Estimating Min-Entropy Using Probabilistic Graphical Models

Darryl Buller (IAD/NSA), Aaron Kaufer (IAD/NSA)

April 15, 2016

Abstract

This paper presents a method for using probabilistic graphical models to discover dependency relationships among the bit positions in a sample of multi-bit blocks of binary data. Since it is computationally infeasible to exhaustively test for all possible dependency combinations, a heuristic is needed. We use Bayesian networks to represent possible dependence structures and a genetic algorithm to evolve increasingly better-scoring Bayesian networks that best identify the statistical dependencies in the sample. A technique for graphical models is used to estimate the min-entropy of the unknown underlying joint probability distribution based on the best-scoring Bayesian network. This method has shown success for block sizes of several dozen bits, where the output spaces cannot be easily decomposed into smaller spaces for independent analysis. Future optimizations may allow for analysis of even larger block-size entropy sources

1 Background

When analyzing a non-deterministic random bit generation (RBG) process, it is important to quantify the entropy provided by the underlying noise sources. There are a number of tools available for this problem, including the entropy estimators proposed in [1], and various other specialized techniques that operate under the assumption that the noise source follows certain types of general statistical models. The success of these approaches depends on the ability to accurately estimate the conditional and unconditional probabilities that compose the appropriate statistical model of the source. In general, this requires that the basic output unit of the source not be excessively large. Otherwise the underlying dependence structure might be too large to be accurately reflected by any sample data set of reasonable size. For example, if a noise source produces 64-bit outputs, the probability distribution on the set of possible values cannot be computed and stored in memory unless the vast majority of 64-bit values can never occur.

In many practical applications the block size is sufficiently small, allowing entropy to be accurately estimated. However, there are some applications where this is not the case, and a different approach is needed. In this paper we describe an approach that uses Bayesian networks to represent the dependence relationships among the individual bits of a data block. In order to find a Bayesian network that best reflects these dependence relationships, we use a genetic algorithm to explore the space of possibilities. This approach has so far shown promising results.

2 Entropy Testing

The concept of entropy is used to assess the quality of a noise source. Roughly speaking, entropy measures the unpredictability of output generated from a particular source. Sources with higher entropy are considered to be more unpredictable than lower entropy sources. Although there are several definitions of entropy, each appropriate for different situations, the min-entropy statistic is typically computed to assess the randomness of the output of sources used to derive cryptographically strong values. Given the distribution of outputs produced by a particular source, the min-entropy of this source is $-\log_2(p_{\max})$, where $p_{\max}$ is the probability of the most likely output. Min-entropy is favored for cryptographic applications because it is an estimator of the average work, measured in bits, needed to correctly guess a particular output produced from a source, as described in Appendix D of [1]. Estimating the min-entropy of sources that produce small, or reduced, output sizes (i.e., at most 8 bits) is the main focus of NIST's Special Publication (SP) 800-90B [1].

3 Preliminary Graphical Models

The motivating question behind this paper is how to reliably estimate the min-entropy of noise source outputs that consist of a large number of bits. Since accumulating enough outputs to get a complete sense of the distribution is typically impossible for sufficiently large sizes, we have to look for an alternative approach. The most straightforward approach that we have considered is to model the dependency structure between bits within fixed positions of the output. For instance, suppose we have 12-bit output blocks. We could ask questions of the form:

- Does the 1st bit depend on the 9th bit?
- Do the 3rd, 6th, and 11th bits influence the 4th and 8th bits?

The following sections present increasingly sophisticated graphical model approaches to solving this problem. The first idea considered accounts for dependencies among subsets of bits, but lacks the ability to optimally configure these subsets or the dependence relationships among them. We then build on this foundation with a technique that determines which of a certain class of models best represents the dependence structure among the bit position subsets. Finally, we move to a generalized family of models capable of representing a very broad class of approximations to the joint probability distribution of the data.

3.1 A Basic Model

One of the simplest ways to address this problem is to partition the bits of the output space into subsets and consider these subsets as random variables. Suppose we have an n-bit output space $\{b_1, b_2, ..., b_n\}$, where b_i is a random variable representing the bit in the ith position of an observed output block. Given a partition of these bits composed of the subsets $X_1, X_2, ..., X_m$, we can construct the following conditional distribution as an approximation to the joint distribution of these bits

$$P(b_1, b_2, ..., b_n) \approx P(X_1) \cdot P(X_2|X_1) \ldots P(X_{m-1}|X_{m-2}) \cdot P(X_m|X_{m-1}).$$

$P(X_k|X_{k-1})$ is a conditional probability distribution that can be written as a $2^u \times 2^v$ table given that X_{k-1} and X_k consist of u and v bits respectively. We define $P_k(y|x)$ to be the probability of $X_k = y$ given $X_{k-1} = x$ in $P(X_k|X_{k-1})$. Likewise, $P_k(x)$ is the probability that $X_k = x$. We note that this approximation is in fact a valid distribution of $X_1, X_2, ...X_m$ because it sums to one, as seen from the fact that the nested summation

$$\sum_{X_1=x_1} P_1(x_1) \sum_{X_2=x_2} P_2(x_2|x_1) \ \ldots \sum_{X_M=x_m} P_m(x_m|x_{m-1})$$

reduces to $\sum_{X_1=x_1} P_1(x_1) = 1$ since each $\sum_{X_k=x_k} P_k(x_k|x_{k-1}) = 1$.[1]

This approximation can be represented as a graphical model by considering the subsets as nodes and placing a directed edge between two subsets that have a conditional dependency. Note that this model is a path, one of the simplest types of graphs. Figure 1 gives an example of this graph.

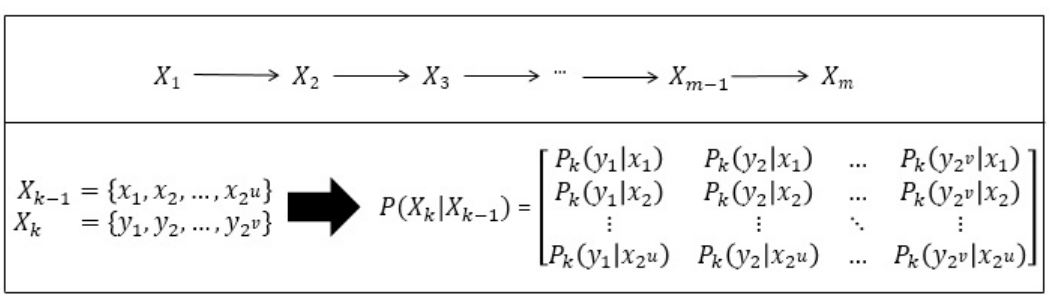


Figure 1: Path Graphical Model

[1] To avoid division by zero, we define $P_k(x_k|x_{k-1}) = 0$ if $P_{k-1}(x_{k-1}) = 0$.

We can compute the min-entropy implied by this type of approximate joint distribution using a slight modification to the Viterbi algorithm. The Viterbi algorithm applies to a hidden Markov chain and is used to find the most likely sequence of states that produced an observed set of outputs under the assumption that each state probabilistically emits a particular output. An in-depth description of this algorithm is provided in [6] (Section 4.11). However we give a brief summary here. Let $V_k(j)$ be the probability corresponding to the most likely k-long sequence of states, ending with state j, that produced the first k outputs. Letting s_k be the kth observed output and $P_{i,j}$ be the transition probability from state i to j, it can be shown that

$$V_k(j) = \begin{cases} P(s_k|j)\max_i(P_{i,j} \cdot V_{k-1}(i)), & 1 < k \leq m; \\ P(s_1|j)P_j, & k = 1. \end{cases}$$

where P_j is the probability that j is the initial state. Therefore $V_k(j)$ can be computed recursively. Given m outputs, the probability corresponding to the most likely sequence of states that generated these outputs is $\max_i(V_m(i))$, where i ranges over all possible states.

For our purposes, we aim to compute the probability of the most likely n-long sequence of bits emitted from the noise source (i.e., the most likely n-bit output block). Instead of a single Markov matrix, we have the $m-1$ conditional probability matrices $P(X_2|X_1)$, $P(X_3|X_2)$, ..., $P(X_m|X_{m-1})$. Since hidden states do not apply to this model, we effectively ignore them by supposing that each state j emits a unique output s_j with probability one. By considering the possible values of each X_k as a state, the Viterbi formula can be written as

$$V_k^{\rightarrow}(y) = \begin{cases} \max_{X_{k-1}=x}(P_k(y|x) \cdot V_{k-1}^{\rightarrow}(x)), & 1 < k \leq m; \\ P_1(y), & k = 1. \end{cases}$$

Although this algorithm as stated can be implemented for the path model, it is not sufficient for more general tree models that are discussed in the following section. To address this issue, we point out that this formula calculates the probability of the most likely sequence of states starting from X_1 and proceeding to X_m. However, this probability can also be computed beginning from X_m and working backwards to X_1. To accomplish this, we define the recursive formula

$$V_k^{\leftarrow}(x) = \begin{cases} \max_{X_{k+1}=y}(P_{k+1}(y|x) \cdot V_{k+1}^{\leftarrow}(y)), & 1 \leq k < m; \\ P_1(x) \cdot V_1^{\leftarrow}(x), & k = 0, \end{cases}$$

subject to the initial condition $V_m^{\leftarrow}(x) = 1$. Figure 2 displays the probability path traversal for both $V^{\rightarrow}$ and $V^{\leftarrow}$. Therefore, these two recursive functions both compute the probability of the most likely output block, and thus, the min-entropy implied by the model can be calculated as either $-\log_2(\max_{X_m=y}(V_m^{\rightarrow}(y)))$ or $-\log_2(\max_{X_1=x}(V_0^{\leftarrow}(x)))$.

