A Well-Behaved Algorithm for Simulating Dependence Structures of Bayesian Networks

Y. Xiang and T. Miller

Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2

Abstract

Automatic generation of Bayesian network (BNs) structures (directed acyclic graphs) is an important step in experimental study of algorithms for inference in BNs and algorithms for learning BNs from data. Previously known simulation algorithms do not guarantee connectedness of generated structures or even successful genearation according to a user specification. We propose a simple, efficient and well-behaved algorithm for automatic generation of BN structures. The performance of the algorithm is demonstrated experimentally.

`Keywords:` directed acyclic graph, graph theory, simulation, Bayesian network.

1 Introduction

Bayesian networks (BNs) [8, 5] have been widely accepted as an effective formalism for inference with uncertain knowledge in artificial intelligent systems [4]. A BN uses a directed acyclic graph (DAG) to represent the dependence structure of a set of domain variables and an underlying probability distribution to quantify the uncertainty of the dependence. A BN can be constructed manually or *learned* from data by automatic construction. Once constructed, it can be used to compute the probability of values for some unobserved variables given observation of some other variables (called *inference*). Studies of better inference algorithms and learning algorithms are two of many active avenues of research.

The study of inference algorithms often includes testing of performance in different BNs (e.g., [7]). The study of learning algorithms often involves testing learning performance using controlled models, where a controlled model may be a given BN (e.g., [3, 11]). BNs used in these studies may be manually created or learned from data, but may also be randomly generated. The generation process creates both a network structure and an underlying probability distribution. The focus of this work is on the generation of the structure. Although DAGs of a small number of nodes can be simulated by a simple generate-and-test, it is not a trivial matter to cleanly create large structures with controlled topological features, as we shall demonstrate through the literature review. The contribution of this work is a well-behaved algorithm and a formal analysis of its properties.

We introduce the necessary terminology in Section 2. In Section 3, we present an overview of related work. In Section 4, we propose a well-behaved algorithm, the properties of which are formally analyzed in Section 5. We demonstrate its performance in Section 6 with experimental implementation and testing.

2 Terminology

For the purpose of this paper, we consider only directed graphs. A *directed graph* is denoted by $G = (V, E)$, where $V = (v_i | 0 \leq i < n, n > 0)$ is a set of nodes and $E = ((u, v) | u, v \in V, u \neq v)$ is a set of arcs. An arc (u, v) is *directed* from u (the *tail*) to v (the *head*). The node u is called a *parent* of v, and v is called a *child* of u.

For any node v, the *degree* $d(v)$ is the number of arcs containing v. The *in-degree* $d^-(v)$ is the number of arcs with head v, and the *out-degree* $d^+(v)$ is the number of arcs with tail v. A node v is a *root* if $d^-(v) = 0$. A node v is a *leaf* if $d^+(v) = 0$.

Two nodes u and v are *adjacent* if $(u, v) \in E$ or $(v, u) \in E$. A *path* is a sequence of nodes such that each pair of consecutive nodes is adjacent. A path is a *cycle* if it contains more than two nodes, and the first node is identical to the last node. A cycle C is *directed* if each node in C is the head of one arc in C and the tail of the other arc in C. A directed graph is *acyclic* or is a DAG if it contains no directed cycles. A graph is *connected* if there exists a path between every pair of nodes. A graph is a *tree* if there exists exactly one path between every pair of nodes; otherwise, the graph is *multiply connected*.

Given a directed graph G, if for each $(v_i, v_j) \in E$, we have $i < j$, then nodes in G are *indexed* according to a *topological order*. For simplicity, we shall say that G is indexed topologically. G can be indexed topologically if and only if it is a DAG.

A BN is a triplet (V, G, P). V is a set of domain variables. G is a DAG whose nodes are labeled by elements of V. The topology of G conveys the dependence/independence among variables through a graphical separation rule called *d-separation* [8]. P is a probability distribution over V. It is defined by specifying, for each node v in G, a distribution $P(v|\pi(v))$, where $\pi(v)$ is the parents of v. More on the semantics of BNs and their applications can be found in [8, 5, 4].

3 Related work

A randomly generated BN structure should satisfy certain topological features. First of all, it must be a DAG. A BN models a problem domain where variables are either directly or indirectly dependent on each other. Hence, the DAG must be connected. Other basic features include the number of root nodes, the number of leaf nodes, the sparseness of the graph, etc.

