks [8]. In [8], we developed a parameterized sampling algorithm that samples from the output of a generalized belief propagation algorithm called Iterative Join Graph propagation (IJGP) [4, 12]. IJGP was shown empirically to be a very good approximation algorithm for probabilistic inference [4] and thus sampling from its output is a reasonable choice.

In our prior work [8], we pre-computed the approximate distribution that we sample from by executing IJGP just once. In this work, we allow re-computation of the distribution that we sample from by executing IJGP periodically as controlled by a parameter *p*. This yields a flexible spectrum of accuracy vs time for us to examine such that when when p=100 we have better accuracy but invest more time while when p=0 we have less accuracy but invest less time and for other values of *p* there is a trade-off between accuracy and time. Incidentally, when p=0 our scheme coincides with [8]. We refer to our resulting algorithm as IJGP(i,p)-sampling.

Our preliminary experiments however revealed that IJGP(i,p)-sampling may fail to output even a single solution for constraint networks that admit few solutions (i.e. for instances in the phase transition). So we propose to enhance IJGP(i,p)-sampling with

systematic search techniques like conflict directed back-jumping and no-good learning. To the best of our knowledge, considering sampling as search has not been investigated before and is one the main contributions of this work. We refer to the resulting search+sampling technique as IJGP(i,p)-search-sampling.

The random solution generation problem is motivated by the task of test program generation in the field of functional verification. The main vehicle for the verification of large and complex hardware designs is simulation of a large number of random test programs [2]. These large hardware designs can be modeled as constraint satisfaction problems and in this case the test programs are solutions to the constraint satisfaction problems. The number of solutions of the modeled program could be as large as 10^{1000} and the typical number of selected test programs are in the range of 10^3 or 10^4. The best test generator is the one that would uniformly sample the space of test programs which translates to generating solutions to the constraint network uniformly at random.

The two previous approaches for generating random solutions are based on the WALKSAT algorithm [11] and the mini-bucket approximation (MBE(i)). The latter converts a constraint network into a belief network and samples from its approximation [3]. Our work can be seen as an extension to the second approach which uses generalized belief propagation [12, 4] instead of MBE(i) for approximation. We also augment MBE(i)-based solution sampler with ideas borrowed from search that improve the speed at which random solutions are generated.

We tested empirically the performance of our sampling algorithm using two evaluation criteria: (a) *accuracy* which measures how random the generated solutions are and (b) *the number of solutions* generated in a stipulated amount of time. We experimented with randomly generated binary constraint networks, randomly generated satisfiability instances and some benchmark instances from SATLIB.

We compared the performance of IJGP(i,p)-search-sampling with MBE(i)-based solution sampler [3] and WALKSAT based solution sampler [11]. We found that on randomly generated binary constraint networks IJGP(i,p)-search-sampling dominates MBE(i) based solution sampler both in terms of accuracy and the number of solutions. On randomly generated SAT problems and benchmarks from SATLIB, we found that IJGP(i,p)-search-sampling is competitive with WALKSAT both in terms of accuracy and the number of solutions. We observe that the accuracy of the samples generated increases as p increases while the number of solutions generated decreases. Thus p gives a user of our algorithm, the option to trade accuracy and time. In our experiments, we found that increasing i may improve accuracy slightly but the number of solutions generated decreases.

Note that in this work, we empirically investigate our sampling schemes on factored probability distributions that express uniform distribution over positive (non-zero) probabilities. However, our sampling schemes are more general and are applicable to any graphical model that can be expressed as a factored probability distribution (e.g. belief and markov networks [9]). Thus, the application of this work goes beyond sampling solutions from a constraint network.

The paper is organized as follows. In section 2, we present some preliminaries. We then present monte-carlo sampling algorithms that generate solutions uniformly at random from a constraint network in section 3. In section 4, we explore approximate

techniques for sampling solutions while in section 5, we present techniques that can be used to improve the performance of approximate sampling algorithms. We present some experimental results in section 6 and end with a brief summary in section 7.

2 Preliminaries

We follow the notation of representing sets by bold letters and members of a set by capital letters. An assignment of a value to a variable is denoted by a small letter while bold small letters indicate an assignment to a set of variables.

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* $\mathbf{S_i} \subseteq \mathbf{X}$*. The relation denotes all compatible tuples of* $\mathbf{D_{S_i}}$ *allowed by the constraint. 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* $\forall\ C_i \in \mathbf{C}$*,* $\mathbf{x_{S_i}} \in \mathbf{R_i}$*. The constraint satisfaction problem (CSP) is to determine if a constraint network has a solution, and if so, to find one. The constraint network represents its set of solutions. The* **primal graph** *(also called the constraint graph) of a constraint network is an undirected graph that has variables as its vertices and an edge connects any two variables involved in a constraint.*

The task of interest in this paper is that of generating solutions to a constraint network uniformly at random and is defined below.

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 the uniform probability distribution* $P_u(\mathbf{x})$ *over* $\mathcal{R}$ *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 the task of generating each solution to* $\mathcal{R}$ *with probability* $\frac{1}{|\mathbf{sol}|}$*.*

3 Monte Carlo Sampling for Generating Solutions Uniformly at Random

In this section, we describe how to generate random solutions using monte-carlo (MC) sampling. Prior work by Dechter et al. [3] relies on using bucket-elimination or its approximations via mini-bucket elimination to convert a constraint network into a belief network and then generating samples from the belief network. Our approach here extends the work in [3] into a more general scheme.