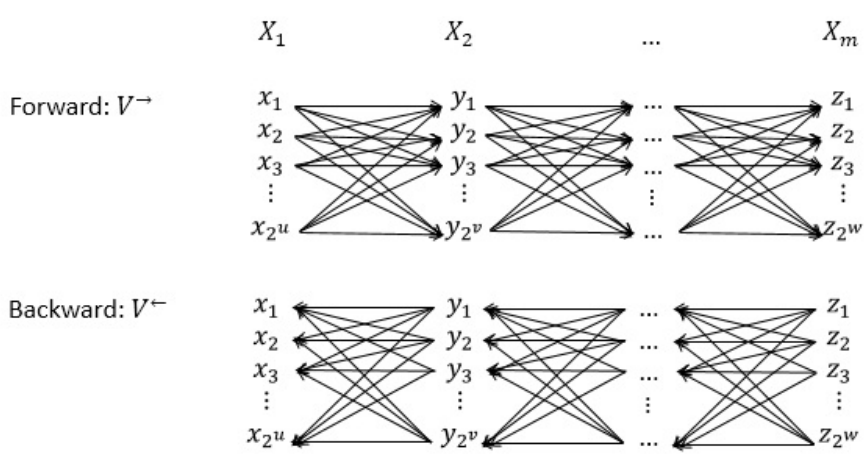


Figure 2: Probability Paths Traversed By $V^{\rightarrow}$ and $V^{\leftarrow}$

Although this path model has motivated our approach, it is in general too primitive to handle data that contain complex dependencies. As an example, if there is a particular subset of bits that are mutually dependent, then unless this subset is selected as part of this model (or at least two subsets containing these bits with an edge between them), the dependency will be missed. A more general model is described in the following section that optimizes the dependence structure given a specific subset partitioning of the bits.

3.2 Chow-Liu Trees

A generalization of the path model approach is to approximate the joint probability distribution of several variables by a structure known as a Chow-Liu tree, described in [2]. Given a set of m random variables, consider these variables as nodes in a graph. Compute the mutual information[2] between each of the $\binom{m}{2}$ pairs of variables, which are used as the edge weights. Arrange these weights in a list in descending order and build a tree by taking the first $m-1$ edges that do not cause an undirected cycle. The resulting graph is a Chow-Liu tree. It is shown in [2] that a Chow-Liu tree minimizes the Kullback-Leibler divergence from the joint distribution across all such tree models, and is therefore the optimum tree approximation to the joint distribution of these m variables.

For the purposes of entropy estimation, we consider the random variables of the joint distribution to be subsets of bits that partition the bits in the output block, as described in Section 3.1. Viewing a Chow-Liu tree in this way provides a graphical model of the dependencies between subsets of bits. As an example, the following is a tree approximation to the joint distribution of the random variables $X_1, X_2, ..., X_6$, depicted in Figure 3,

$$P(X_1, X_2, ..., X_6) \approx P(X_3) \cdot P(X_2|X_3) \cdot P(X_4|X_3) \cdot P(X_6|X_3) \cdot P(X_5|X_2) \cdot P(X_1|X_5).$$

It can be seen that a tree model is a valid distribution (i.e., the probabilities sum to one). To show this explicitly, we can sum the probabilities using the ordering returned by the following procedure. Given a tree T and empty list L:

1. Prepend the variables corresponding to the leaf nodes of T to L.
2. Define T' to be the subgraph induced by removing the leaf nodes from T.
3. If T' is the null graph, return L as the ordering. Otherwise, apply Step 1 to T'.

This eliminates the leaves at each step, while working backwards towards the root. For example, a possible ordering of the variables returned by this procedure applied to the tree in Figure 3 is $X_3, X_2, X_5, X_1, X_4, X_6$. This ordering yields the following sum, which is equal to 1,

$$\sum_{X_3=x_3} P_3(x_3) \sum_{X_2=x_2} P_{3,2}(x_2|x_3) \sum_{X_5=x_5} P_{2,5}(x_5|x_2) \sum_{X_1=x_1} P_{5,1}(x_1|x_5) \sum_{X_4=x_4} P_{3,4}(x_4|x_3) \sum_{X_6=x_6} P_{3,6}(x_6|x_3),$$

where $P_{i,j}(x_j|x_i)$ is the probablity that $X_j = x_j$ given $X_i = x_i$ in $P(X_j|X_i)$, and $P_i(x_i)$ is the probability that $X_i = x_i$.

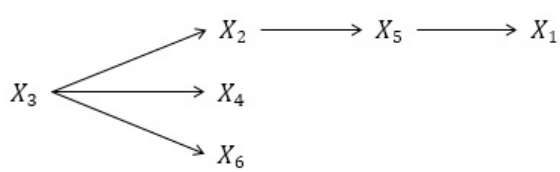


Figure 3: Example of Tree Graphical Model

[2]The mutual information between two random variables X and Y is

$$I(X;Y) = \sum_{Y=y} \sum_{X=x} P(x,y) \log \left(\frac{P(x,y)}{P(x)P(y)} \right).$$

Note that this quantity can be expressed as $H(X) + H(Y) - H(X,Y)$, where H is the Shannon entropy function.

In order to compute the min-entropy associated with a Chow-Liu tree, we extend the definition of the Viterbi algorithm, specifically $V^{\leftarrow}$, described in Section 3.1. We point out that by traversing the probability paths, beginning from the leaf nodes and working towards the root, these paths are independent prior to a junction (parent node), and can be merged at each junction (see Figure 4). Let C_i be the set of variables that are the child nodes of a variable X_i. Given a tree model on the variables $X_1, X_2, ..., X_m$, we relabel the indices $X_{1'}, X_{2'}, ..., X_{m'}$ to denote an ordering returned by the procedure described in the above paragraph. Note that $X_{1'}$ is the root. We now extend the definition of $V^{\leftarrow}$ as

$$V_k^{\leftarrow}(x) = \begin{array}{ll} \prod_{X_{l'} \in C_{k'}} \max_{X_{l'}=y} \left(P_{k',l'}(y|x) \cdot V_{l'}^{\leftarrow}(y)\right), & 1 \leq k \leq m; \\ P_{1'}(x) \cdot V_1^{\leftarrow}(x), & k = 0. \end{array}$$

We define $V_k^{\leftarrow}(x) = 1$ when $X_{k'}$ is a leaf node (i.e., $C_{k'} = \emptyset$) so that $V_k^{\leftarrow}$ is consistent with subsequent computations. The min-entropy implied by the tree is $-\log_2(\max_{X_{1'}=x}(V_0^{\leftarrow}(x)))$.

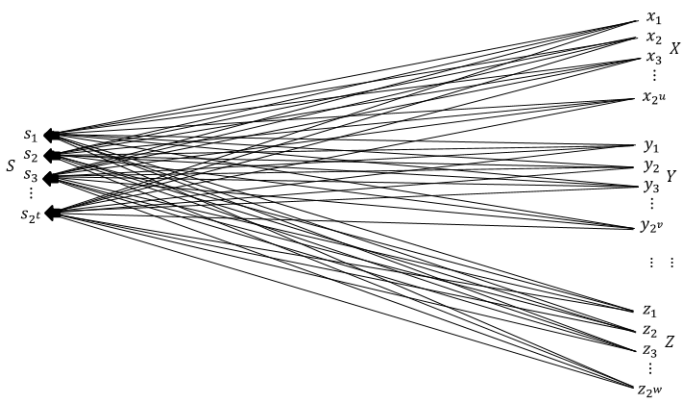


Figure 4: Consolidating Probability Paths at a Junction

To make this more concrete, we show the recursive step used to compute $V^{\leftarrow}$ when applying this function to the tree in Figure 3. We use the ordering $X_{1'} = X_3$, $X_{2'} = X_2$, $X_{3'} = X_5$, $X_{4'} = X_1$, $X_{5'} = X_4$, $X_{6'} = X_6$ to obtain

$$\begin{aligned}
V_6^{\leftarrow}(x) &= 1, \\
V_5^{\leftarrow}(x) &= 1, \\
V_4^{\leftarrow}(x) &= 1, \\
V_3^{\leftarrow}(x) &= \max_{X_1=y}(P_{5,1}(y|x) \cdot V_4^{\leftarrow}(y)), \\
V_2^{\leftarrow}(x) &= \max_{X_5=y}(P_{2,5}(y|x) \cdot V_3^{\leftarrow}(y)), \\
V_1^{\leftarrow}(x) &= \max_{X_2=y}(P_{3,2}(y|x) \cdot V_2^{\leftarrow}(y)) \cdot \max_{X_4=y}(P_{3,4}(y|x) \cdot V_5^{\leftarrow}(y)) \cdot \max_{X_6=y}(P_{3,6}(y|x) \cdot V_6^{\leftarrow}(y)), \\
V_0^{\leftarrow}(x) &= P_3(x) \cdot V_1^{\leftarrow}(x).
\end{aligned}$$

The min-entropy of this tree distribution is $-\log_2(\max_{X_3=x}(V_0^{\leftarrow}(x)))$.

