Generating Solutions to Constraint Satisfaction Problems with Arbitrary Random Biasing

David Larkin

University of California, Irvine

1 Introduction

In this abstract we present a method of generating solutions to a constraint satisfaction problem according to an arbitrary probability distribution. If S is the set of solutions, and P is a probability density function, solution x should be generated with probability $P(x)/\sum_{y\in S} P(y)$. We allow P to be specified directly, in the form of a Bayesian network, or indirectly, as a set of probabilistic biases.

Biases are judgments on the probability of certain events. They delimit a set of consistent probability distributions. For example, if the bias $P(\neg X_1) = 1/2$ is given, then one consistent distribution on the three binary variables $\{X_1, X_2, X_3\}$ might give equal probability to each of the $2^3 = 8$ possible assignments. Another consistent distribution might give probability $1/2$ to the event in which all variables are true and $1/2$ to the event in which they are all false. In general, many distributions may be consistent with a given set of biases.

So given a fully or partially specified probability function, the task is to generate solutions in accordance with it. This has application in the generation of test programs for functional verification, in which the space of test programs to explore may be characterized by a set of probabilistic and deterministic constraints [4, 5, 14, 11]. When a piece of hardware is being tested for bugs, a large volume of programs is required to run on it. Often these are generated by computer [2]. Each potential test program is associated with a set of bugs it might uncover, but the distribution of bugs in program space is highly uneven. In general they will be highly concentrated in a few key areas which exercise some special functionality of the device being tested. For example, consider an AND gate with 32 inputs. The stuck-at-zero fault would only appear when all the inputs are one, which is one out of 2^{32} possible programs. A pair of typical, random programs with about equally many ones and zeroes are unlikely to be very different as far as the device is concerned. So the search for bugs via testing, in which elements of the program space are randomly sampled in order to uncover faults, has to be directed to the high-risk areas which are likely to exercise special logic and functionality. For this two forms of information are supplied by the user. First is a set of probabilistic biases, for example $P(\text{NextInstruction} = \text{ADD}) = 0.25$. These may be defined on one or several variables and give different weights to different assignments in order the emphasize the corner cases and deemphasize the large number of routine cases. Next is given a set of deterministic constraints. After various regions of the test program space have been magnified or demagnified via the biasing process, "illegal" regions are cut out entirely with the deterministic constraints. For example, if an FPU is being tested, instructions that do not interact in any way with this unit might be disallowed. The remaining, legal program space is then rescaled (normalized) so that the relative probabilities of any pair of legal test programs will be the same both before and after the incorporation of the deterministic constraints. After this, suitable test programs are generated by drawing random elements from this space.

Our algorithm will be divided into two parts. The first problem is finding a probability distribution P that satisfies the biases. It is to be encoded economically in a Bayesian network. Then, the deterministic constraints are to be considered. The probability of every assignment that violates the constraints is to be set to zero, and the other probabilities are to be normalized. This will result in a modified Bayesian network. Using an algorithm such as logic sampling [8], well-distributed solutions are to be generated from this network.

In the following sections we give the details of this algorithm. After this introduction, in Section 2 we cover the problem of generating an encoding of a consistent probability distribution. In this Section we consider only the probabilistic biases and not the deterministic constraints. We also briefly develop an approximation scheme. In Section 3 we show how the distribution computed in the previous section can be modified to reflect the deterministic constraints. This may increase the complexity of the network. An approximation method for this is also outlined. Finally in Section 4 we discuss applications, related work, and conclude.

2 Generating a Consistent Distribution

In the following we assume without loss of generality that we are dealing with a set of binary variables $X = \{X_1, X_2, ..., X_n\}$.

Definition 1. *A bias is a statement of the form $P(W = w) = \alpha$, where $\alpha \in [0..1]$, $W \subseteq X$, and w is an assignment to the variables in W. The equality may also be replaced by an inequality, and α by a linear expression in some variables. W is called the scope of the bias and w is called its assignment. A probability distribution is consistent with a set of biases if the probability it gives to every assignment w to a scope W satisfies the corresponding bias.*

In this Section we consider the problem of generating a probability distribution that is consistent with a set of biases. The closely related problem of probabilistic logic, as rediscovered by [13], is to determine whether a set of biases is consistent (i.e., is there a probability distribution that satisfies them), or what bounds on an unknown probability are implied by a set of such biases. These tasks are known to be NP complete even when the biases only involve pairs of binary variables [10]. There are two well-known methods of attacking this problem: global optimization [13, 12] and local deduction [7]. Local deduction repetitively applies inference rules in order to derive implied probabilities from known ones. The final result is reached by following a sequence of steps in which only a small fraction of the knowledge base is consulted at each point. These procedures typically suffer from incompleteness or difficult proof procedures. Global optimization, on the other hand, derives its result in a single step by considering the entire knowledge base at once. It is accurate and algorithmically simple, and works by actually deriving a representation of the entire space of consistent distributions (from which it is simple to generate one, if an explicit solution is desired). However it generally has a complexity which is exponential in the number of variables and cannot even be attempted except for small problems.