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. To do this, we consider the constraints $C_i(\mathbf{S_i}, R_i)$ as cost functions f_i in which we assign a value 1 for each allowed tuple in the constraint and 0 for each unallowed tuple. Formally,

$$\mathcal{P}(\mathbf{X}) = \alpha \prod_i f_i(\mathbf{S_i} = \mathbf{s_i}) \quad f_i(\mathbf{S_i} = \mathbf{s_i}) = \begin{cases} 1 & If\ \ \mathbf{S_i} = \mathbf{s_i} \in R_i \\ 0 & otherwise \end{cases}$$

Here, $\alpha = 1/\sum\prod_i f_i(\mathbf{S_i})$ is the normalization constant. It is easy to see that for any consistent assignment $\mathbf{X} = \mathbf{x}$, $\mathcal{P}(\mathbf{x}) = 1/|\mathbf{sol}|$ where $|\mathbf{sol}|$ is the number of solutions to the constraint network. In other words, any algorithm that generates random samples according to $\mathcal{P}$, also generates solutions uniformly at random from the constraint network $\mathcal{R}$.

Expressing the constraint network as a factored probability distribution $\mathcal{P}$ allows us to use the standard monte-carlo (MC) sampler to sample from $\mathcal{P}$. In particular, given an ordering of variables $o = \langle X_1, \ldots, X_n \rangle$, we can generate samples from $\mathcal{P}$ using the following scheme:

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

The conditional distribution $P(X_i|X_1, \ldots, X_{i-1})$ can be computed by its definition:

$$P(X_i|X_1, \ldots, X_{i-1}) = \frac{\sum_{X_{i+1} \ldots X_n} \mathcal{P}(\mathbf{X})}{\sum_{X_i, \ldots, X_n} \mathcal{P}(\mathbf{X})} \quad (1)$$

Hence forth, we will abuse notation and use P to denote the conditional distribution $P(X_i|X_1, \ldots, X_{i-1})$. It is clear from equation 1 that computing P is exponential in the number of variables in the constraint network. If a parameter called the tree-width of the constraint network is small, we can use bucket elimination to compute P (see [3]).

However, for most real-world constraint networks the tree-width is large and thus exact methods for computing P are infeasible. This is to be expected since the random solution generation task is a well known #-complete problem. So we consider various approximations for P in the next section.

4 Approximating the Conditional Distributions using Iterative Join Graph Propagation

In this section, we show how to use a generalized belief propagation algorithm [12] called Iterative Join Graph Propagation (IJGP) [4] to compute an approximation to P. In an earlier work [8], we used the IJGP approximation to compute P prior to sampling. In this work, we investigate schemes that recompute P several times during sampling by using a control parameter p. Also, prior to that Dechter et al. [3] used mini-bucket elimination (MBE) to convert a constraint network into an approximate belief network and then generated solution samples from the approximate belief network. Our approach here extends this earlier work [3, 8] into a more general scheme.

Iterative Join Graph Propagation (IJGP) [4](presented for completeness sake in Figure 1) takes as input two quantities: (1) a collection of functions whose normalized

Algorithm IJGP(i)
Input: A factored probability distribution $\mathcal{P}(\mathbf{X}) = \prod_i F_i(\mathbf{S_i})$ where $\mathbf{S_i} \subseteq \mathbf{X}$. Evidence variables $\mathbf{I} = \mathbf{i}$ where $\mathbf{I} \subseteq \mathbf{X}$
Output: A join graph containing original functions along with messages received from all its neighbors

1. Convert the factored-distribution $\mathcal{P}$ into a join-graph $< JG, \chi, \psi >, JG = (V, E)$ using a method given in [4].
   Select an Activation schedule $d = (u_1, v_1), \ldots, (u_{2*|E|}, v_{2*|E|})$.
   Denote by $h_{(u,v)}$ the message from vertex u to v.
   $cluster(u) = \psi(u) \cup \{h_{(v,u)} | (v,u) \in E\}$, $cluster_v(u) = cluster(u)$ excluding message from v to u.
   Let $h_{(u,v)}(j)$ be $h_{(u,v)}$ computed during the j-th iteration of IJGP.
2. **Process observed variables**:
   Assign relevant evidence to all $R_k \in \psi(u)$ $\chi(u) := \chi(u) - I, \ \ \forall u \in V$.
3. **Repeat iterations of IJGP** :
   - Along d, for each edge (u_i, v_i) in the ordering,
   - compute
   $$h_{(u_i,v_i)} = \alpha \sum_{\chi(u_i) - sep(u_i,v_i)} \prod_{f \in cluster_{v_i}(u_i)} f$$
4. **until**:
   - Max number of iterations is exceeded
   - The algorithm converges i.e. KL distance between old and new messages is less than some threshold.

