Generating Random Solutions for Constraint Satisfaction Problems

Rina Dechter **Kalev Kask**
University of California, Irvine
{dechter, kkask}@ics.uci.edu

Eyal Bin **Roy Emek**
IBM Research Laboratory in Haifa
{bin, emek}@il.ibm.com

Abstract

The paper presents a method for generating solutions of a constraint satisfaction problem (CSP) uniformly at random. The main idea is to transform the constraint network into a belief network that expresses a uniform random distribution over its set of solutions and then use known sampling algorithms over belief networks. 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.

Introduction and Motivation

The paper presents a method for generating uniformly distributed random solutions for a CSP. The method we propose is based on a transformation of the constraint network into a belief network that expresses a uniform random distribution over the CSP's set of solutions. We then can use known sampling methods for belief networks to generate the desired solution samples. The basic algorithm we propose uses a variable elimination approach and its complexity is time and space exponential in the induced-width of the constraint problem. Because of this complexity the approach will not be practical in most real life situations and we therefore propose a general partition-based scheme for approximating the algorithm.

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 (Bergeron 2000). The generation of such programs therefore plays a central role in the field of functional verification.

The input for a test program generator is a specification of a test template. For example, tests that exercise the data cache of the processor and that are formed by a series of double-word `store` and `load` instructions. The generator generates a large number of *distinct* well-distributed test program instances, that comply with the user's specification. In addition, generated test programs must meet two inherent classes of requirements: (a) Tests must be *valid*. That is, their behavior should be well defined by the specification of the verified system; (b) Test programs should also be of high *quality*, in the sense that they focus on potential bugs.

The number of potential locations of bugs in a system and the possible scenarios that can lead to their discovery is huge: In a typical architecture, there are from $10^{1,000}$ to $10^{10,000}$ programs of 100 instructions. It is impossible to exactly specify all the test programs that we would like to use out of the above combinations, and even harder to generate them. This means that users of test generators intentionally under-specify the requirements of the tests they generate, and expect the generators to fill in the gaps between the specification and the required tests. In other words, a test generator is required to explore the unspecified space and to help find the bugs for which the user is not directly looking (Hartman, Ur, & Ziv 1999).

There are two ways to explore this unspecified space, systematically or randomly. A systematic approach is impossible when the explored space is large and not well-understood. Therefore, the only practical approach is to generate pseudo-random tests. That is, tests that satisfy user requirements and at the same time uniformly sample the derived test space (Fournier, Arbetman, & Levinger 1999).

The validity, quality, and test specification requirements described above are naturally modeled through constraints (Bin *et al.* ; Chandra & Iyengar 1992). As an example of a validity constraint, consider the case of a translation table: $RA = trans(EA)$, where EA stands for the effective address and RA stands for the real (physical) address. For CSP to drive test program generation, the program should be modeled as constraint networks. The requirement to produce a large number of random, well-distributed tests is viewed, under the CSP modeling scheme, as a requirement to produce a large number of random solutions to a CSP. This stands in contrast to the traditional requirement of reaching a single solution, all solutions, or a 'best' solution (Dechter 1992; Kumar 1992).

Related work. The problem of generating random solutions for a set of constraints, in the context of hardware verification, is tackled in (Yuan *et al.* 1999). The Authors deal with Boolean variables and constraints over them, but do not use the CSP framework. Instead, they construct a single BDD that represents the entire search space, and develop a sampling method which uses the structure of the BDD. A BDD based constraint satisfaction engine imposes a restriction on the size of the problems that can be solved, since the BDD approach often requires exponential time and space. No approximation alternative was presented. As far as we know, in the CSP literature the task of random solution generation was not addressed.

Preliminaries

DEFINITION **1 (Constraint Networks)** *A* Constraint Network (CN) *is defined by a triplet* (X, D, C) *where* X *is a set of variables* $X = \{X_1, ..., X_n\}$*, associated with a set of discrete-valued domains,* $D = \{D_1, ..., D_n\}$*, and a set of constraints* $C = \{C_1, ..., C_m\}$*. Each constraint* C_i *is a pair* (S_i, R_i)*, where* R_i *is a relation* $R_i \subseteq D_{S_i}$ *where* $D_{S_i} = \bowtie_{X_j \in S_i} D_j$*, defined on a subset of variables* $S_i \subseteq X$ *called the scope of* C_i*. The relation denotes all compatible tuples of* D_{S_i} *allowed by the constraint. The constraint graph of a constraint network, has a node for each variable, and an arc between two nodes iff the corresponding variables participate in the same constraint. A* solution *is an assignment of values to variables* $x = (x_1, ..., x_n)$*,* $x_i \in D_i$*, such that no constraint is violated.*

EXAMPLE **1** *Consider a graph coloring problem that has four variables (A, B, C, D), where the domains of A and C are* $\{1, 2, 3\}$ *and the domains of B and D are* $\{1, 2\}$*. The constraint are not-equal constraints between adjacent variables. The constraint graph is given in Figure 4 (top, left).*

Belief networks provide a formalism for reasoning about partial beliefs under conditions of uncertainty.