3.3 An Example Distribution

Suppose we have a noise source that produces 10-bit outputs according to the distribution in Figure 5. We label the first bit b_1, the second bit b_2, and so on, up to b_{10}. In this figure we see that, for example, bit b_1 has no dependencies on other bits, while the value of bit b_3 depends on the values of bits b_1 and b_2.

$$b_1: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.5 \quad 0.5\ ] \end{array} \qquad b_5: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.4 \quad 0.6\ ] \end{array} \qquad b_{10}: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.3 \quad 0.7\ ] \end{array}$$

$$b_2: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.5 \quad 0.5\ ] \end{array} \qquad b_8: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.3 \quad 0.7\ ] \end{array}$$

$$b_4: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.8 \quad 0.2\ ] \end{array} \qquad b_9: \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ [\ 0.5 \quad 0.5\ ] \end{array}$$

$$b_3: \begin{array}{c} \\ b_2b_1 = 00 \\ b_2b_1 = 01 \\ b_2b_1 = 10 \\ b_2b_1 = 11 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \end{array} \qquad b_6: \begin{array}{c} \\ b_5b_4 = 00 \\ b_5b_4 = 01 \\ b_5b_4 = 10 \\ b_5b_4 = 11 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \end{array} \qquad b_7: \begin{array}{c} \\ b_8b_5 = 00 \\ b_8b_5 = 01 \\ b_8b_5 = 10 \\ b_8b_5 = 11 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{bmatrix} 0.3 & 0.7 \\ 0.2 & 0.8 \\ 0.1 & 0.9 \\ 0.2 & 0.8 \end{bmatrix} \end{array}$$

Figure 5: An Example 10-Bit Distribution

Consider the following four bit partitions:

$$P_1 = \{S_1 = \{b_1\}, S_2 = \{b_2\}, S_3 = \{b_3\}, S_4 = \{b_4\}, S_5 = \{b_5\}, S_6 = \{b_6\}, S_7 = \{b_7, b_8, b_9, b_{10}\}\},$$
$$P_2 = \{T_1 = \{b_1, b_4, b_8\}, T_2 = \{b_2\}, T_3 = \{b_3\}, T_4 = \{b_5, b_{10}\}, T_5 = \{b_6\}, T_6 = \{b_7, b_9\}\},$$
$$P_3 = \{U_1 = \{b_1, b_2\}, U_2 = \{b_3, b_4\}, U_3 = \{b_5, b_6\}, U_4 = \{b_7, b_8\}, U_5 = \{b_9, b_{10}\}\},$$
$$P_4 = \{V_1 = \{b_1, b_4\}, V_2 = \{b_7, b_{10}\}, V_3 = \{b_2, b_5\}, V_4 = \{b_3, b_8\}, V_5 = \{b_6, b_9\}\}.$$

The min-entropy estimates derived from the corresponding Chow-Liu trees displayed in Figure 6 are 6.5851, 6.5240, 5.3867, and 6.5565 for P_1, P_2, P_3, and P_4 respectively. Note that the edge weights, representing the mutual information between subsets, are provided as well.

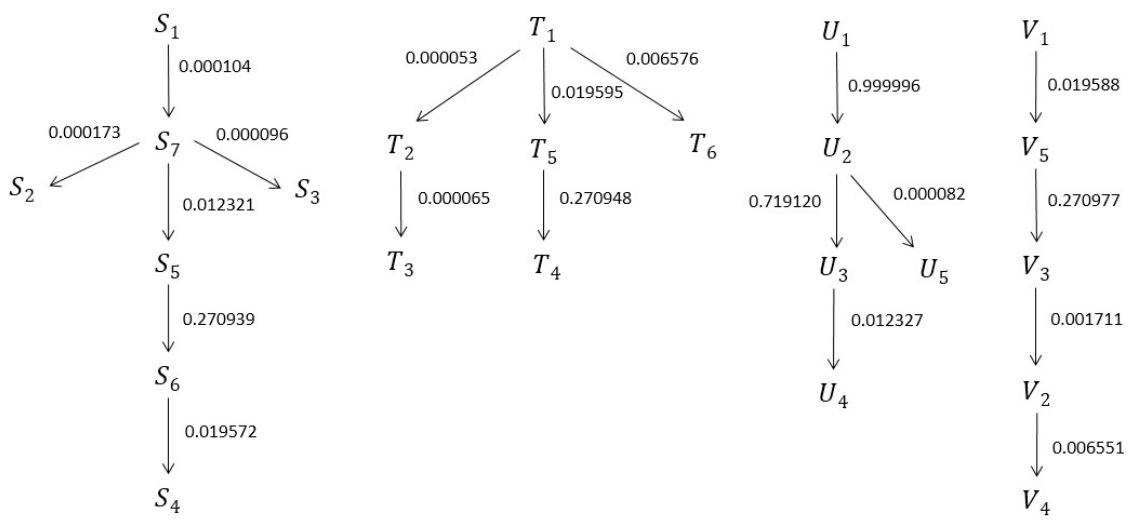


Figure 6: Chow-Liu Trees for Four Different Partitions

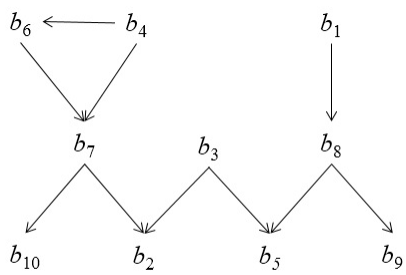
This example shows that different subset partitions of the bits will in general yield different approximations to the joint distribution. For this noise source the actual min-entropy is known to be 5.40997, and although P_3 results in the closest approximation to the actual min-entropy, there does not appear to be a straightforward way to check partitions that are "close" to P_3. In other words, it is not clear how to converge to partitions that give better approximations to the actual joint distribution of the bits. In order to make progress on this problem, two concepts are explored in the following sections. The first is a Bayesian network, which is a generalization of a Chow-Liu tree, and is used to represent the dependencies between bits. The second is a genetic algorithm, which is used to converge to a Bayesian network that gives an optimal approximation of the joint distribution.

4 Bayesian Networks

A Bayesian network (BN) is a type of probabilistic graphical model and, in our context, can be thought of as a generalization of a Chow-Liu tree (Section 3.2). The main advantage of using a BN over a Chow-Liu tree is that a BN allows for greater versatility in expressing relationships between the individual bits of an output block and provides more granularity when visualizing these dependencies among the bits as opposed to subsets of bits.

4.1 General Concepts

A BN is a directed acyclic graph (DAG) where each node is a random variable and the edges indicate relationships between these variables. These relationships are encoded as conditional probabilities between the random variables. To illustrate this, Figure 7 gives an example of a BN for a 10-bit output block along with the corresponding adjacency matrix. By summing the variables according to a topological[3] ordering, it can be shown that the approximation to the joint distribution implied by a BN sums to one. Note that the procedure described in Section 3.2 provides such an ordering.



$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 7: Example of Bayesian Network and Adjacency Matrix

In Figure 7, b_1, b_2, ..., b_{10} are random variables representing the value of each respective bit in a given output block. This BN approximates the joint distribution of these bits by

$$P(b_1) \cdot P(b_3) \cdot P(b_4) \cdot P(b_6|b_4) \cdot P(b_8|b_1) \cdot P(b_9|b_8) \cdot P(b_{10}|b_7) \cdot P(b_7|b_4, b_6) \cdot P(b_2|b_3, b_7) \cdot P(b_5|b_3, b_8).$$

According to this model, the variables b_1, b_3, and b_4 are not dependent on any other bits, and therefore each has an unconditional probability distribution on the two possible values 0 and 1. Each unconditional distribution can be thought of as a 1×2 table. On the other hand, b_6, b_8, b_9, and b_{10} each depend on a single bit, and so, have conditional probability distributions, each represented by a 2×2 table. Likewise, b_2, b_5, and b_7 each depend on two bits, yielding a 4×2 table for each of the conditional distributions. Figure 8 gives an example of each type of distribution. In general, if a BN implies that a particular bit depends on m other bits, then this conditional bit distribution can be represented by a $2^m \times 2$ table, since the joint distribution of the m bits has 2^m possible values, and there are only two possible values for the dependent bit.

Suppose that a particular BN posits that a bit b is dependent on m bits, c_1, c_2, ..., c_m. Consider the scenario where b is actually dependent on a subset consisting of only r of these m bits, such that $0 \leq r < m$. We relabel the indices of these bits $c_{1'}$,$c_{2'}$, ..., $c_{r'}$, along with the remaining bits $c_{(r+1)'}$, $c_{(r+2)'}$, ..., $c_{m'}$. We further assume that any random variable within one of the two sets $\{c_{1'}, c_{2'}, ..., c_{r'}\}$ and $\{c_{(r+1)'}, c_{(r+2)'}, ..., c_{m'}\}$ is independent of all of the variables within the other set. These conditions imply (using lax notation)

$$\begin{aligned} P(b|c_1, c_2, ..., c_m) = \frac{P(b, c_1, c_2, ..., c_m)}{P(c_1, c_2, ..., c_m)} &= \frac{P(b, c_{1'}, c_{2'}, ..., c_{r'}) \cdot P(c_{(r+1)'}, c_{(r+2)'}, ..., c_{m'})}{P(c_{1'}, c_{2'}, ..., c_{r'}) \cdot P(c_{(r+1)'}, c_{(r+2)'}, ..., c_{m'})} \\ &= \frac{P(b, c_{1'}, c_{2'}, ..., c_{r'})}{P(c_{1'}, c_{2'}, ..., c_{r'})} = P(b|c_{1'}, c_{2'}, ..., c_{r'}). \end{aligned}$$

[3] A topological ordering is defined such that for a pair of nodes X and Y, X precedes Y in the ordering if and only if there is a directed edge from X to Y. For details, refer to [4] (page 36).