Fig. 1. Algorithm IJGP(i)

product is a probability distribution $\mathcal{P}(\mathbf{X})$ (i.e. a belief or a markov network) and (2) Evidence $\mathbf{E} = \mathbf{e}$ where $\mathbf{E} \subseteq \mathbf{X}$. It then decomposes these functions into a graph structure called a join-graph satisfying the following independence properties:

Definition 3 (Join-Graph). *[4] Given a factored distribution $\mathcal{P}(\mathbf{X}) = \prod_i C_i(\mathbf{S_i})$ where $\mathbf{S_i} \subseteq \mathbf{X}$ and $C_i \in \mathbf{C}$ are the set of functions or factors of the distribution, a join-graph for $\mathcal{R}$ is a triple $JG =< G, \chi, \psi >$, where $G = (\mathbf{V}, \mathbf{E})$ is a graph, and χ and ψ are labeling functions which associate with each vertex $U \in \mathbf{V}$ two sets, variable label $\chi(U) \subseteq \mathbf{X}$ and function label $\psi(U) \subseteq \mathbf{C}$: (1) For each function $C_i \in \mathbf{C}$, there is exactly one vertex $U \in \mathbf{V}$ such that $C_i \in \psi(U)$, and $\mathbf{S_i} \subseteq \chi(\mathbf{U})$ and (2) For each variable $X_i \in \mathbf{X}$, the set $\{U \in \mathbf{V} | X_i \in \chi(\mathbf{V})\}$ induces a connected subgraph of G.*

Example 1. Figure 2 shows a constraint network and a possible join graph for the constraint network. The functions associated with each cluster are written on the top of each cluster.

IJGP (see Figure 1) is a form of sum-product belief propagation [12, 4] in which messages are passed between adjacent clusters of the join-graph by taking a product of messages and functions in each cluster and projecting it to the labeled edge between the two clusters. IJGP performs message-passing until a pre-specified number of iterations has been reached or until it converges (i.e. the messages do not change between subsequent iterations). If the number of variables in each cluster is bounded by i (referred to as 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 (see [4] for details).

Example 2. Initially in Figure 2, no messages are exchanged between any clusters. The first message exchanged between cluster (ABD) and (ABC) is given below.

$$M(ABD \rightarrow ABC) = \alpha \sum_{BD} [\neq (A,B) * \neq (B,D)]$$

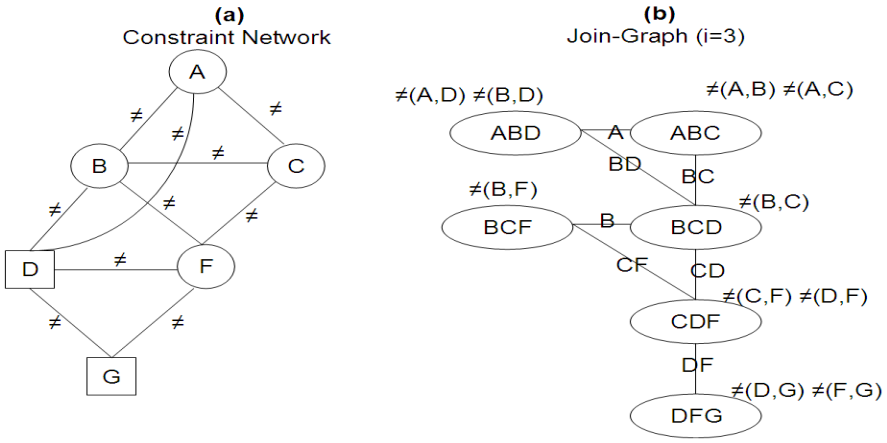

Fig. 2. (a) Constraint Network with not-equal constraints (b) Join-graph with a i-bound of 3

.

$$= \alpha * \sum_{BD}[(0,1,1,1,0,1,1,1,0) * (0,1,1,1,0,1,1,1,0)]$$
$$= \alpha * \sum_{BD}[0,0,0,1,0,1,1,1,0,0,1,1,0,0,0,1,1,0,0,1,1,1,0,1,0,0,0]$$
$$= \alpha(4,4,4) = (0.33,0.33,0.33)$$

The output of IJGP(i) is a collection of functions and messages, each bounded exponentially by i. We can use the output of IJGP to compute an approximation $Q(X_j|\mathbf{e})$ of $P(X_j|\mathbf{e})$ where $X_j \in \mathbf{X} \setminus \mathbf{E}$ by selecting a cluster u in the join-graph such that $X_j \in \chi(u)$, and using the following equation:

$$Q(X_j|\mathbf{e}) = \alpha \sum_{\chi(u)-\{X_j\}} (\prod_{f \in cluster(u)} f) \tag{2}$$

ALGORITHM IJGP-SAMPLING (i,p)
Input: A constraint network $\mathcal{R}$ and i-bound i.
Output: A set of randomly generated solutions.

1. Write the constraint network $\mathcal{R}$ as a probability distribution $\mathcal{P}$
2. Repeat Until there is time
3. FOR j=1 to n do
   (a) Run IJGP(i) with evidence $X_1 = x_1, \ldots, X_{j-1} = x_{j-1}$ and $\mathcal{P}$ as input.
   (b) Compute $Q(X_j|X_1 = x_1, \ldots, X_{j-1} = x_{j-1})$ from the output of IJGP(i) using equation 2.
   (c) Sample $X_j = x_j$ from $Q(X_j|X_1 = x_1, \ldots, X_{j-1} = x_{j-1})$
4. End FOR
5. If $X_1 = x_1, \ldots, X_n = x_n$ is a solution output it
6. End Repeat

Fig. 3. Algorithm IJGP-Sampling to randomly generate solutions of a CSP

Next we show how to use IJGP to approximate the conditional distribution P at each step of our monte-carlo sampler. This algorithm IJGP(i)-sampling (see also [8]) is presented in Figure 3. At each step of sampling, we run IJGP(i) over $\mathcal{P}$ with evidence $X_1 = x_1, \ldots, X_j = x_j$ and compute the approximation $Q(X_j|X_1, \ldots, X_{j-1})$ (using equation 2) from the output of IJGP(i) and then sample X_j from Q. We repeat this process until a given time-bound expires.