DEFINITION **2 (belief network)** *Let* $X = \{X_1, ..., X_n\}$ *be a set of random variables over multi-valued domains,* $D_1, ..., D_n$*, respectively. A* belief network *is a pair* (G, P) *where* $G = (X, E)$ *is a directed acyclic graph over the variables, and* $\mathcal{P} = \{P_i\}$*, and* P_i *denote Conditional Probability Tables (CPTs)* $P_i = \{P(X_i|pa_i)\}$*, where* pa_i *is the set of* parents *nodes pointing to* X_i *in the graph. The belief network represents a probability distribution* $P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i|x_{pa_i})$ *where an assignment* $(X_1 = x_1, ..., X_n = x_n)$ *is abbreviated to* $x = (x_1, ..., x_n)$ *and where* x_S *denotes the restriction of a tuple* x *over a subset of variables* S*. The* moral graph *of a directed graph is the undirected graph obtained by connecting the parent nodes of each variable and eliminating direction.*

DEFINITION **3 (Induced-width)** An *ordered graph* is a pair (G, d) where G is an undirected graph and $d = X_1, ..., X_n$ is an ordering of the nodes. The *width of a node* in an ordered graph is the number of the node's neighbors that precede it in the ordering. The *width of an ordering d*, denoted $w(d)$, is the maximum width over all nodes. The *induced width of an ordered graph*, $w^*(d)$, is the width of the induced ordered graph obtained as follows: nodes are processed from last to first; when node X is processed, all its preceding neighbors are connected. The *induced width of a graph*, $w*$, is the minimal induced width over all its orderings (Arnborg 1985).

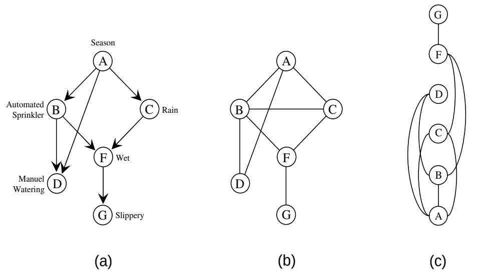


Figure 1: (a) Belief network (b) its moral graph (c) Induced width

EXAMPLE **2** *The network in Figure 1a expresses probabilistic relationship between 6 variables* A, B, C, D, F, G*. The moral graph is in Figure 1b and the induced-width along* $d = (A, B, C, D, F, G)$ *is 2, as shown in Figure 1c.*

The Random Solution Task

Given a constraint network $\mathcal{R} = (X, D, C)$ we define the uniform probability distribution $P_u(\mathcal{R})$ over X such that for every assignment $\vec{x} = (x_1, ..., x_n)$ to all the variables,

$$P_u(\vec{x}) = \begin{cases} \frac{1}{|sol|} & \text{if } \vec{x} \text{ is a solution} \\ 0 & \text{otherwise} \end{cases}$$

Where $|sol|$ is the number of solutions to $\mathcal{R}$. We consider in this paper the task of generating random solutions to a CSP, from a uniform distribution over the solution space. A *naive approach* to this task is to randomly generate solutions from the problem's search space. Namely, given a variable ordering, starting with the first variable X_1, we can randomly assign it a value from its domain (choosing each value with equal probability). For the second variable, X_2 we compute values that are consistent with the current assignment to X_1 and choose one of these values with equal probability, and so forth.

This approach is of course incorrect. Consider a constraint network over Boolean variables where the constraints are a set of implications: $\rho = \{A \to B,\ A \to C,\ A \to D,\ A \to E\}$. When applying the naive approach to this formula along the variable ordering A, B, C, D, E, we select the value $A = 1$ or $A = 0$ with equal probabilities of 1/2. If the value $A = 1$ is chosen, all the other variables must be set to 1, as there is a single solution consistent with $A = 1$. On the other hand, if $A = 0$ is generated, any assignment to the rest of the variables is a solution. Consequently, the naive approach generates solutions from the distribution:

$$P(a, b, c, d, e) = \begin{cases} 1/2 & \text{(a=1,b=1,c=1,d=1,e=1)} \\ 1/16 & \text{otherwise} \end{cases}$$

rather than from the uniform distribution $P_u(\rho)$. From the example it is clear that the naive method's accuracy will not improve by making the problem backtrack-free.

On the other extreme lies the *brute-force* approach. Namely, generate (by some means) all the solutions and subsequently uniformly at random select one. The brute-force approach is correct but impractical when the number of solutions is very large. We next present a range of schemes that lie between the *naive approach* and an exact approach, which permits trading accuracy for efficiency in anytime fashion.