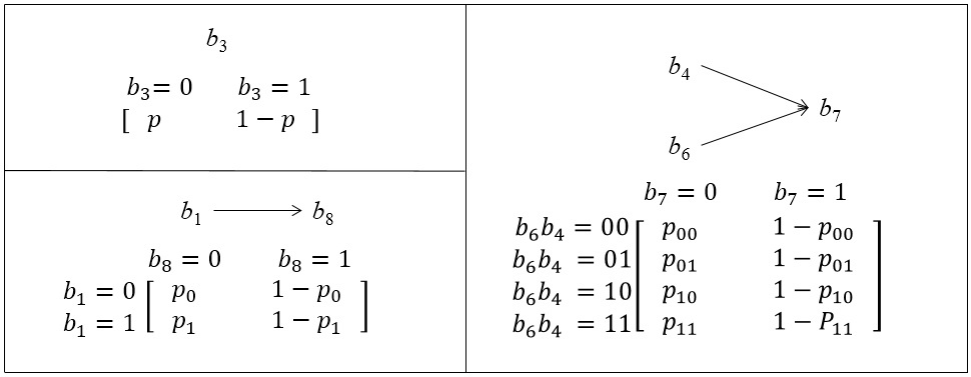


Figure 8: Examples of Bit Distributions in a Bayesian Network

This equation states that under the assumed conditions, $P(b|c_1, c_2, ..., c_m) = P(b|c_{1'}, c_{2'}, ..., c_{r'})$. Therefore, the $m - r$ bits that are independent of b and $c_{1'}$,$c_{2'}$, ..., $c_{r'}$, do not affect the distribution of b, and thus, their edges to b are unnecessary. A fitness function for a genetic algorithm that penalizes BNs with such extraneous edges will be discussed in Section 5.2.2.

4.2 Implementation Details

The BN object that we define consists simply of its corresponding adjacency matrix (as seen in Figure 7) as well as its associated score from the fitness function used in the genetic algorithm (described in Section 5.2.2). In the first of the following sections, we discuss a technique for efficiently storing the BN's probability tables that significantly improves the speed of generating and scoring each BN. Note that this section is not crucial for understanding the primary concepts of this paper, and can therefore be skipped. In the second section, we explore how different methods of sampling the output bits can, in some cases, enhance the effectiveness of the BN to approximate the joint distribution of these bits.

4.2.1 Storing Probability Tables in Red-Black Trees

A straightforward implementation would have a BN object, in addition to its adjacency matrix, include each of its probability tables. However, this is (not surprisingly) inefficient when dealing with sufficiently large output block sizes, since any probability table common to multiple BNs must be regenerated and stored for each of these BNs. A simple solution is to store each probability table as an entry in a global binary search tree. The downside to a standard binary search tree is that it may become increasingly unbalanced, resulting in search time that is closer to $O(M)$ rather than $O(\log_2 M)$, where M is the number of probability tables in the tree. We therefore choose to implement a *red-black tree*, which has the property that no path from the root to a leaf is more than twice as long as any other path from the root to a leaf. This is achieved by imposing a suitable red-black coloring of each node when building the tree. This type of tree maintains a search time of $O(\log_2 M)$. For a rigorous discussion of red-black trees, refer to [3] (Chapter 13).

Our implementation utilizes two red-black trees, denoted as T_c and T_g, each storing the same set of pointers[4] to previously generated probability tables. In T_c, the pointers are ordered according to the variables involved in the respective probability table, and in T_g, they are ordered by the most recent generation in which a BN with this table was created (refer to the discussion of genetic algorithms in Section 5). Ties in T_g are decided by the ordering used in T_c. We point out that the ordering used in T_c can be achieved by noticing that each column of the BN's adjacency matrix corresponds to a probability distribution and also specifies the variables involved. For example, the 7th column of the adjacency matrix in Figure 7 implies that b_7 is dependent on b_4 and b_6, resulting in the probability table displayed in Figure 8. This column (transposed) is [7 0 0 0 1 0 1 0 0 0 0], where the column number is prepended to the column in order to differentiate columns that are identical. We can therefore index a probability table in T_c by its corresponding column in the adjacency matrix and order the columns lexicographically

[4]In order to save memory, we use pointers, which avoids duplication of probability tables across the two trees.

(e.g., [7 0 0 0 1 0 1 0 0 0 0] > [2 0 0 1 0 0 0 1 0 0 0]). An example of these two orderings used for both T_c and T_g, across three generations, is provided in Figure 9. Note that a repeated column entry is updated with the most recent generation in which the corresponding BN occurs.

GEN = 1:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

GEN = 2:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

GEN = 3:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

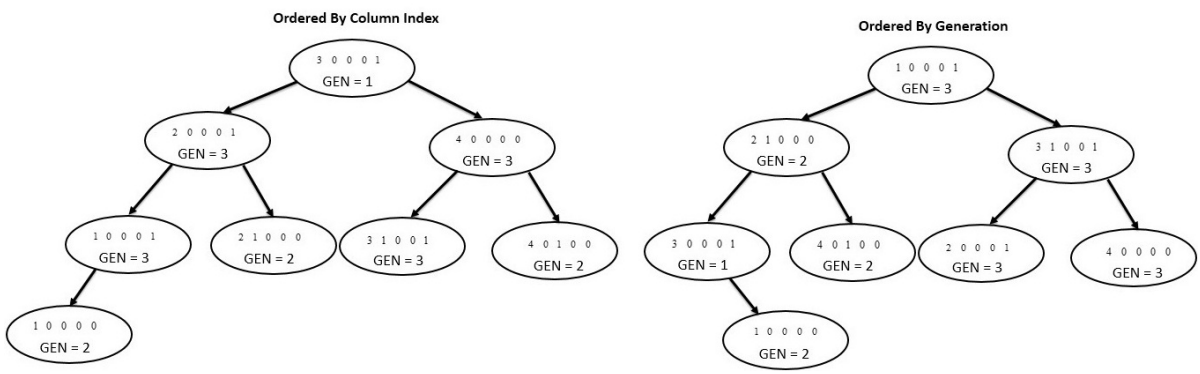


Figure 9: Binary Search Tree Ordered By Column Index (Left) and Generation (Right)

The purpose of T_c is to maintain a shared space in memory where each BN object can access probability tables that have already been created. However, too many such tables might be stored at a time and will need to be aged off. T_g allows us to efficiently identify the "eldest" (i.e., smallest generation) tables and remove them from both trees, as these tables are the least recently used. The frequency at which these tables are removed depends on the memory constraints for a given platform. Therefore, assuming that recurring tables are not frequently recreated and removed, most of the tables only need to be built once, and as the genetic algorithm converges, fewer new tables need to be built. Although we do not have a proof about the rate of convergence, in practice we have seen a considerable increase in the performance of the program as the generations progress.

The use of binary search trees also allows us to efficiently compute the likelihood[5] of observing a set of n-bit output blocks given a BN that models the n bits of each block as the random variables $b_1, b_2, ..., b_n$. We denote this BN as a model M. Suppose there are N such output blocks, labeled $B_1, B_2, ..., B_N$, with B_i consisting of the bits $B_{i,1}, B_{i,2}, ..., B_{i,n}$. Then the probability of observing this set of output blocks given M is provided by the likelihood function $L(M : \{B_1, B_2, ..., B_N\}) = \ _{i=1}^{N} P(B_i|M)$. In order to determine $P(B_i|M)$, we notate the probability distribution, posited by M, of a bit b_j as P_{S_j,b_j}, where S_j is the set consisting of the bits that influence b_j. Note that b_j is unconditionally distributed if $S_j = \emptyset$. We define $S_j^{B_i}$ to be the values of each random variable in S_j corresponding to B_i. Then $P(B_i|M) = \ _{j=1}^{n} P_{S_j,b_j}(B_{i,j}|S_j^{B_i})$, and so,

$$L(M : \{B_1, B_2, ..., B_N\}) = \prod_{i=1}^{N} \prod_{j=1}^{n} P_{S_j,b_j}(B_{i,j}|S_j^{B_i}).$$

Alternatively, we can compute this likelihood by interchanging these products as

$$L(M : \{B_1, B_2, ..., B_N\}) = \prod_{j=1}^{n} \prod_{i=1}^{N} P_{S_j,b_j}(B_{i,j}|S_j^{B_i}).$$

[5]Note that computing the likelihood of the data for a particular model is required to evaluate the fitness function described in Section 5.2.2 (refer to Chapter 17 of [4] for an in-depth discussion of the likelihood function).

This allows us to calculate the likelihood of observing the output blocks by independently determining the likelihood of each b_j. Since we can easily compute $\pi_j = \prod_{i=1}^{N} P_{S_j,b_j}(B_{i,j}|S_j^{B_i})$ when building each probability table, we simply store this value in the binary search tree as well. Given a BN, we then compute the likelihood by retrieving each probability table in the search tree and multiplying the corresponding π_j's.