Note that in IJGP(i)-sampling (see Figure 3), IJGP(i) should be executed n times, one for each instantiation of variable X_j in order to generate one full sample. This process may be slow because the complexity of running IJGP(i) is exponential in i and therefor the complexity of generating N samples is $O(N * n * exp(i))$. To speed-up the sampling process, in Gogate and Dechter [8] we pre-computed the approximation P of Q by executing IJGP(i) just once yielding a complexity of $O(N * n + exp(i))$ for generating N samples.

In our preliminary empirical testing, we observed that changing only one (or a few) variables to become instantiated often does not impact the approximation Q computed by IJGP(i). Since these reruns of IJGP can present a significant overhead, it can be more cost-effective to rerun IJGP only periodically during sample generation. Therefore, we introduce a control parameter p which allows running IJGP(i) every p% of the possible n variable instantiations. We will refer to the resulting algorithm as IJGP(i,p)-sampling which can be obtained by replacing line 3(a) in Figure 3 by the following:

– If $100\%j = p$ OR if $j = 1$ Then
  Run IJGP(i) with evidence $X_1 = x_1, \ldots, X_{j-1} = x_{j-1}$ and $\mathcal{P}$ as input.

Note that when $p = 0$, our scheme coincides with the scheme presented in our previous work [8]. Also note that if we use mini-bucket-elimination (MBE(i)) instead of IJGP(i) with $p = 0$, our scheme coincides with the one presented in [3]. In principle, one could use any policy p to rerun IJGP(i) for a subset of the possible n variable instantiations. Here, we investigate a periodic policy.

Theorem 1. *The time required to generate N samples using IJGP(i,p)-sampling is* $O([\frac{p}{100} * n * exp(i) * N] + N * n + exp(i))$ *where n is the number of variables and N is the number of samples generated.*

An important property of completeness is expressed in the following theorem.

Theorem 2 (Completeness). *IJGP(i,p)-sampling has a non-zero probability of generating any arbitrary solution to a constraint satisfaction problem.*

Proof. According to the monte-carlo sampling theory[7], if any approximation Q of P satisfies $P > 0 \Rightarrow Q > 0$, then the monte-carlo sampling algorithm that samples from Q will reach all consistent assignments i.e. $(\mathcal{P}(\mathbf{X} = \mathbf{x}) > 0)$ with a non-zero probability. It was proved in[5] that all zero probabilities inferred by IJGP are sound. In other words, for any IJGP approximation $P > 0 \Rightarrow Q > 0$ and so the theorem follows.

5 Improving the Approximate Sampling Algorithm

5.1 Rejection of Samples

It is important to note that when the conditional distributions P are exact in our monte-carlo sampler, all samples generated are guaranteed to be solutions to the constraint net-

work. However, when we approximate P such guarantees do not exist and our scheme will attempt to generate samples that are not consistent. Specifically when using IJGP to approximate P, samples will be rejected because of IJGP's inability to infer a probability of zero for some partial assignments that cannot be extended to a solution.

Our preliminary experiments showed us that the rejection rate of IJGP(i,p)-sampling was very low for networks that admit relatively large number of solutions (under-constrained instances) but was quite high for constraint networks that admit few solutions (phase transition instances). So in this section, we discuss how to decrease the rejection rate of IJGP(i,p)-sampling by utilizing pruning schemes used in systematic search such as conflict-directed backjumping and no-good learning.

5.2 Introducing Backjumping

Traditional sampling algorithms often assume that the probability distribution we sample from is completely positive. However, it is often the case that in probabilistic reasoning the distribution has a lot of zero probabilities (determinism). This is obviously the case when we sample from a constraint-based distribution. Therefore, we can integrate ideas from backtracking search and its pruning techniques into the naive sample generation schemes. Whenever a sample gets rejected, naive sampling algorithms start sampling anew from the first variable in the ordering. Instead, we could backtrack to the previous variable, update the conditional distribution to reflect the dead-end and re-sample the previous variable. For example,

Example 3. Let us assume that for a variable X_j with domain $\{1,2,3\}$, the conditional distribution is $Q(X_j|X_1 = x_1,\ldots,X_{j-1} = x_{j-1}) = (0.33, 0.33, 0.33)$. Let $X_j = 2$ be a sampled value of X_j which causes a dead-end in X_{j+1}. Conventional monte-carlo sampler would then start sampling anew from variable X_1. Instead we could change the conditional distribution $Q'(X_j|X_1 = x_1,\ldots,X_{j-1} = x_{j-1}) = (0.5, 0, 0.5)$ to reflect the fact that $X_j \neq 2$ and sample X_j from Q'.

We propose to use conflict-directed back-jumping instead of pure chronological backtracking for obvious efficiency reasons. An important thing to note is that once a solution is generated (i.e. an assignment that satisfies all constraints), we start anew from the first variable in the ordering to generate the next solution (sample).

5.3 Utilizing Separators

The time required to generate each sample in IJGP-Sampling can be improved by utilizing graph separators. A separator is a set of variables that separates the join graph into two or more connected components. The idea is that after a separator is instantiated, we can recursively assign values to variables in different connected components by running IJGP just once. For example,

Example 4. Let us assume that $p = 100$ for algorithm IJGP(i,p)-Sampling. Without taking advantage of graph separation, IJGP(i) should be invoked 6 times one for each instantiation of variables $\{$A,B,C,D,F,G$\}$ for the join-graph shown in Figure 2. After instantiating the separator *CDF* in the join graph of Figure 2 by running IJGP(i) thrice, we can instantiate *A* and *G* simultaneously by running IJGP(i) just once. Later, we instantiate *B* by running IJGP(i) one more time. Thus IJGP(i) is run for 5 times.