Algorithm elim-count
Input: A constraint network $\mathcal{R} = (X, D, C)$, ordering d.
Output: Augmented output buckets including the intermediate count functions. The number of solutions.
1. **Initialize:** Partition C into $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all constraints whose latest (highest) variable is X_i. We denote a function in a bucket N_i, its scope S_i.)
2. **Backward:** For $p \leftarrow n$ downto 1, do
   Generate the function N^p: $N^p = \sum_{X_p} \prod_{N_i \in bucket_p} N_i$.
   Add N^p to the bucket of the latest variable in $\bigcup_{i=1}^{j} S_i - \{X_p\}$.
3. **Return** the number of solutions, N^1 and the set of output buckets with the original and computed functions.

Figure 2: Algorithm *elim-count*

Uniform Solution Sampling Algorithm

Our idea is to transform the constraint network into a belief network that can express the desired uniform probability distribution. Once such a belief network is available we can apply known sampling algorithms for belief networks (Pearl 1988). The transformation algorithm is based on a variable elimination algorithm that counts the number of solutions of a constraint network and the number of solutions that can be reached by extending certain partial assignments. Clearly, the task of counting is known to be difficult ($\#P$-complete) but when the graph's induced-width is small the task is manageable.

We describe the algorithm using the bucket-elimination framework. *Bucket elimination* is a unifying algorithmic framework for dynamic programming algorithms (Bertele & Brioschi 1972; Dechter 1999). The input to a bucket-elimination algorithm consists of relations (functions, e.g., constraints, or conditional probability tables for belief networks). Given a variable ordering, the algorithm partitions the functions into buckets, where a function is placed in the bucket of its latest argument in the ordering. The algorithm processes each bucket, from the last variable to the first, by a variable elimination operator that computes a new function that is placed in an earlier bucket.

For the counting task, the input functions are the constraints, expressed as cost functions. A constraint R_S over scope S is a cost function that assigns 0 to any illegal tuple and 1 otherwise. When the bucket of a variable is processed the algorithm multiplies all the functions in the bucket and sums over the bucket's variable. This yields a new function that associates with each tuple (over the bucket's scope excluding the bucket's variable) the number of extensions to the eliminated variable. Figure 2 presents algorithm *elim-count*, the bucket-elimination algorithm for counting. The complexity of elim-count obeys the general time and space complexity of bucket-elimination algorithms (Dechter 1999).

THEOREM **1** *The time and space complexity of algorithm elim-count is $O(n \cdot exp(w^*(d))$ where $n = |X|$ and $w^*(d)$ is the induced-width of the network's ordered constraint graph along d.* □

Let $\vec{x}_i = (x_1, ..., x_i)$ be a specific assignment to the first set of i variables and let N_k^j denotes a new function that resides in bucket X_k and was generated in bucket X_j, for $j > k$.

THEOREM **2** *Given an assignment $\vec{x}_i = (x_1, ..., x_i)$, the number of consistent extensions of $\vec{x}_i$ to full solutions is*[1] $\prod_{\{N_k^j | 1 \le k \le i,\ i+1 \le j \le n\}} N_k^j(\vec{x})$.

EXAMPLE **3** *Consider the constraint network of Example 1 and assume we use the variable ordering (D, C, B, A), the initial partitioning of functions into buckets is given in the table below in the middle column.*

Processing bucket A we generate the function (We use the notation N^X for the function generated by eliminating variable X) $N^A(B, D) = \sum_A R(A, B) \cdot R(A, D)$ and place it in the bucket of B. Processing the bucket of B we compute $N^B(C, D) = \sum_B R(B, C) \cdot N^A(B, D)$ and place it in the bucket of C. Next we process C generating the function $N^C(D) = \sum_C R(C, D) \cdot N^B(C, D)$ placed it in bucket D and finally we compute (when processing bucket D) all the solutions $N^D = \sum_D N^C(D)$. The output buckets are:

Bucket	*Original constraints*	*New constraints*
$bucket(A)$	$R(A, B),\ R(A, D)$	
$bucket(B)$	$R(B, C)$	$N^A(B, D)$
$bucket(C)$	$R(C, D)$	$N^B(C, D)$
$bucket(D)$		$N^C(D)$
$bucket(0)$		N^D

The actual N functions are displayed in the following table:

$N^A(b,d)$ *: (b, d)*	$N^B(c,d)$ *: (c, d)*	$N^C(d)$*: (d)*	N^D
2: (1,1) or (2,2)	*2 : (2,1) or (1,2)*	*5: (1)*	*10*
1: (1,2) or (2,1)	*3 : (3,1) or (3,2)*	*5: (2)*	
	1 : (1,1) or (2,2)		

We next show, using the example, how we use the output of the counting algorithm for sampling. We start assigning values along the order D, C, B, A. The problem has 10 solutions. According to the information in bucket D, both assignments $D = 1$ and $D = 2$ can be extended to 5 solutions, so we choose among them with equal probability. Once a value for D is selected (lets say $D = 1$) we compute the product of functions in the output-bucket of C which yields, for any assignment to D and C, the number of full solutions they allow. Since the product functions shows that the assignment $(D = 1, C = 2)$ has 2 extensions to full solutions, $(D = 1, C = 3)$ has 3 extensions while $(D = 1, C = 1)$ has none, we choose between the values 2 and 3 of C with ratio of 2 to 3. Once a value for C is selected we continue in the same manner with B and A. Algorithm *solution-sampling* is given in Figure 3. Since the algorithm operates on a backtrack-free network created by elim-count, it is guaranteed to be linear in the number of samples generated.