4.2.2 Relationship Between Bit Positions and Bayesian Network Structure

Up to this point we have assumed a straightforward correspondence between the n bit positions of each observed sample value and the n nodes in a BN model of the data. However, there is flexibility in the interpretation of bit positions as nodes in a BN structure that can increase the utility of the method described in this paper. In particular, we can define the set of bit positions in the sample data in a way that reflects the extent of the dependence structure potentially existing in the data. For example, if it is believed that the n-bit sample values are independent of each other, but we want to identify possible dependencies among the bit positions in each n-bit block, then the straightforward approach would be to search for the BN that best describes the dependency relationship among the n bit positions.

In some cases, a data sequence may consist of individually generated bits and therefore have no "natural" block size (other than 1). Suppose such a data sample has the property that a certain dependence structure applies to combinations of bits in regularly spaced positions. As an example, the bit position sets $\{1, 8, 11\}$, $\{51, 58, 61\}$, $\{101, 108, 111\}$, etc., could each be explained by a dependence where the bit in the third position in each set is exactly determined by the bits in the first two positions of that set with probability 0.6. There are several ways to handle this data that would allow us to identify this dependency. We could simply interpret the data as non-overlapping 50-bit blocks. However, if we have hints about the approximate locations and extent of the dependencies, we could choose a smaller block size and specify a corresponding skip distance. Depending on the values chosen, this allows us to omit any regions not of interest in our analysis. In this example we could incorporate the known relationship between the three bits in each set and also search for other nearby dependencies by choosing a block size of, say 20, and a skip distance of 50. This approach is illustrated in Figure 10. Note that this reinterpretation of block size (possibly in conjunction with a skip distance) can be performed even if the data sample does have an apparent block size. The point is that we are free to define a block in many different ways, resulting in the ability to extend the search for dependence outside the constraints of the formatting of the data.

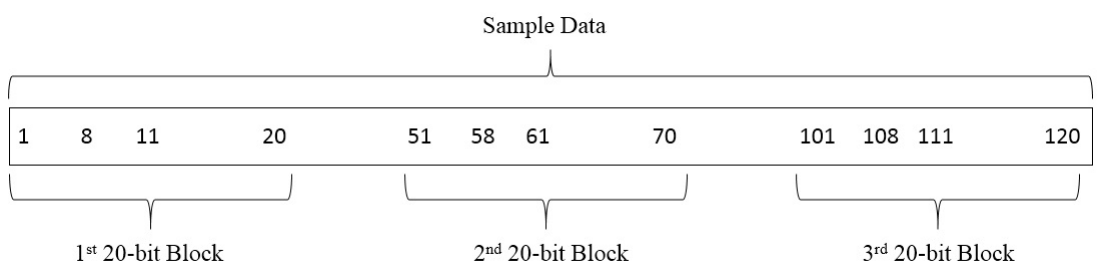


Figure 10: Block-Size Interpretation

4.3 Estimating Min-Entropy

Neither the Viterbi algorithm, nor its extension (described in Section 3.2), are sufficient to compute the min-entropy implied by a general BN. Instead, we use the *max-product variable elimination* algorithm, which can be considered a generalization of the Viterbi algorithm. This method works by choosing one of the random bit variables b_i and computing the corresponding *max-marginalization factor* resulting from the elimination of b_i. This process is then repeated for each remaining bit. We give a brief description of this algorithm, and adopt the notation used in Section 13.2 of [4], where a more elaborate description is provided.

In order to compute the max-marginalization factor resulting from the elimination of a variable b, the following steps are performed.

1. Each probability distribution specified by the BN is converted into a *factor*, denoted as ϕ, which (for our purpose) can be thought of as "unrolling" a conditional probability table into a one-dimensional list. For instance, given the conditional distribution $P(b|c_1, c_2, ..., c_m)$, represented by a $2^m \times 2$ table, $\phi(c_m, c_{m-1}, ..., c_1, b)$ is a list consisting of these 2^{m+1} conditional probabilities, corresponding to all the possible combinations of the values of $c_m, c_{m-1}, ..., c_1, b$. Note that any ordering of the bits can be chosen for the factor.
2. The product of factors, ψ, that includes all of the bits that have a common dependence with b, is computed by "multiplying" all of the factors that have b in common.
3. The max-marginalization factor τ, obtained by eliminating b, is constructed by taking the maximum entry when $b = 0$ and $b = 1$ for each combination of variables in ψ. This incorporates into τ the maximum probability that b contributes to the BN.

An example of creating a max-marginalization factor by eliminating a bit b is shown in Figure 11. In this diagram, $\phi_\alpha(c_2, c_2, b)$ is the factor derived from $P(b|c_1, c_2)$. Supposing that ϕ_α and another factor ϕ_β are the only two factors that have b in common, $\psi(c_2, c_1, b)$ is constructed by multiplying the probabilities of the matching values of the bits between these factors. The max-marginalization factor $\tau(c_2, c_1)$ is created by taking the maximum of the probabilities in each pair of rows where the values of c_1 and c_2 are the same.

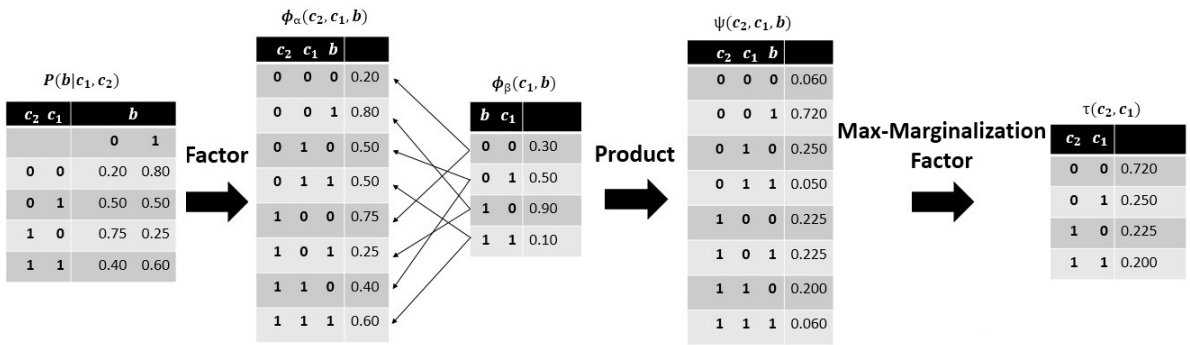


Figure 11: Process of Generating a Max-Marginalization Factor

Once the factors used to construct ψ are incorporated into τ, they are no longer needed. This process is repeated by selecting one of the remaining bits and computing its max-marginalization factor using the remaining factors along with τ. The max-marginalization factor of the last remaining bit is the *max-marginal*. The maximium probability entry of this max-marginal is the maximum probability of the joint bit distribution implied by the BN, labeled $p_{\max}$. The min-entropy is therefore $-\log_2(p_{\max})$. Note that if a BN consists of multiple components, the cumulative min-entropy is the sum of the min-entropy indicated by each component. Although the max-product variable elimination algorithm succeeds for any bit-ordering, some orderings are more computationally intensive than others. However, determining an optimal elimination ordering is in general an NP-hard problem (Section 9.4.3 of [4]). We simply use a topological ordering as described in Section 3.2.

Figure 12 provides an example of the steps performed in the max-product algorithm for a BN with the four bits b_1, b_2, b_3, b_4. Using the ordering b_1, b_3, b_4, b_2, we begin by eliminating b_1. Since $\phi_1(b_3, b_2, b_1)$ is the only factor that contains b_1, the product $\psi_1(b_3, b_2, b_1)$ is equal to this factor. The resulting max-marginalization factor $\tau_1(b_3, b_2)$ replaces $\phi_1(b_3, b_2, b_1)$. The next variable to eliminate is b_3. The only factors that include b_3 are $\tau_1(b_3, b_2)$ and $\phi_3(b_4, b_2, b_3)$, and so $\psi_3(b_4, b_2, b_3)$ is equal to the product of these factors. Once $\tau_3(b_4, b_2)$ is created by eliminating b_3 in ψ_3, it is multiplied by $\phi_4(b_4)$, resulting in $\psi_4(b_4, b_2)$, since these are the only factors that contain b_4. After $\tau_4(b_2)$ is computed by eliminating b_4, b_2 is the only remaining bit. The product of the last remaining factors $\phi_2(b_2)$ and $\tau_4(b_2)$ forms $\psi_2(b_2)$ (i.e., the max-marginal). The two entries in $\psi_2(b_2)$ correspond to the probability of the most likely values of b_1, b_3, and b_4 when $b_2 = 0$ and $b_2 = 1$, respectively. Therefore, $p_{\max}$ is the largest entry of $\psi_2(b_2)$.