Although simulation of BN structures for experimental study has been widely used, algorithms used for random generation of DAGs are rarely published (e.g., [1, 6]). We review two published algorithms here:

Spirtes et al. [9] used a simple algorithm to simulate BNs for testing their learning algorithms. The algorithm takes the average node degree and the number of nodes as input from which a threshold p is computed. For each pair of variables, a random number $x \in [0, 1]$ is generated. The pair is connected with an arc if $x \leq p$. The algorithm has a complexity of $O(n^2)$. However, the graph produced may be disconnected.

Chu [2] designed an BN simulation algorithm also for testing learning algorithms. The algorithm takes three parameters as input: the number of nodes, the maximum in-degree d^-, and the maximum out-degree d^+. For each node v, $d^-(v)$ ($\leq d^-$) is assigned first. Then $d^-(v)$ parent nodes are selected from nodes whose out-degree is less than d^+. The algorithm has a complexity of $O(n^2)$. Given the input, it may fail, however, to generate a DAG accordingly since when choosing a parent for some node v, it may happen that all potential candidates have already reached out-degree limit d^+.

Although DAGs of a small number of nodes can be simulated by a simple generate-and-test, the above review demonstrates that it is not a trivial matter to *cleanly* create large DAGs with controlled topological features. Furthermore, the generation of a DAG *composed* of multiple subDAGs under certain constraints may be necessary, e.g., in study of algorithms for learning *embedded pseudo-independent models* [11] and in study of algorithms for inference in *multiply sectioned Bayesian networks* [10]. Generation of such composed structures should be based on a well-behaved algorithm for generating a single DAG.

In this work, we develop a simple algorithm. It generates, in a single pass, a connected DAG of a given number of nodes, a given number of roots and a given maximum in-degree.

4 Simulating DAG structures

We allow the user to specify the total number n of nodes, the total number r of root nodes, and the maximum in-degree m of any node. Unlike the algorithm in [2] which accepts and proceeds with any input parameters but may fail to satisfy them successfully, we identify conditions for unreasonable input which are then used to reject such input from the user at the outset:

A non-trivial graph must have at least two nodes ($n \geq 2$). A DAG has at least one root, and a connected DAG cannot have all nodes being roots. Hence, we require $1 \leq r < n$. The in-degree of each node must be less than n. Hence, we require $m < n$. To be a connected graph, it must be the case $m \geq 1$.

Given the above bounds, r and m still cannot be independently specified. For instance, given $n = 5$ and $r = 4$, the unique connected DAG has a single child node, which requires $m \geq 4$. We provide a constraint on parameters (n, r, m) specified as follows:

Condition 1 *Let n, r, m $(n \geq 2, 1 \leq r < n, 1 \leq m < n)$ be three positive integers such that*

- *if $r \geq m$, then $m\ (n - r) \geq n - 1$;*
- *otherwise*
$$m\ (n-m) + \frac{m\ (m-1) - r\ (r-1)}{2} \geq n-1.$$

Theorem 2 is a necessary condition of a connected DAG. It says that if (n, r, m) violates Condition 1, then no connected DAG can be constructed. Hence Condition 1 can be used to reject all invalid parameters before construction of a target DAG starts.

Theorem 2 *Let G be a connected DAG of n nodes, r roots and maximum in-degree m. Then (n, r, m) satisfies Condition 1.*

Proof:

Without loss of generality, we assume that G is indexed topologically and hence $v_0, ..., v_{r-1}$ are roots. Topological indexing implies that node v_i can have no more than i incoming arcs. Furthermore, each node can have no more than m incoming arcs. Let k denote the total number of arcs. Then $\sum_{i=r}^{n-1} \min(i, m) \geq k$. For any connected graph, $k \geq n - 1$. Hence we obtain $\sum_{i=r}^{n-1} \min(i, m) \geq n - 1$.

The summation can be simplified to reduce the complexity of verification from linear to constant: If $r \geq m$, then $\min(i, m) = m$ and we have $\sum_{i=r}^{n-1} \min(i, m) = m\ (n - r)$. If $r < m$, we have

$$\sum_{i=r}^{n-1} \min(i,m) = \sum_{i=r}^{m-1} i + \sum_{i=m}^{n-1} m = m\ (n-m) + \frac{m\ (m-1) - r\ (r-1)}{2}.$$

The result now follows. □

We propose an algorithm to generate a connected DAG given (n, r, m). Its pseudocode is presented as Algorithm 1. A brief explanation is as follows:

Algorithm 1

```
Input:  $(n, r, m)$ such that Condition 1 is satisfied.
Output: Return a directed graph $G$.
begin
1    if $r \geq m$, then $arcBound = m\ (n - r)$;
2    else $arcBound = m\ (n - m) + 0.5\ (m\ (m - 1) - r\ (r - 1))$;
3    select at random the number $e$ of arcs from $[n - 1, arcBound]$;

4    assign each $v_i$ $(0 \leq i \leq r - 1)$ an in-degree $d^-(v_i) = 0$;
5    assign each $v_i$ $(i \geq r)$ an in-degree $d^-(v_i)$ from $[1, \min(i, m)]$
         such that $\sum_{i=r}^{n-1} d^-(v_i) = e$;
6    for each $v_i$ $(i \geq r)$, set $p(v_i)$ (the number of parents
         left to connect) to $d^-(v_i)$;

7    set $G = (V, E)$ where $V = \{v_i | 0 \leq i \leq n - 1\}$ and $E = \emptyset$;
8    for $i = r$ to $n - 1$, do
9        insert $(v_j, v_i)$ in $E$, where $j$ is chosen randomly from $[r - 1, i - 1]$;
10       decrement $p(v_i)$;
11   for $i = 0$ to $r - 2$, do
12       insert $(v_i, x)$ in $E$, where $x$ is chosen randomly from the
             set $\{v_j | r \leq j \leq n - 1$ and $p(v_j) \geq 1\}$;
13       decrement $p(x)$;
14   for $i = r$ to $n - 1$, do
15       while $p(v_i) \geq 1$, do
16           insert $(x, v_i)$ in $E$, where $x$ is chosen randomly from the
                 set $\{v_j | 0 \leq j \leq i - 1$ and $(v_j, v_i) \notin E\}$;
17           decrement $p(v_i)$;
18   return G;
end
```

Lines 1 to 3 compute the number e of arcs for the target DAG.

Line 4 specifies v_0 to v_{r-1} as roots. Line 5 specifies the number of incoming arcs for each nonroot node such that each nonroot has at least one incoming arc and the total number of arcs in the DAG is e. The number is copied to a counter in line 6.

The remaining lines add the e arcs to the graph. Line 7 initializes the empty graph. The for loop in lines 8 through 10 connects nodes v_{r-1} through v_{n-1} into a tree with v_{r-1} as the unique root. The for loop in lines 11 through 13 enlarges the tree by adding v_0 to v_{r-2}. Finally, lines 14 through 17 add the remaining number of arcs (if any) to the tree, making it a multiply connected DAG of e arcs.

5 Properties of the algorithm

Theorem 2 is a necessary condition of connected DAG. It is still unclear whether a connected DAG exists given a triple (n, r, m) satisfying Condition 1. Below we follow the approach of a constructive proof to show that it is indeed the case. In Theorem 3, we show that given any input that satisfies Condition 1, Algorithm 1 will return a graph successfully. In Theorems 4 and 5, we show that the returned graph is a DAG and is connected. In Theorem 6, we show that the returned graph is consistent with the given input (n, r, m).

Theorem 3 *Given any valid input, Algorithm 1 executes to completion.*

Proof:

Lines 1 to 3 compute the number of arcs. They will succeed since the input satisfies Condition 1 and ensures $arcBound \geq n-1$ (Theorem 2).

Line 4 will succeed since $1 \leq r < n$. For line 5, we have $min(i, m) \geq 1$ since $r \geq 1$ and $m \geq 1$. From the proof of Theorem 2, $d^-(v_i)$s can be found to make the equation hold.

Line 9 in the for loop will succeed since $r-1 \geq 0$ and $i-1 \geq r-1$. Line 10 will succeed since $p(v_i) \geq 1$ by line 5. The loop will add $n-r$ arcs to G. Since $e \geq n-1$ (line 3), there are at least $(n-1)-(n-r) = r-1$ arcs not yet added. Hence lines 11 through 13 will succeed, which add exactly $r-1$ arcs to G.

Each iteration of the last for loop is executed conditioned on $p(v_i) \geq 1$. No matter the condition holds or not, it always succeeds. □

Theorem 4 *The graph G returned by Algorithm 1 is a DAG and is topologically indexed.*

Proof:

Arcs are added to G in lines 9, 12 and 16. For each arc (v_i, v_j) added, we have $i < j$ and hence G is acyclic and is topologically indexed. □

Theorem 5 *The DAG G returned by Algorithm 1 is connected.*

Proof:

First, we show that lines 8 to 10 connect v_{r-1} through v_{n-1}. In the first iteration of the for loop, (v_{r-1}, v_r) is added, making the two nodes connected. In the second iteration, either (v_{r-1}, v_{r+1}) or (v_r, v_{r+1}) is added, making the three nodes $v_{r-1},\ v_r,\ v_{r+1}$ connected. In each of the remaining iteration, one additional node is connected to the above connected subgraph, and hence the for loop connects v_{r-1} through v_{n-1}.