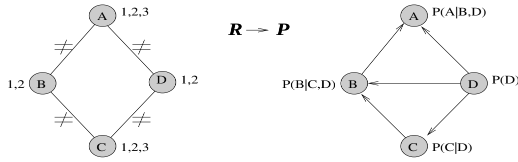
The Transformed Belief Network

Given a constraint network $\mathcal{R} = (X, D, C)$ and its output-buckets generated by elim-count applied along ordering d,

[1] We abuse notation denoting by $N(\vec{x})$ the function $N(\vec{x}_S)$, where S is the scope of N.

Algorithm solution-sampling
Input: A constraint network $\mathcal{R} = (X, D, C)$, an ordering d. The output buckets along d, produced by elim-count.
Output: Random solutions generated from $P_u(\mathcal{R})$.
1. **While** not enough solutions generated, do
2. For $p \leftarrow 1$ to n, do
Given the assignment $\vec{x}_p = (x_1, ...x_p)$ and $bucket_p$ with functions $\{N_1, N_2, \ldots\}$, compute the frequency function of $f(X_{p+1}) = \prod_j N_j(\vec{x}_p, X_{p+1})$ and generate sample for X_{p+1} according to f.
3. **Endwhile**.
4. **Return** the generated solutions.

Figure 3: Algorithm *solution-sampling*



$$A: \underbrace{R(A,B), R(A,D)}_{\searrow} \longrightarrow P(A|B,D) = \frac{R(A,B)\cdot R(A,D)}{\sum_A R(A,B)\cdot R(A,D)}$$

$$B: \underbrace{R(B,C), N^A(B,D)}_{\searrow} \longrightarrow P(B|C,D) = \frac{R(B,C)\cdot N^A(B,D)}{\sum_B R(B,C)\cdot N^A(B,D)}$$

$$C: \underbrace{R(C,D), N^B(C,D)}_{\swarrow} \longrightarrow P(C|D) = \frac{R(C,D)\cdot N^B(C,D)}{\sum_B R(C,D)\cdot N^B(C,D)}$$

$$D: N^C(D) \longrightarrow P(D) = \frac{N^C(D)}{\sum_D N^C(D)}$$

Figure 4: The conversion process

$B_{(\mathcal{R},d)}$ is the belief network defined over X as follows. The directed acyclic graph is the induced graph of the constraint problem along d where all the arrows are directed from earlier to later nodes in the ordering. Namely, for every variable X_i the parents of X_i are the nodes connected to it in the induced-graph that precede it in the ordering. The conditional probability table (CPT) associated with each child variable X_i can be derived from the functions in the output-bucket by

$$P(X_i|pa_i) = \frac{\prod_j N_j(X_i, pa_i)}{\sum_{X_i} \prod_j N_j(X_i, pa_i)} \tag{1}$$

where N_j are both the original and new functions in the bucket of X_i.

EXAMPLE **4** *In our example, the parents of A are B and D, the parents of B are D and C, the parents of C is D and D is a root node (see Figure 4). The CPTs are given by the following table:*

$P(a\|b,d)$	(d, b, a)	$P(b\|c,d)$	(c, d, b)	$P(c\|d)$	(d, c)
1/2	(1,1,2)	1	(1,1 or 2,2)	2/5	(1,2)
1/2	(1,1,3)	1	(2,1 or 2,1)	2/5	(2,1)
1/2	(2,2,1)	2/3	(3,1,1)	3/5	(1,3)
1/2	(2,2,3)	1/3	(3,1,2)	3/5	(2,3)
1	(1,2,3)	1/3	(3,2,1)		
1	(2,1,3)	2/3	(3,2,2)		

The conversion process is summarized in Figure 4.We can show

THEOREM **3** *Given a constraint network $\mathcal{R}$ and an ordering d of its variables, the Belief network $B_{(\mathcal{R},d)}$ (defined by Eq. 1), expresses the uniform distribution $P_u(\mathcal{R})$.*

Given a belief network that expresses the desired distribution we can now use well known sampling algorithms to sample solutions from the belief network. In particular, the simplest such algorithm that works well when there is no evidence, is *Logic sampling* (Henrion 1986). The algorithm samples values of the variables along the topological ordering of the network's directed graph. Given an assignment to the first $(i-1)$ variables it assigns a value to X_i using the probability distribution $P(X_i|pa_i)$, as the parents of X_i are already assigned. We can show that