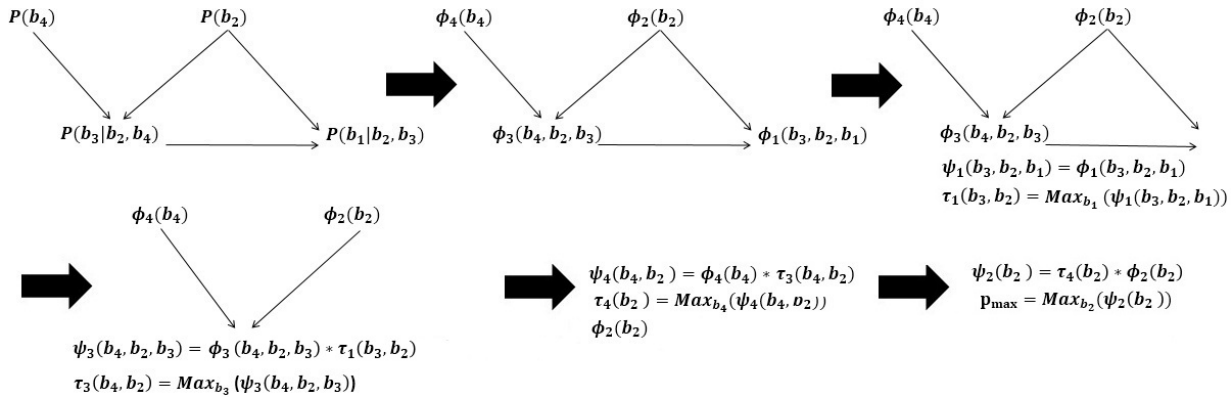


Figure 12: Max-Product With Bit Ordering b_1, b_3, b_4, b_2

5 Genetic Algorithms

The previous section laid the groundwork for using a Bayesian network (BN) to represent the dependence relationships among the bit positions in the data blocks produced by the noise source. We now address the problem of finding the particular BN that best models the sample data. In general, this is is an NP-hard problem (see [4], Theorem 18.5), so it will not be possible to deterministically find the optimal BN for practical problems. What is needed, therefore, is a heuristic technique for exploring the space of possible solutions that has a good chance of finding a nearly optimal BN. The approach we will use is known as a genetic algorithm (GA).

5.1 General Concepts

A GA is a technique for finding a set of parameter values that maximize or minimize a given function. The approach taken is to mimic biological adaptation of traits to a species' environment by forming an initial population of candidate solutions, then forming successive generations of new candidate solutions. Each new generation is created from the previous generation through a combination of replicating a certain number of best-scoring candidates (elitist selection), creating new candidates by combining features from pairs of candidates (crossover), and introducing small random changes in a few candidates (mutation). The intent is for the average fitness score for the candidate solutions in a generation (based on the function being optimized) to increase with subsequent generations, until a nearly optimal candidate solution is found. Section 5.2 presents a more detailed description of each of the components required to perform a GA for our specific application. We provide a summary of these components here.

1. In order to use a GA to explore the space of possible candidate solutions and converge toward an optimal solution, we need a way to encode candidate solutions. For each candidate solution, this encoding scheme gives a representation of the parameter values that distinguish that particular candidate solution. This encoding scheme must be capable of representing the full range of potential solutions, and must be structured in a way that facilitates the crossover and mutation operations described below.

2. A fitness function is required to quantify the progress of the GA toward an optimal solution. This function assigns to each candidate solution a numerical indication of its quality as a solution to the optimization problem. It provides the basis for comparing the fitness of multiple candidates, which allows the prioritization of higher-fitness candidate solutions in the selection process described below. For many applications, the fitness function is simply the function to be optimized. In other applications, the fitness function must be augmented with additional components to account for other necessary factors. This is the case for our application.

3. In order to encourage the development of "beneficial" traits in subsequent generations of candidate solutions, a GA can use the process of *crossover*. In biological reproduction, an offspring (typically) is a combination of features of two parents. A GA can mimic this process using the concept of crossover, giving the GA a way to form new candidate solutions that combine features of current

candidate solutions. The intended benefit of crossover can be explained as follows. Combining pairs of candidates to form new candidate solutions results in the traits of the individual candidates being mixed in various ways. Whenever each candidate in a pair includes a portion of a beneficial trait (which might not be beneficial on its own), the offspring of that pair can potentially include the full beneficial trait. The selection process used to form subsequent generations will reward this trait with ongoing reproductive success, leading to the spread of this trait (which may be a component of the optimal solution) throughout the population. Thus, in order to produce improved candidate solutions, the GA explores various combinations of traits included in the previous generation.

4. A GA can also cause slight random changes in a small proportion of candidate solutions in each generation as an additional way of introducing novel traits into a population. These *mutations* provide extra variability in the population as the GA combines and assesses candidate solutions. Since each mutation is random, it may or may not contribute to the formation of new beneficial traits, and may in fact either increase or decrease the fitness level for the offspring candidates that experience the mutation.

5. As the GA forms each subsequent generation, it uses a selection process to prioritize the choice of candidate solutions for crossover from the current generation according to their fitness. The selection process may also be designed to directly copy one or more candidates from the current generation to the new generation. This selection process is probabilistic, assigning a higher probability of selection to better-fit candidates. This results in better-fit candidates having more reproductive success than worse-fit candidates, which encourages the spread of beneficial traits (characteristics of better-fit candidate solutions) throughout the population, and the elimination of harmful traits (characteristics that worsen the fitness score) from the population. However, the probabilistic nature of the selection function occasionally allows worse-fit candidates to be selected for crossover or copying, which is useful when an offspring candidate includes a portion of a poorer-fitting trait combined with another trait from a different candidate to form a more beneficial trait.

A candidate solution encoding, as well as the crossover and mutation operations are illustrated in Figure 13, while Figure 14 displays the formation of successive populations using these operations.

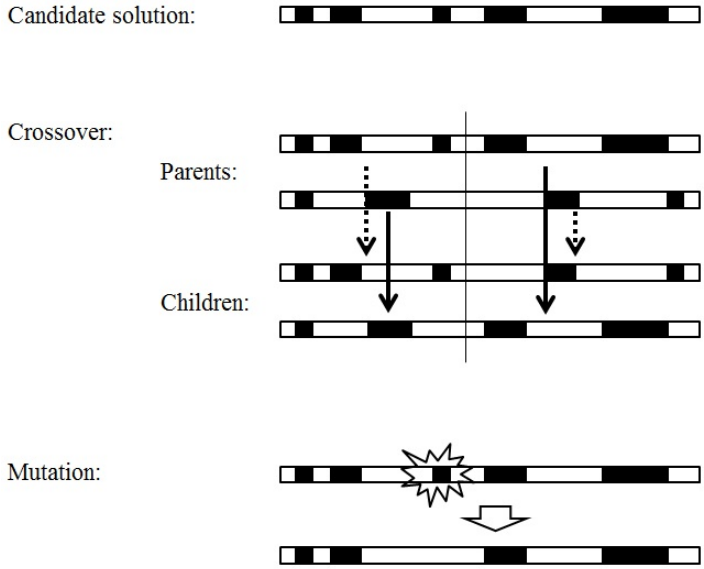


Figure 13: Genetic Algorithm Operations

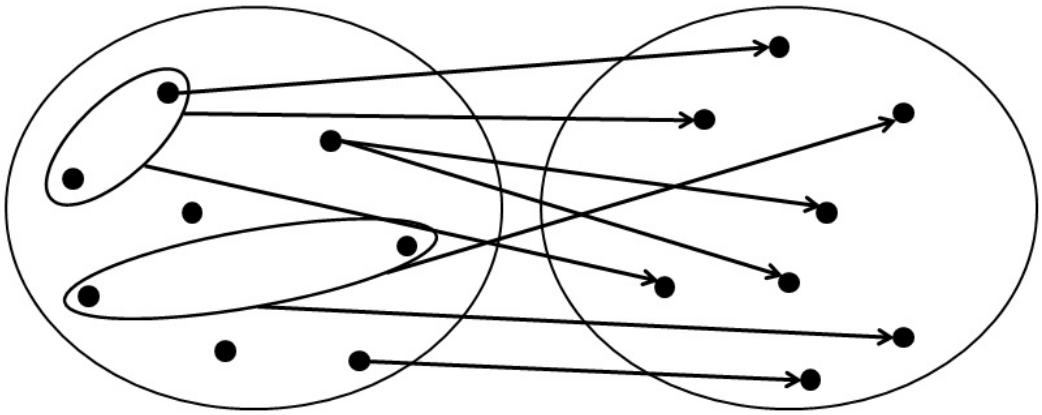


Figure 14: Formation of Successive Genetic Algorithm Generations

5.2 Implementation Details

In the following sections, we provide additional detail on the major GA components that are used in our implementation. Although there are many design decisions that can be made, our instantiation of these components has proven quite successful in practice.

5.2.1 Encoding of Candidate Solutions

There is a very simple way to encode a candidate BN for use in a GA. Given the definition of a BN in Section 4.1, we can simply represent a BN describing the dependencies in a sequence of n-bit blocks as an $n \times n$ adjacency matrix, in which the entry in row i and column j is a one if b_j (the bit in position j) is dependent on bit b_i (the bit in position i), and zero otherwise. For example, Figure 7 shows a BN representing the dependence relationships for a 10-bit block-size data sample, and the corresponding adjacency matrix.

For each of the nodes in the BN, we also require the probability distribution for the corresponding value at a given bit position be computed from the observed n-bit values. Note that this distribution is conditional or unconditional depending on whether or not any edges point to that node. These probability distributions are discussed in Section 4.1 and illustrated in Figure 8. This encoding of a BN as a candidate solution allows straightforward implementations of crossover and mutation as described below.

5.2.2 Fitness function

For our application, the goal is to find the "best" BN describing the dependence structure among the bit positions in a sample data set, although it is not immediately clear what is meant by "best" candidate. In general, a good statistical model that describes a given set of observed data is one for which the likelihood function, the probability of the data sample given the model, is high. Intuitively then, we might suspect that since we should search for the BN that maximizes this function, the fitness function should simply be the likelihood function. The probability of each observed n-bit value is the product of the appropriate probabilities indicated by the BN, and the overall likelihood is the product of these probabilities for all n-bit sample values.[6] An efficient method for computing the likelihood is described in Section 4.2.1.