In lines 11 through 13, each of the remaining $r-1$ roots is connected to the above connected subgraph. Since arcs are only added (vs. deleted) afterwards, the returned graph G is connected. □

Theorem 6 *The DAG G returned by Algorithm 1 satisfies the parameters (n, r, m).*

Proof:

Line 7 sets the number of nodes to n.

Lines 4 and 5 set the in-degrees of only v_0 to v_{r-1} to zero. In the remaining part of the algorithm, these nodes are given no parents, but each other node is given at least one parent (lines 8 to 10). Hence G has exactly r roots.

Line 5 assigns an in-degree to each node that is less than or equal to m, which is respected in the remaining part of the algorithm. Hence the parameter m is satisfied. □

The above four theorems imply that given any input (n, r, m) that observes Condition 1, Algorithm 1 guarantees to return a connected DAG accordingly. Hence, Condition 1 is both necessary and sufficient to construct a connected DAG.

We now analyze the complexity of Algorithm 1. The complexity of lines 1 through 6 is $O(n)$. Lines 7 through 17 have exactly e nontrivial iterations. From lines 1, 2 and 3, it is easy to see that the complexity is $O(m\ n)$. Hence the complexity of Algorithm 1 is $O(m\ n)$. It is more efficient than both algorithms in [9, 2].

It is worth mentioning that this algorithm requires no backtracking (i.e. trial and error), and is hence easy to comprehend. This also contributes to its efficiency.

6 Experimental results

The algorithm has been implemented in Java as one module in the WEBWEAVR-III toolkit [4]. Below we demonstrate the performance using a few different sets of parameters.

Our implementation enforces the necessary conditions developed in Section 4. It checks the user input before proceeding to execute Algorithm 1. The following shows rejection of two sets of invalid input which may look reasonable to a user:

```
spither[339]% java RandBn.RandBn 50 10 1 jk.bn
n=50 r=10 m=1 file=jk.bn
Invalid input entered.
spither[340]% java RandBn.RandBn 50 30 2 jk.bn
n=50 r=30 m=2 file=jk.bn
Invalid input entered.
```

In the following, we show four DAGs generated with valid input. They are displayed using the Editor module of WEBWEAVR-III. Figure 1 (*left*) is a DAG generated using the parameters $(10, 1, 1)$; it is a tree with a single root. Figure 1 (*right*) contains more nodes. It is also a tree but has more roots.

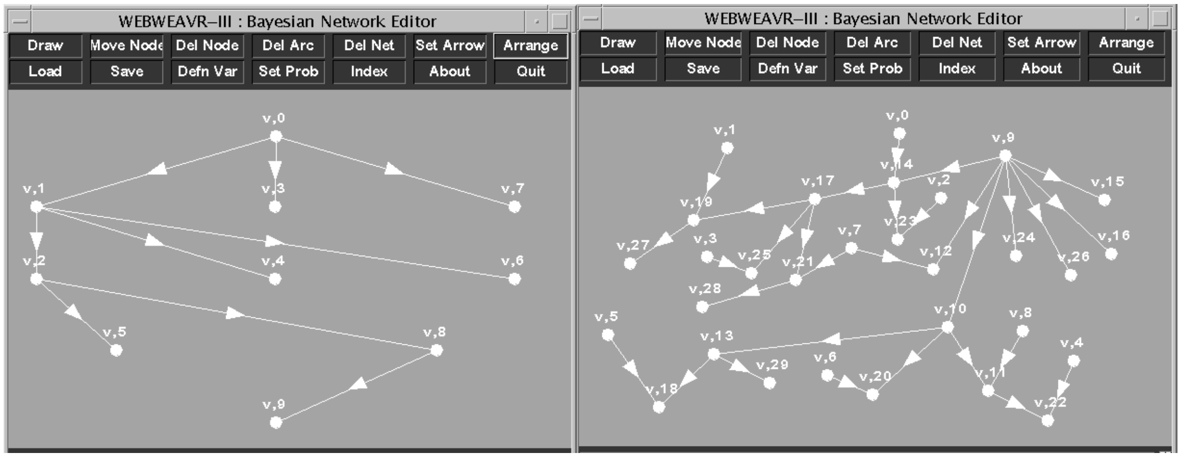

Figure 1: *Left*: a single-rooted tree simulated with $(10, 1, 1)$. *Right*: a multi-rooted tree simulated with $(30, 10, 2)$.