Proposition 1 *Given a constraint network $\mathcal{R}$. Algorithm* solution-sampling *applied to the output-buckets of elim-count along d, is identical to logic sampling on the belief network $B_{(\mathcal{R},d)}$.* □

Mini-bucket Approximation for Sampling

The main drawback of bucket elimination algorithms is that, unless the problem's induced-width is bounded, they are time and space exponential. *Mini-Bucket Elimination* is an approximation designed to reduce the space and time problem of full bucket elimination (Dechter & Rish 1997) by partitioning large buckets into smaller subsets called mini-buckets, which are processed independently. Here is the rationale. Let $h_1, ..., h_j$ be the functions in $bucket_p$ and let $S_1, ..., S_j$ be the scopes of those functions. When *elim-count* processes $bucket_p$, it computes the function $N^p = \sum_{X_p} \prod_{i=1}^{j} N_i$. The scope of N^p is $U_p = \bigcup_{i=1}^{j} S_i - \{X_p\}$. The Mini-Bucket algorithm, on the other hand, creates a partitioning $Q\prime = \{Q_1, ..., Q_t\}$ where the mini-bucket Q_l contains the functions $\{N_{l_1}, ..., N_{l_k}\}$. We can rewrite the expression for N^p as follows: $N^p = \sum_{X_p} \prod_{l=1}^{t} \prod_{l_i} N_{l_i}$. Now, if we migrate the summation inside the multiplication we will get a new function g^p defined by: $g^p = \prod_{l=1}^{t} \sum_{X_p} \prod_{l_i} N_{l_i}$. It is not hard to see that g^p is an upper bound on N^p: $N^p \leq g^p$. Thus, the approximation algorithm can process each mini-bucket separately (by using the summation and product operators), therefore computing g^p rather than N^p.

A tighter upper-bound for N^p can be obtained by bounding all the mini-buckets but one, by a maximization instead of summation, namely, $N^p \leq (\sum_{X_p} \prod_{l_1} N_{l_1}) \cdot (\prod_{l=2}^{t} \max_{X_p} \prod_{l_i} N_{l_i})$. Alternatively, we can minimize over the mini-bucket's variables, except one mini-bucket to yield a lower bound (replacing max by min), or apply averaging, yielding a mean-value approximation.

The quality of the approximation depends on the degree of partitioning into mini-buckets. Given a bounding parameter i, the algorithm creates an i-partitioning, where each mini-bucket includes no more than i variables[2]. The choice of

[2] We assume that i is at least as large as the size of scopes of

Algorithm MBE-count(i)
Input: A constraint network $\mathcal{R} = (X, D, C)$ an ordering d; parameter i
Output: A bound on (upper, lower or approximate value) of the count function computed by elim-count. A bound on the number of solutions
1. **Initialize:** Partition the functions in C into $bucket_1, \ldots, bucket_n$
2. **Backward** For $p \leftarrow n$ downto 1, do
• Given functions $N_1, N_2, ..., N_j$ in $bucket_p$, generate an (i)-partitioning, $Q' = \{Q_1, ..., Q_t\}$. For Q_1 containing $N_{1_1}, ...N_{1_t}$ generate $N^1 = \sum_{X_p} \prod_{i=1}^{t} N_{l_i}$. For each $Q_l \in Q'$, $l > 1$ containing $N_{l_1}, ...N_{l_t}$ generate function N^l, $N^l = \Downarrow_{U_l} \prod_{i=1}^{t} N_{l_i}$, where $U_l = \bigcup_{i=1}^{j} scope(N_{l_i}) - \{X_p\}$ (where $\Downarrow$ is max, min or mean). Add N^l to the bucket of the largest-index variable in its scope.
4. **Return** the ordered set of augmented buckets and number of solutions.

Figure 5: Algorithm, MBE-count(i)

i-partitioning affects the accuracy of the mini-bucket algorithm. Algorithm *MBE-count(i)* described in Figure 5, is parameterized by this i-bound. The algorithm outputs not only a bound on the number of solutions but also the collection of augmented buckets. It can be shown ((Dechter & Rish 1997)),

THEOREM **4** *Algorithm MBE-count(i) generates an upper (lower) bound on the number of solutions and its complexity is time $O(r \cdot exp(i))$ and space $O(r \cdot exp(i-1))$, where r is the number of functions.* □

EXAMPLE **5** *Considering our example and assuming $i = 2$, processing the bucket of A we have to partition the two functions into two separate mini-buckets, yielding two new unary functions: $N^A(B) = \sum_A R(A, B)$, $N^A(D) = \max_A R(A, D)$ placed in bucket B and bucket D, respectively (only one, arbitrary, mini-bucket should be processed by summation). Alternatively, we get $N^A(D) = 0$ if we process by* min *operator, $N^A(D) = 2$ by summation and $N^A(D) = \frac{1}{|D(A)|} \cdot \sum_A R(A, D) = 2/3$, by mean operator. Processing bucket B we compute $N^B(C) = \sum_B R(B, C) \cdot N^A(B)$ placed in bucket of C, and processing C generates $N^C(D) = \sum_C R(C, D) \cdot N^B(C)$ placed in bucket D. Finally we compute, when processing bucket D, an upper bound on the number of solutions $N^D = \sum_D N^C(D) \cdot N^A(D)$. The output buckets are given by the table below*