The correctness of this scheme is due to the theorem below [6]:

Theorem 3. *Let $G(V,E)$ be the join-graph of a constraint network, let $W \subseteq V$ be a subset of vertices that separates the graph $G[V \setminus W]$ into two or more connected components $R_1, \ldots, R_m$. Let $X \subseteq R_i$ and $Y \subseteq V \setminus (W \cup R_i), i \neq j$ and let Q be the distribution computed by running IJGP on graph G. Then $Q(X|W = w, Y = y) = Q(X|W = w)$.*

5.4 No-good learning

Conventional monte-carlo sampling methods do not learn no-good from dead-ends once a sample is rejected. Thus they are subjected to thrashing as happens during systematic search. So we augment our sampling schemes that employ back-jumping search with no-good learning schemes as in [1]. Consistent with our *i*-bounded inference, we propose to learn no-goods or conflict sets that are bounded by *i* [1].

Since each no-good can be considered as a constraint, they can be inserted into any cluster in the join graph that includes the scope of the no-good. Indeed, in our preliminary experiments we found that the approximation to P computed by IJGP(i) with learnt no-goods was better than the approximation computed by IJGP(i) without learnt no-goods. So each time a no-good bounded by *i* is discovered, we check if the no-good can be added to a cluster in the join-graph without changing its structure (see definition 3) and subsequent runs of IJGP utilizes this no-good.

We refer to the algorithm resulting from adding back-jumping search and no-good learning to IJGP(i,p)-sampling as IJGP(i,p)-search-sampling.

6 Experimental Evaluation

We tested the performance of IJGP(i,p)-search-sampling on randomly generated binary constraint networks, randomly generated satisfiability (SAT) instances and hard satisfiability (SAT) benchmarks available from SATLIB [1]. The specific algorithms that we experimented with are discussed below.

6.1 Competing Algorithms

IJGP(i,p)-search-sampling Note that at a higher level of abstraction, IJGP(i,p)-search-sampling can be looked at as a search procedure in which at each decision of value selection for a variable X_j, IJGP(i) is run and a value is selected for X_j by sampling X_j from the conditional distribution $Q(X_j|X_1 = x_1, \ldots, X_{j-1} = x_{j-1})$ computed using Equation 2. We refer to this process of selecting a value as value sampling (just like value ordering except that we select a value using sampling). This view allows using any of-the-shelf search algorithm for our sampling scheme by running the IJGP(i,p)-sampling scheme only for some value selection points.

Therefore, in order to be competitive on SAT problems, we implemented IJGP(i) in C++ and integrated it as a value sampling scheme with the RELSAT solver [1]. All experiments on SAT instances are performed using this RELSAT-based-sampler.

[1] http://www.satlib.org/

We also implemented IJGP(i) as a value sampling scheme in a MAC-based binary CSP solver by Tudor Hulubei [2] for solving binary constraint satisfaction problems. All experiments on random binary CSP instances are performed using this MAC-based-sampler.

We implemented our own no-good learner on top of these two solvers. Note that our no-good learner was used to improve the estimates of IJGP(i) and not integrated with the constraint/SAT solver[3]. This can be done in future enhancements. We experimented with i-bounds of 3 and 6 for all the problems while parameter p was selected from the set $\{0, 10, 50, 100\}$. Without loss of generality, we will refer to both RELSAT based solution sampler and MAC based solution sampler as IJGP(i,p)-search-sampling.

MBE(i)-search-sampling We also implemented mini-bucket-elimination or MBE(i) using C++ and used it as a value sampling scheme in the MAC-based CSP solver mentioned above. Note that unlike IJGP(i,p)-search-sampling our current MBE(i) sampler does not have a parameter p. We experimented with i-bounds of 3 and 6 for MBE(i). Our current implementation of MBE(i)-based random solution generator uses backtracking (CBJ to be precise) search while the MBE(i)-based random solution generator used in the previous paper by Dechter et al. [3] does not perform any backtracking. We refer to our current implementation as MBE(i)-search-sampling.

WALKSAT On all SAT instances, we ran an implementation of WALKSAT available on Wei Wei's web-site [4]. The WALKSAT solution sampler uses a random walk procedure for generating random solution samples. It starts with a random truth assignment. Assuming this randomly guessed assignment does not already satisfy the formula, one selects an unsatisfied clause at random, and flips the truth assignment of one of the variables in that clause. This will cause the clause to become satisfied but, of course, one or more other clauses may become unsatisfied. Such flips are repeated until one reaches an assignment that satisfies all clauses or until a pre-defined maximum number of flips are made. WALKSAT has numerous heuristics for flipping variables in a clause. Here we chose to use the heuristic SABEST which selects with probability p a random walk style move and with probability 1 - p, a simulated annealing step. This heuristic was shown to have a superior performance than other heuristics (see [11] for details).