There is a problem with this approach, however, as alluded to at the end of Section 4.1. Suppose that bit position j is not dependent on bit position i, and that a candidate BN reflects this fact by having a zero in position (i, j) of its adjacency matrix. Suppose this zero is replaced with a one, thereby allowing the BN to account for this possible dependency, even though this dependency is not present in the sample data. There is no influence from b_i on the resulting conditional probability distribution for b_j,

[6] In practice, we replace multiplication of these probabilities with summing of the logarithms of the probabilities in order to avoid dealing with extremely small numbers.

so the likelihood for this candidate will not be significantly changed by this modification. However, since the crossover and mutation operations described below may result in several replacements of zeroes with ones (many of which may be unwarranted) in the candidates as the GA proceeds, and there is no penalty in terms of a decrease in the likelihood for these changes, the population of candidates soon consists of adjacency matrices filled with ones. Although the optimal BN is contained within this "solution," this is of course an unhelpful answer to our optimization problem. What is needed is a way to penalize hypothesized dependencies (ones in the adjacency matrices) that merely add complexity to the model without significantly increasing the model's explanatory power.

We address this problem by choosing a fitness function that takes model complexity into account. Specifically, we use the Bayes Information Criterion (BIC) as our fitness function ([4], Section 18.3.5).[7] The BIC is defined as $-2\log(L) + k\log(N)$, where L is the likelihood of the sample data given the BN, N is the sample size (the number of n-bit sample values), and k is the number of free parameters in the BN model. Since we prefer a high likelihood value and low complexity, a lower BIC is preferable to a higher BIC. For a BN, the model parameters consist of the tables of conditional or unconditional probabilities indicating the distribution of the n bits in the graph. The value of k can be computed as ${}_{j=1}^{n} 2^{k_j}$, where k_j is the number of bits that b_j is dependent on according to that BN. This can be seen by noting that for each combination of values for the bits that b_j is dependent on, the model requires one free parameter – the probability that b_j is a zero given that particular combination of values. On the other hand, the conditional probability that b_j is a one is determined by the probability that b_j is a zero, and is therefore not a free parameter.

The effect of using the BIC as our fitness function is that the GA is forced to balance the goals of increasing the likelihood and avoiding unnecessary model complexity. For models of equal complexity, we prefer the model with the highest likelihood. For models having equal likelihoods, we prefer the model with the lowest complexity. In general, decreased explanatory power can be compensated by a sufficient decrease in complexity, and increased complexity can be compensated by a sufficient increase in explanatory power.

5.2.3 Crossover

Our implementation performs crossover on a pair of candidate solutions in a straightforward way. Since each candidate BN is represented by a $n \times n$ adjacency matrix, we simply choose a crossover point at random. We then form the first (second) child candidate by joining the portion of the first (second) parent's adjacency matrix lying above the crossover point with the portion of the second (first) parent's adjacency matrix lying below the crossover point. This is illustrated in Figure 15.

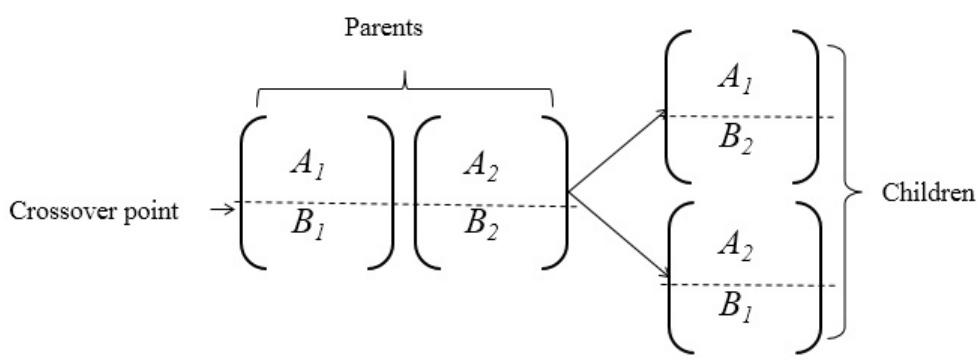


Figure 15: Example of Crossover

[7] An observation regarding why we do not use min-entropy as a fitness function is in order. We believe it is the case that as a model incorporates additional dependencies present in the sample data, the average work required by the optimum guessing attack for that model decreases. However, this does not hold in general for min-entropy. We suspect this is due to the fact that min-entropy is an approximation of the logarithm of this average work (refer to Appendix D of [1]). In fact, we have observed occasional increases in min-entropy within the overall downward trend when using this GA approach. We therefore conclude that min-entropy is not an appropriate choice for our fitness function.

Note that it is possible for this method of crossover to result in an adjacency matrix containing cycles, which cannot exist in a valid BN. For example, the first part of the first parent could specify that node i influences node j, and the second part of the second parent could specify that node j influences node i. Since each of the offspring candidate solutions resulting from crossover must be a valid BN, we remove these cycles. To accomplish this, our implementation uses a method that recursively traverses the edges of the graph and randomly selects an edge to remove from each cycle found. It is important to note that the cycle removal process must not introduce bias into the resulting offspring. For example, suppose that when we locate a cycle, we always remove an outward edge from the lowest-numbered node included in the cycle. This would favor the removal of edges from lower-numbered nodes, thereby giving the GA a preference for certain regions of the search space at the expense of other regions.

5.2.4 Mutation

In our implementation, a mutation consists of a slight change in the candidate's adjacency matrix. Even though a mutation causes only a small change in the visual appearance of the graph associated with the BN, it can potentially lead to a significant increase or decrease in the fitness value. Our implementation uses the following five types of mutations:

- Add a new edge from node i to node j.
- Delete an existing edge from node i to node j.
- Reverse an existing edge from node i to node j, making it an edge from node j to node i.
- Change the destination of an existing edge from node j to node j'.
- Change the origin of an existing edge from node i to node i'.

These types of mutations are illustrated in Figure 16. Reference [4] (Section 18.4.3.1) suggests the first three of these mutation types as "operators" connecting BNs that are "neighbors" of a given BN, as a way of establishing a local search space for optimization. The additional mutation types can occur as a combination of the first two mutations, but we include them as separate mutation types in order to allow the possibility of somewhat larger changes in a candidate solution. Note that with the exception of an edge deletion, a mutation can result in a cycle being formed. We therefore use the cycle removal procedure described for crossover.

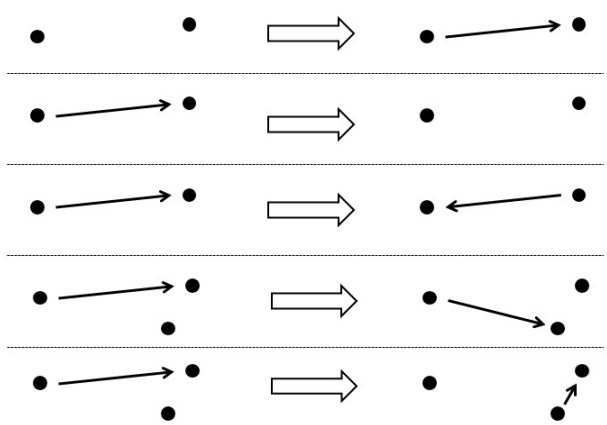


Figure 16: Types of Mutations in a Bayesian Network Adjacency Matrix

5.2.5 Selection

Selection is the process of choosing candidate solutions from the current population to influence the next generation in a way that favors better-fit candidates while still occasionally choosing worse-fit candidates. This process encourages the spread of beneficial traits throughout the population. Note that various selection methods exist. One method, called *roulette wheel selection*, chooses a candidate with probability proportional to the candidate's fitness score. That is, candidate i is selected with

probability $\frac{S_i}{\sum_{j=1}^{M} S_j}$, where S_i is the fitness score for this candidate and M is the number of candidates in the population. This method is not particularly useful in two types of situations. The first situation is where a small number of candidates have fitness values that are much higher than those of the other candidates. In such cases, roulette wheel selection gives too much priority to the better-fit candidates and rarely selects any of the others. The second situation is the case in our application, where the fitness scores (BIC values) tend to be fairly close to each other, which makes the selection probabilities essentially a uniform distribution and leaves the selection process having little preference among the candidates. We use an alternative selection method, known as *rank selection*. This method selects each candidate with probability proportional to its ranking when the fitness scores are sorted in descending order. That is, if the fitness scores are sorted and relabeled so that $S_M \geq S_{M-1} \geq ... \geq S_1$, then candidate i is selected with probability $\frac{i}{\sum_{j=1}^{M} j} = \frac{2i}{M(M+1)}$.

Prior to using the selection process, our implemementation directly copies the best-fit candidate solution from the current generation to the new generation. This is known as *elitist selection* and is performed in order to ensure that the best-fit candidate in each generation is at least as good as the best-fit candidate in the previous generation. We then select $\left\lfloor \frac{M}{2} \right\rfloor$ pairs of candidates to perform crossover (as displayed in Figure 15), producing $M-1$ offspring candidate solutions for the next generation. Note that we make no attempt to prevent candidate solutions from being selected more than once for crossover.