Bucket	*Original constraints*	*New constraints*
$bucket(A)$	$R(A, B),\ R(A, D)$	
$bucket(B)$	$R(B, C)$	$N^A(B)$
$bucket(C)$	$R(C, D)$	$N^B(C)$
$bucket(D)$		$N^A(D),\ N^C(D)$
$bucket(0)$		N^D

input functions.

The actual N functions using the max operator are:

$$N^A(b) = 2 \quad N^B(c) = \begin{cases} 2 & \text{if (c=1)} \\ 2 & \text{if (c=2)} \\ 4 & \text{if (c=3)} \end{cases}$$

$$N^A(d) = 1 \quad N^C(d) = \begin{cases} 6 & \text{if (d=1)} \\ 6 & \text{if (d=2)} \end{cases} \quad N^D = 12$$

We see that the bound is quite good. Note that had we processed both mini-bucket by summation we would get a bound of 24 on the number of total solutions, 0 solutions using min and 8 solutions using the mean operator.

Given a set of output buckets generated by MBE-count(i) we can apply algorithm solution-sampling as before. There are, however, a few subtleties here. First we should note that the sample generation process is *no longer* backtrack-free. Many of the samples can get into a "dead-end" because the generated network is not backtrack-free. Consequently, the complexity of solution-sampling algorithm is no longer output linear. The lower the i-bound, the larger the time overhead per sample.

Can we interpret the sampling algorithm as sampling over some belief network? If we mimic the transformation algorithm used in the exact case, we will generate an *irregular* belief network. The belief network generated is irregular in that it is *not* backtrack-free, while by definition, all belief networks are backtrack-free, because regular CPTs must sum to 1.

An irregular belief network includes an irregular CPT $P(X_i|pa_i)$, where there could be an assignment to the parents pa_i such that $P(x_i|t_{pa_i}) = 0$ for *every* value x_i or X_i. An irregular belief network represent the probability distribution $P(x_1, ..., x_n) = \alpha \cdot \prod_{i=1}^{n} P(x_i|x_{pa_i})$ where α is a normalizing constant. It is possible to show that the sampling algorithm that follows MBE-count(i) is identical to logic sampling over such an irregular network created by the transformation applied to the output buckets generated by MBE-count(i).

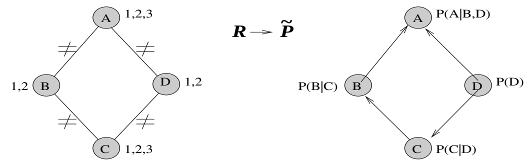
EXAMPLE **6** *The belief network that will correspond to the mini-bucket approximation still has D, B as the parents of A, C is the parent of B and D is the parent of C. The probability functions that can be generated for our example are given below. For this example, sampling is backtrack-free. For variable A, after normalization, we get the same function as the exact one (see Example 4). The CPT for B, C and D are:*

$P(b\|c)$	*(c,b)*	$P(c\|d)$	*(d,c)*	$P(d)$	*(d)*
1	*(1,2) or (2,2)*	*1/3*	*(1,2) or (2,1)*	*1/2*	*(1)*
1/2	*(3,1) or (3,2)*	*2/3*	*(1,3) or (2,3)*	*1/2*	*(2)*

It is interesting to note that if we apply the sampling algorithm to the initial bare buckets, which can be perceived to be generated by MBE-count(0), we just get the naive-approach we introduced at the outset.

Empirical evaluation

We provide preliminary empirical evaluation, demonstrating the anytime behavior of the mini-bucket scheme for sampling. We used as benchmark randomly generated binary CSPs generated according to the well-known four-parameter



$$
\begin{array}{llcll}
A: & \underbrace{R(A,B)}, \underbrace{R(A,D)} & \rightarrow & P(A|B,D) = & \frac{R(A,B)\cdot R(A,D)}{\sum_A R(A,B)\cdot R(A,D)} \\
B: & \underbrace{R(B,C), N^A(B)} & \rightarrow & P(B|C) = & \frac{R(B,C)\cdot N^A(B)}{\sum_B R(B,C)\cdot N^A(B)} \\
C: & \underbrace{R(C,D), N^B(C)} & \rightarrow & P(C|D) = & \frac{R(C,D)\cdot N^B(C)}{\sum_B R(C,D)\cdot N^B(C)} \\
D: & N^C(D), N^A(D) & \rightarrow & P(D) = & \frac{N^C(D)\cdot N^A(D)}{\sum_D N^C(D)\cdot N^A(D)}
\end{array}
$$

Figure 6: The conversion process

model (N, K, C, T) where N is the number of variables, K is the number of values, T is the tightness (number of disallowed tuples) and C is the number of constraints. We also tested the special structure of square grid networks.