6.2 Performance Criteria

For each network, we compute the exact marginal distribution for each variable in the constraint network using $P_e(X_i = x_i) = \frac{N_{x_i}}{N}$ where N_{x_i} is the number of solutions that the assignment $X_i = x_i$ participates in and N is the number of solutions. The number of solutions for the SAT problems were computed using RELSAT while the number of solutions for binary constraint networks were computed using the MAC implementation by Tudor Hulubei. After running our sampling algorithms, we get a set of solution

[2] www.hulubei.net/tudor

[3] The SAT solver RELSAT however has a no-good learner which we turned off

[4] http://www.cs.cornell.edu/ weiwei/

samples say ϕ from which compute the approximate marginal distribution: $P_a(X_i = x_i) = \frac{N_{\phi(x_i)}}{|\phi|}$ where $N_{\phi(x_i)}$ is the number of solutions in the set ϕ with X_i assigned the value x_i. We then compare the exact distribution with the approximate distribution using two error measures: (a) *Mean Square error* - the square of the difference between the approximate and the exact, averaged over all values, all variables and all problems 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.

Mean-Square Error and K-L distance provide us with an estimate of how close to the uniform distribution our samples are while another important criteria is also the number of solutions generated which we report for each set of experiments.

Problems (N,K,C,T)	Time	IJGP(i,p)-search-sampling No learning				IJGP(i,p)-search-sampling learning				MBE
		p=0	p=10	p=50	p=100	p=0	p=10	p=50	p=100	p=0
		KL	KL	KL	KL	KL	KL	KL	KL	KL
		MSE	MSE	MSE	MSE	MSE	MSE	MSE	MSE	MSE
		#S	#S	#S	#S	#S	#S	#S	#S	#S
		i=3								
50,4,150,4	300s	0.0211	0.0284	0.0192	0.0076	0.0187	0.0102	0.0112	0.009	0.102
		0.0031	0.0047	0.0027	0.0003	0.0028	0.0019	0.0021	0.0003	0.089
		138322	72744	47278	19002	162387	97560	38281	15347	78218
50,4,180,4	300s	0.0278	0.0343	0.0221	0.0092	0.0285	0.0119	0.0185	0.0091	0.1116
		0.0038	0.0047	0.0035	0.0006	0.0028	0.0024	0.0029	0.0006	0.0695
		88329	59209	37012	23671	74126	61822	24102	11438	45829
100,4,350,4	1000s	0.0346	0.0319	0.0108	0.011	0.0403	0.0172	0.013	0.0053	0.134
		0.0074	0.0061	0.0028	0.0017	0.0086	0.0048	0.0026	0.0008	0.073
		82290	42398	19032	11792	103690	37923	25631	9872	93823
100,4,370,4	1000s	0.0249	0.0235	0.0267	0.0156	0.0167	0.0188	0.0143	0.0106	0.107
		0.0089	0.0062	0.0084	0.0037	0.0058	0.0061	0.0049	0.0019	0.0332
		18894	17883	2983	1092	28346	14894	3329	1981	33895
		i=6								
50,4,150,4	300s	0.0178	0.0182	0.011	0.0078	0.093	0.084	0.088	0.009	0.065
		0.0027	0.0017	0.0009	0.0007	0.0011	0.0016	0.0008	0.0003	0.089
		145936	18942	9832	1923	108943	28304	7827	2872	88378
50,4,180,4	300s	0.0245	0.0295	0.0178	0.0109	0.0176	0.0209	0.0148	0.0073	0.093
		0.0057	0.0039	0.0028	0.0068	0.0083	0.0057	0.0029	0.0004	0.0457
		89542	23392	8232	1895	87356	45228	9237	1093	75643
100,4,330,4	1000s	0.0307	0.0346	0.0153	0.0129	0.0386	0.0151	0.0176	0.0008	0.0925
		0.0071	0.0068	0.0018	0.0014	0.0077	0.0056	0.0033	0.0011	0.073
		72306	18383	12832	1038	63594	17387	2382	1822	43872
100,4,370,4	1000s	0.0178	0.0213	0.0191	0.0141	0.0217	0.0119	0.0093	0.0084	0.1058
		0.0077	0.0094	0.0076	0.0054	0.0083	0.0068	0.0058	0.0027	0.0333
		20395	19032	8983	1982	27823	10832	6290	782	14383

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

6.3 Results on randomly generated CSPs

We experimented with two-sets of randomly generated constraint networks; one having 50-variables and the second having 100-variables. All problems are consistent. The benchmarks are generated according to the well-known four-parameter $(N, K = 4, C, T = 4)$ model-B generator [10] where N is the number of variables, K is the number of values, C is the tightness (number of disallowed tuples) and T is the number of

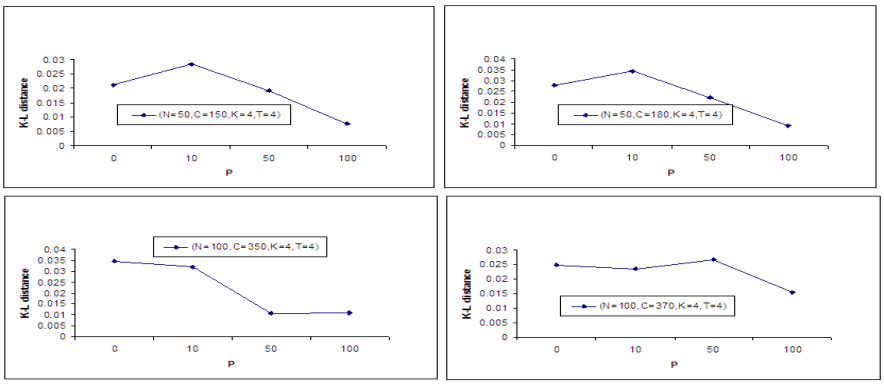

Fig. 4. Effect of increasing p on accuracy (K-L distance) of IJGP(3,p)-search-sampling

constraints. We had to stay with relatively small problems in order to apply the MAC algorithm to count the solutions that each variable-value pair participates in. All approximate sampling algorithms were given the same amount of time to generate solution samples which is given by the column *Time* in Table 1. The results are averaged over 1000 instances each for 50-variable problems and 100 instances each for 100 variable problems.