In this Section we outline a method of generating a satisfying distribution in the global optimization style, which also implicitly solves the problem of determining consistency. Its complexity is exponential in the induced width of the bias graph. Arnborg [1] gives an algorithm with similar complexity for this same problem. However he does not discuss approximations, as we will, and his method is somewhat more complex in that it relies upon a clique tree instead of the one-dimensional variable elimination method. In the succeeding Section we will address the problem of using this distribution to sample CSP solutions.

The algorithm basically works as follows. First, we generate the bias graph. This is a graph with a node for every variable and an edge connecting every pair of variables appearing in the same bias scope W. This is then triangulated. The variables are ordered somehow, then processed from last to first. When variable X_i is processed, all of its preceding neighbors are connected. We use Q_i to denote the clique in the triangulated graph consisting of X_i and all of its preceding neighbors. We also call this graph the induced graph. The width (also called the induced width) of the induced graph is the maximum number of preceding neighbors any variable has .

For every assignment a_j to the variables in Q_i, we introduce a variable p^i_j. We may also write out the assignment explicitly in the subscript, as in $p^i_{Q_i=a_j}$. This variable represents the probability of the event $Q_i = a_j$ in the distribution we are going to construct. We will generate a linear program in these variables in such a way that any solution can be transformed directly into the Bayesian network we require that satisfies the biases. To do this we introduce three types of constraints. First, $\forall i, j. p^i_j \geq 0$ and $\forall i. \sum_j p^i_j = 1$. These ensure that assignments to the variables that are inconsistent with the basic laws of probability are not allowed. Second, we produce constraints that enforce the biases. For every bias $P(W = w) = \alpha$ in our input, we find a clique Q_i such that $W \subseteq Q_i$ (there always is one by definition), then we represent the bias as a constraint on the clique probabilities. We do this with the equality $\sum_a p^i_{W=w,\{Q_i-W\}=a} = \alpha$. For example, if we have a clique $Q_1 = \{X_1, X_2\}$ and a bias $P(\neg X_1) = 1/3$, we introduce the constraint $\sum_{b_2} p^1_{X_1=F,X_2=b_2} = 1/3$. Lastly, we bring in consistency requirements. For each clique $Q_i = \{X_i\} \cup N_i$, let X_k be the highest ordered variable in N_i. By definition $N_i \subseteq Q_k$. Consistency between the distributions on

Q_i and on Q_k is enforced with the constraint

$$\forall u. \sum_a p^k_{N_i=u,\{Q_k-N_i\}=a} = \sum_{b_i} p^i_{N_i=u,X_i=b_i}$$

This completes the definition of the linear program. Once this is solved with an algorithm such as Simplex, two results are possible. First, Simplex might report that the system is inconsistent, in which case so are the biases. Second, Simplex might return a satisfying assignment. In this case we can map this to a Bayesian network with the equations

$$\forall i, b, u. P(X_i = b|N_i = u) = \frac{p^i_{X_i=b,N_i=u}}{\sum_{b_i} p^i_{X_i=b_i,N_i=u}}$$

It can be shown that this network represents a distribution that satisfies the biases.

This algorithm can be easily represented in the bucket elimination framework, as we did in [11]. Its complexity is exponential in the induced width of the bias graph. This may be intractable for large and highly connected problems. Therefore a less accurate but less expensive computation may be desired. This can be done as follows, where the complexity is controlled in an anytime style with a parameter k. For every variable X_i, let Q'_i be the set containing X_i and all of its preceding neighbors in the regular (non-triangulated) bias graph. Choose a set W_i such that $Q'_i \subseteq W_i \subseteq Q_i$ and $|W_i| \leq k$. We will introduce variables p^i_j for every assignment to W_i. As before, we generate type-one constraints $\forall i, j. p^i_j \geq 0$ and $\forall i. \sum_j p^i_j = 1$. For type two, we note that the scope W of every bias is a subset of some W_i. Therefore we can enforce it with the same sort of constraint on the distribution of W_i as before. The approximation comes in with the third type of constraint. Let X_i be a variable and $Q_i = \{X_i\} \cup N_i$ be its clique in the induced graph. We partition N_i into sets $\{N^1_i, N^2_i, ..., N^m_i\}$ such that for all j, $N^j_i \subseteq W_k$ for some $k < i$. For each such N^j_i we generate the constraint $\forall u. \sum_a p^k_{N^j_i=u,\{W_k-N^j_i\}=a} = \sum_b p^i_{N^j_i=u,\{Q_i-N^j_i\}=b}$ Thus we replace the single constraint generated by the exact method with possibly many constraints, which in their combined influence are weaker than the exact constraint. Thus the approximate method will never introduce inconsistency, but it may introduce inaccuracy. The encoding of the final distribution in the Bayesian network is produced the same way as before.