Measures. We assume that the accuracy of the distribution obtained for the first variable is representative of the accuracy of the whole distribution. We therefore compare the approximated distribution associated with the first variable in the ordering (the probability of the first node in the ordering) computed by MBE-count(i) against its exact distribution, using several error measures. The primary measure is the KL distance which is common for comparing the distance between two probabilities (Chow & Liu 1968). Let $P(x)$ be a probability distribution and $P_a(x)$ its approximation. The KL-distance of P and P_a is defined as $KL(P, P_a) = \sum_x P(x)log(P(x)/P_a(x))$. It is known that $KL(P, Pa) \geq 0$, and the smaller $KL(P, P_a)$, the closer $P_a(x)$ is to $P(x)$, with $KL(P, P_a) = 0$ iff $P = P_a$. We also compute the absolute error[3] and the relative error[4]. Finally, for comparison we also compute the KL distance between the exact distribution and the uniform, KL_u of the first variable.

Benchmarks and results. We experimented with random problems having 40 and 50 variables with domains of 5 values and 8x8 grids with domains of 5 values. All problems are consistent. We had to stay with relatively small problems in order to apply the exact algorithm. The results with random CSPs are given in Tables 1 and 2, and results with 8x8 grids are given in Table 3. In the first column we have tightness T, in the second column the KL-distance between the exact distribution and uniform distribution (KL_u), and in the remaining columns various values of the i-bound. First we report the average time of MBE-count(i) per problem for each i. The remainder of the table consists of horizontal blocks, corresponding to different values of T. In columns corresponding to values of i-bound, we report, for each value of i, KL-distance between the exact probability and MBE(i) probability (KL_i), absolute error and relative error, averaged over all problem instances.

[3] $\epsilon_{abs} = \sum_i |P(x = i) - P_a(x = i)|/K$.

[4] $\epsilon_{rel} = \sum_i (|P(x = i) - P_a(x = i)|/P(x = i))/K$.

$N = 40, K = 5, C = 90.$ w^*=10.8. 20 instances.								
		i=4	i=5	i=6	i=7	i=8	i=9	i=10
time		0.05	0.09	0.33	1.3	5.2	20	86
T	KL_u	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e
8	0.398	0.223 0.106 1.56	0.184 0.095 1.13	0.144 0.081 0.86	0.086 0.058 0.65	0.091 0.058 0.64	0.063 0.045 0.48	0.020 0.026 0.21
9	0.557	0.255 0.110 37	0.323 0.125 28	0.303 0.112 23	0.132 0.074 5.16	0.109 0.064 1.76	0.082 0.053 0.99	0.085 0.045 0.61
10	0.819	0.643 0.164 28	0.480 0.124 7.51	0.460 0.123 9.41	0.340 0.108 5.41	0.295 0.105 4.31	0.401 0.098 2.69	0.228 0.064 0.81
11	1.237	0.825 0.203 1.33	0.803 0.184 1.65	1.063 0.209 2.71	0.880 0.166 1.15	0.249 0.088 0.88	0.276 0.098 1.24	0.193 0.068 0.33

Table 1: Accuracy and time on Random CSPs.

$N = 50, K = 5, C = 110.$ w^*=12.7. 10 instances.							
		i=6	i=7	i=8	i=9	i=10	i=11
time		0.44	1.64	6.60	29	125	504
T	KL_u	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e
10	1.044	0.372 0.127 52	0.599 0.147 120	0.442 0.135 81	0.631 0.100 79	0.295 0.098 8.12	0.278 0.041 0.91
10.5	0.923	0.502 0.150 1.97	0.285 0.109 1.93	0.137 0.069 0.44	0.215 0.073 1.60	0.214 0.079 1.28	0.464 0.143 3.73
11	1.246	0.781 0.208 116	0.851 0.186 81	0.550 0.156 44	0.490 0.134 91	1.670 0.177 100	- - -
11.5	1.344	0.577 0.160 5.69	0.660 0.180 3.40	0.333 0.180 3.02	0.231 0.061 2.70	0.088 0.042 0.94	- - -

Table 2: Accuracy and time on Random CSP.

The first thing to observe from the tables is that even the weakest (but most efficient) level of approximation is superior to the naive uniform distribution, sometimes substantially. We also see from Table 1 that as i increases, the running time of MBE(i) increases, as expected. Looking at each horizontal block, corresponding to a specific value of T, we see that as i increases, the KL-distance as well as absolute and relative error decrease. For large values of i, KL_i is as much as an order of magnitude smaller than KL_u, indicating the quality of the probability distribution computed by MBE-count(i). We see similar results from Table 2.