Note that the problems become harder as we increase the number of constraints for a fixed number of variables (phase transition). The results are summarized in Table 1 which gives MSE and K-L distance, the time-bound for each problem set and the number of solutions generated for each algorithm. We can see that in the case of IJGP(i,p)-search-sampling, for a fixed *i*-bound both error measures decrease as we increase *p* (see Figure 4). However, it can also be seen that as we increase *p*, the number of solutions computed decrease considerably. 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)-search-sampling is better than MBE(i)-search-sampling both in terms of the number of solutions generated and also in terms of the error measures. Increasing the *i*-bound from 3 to 6 (see Table 1) increases accuracy slightly. However the savings are not cost-effective because the number of solutions generated decreases considerably as we increase *i* (except for p=0). It can be seen from Table 1 that no-good learning improves the accuracy of IJGP(i,p)-search-sampling in most cases.

6.4 Results on randomly generated SAT problems

We experimented with two sets of randomly generated 3-SAT problems, one having 50-variables and the second having 100-variables. All problems are consistent. Again, we had to stay with relatively smaller problems because when the number of solutions are large, our complete algorithm RELSAT takes a lot of time to count them. The results are based on 1000 instances of 50-variable SAT problems and 100 instances of 100 variable SAT problems.

Problems (N,K,C,T)	Time	IJGP(i,p)-search-sampling No learning				IJGP(i,p)-search-sampling learning				WALKSAT
		p=0	p=10	p=50	p=100	p=0	p=10	p=50	p=100	p=0
		KL	KL	KL	KL	KL	KL	KL	KL	KL
		MSE	MSE	MSE	MSE	MSE	MSE	MSE	MSE	MSE
		#S	#S	#S	#S	#S	#S	#S	#S	#S
		i=3								
50,150	20s	0.124	0.092	0.083	0.088	0.109	0.089	0.085	0.073	0.114
		0.019	0.007	0.006	0.006	0.013	0.006	0.005	0.005	0.017
		28002	18203	9032	1033	31093	11903	7392	1208	45838
50,200	20s	0.117	0.087	0.086	0.0896	0.1068	0.085	0.084	0.078	0.103
		0.018	0.0069	0.0062	0.0052	0.0124	0.0055	0.005	0.0057	0.016
		48917	18772	8944	1292	50962	12636	7793	1172	42950
100,350	100s	0.123	0.089	0.074	0.082	0.127	0.088	0.074	0.068	0.14
		0.022	0.009	0.008	0.008	0.023	0.009	0.007	0.005	0.024
		89029	54832	17945	1833	79293	42894	27983	2094	103934
100,400	100s	0.107	0.077	0.049	0.024	0.128	0.059	0.039	0.019	0.093
		0.029	0.0084	0.0075	0.0038	0.039	0.0081	0.0077	0.0023	0.019
		70298	28901	11309	1003	60934	39782	9462	1284	93024
		i=6								
50,150	20s	0.112	0.104	0.093	0.099	0.134	0.091	0.085	0.073	0.114
		0.016	0.012	0.009	0.011	0.023	0.009	0.005	0.005	0.017
		24083	2420	1038	378	27910	1983	1123	392	45838
50,200	20s	0.129	0.092	0.102	0.093	0.13	0.079	0.08	0.077	0.103
		0.021	0.008	0.016	0.008	0.024	0.006	0.006	0.004	0.016
		32001	3467	875	297	26801	5487	907	176	42950
100,350	100s	0.1327	0.093	0.066	0.086	0.136	0.088	0.074	0.0713	0.14
		0.022	0.0086	0.0079	0.0087	0.023	0.0082	0.0077	0.00771	0.024
		29841	5216	1828	154	29914	4312	2827	252	103934
100,400	100s	0.109	0.0785	0.052	0.039	0.086	0.057	0.035	0.032	0.093
		0.028	0.0079	0.0052	0.0045	0.019	0.007	0.0049	0.0037	0.019
		25603	7355	946	190	30079	5459	1361	243	93024

Table 2. Performance of IJGP(i,p)-search-sampling and WALKSAT on randomly generated 3-SAT problems.

The results are summarized in Table 2 which gives MSE and K-L distance, the time-bound for each problem set and the number of samples generated. In terms of changing p and i, the results on SAT problems are similar to those on random binary csps (see Table 2) and so we do not comment on it here.

When we compare the results of IJGP(i,p)-search-sampling with WALKSAT, we see that the performance of WALKSAT is slightly better than IJGP(i,p)-search-sampling when $p = 0$ in terms of both error measures. However, as we increase p, the performance of IJGP(i,p)-search-sampling is better than WALKSAT. It is easy to see that WALKSAT dominates IJGP(i,p)-search-sampling by an order of magnitude in terms of the number of solutions computed for $p = 50, 100$. However for $p = 0, 10$ the number of solutions generated by WALKSAT are comparable to IJGP(i,p)-search-sampling.