3 Generating Well Distributed CSP Solutions

In this section we consider the problem of generating solutions from a CSP in accordance with a probability distribution P supplied in the form of a Bayesian network. If x is a solution, we want to generate it with probability $P(x)/\sum_{y \in \text{Soln's}} P(y)$. To do this, we modify the Bayesian network representing P so that it assigns zero probability to every variable instantiation that is not a solution of the CSP and normalizes the rest. Once the new network is constructed, a method such as logic sampling [8] can be used to draw well-distributed solutions. So the problem reduces to modifying the network, which is done as follows. First all the constraints in the CSP are represented as boolean functions and all the CPTs representing P are considered functions with values between 0 and 1. Setting F to be the union of all CPTs and constraints, we process the variables from last to first as follows. When processing X_i, we remove all functions from F mentioning X_i in their scope. We define and save the function B_i consisting of the product of all these. Let $Q_i = N_i \cup \{X_i\}$ be B_i's scope. Then we generate the function f^i such that $\forall u. f^i(N_i = u) = \sum_{b_i} B_i(X_i = b_i, N_i = u)$ and put it back in F. When we are finished, we generate the CPTS

$$\forall i, b, u. P(X_i = b | N_i = u) = \frac{B_i(X_i = b, N_i = u)}{f^i(N_i = u)}$$

These represent the modified network.

This process can be approximated with the mini-bucket method [6]. It is used in almost the same way as in [5]. As in that paper, an "irregular Bayesian network" is created which defines a probability distribution in which the sum of the probabilities of all events may not add up to one. As variables are instantiated according to this network's CPTs in the logic sampling process, a deadend may be reached in which some variable has zero probability of assuming *any* value given the current instantiation of its parents. In this case backtracking must occur. So the solution in this case is generated by a standard CSP solver, but the mini-buckets produced by the approximation algorithm can be used as value-ordering heuristics in order to approximate generation of a solution over the required distribution.

4 Conclusion

In this abstract we presented a method of generating solutions of a CSP in accordance with a partially or fully defined probability distribution. Our exact algorithms were exponential in the induced width of the constraint graphs, so we also outlined approximation methods. This has application in the generation of test programs for functional verification [14, 5].

The problem of generating solutions to a CSP in some kind of well-distributed way for functional verification has been addressed before. In [14], the authors present a method which allows biases to be specified over individual variables. They express the need to allow biases to be defined over sets of variables, but note that the problem is NP-complete and leave the question of suitable approximations or heuristics open. In this work, we provide exact algorithms which are exponential in the induced width of the constraint graph, as well as approximations. An algorithm with similar complexity, which also allows approximations, was introduced in [5]. However that work considers only the problem of generating uniformly-distributed solutions. A more ad-hoc method was presented in [3], in which a random solution is approximated by using an ordinary CSP solver with random variable and value-ordering strategies. Earlier work [4] also used an ordinary CSP solver, in which the distribution is made approximately uniform over a few key variables by picking random values for those variables first. As far as we know, we are the first to consider the problem of generating CSP solutions according to an arbitrary probability distribution.

Directions for future work include implementing these algorithms and testing their efficiency/accuracy on a variety of real and random problems. It would be interesting to find good heuristics to guide the mini-buckets partitioning of both approximate methods. Also, the current method of choosing a consistent distribution among those possible is fairly arbitrary. It would be useful to find a more principled method such as choosing the distribution with maximum entropy [9].

References

[1] Stefan Arnborg. Learning in prevision space. In *First International Symposium on Imprecise Probabilities and Their Applications*, 1999.

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

[3] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using constraint satisfaction formulations and solution techniques for random test program generation. *IBM System Journal*, August 2002.

[4] A. K. Chandra and V. S. Iyengar. Constraint solving for test case generation. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 245–248, 1992.

[5] Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *Proceedings of AAAI*, 2002.

[6] Rina Dechter and Irina Rish. A scheme for approximating probabilistic inference. In *Proc. of the Conference on Uncertainty in Artificial Intelligence*, pages 132–141, 1997.

[7] H. Thöne et al. Towards precision of probabilistic bounds propagation. In *Proc. of the 8th Conf. on Uncertainty in Artificial Intelligence*, pages 315–322, 1992.

[8] M. Henrion. Propagating uncertainty by logic sampling. Technical report, Carnegie Mellon University, 1986.

[9] E. T. Jaynes. Where do we stand on maximum entropy? In *The Maximum Entropy Formalism*. M. I. T. Press, 1979.

[10] Daphne Koller and Nimrod Megiddo. Constructing small sample spaces satisfying given constraints. *SIAM Journal of Discrete Mathematics*, 7(2):260–274, May 1994.

[11] David Larkin. Generating solutions to constraint satisfaction problems with arbitrary random biasing. In *First International Workshop on Constraints in Formal Verification (submitted)*, 2002.

[12] C. Luo, C. Yu, J. Lobo, G. Wang, and T. Pham. Computation of best bounds of probabilities from uncertain data. *Computational Intelligence*, 12(5), November 1996.

[13] N. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–88, 1986.

[14] Jun Yuan, Kurt Schultz, Carl Pixley, Hiller Miller, and Adnan Aziz. Modeling design constraints and biasing using BDDs in simulation. In *Proc. of International Conference on Computer-Aided Design*, November 1999.