Conclusion

The paper introduces the task of 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 are based on exact and approximate variable elimination algorithms for counting the number of solutions, that can be viewed as transformation of constraint networks into belief networks.

The result is a spectrum of parameterized anytime algorithms controlled by an i-bound, that, starts with the naive approach on one end and the exact approach on the other. As i increases, we are likely to have more accurate sam-

8x8 grid, $K = 5$. w^*=10. 25 instances.								
		i=4	i=5	i=6	i=7	i=8	i=9	i=10
time		0.01	0.04	0.12	0.56	1.8	5.7	16
T	KL_u	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e	KL_i abs-e rel-e
5	0.013	0.001 0.016 0.091	3e-4 0.008 0.044	2.3e-5 0.002 0.012	2.2e-5 0.002 0.010	1e-6 3.3e-4 0.002	0 4.6e-5 2.7e-4	0 1e-6 4e-6
7	0.022	0.002 0.021 0.133	7.2e-4 0.012 0.076	1.1e-4 0.005 0.026	1.2e-4 0.005 0.025	7e-6 0.001 0.006	0 2.1e-5 0.001	0 4e-6 3e-5
9	0.049	0.009 0.045 0.440	0.002 0.022 0.215	4.1e-4 0.009 0.069	2.8e-4 0.007 0.056	3.8e-5 0.003 0.020	5e-6 0.001 0.006	0 6e-5 5e-4
11	0.073	0.020 0.060 1.63	0.005 0.031 0.342	0.003 0.021 0.265	0.002 0.017 0.223	6.8e-4 0.009 0.118	9.4e-5 0.003 0.039	1e-6 3e-4 0.003

Table 3: Accuracy and time on 8x8 grid CSPs.

ples that takes less overhead to generate during the sampling phase. Our preliminary evaluations show that the scheme provides substantial improvements over the naive approach even when using its weakest version. More importantly, it demonstrate the anytime behavior of the algorithms as a function of the i-bound. Further experiments clearly should be conducted on the real application of test program generation. In the future we still need to test the sampling complexity of the approximation and its accuracy on the full distribution.

Our approach is superior to ordinary OBDD-based algorithms which are bounded exponentially by the *path-width*, a parameter that is always larger than the induced-width. However, another variants of BDDs, known as tree-BDDs (McMillan 1994) extends OBDDs to trees that are also time and space exponentially bounded in the induced-width (also known as tree-width). As far as we know all BDD-based algorithms for random solution generation, use ordinary OBDDs rather than tree-BDDS.

The main virtue of our approach however is in presenting an anytime approximation scheme which is so far unavailable under the BDD framework.

Acknowledgement

This work was supported in part by NSF grant IIS-0086529 and by MURI ONR award N00014-00-1-0617.

References

Arnborg, S. A. 1985. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT* 25:2–23.

Bergeron, J. 2000. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers.

Bertele, U., and Brioschi, F. 1972. *Nonserial Dynamic Programming*. Academic Press.

Bin, E.; Emek, R.; Shurek, G.; and Ziv, A. What's between constraint satisfaction and random test program generation. Submitted to IBM System Journal, Aug. 2002.

Chandra, A. K., and Iyengar, V. S. 1992. Constraint solving for test case generation. In *International Conference on Computer Design, VLSI in Computers and Processors*, 245–248. Los Alamitos, Ca., USA: IEEE Computer Society Press.

Chow, C. K., and Liu, C. N. 1968. Approximating discrete probability distributions with dependence trees. *IEEE Transaction on Information Theory* 462–67.

Dechter, R., and Rish, I. 1997. A scheme for approximating probabilistic inference. In *Proceedings of Uncertainty in Artificial Intelligence (UAI'97)*, 132–141.

Dechter, R. 1992. Constraint networks. *Encyclopedia of Artificial Intelligence* 276–285.

Dechter, R. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113:41–85.

Fournier, L.; Arbetman, Y.; and Levinger, M. 1999. Functional verification methodology for microprocessors using the Genesys test program generator. In *Design Automation & Test in Europe (DATE99)*, 434–441.

Hartman, A.; Ur, S.; and Ziv, A. 1999. Short vs long size does make a difference. In *HLDVT*.

Henrion, M. 1986. Propagating uncertainty by logic sampling. In *Technical report, Department of Engineering and Public Policy, Carnegie Melon University*.

Kumar, V. 1992. Algorithms for constraint-satisfaction problems: A survey. *A.I. Magazine* 13(1):32–44.

McMillan, K. L. 1994. Hierarchical representation of discrete functions with application to model checking. In *Computer Aided Verification, 6th International conference, David L. Dill ed.*, 41–54.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Yuan, J.; Shultz, K.; Pixley, C.; Miller, H.; and Aziz, A. 1999. Modeling design constraints and biasing in simulation using BDDs. In *International Conference on Computer-Aided Design (ICCAD '99)*, 584–590. Washington - Brussels - Tokyo: IEEE.