5.2.6 General Outline

When designing a GA, several options are available, even after deciding on the details of the components described in Sections 5.2.1 - 5.2.5. Our GA implementation is essentially as described in [5] (page 251). The following pseudocode gives the general structure of our implementation.

```
Input sample data
Input GA parameters (population size, mutation probability, number of generations)

Do for each member of initial population:
    Generate random adjacency matrix
    Remove cycles
    Compute all probability distributions needed
    Compute fitness function (BIC)
End Do

Do until maximum generation count or fitness threshold reached:
    Copy best-fit candidate solution into new generation
    Do until new population is filled:
        Select pair of candidate solutions using rank selection
        Produce two offspring candidate solutions using crossover

        For each of the two offspring:
             If Random number in range [0, 1] <= specified mutation probability,
                 Generate one of the random mutation types (1-5)
                 Apply mutation to offspring
             End If

             Remove cycles in offspring
             Compute all probability distributions implied by offspring
             Compute fitness function (BIC) for offspring
        End For
    End Do

    Replace current generation with new generation
End Do
```

After the GA has completed for the specified number of generations, there may still be extraneous edges that have not been pruned from the most-fit candidate. We check for such edges by removing each edge, one at a time, from this solution and recomputing the BIC to determine if the fitness score has

improved. If this score has not improved, the removed edge is restored. We consider the resulting solution to be the "best-fit" candidate returned by the program. This candidate solution is then saved to a file, which gives us the option to restart the program with the best-fit candidate in the initial generation. This allows us to continue searching for better-fit candidates based on the previous best-fit solution, and also serves as a "meteorite" effect, meaning that since the initial population is randomly generated, the best-fit candidate now has the chance to crossover and mutate within a brand-new population.

6 An Example

We now illustrate the operation of our implementation of this technique with the following example. Suppose a noise source produces a sequence of 32-bit outputs, where the portions in bit positions 4-8, 11-15, 18-22, and 25-29 each follow a biased joint distribution on five bits, while the remaining twelve bits are independent and unbiased. This is illustrated in Figure 17.

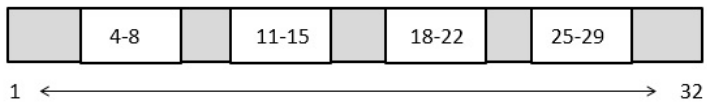


Figure 17: Sample Noise Source Data Dependence Structure

We generate 15,000 32-bit sample outputs from this distribution. Given the specific probability distribution governing the four five-bit segments, we were able to determine that the true min-entropy for the 32-bit blocks is 17.2877. We ran our min-entropy estimation program on this sample data using the following parameter values:

- Block size: 32
- Population size: 100 (number of candidates in each generation)
- Number of generations: 5000
- Skip distance: 32 (data blocks will be interpreted as non-overlapping)
- Mutation probability: 0.1 (probability that a given candidate experiences a mutation)
- Edge probability: 0.1 (probability that a pair of nodes in initial population is connected)

For this example, the program converges to a good approximation to the unknown joint distribution on 32-bit blocks within a few hundred generations. The accuracy of the approximation to the joint distribution is attested to by the fact that the estimated min-entropy per 32-bit block is 17.0506, which is very close to the true min-entropy. Figures 18 - 22 show the progress made by the GA as it explores the solution space and converges toward an optimal BN. The GA process has effectively explored the space of possible joint probability distributions on 2^{32} 32-bit blocks and converged on a reasonable approximation to the true joint distribution. Note that in the best BN in the final generation, the program has found different graphical representations of the dependencies among the four sets of dependent bits, even though these four sets of bits follow the same joint distribution. This is not surprising, since the program has no way of knowing that these underlying joint distributions are the same, and there will in general be many reasonable approximations to this joint distribution. It is interesting to observe the effect of our fitness function as subsequent generations are produced. Because this function addresses both likelihood and model complexity, we see the gradual elimination of unhelpful edges and the inclusion of edges useful to a good model of the data.

ADJACENCY MATRIX:

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
01																					*											*
02																																
03								*								*														*		
04							*														*											
05																*				*					*						*	
06			*															*				*				*						
07					*									*			*													*	*	
08																																
09			*	*																								*		*		
10								*																								
11				*	*									*																		*
12					*		*		*							*		*														*
13				*			*				*	*																				
14																																
15													*																			
16										*															*							
17										*																						
18			*																													
19				*																					*	*						
20																						*			*							
21																																
22								*									*	*														
23																																
24		*						*							*																	
25																											*					
26				*																												
27														*									*						*			
28	*																									*			*			
29		*															*				*											
30								*																								
31																*																
32																	*		*													

BIC: 813033.463068

Figure 18: Best-scoring Candidate Solution in Initial Generation

ADJACENCY MATRIX:

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
01																																
02	*												*																			
03																																
04					*		*	*																								
05																*				*												
06	*	*		*	*			*																							*	
07					*			*																								
08																																
09																													*			
10																																
11							*								*																	
12															*																	
13												*			*	*																
14											*	*			*																	
15																																
16																																
17																								*								
18		*										*								*	*											
19																						*										
20																														*		
21																							*									
22																		*		*												
23																																
24			*																													
25																										*	*		*		*	
26																																
27														*									*			*			*			
28				*	*															*					*				*			
29																																
30																																
31																																
32																																

BIC: 683288.866511

Figure 19: Best-scoring Candidate Solution After 100 Generations

ADJACENCY MATRIX:

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
01																																
02	*												*																			
03																																
04					*			*																								
05																*																
06	*			*	*			*																								
07				*	*			*																								
08																																
09																																
10																																
11												*			*																	
12															*																	
13											*	*			*																	
14											*	*			*																	
15																																
16																																
17																																
18																				*	*											
19																		*			*	*										
20																					*									*		
21																																
22																		*		*												
23																																
24																																
25																										*	*		*			
26																													*			
27																										*			*			
28																				*					*	*	*		*			
29																																
30																																
31																																
32																																

BIC: 658468.903248

Figure 20: Best-scoring Candidate Solution After 250 Generations

ADJACENCY MATRIX:

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
01																																
02																																
03																																
04					*			*																								
05																*																
06				*	*			*																								
07				*	*			*																								
08																																
09																																
10																																
11												*			*																	
12															*																	
13											*	*			*																	
14											*	*			*																	
15																																
16																																
17																			*													
18																				*	*											
19																		*			*	*										
20																					*											
21																																
22																		*		*	*											
23																																
24																																
25																										*	*		*			
26																																
27																										*			*			
28																									*	*	*		*			
29																										*						
30																																
31																																
32																																

BIC: 657256.585419

Figure 21: Best-scoring Candidate Solution After 375 Generations

ADJACENCY MATRIX:

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
01																																
02																																
03																																
04								*																								
05				*				*																								
06				*	*			*																								
07				*	*			*																								
08																																
09																																
10																																
11												*			*																	
12															*																	
13											*	*			*																	
14											*	*			*																	
15																																
16																																
17																			*													
18																				*	*											
19																		*		*	*	*										
20																					*											
21																																
22																		*		*	*											
23																																
24																																
25																										*	*		*			
26																																
27																										*			*			
28																									*	*	*		*			
29																										*						
30																																
31																																
32																																

BIC: 651783.140737

ESTIMATED MIN ENTROPY : 17.050647

Figure 22: Best-scoring Candidate Solution After Final Generation (With Edge Pruning)

7 Future Work

There are several aspects of our approach that could be examined for possible improvements. Three noteworthy areas to consider are the implementation of the GA, the storage technique used for the adjacency matrices, and the computation of the probability of the most likely output bit sequence given a BN.

As discussed in Section 5.2, there are several choices regarding the implementation of a GA. The process of forming each new generation could be modified to make greater use of elitist selection to directly copy other candidates in addition to the best-scoring one. We could also consider a broader class of mutation types, giving the potential for more significant changes in a candidate. For example, this could include mutations that affect a larger number of edges. Further investigation of the effect of particular parameter values – various mutation rates, density of the adjacency matrices for the initial generation, and population size – may lead to more reliable convergence toward an optimal solution.

We currently use a two-dimensional array to store each of the adjacency matrices. Since in many real-world applications we do not expect many dependencies to exist, most of the adjacency matrices will be sparse. A more efficient implementation would utilize a sparse matrix structure. This would potentially be a significant benefit in the case of large block sizes.

We currently use the max-product variant of the variable elimination algorithm (explained in Section 4.3) to compute the probability of the most likely output block predicted by the BN. However, identifying the optimal variable ordering for this algorithm is an NP-hard problem. We use a straightforward topological ordering, with the understanding that we may forego an optimal ordering. This can result in the inability to complete this algorithm when estimating min-entropy, although the algorithm might have completed successfully using a different ordering. An alternative approach would be to investigate the effects of the belief propagation method for estimating min-entropy.

References

[1] Meltem Sonmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. *Recommendation for the Entropy Sources Used for Random Bit Generation*, Second DRAFT, NIST Special Publication 800-90B, January 2016.

[2] C.K. Chow and C.N. Liu. *Approximating Discrete Probability Distributions with Dependence Trees*, IEEE Transactions on Information Theory, Vol. IT-14, No. 3, May 1968.

[3] Thomas Corman, Charles Leiserson, Ronald Rivest, Clifford Stein. *Indroduction to Algorithms*, Third Ed., 2009.

[4] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*, The MIT Press, 2009.

[5] Tom M. Mitchell. *Machine Learning*, McGraw-Hill, 1997.

[6] Sheldon M. Ross. *Introduction to Probability Models*, 9th Edition, Academic Press, 2007.