6.5 Results on SAT benchmarks from SATLIB

We also experimented with some benchmark problems available from SATLIB. Here, we only experimented with our best performing algorithm IJGP(i,p)-search-sampling with i=3 and p=10 (considering a combination of the number of solutions computed and the error measures). In particular, we experimented with 3 logistics instances and 2 instances from the Verification domain (Tables 3 and 4 respectively) On all the SAT

	Logistics.a			Logistics.b			Logistics.d		
	N=828,Time=1000s			N=843,Time=1000s			N=4713,Time=1000s		
	IJGP(3,10)		WALK	IJGP(3,10)		WALK	IJGP(3,10)		WALK
	No Learn	Learn		No Learn	Learn		No Learn	Learn	
KL	0.00978	0.00193	0.01233	0.00393	0.00403	0.0102	0.0009	0.0003	0.0008
MSE	0.001167	0.00033	0.00622	0.00194	0.00243	0.0097	0.00073	0.00041	0.0002
#S	23763	32893	882	11220	21932	93932	10949	19203	28440

Table 3. KL distance, Mean-squared Error and number of solutions generated by IJGP(3,10)-sampling and Walksat on logistics benchmarks

	Verification1			Verification2		
	N=2654,Time=10000s			N=4713,Time=10000s		
	IJGP (3,10)		WALK	IJGP (3,10)		WALK
	No Learn	Learn		No Learn	Learn	
KL	0.0044	0.0037	0.003	0.0199	0.0154	0.01
MSE	0.0035	0.0021	0.0012	0.009	0.0088	0.0073
#S	1394	945	11342	1893	1038	8390

Table 4. KL distance, Mean-squared Error and number of solutions generated by IJGP(3,10)-sampling and Walksat on verification benchmarks

benchmarks that we experimented with, we had to reduce the number of solutions that each problem admits by adding unary clauses. This is because when the number of solutions is large, RELSAT takes a lot of time to count them and note that we have to count the number of solutions that each variable-value pair participates in. From Tables 3 and 4, we can see that our best performing algorithm IJGP(3,10)-search-sampling is competitive with WALKSAT both in terms of the error measures and also in terms of the number of solutions generated. In particular, on the logistics benchmarks, IJGP(3,10)-search-sampling is slightly better than WALKSAT both in terms of both error measures while on the verification benchmarks WALKSAT is slightly better than IJGP(3,10)-search-sampling in terms of both error measures. WALKSAT however dominates our algorithms in terms of the number of solutions generated except for the modified Logistics.a instance (see Table 3).

7 Summary and Conclusion

The paper presents a new algorithm for generating random, uniformly distributed solutions for constraint satisfaction problems. The origin of this task is the use of CSP based methods for the random test program generation. 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 called Iterative Join Graph Propagation (IJGP) and extend our previous work [8]. Using a parameter $p \in [0,100]$, we develop novel sampling algorithms such that when $p = 100$, we have a sampling algorithm that is more accurate but takes more time to generate each sample while when $p = 0$ we have an algorithm that is less accurate but takes very little time to generate each sample while for other values of $p \in [1,99]$ we see a trade-off between accuracy and the number of solutions generated. Thus p gives a user of our algorithms, the option to trade accuracy and time.

Dechter et al. [3] present a method to generate near random solutions using mini-bucket elimination (MBE). In this work, we show empirically that sampling from the output of IJGP yields better results than MBE. We also augment the algorithm presented in [3] with systematic search.

Specifically, we show how to improve upon conventional monte-carlo sampling methods by integrating sampling with back-jumping search and no-good learning and is the main contribution of our work. This has the potential of improving the performance of monte-carlo sampling methods used in the belief network literature [13, 8], especially on networks having large number of zeros probabilities. Note that unlike WALKSAT, our search+sampling schemes are applicable to any graphical model (constraint, belief and markov networks).

Our results look promising in that we are consistently able to generate uniform random solution samples. Our best-performing schemes are competitive with the state-of-the-art SAT solution samplers [11] in terms of accuracy and thus presents a Monte-carlo (MC) style alternative to random walk solution samplers like WALKSAT and Markov Chain Monte Carlo (MCMC) methods like Simulated Annealing.

References

1. R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *AAAI-96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, 1996.
2. J Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.
3. Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *AAAI*, 2002.
4. Rina Dechter, Kalev Kask, and Robert Mateescu. Iterative join graph propagation. In *UAI '02*, pages 128–136. Morgan Kaufmann, August 2002.
5. Rina Dechter and Robert Mateescu. A simple insight into iterative belief propagation's success. *UAI-2003*, 2003.
6. Rina Dechter and Robert Mateescu. Mixtures of deterministic-probabilistic networks and their and/or search space. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 120–129, Arlington, Virginia, United States, 2004. AUAI Press.
7. John Geweke. Bayesian inference in econometric models using monte carlo integration. *Econometrica*, 57(6):1317–39, 1989.
8. Vibhav Gogate and Rina Dechter. Approximate inference algorithms for hybrid bayesian networks with discrete constraints. *UAI-2005*, 2005.
9. J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.
10. Barbara Smith. The phase transition in constraint satisfaction problems: A CLoser look at the mushy region. In *Proceedings ECAI'94*, 1994.
11. Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, 2004.
12. Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *NIPS*, pages 689–695, 2000.
13. C. Yuan and M. J. Druzdzel. Importance sampling algorithms for Bayesian networks: Principles and performance. *To appear in Mathematical and Computer Modelling*, 2005.