Figure 2 (*left*) is a multiply connected DAG with a few undirected cycles. It is generated by $(30, 10, 2)$. Figure 2 (*right*) is also multiply connected but is much more densely connected due to the larger m value.

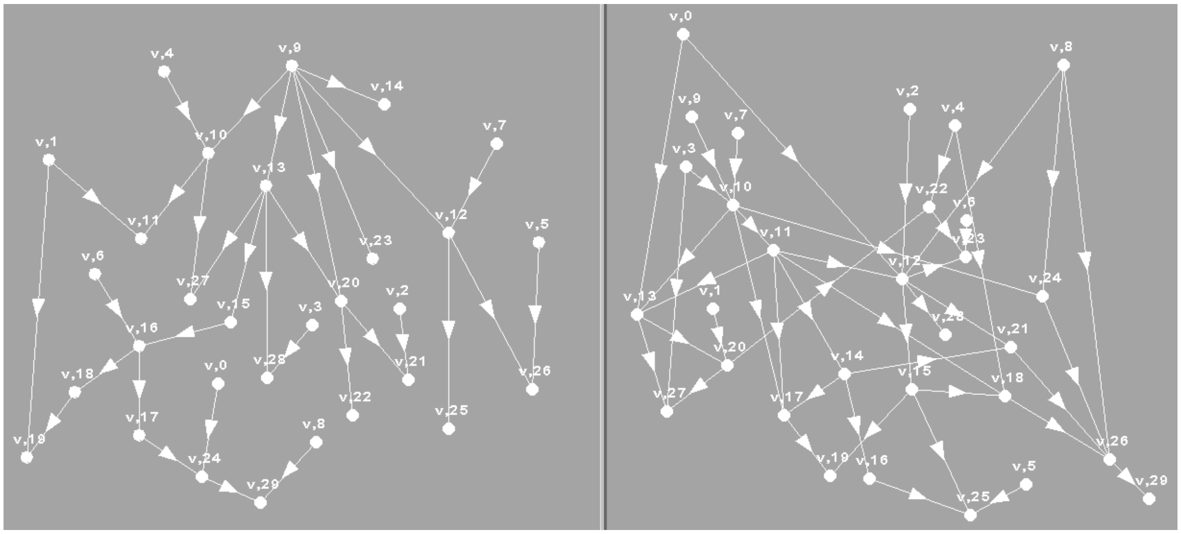

Figure 2: *Left*: a sparse multiply connected DAG simulated with $(30, 10, 2)$. *Right*: a dense multiply connected DAG simulated with $(30, 10, 4)$.

7 Conclusion

We have presented a simple algorithm to simulate the structure of a BN. It creates randomly a connected DAG with a given number n of nodes, a given number r of roots and a given maximum in-degree m. We have identified a necessary and sufficent condition of the input parameters to construct such a DAG. Unlike other known algorithms which may fail to generate a connected DAG with a given requirement, our algorithm, equipped with this condition, will reject any invalid input and guarantees successful generation in a single pass without backtracking. It is also more efficient than existing algorithms, with a complexity of $O(m\ n)$ instead of $O(n^2)$. The performance has been demonstrated through experimental implementation and testing.

Acknowledgement

This work is supported by the Research Grant OGP0155425 from NSERC to the first author. Assistance from Chong Li on literature search is acknowledged.

References

[1] R.R. Bouckaert. Properties of Bayesian belief network learning algorithms. In R. Lopez de Mantaras and D. Poole, editors, *Proc. 10th Conf. on Uncertainty in Artificial Intelligence*, pages 102–109, Seattle, Washington, 1994. Morgan Kaufmann.

[2] T. Chu. Parallel learning of belief networks from large databases. Master's thesis, University of Regina, 1997.

[3] G.F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, (9):309–347, 1992.

[4] P. Haddawy. An overview of some recent developments in bayesian problem-solving techniques. *AI Magazine*, 20(2):11–19, 1999.

[5] F.V. Jensen. *An introduction to Bayesian networks*. UCL Press, 1996.

[6] W. Lam and F. Bacchus. Learning Bayesian networks: an approach based on the MDL principle. *Computational Intelligence*, 10(3):269–293, 1994.

[7] A.L. Madsen and F.V. Jensen. Lazy propagation in junction trees. In *Proc. 14th Conf. on Uncertainty in Artificial Intelligence*, 1998.

[8] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[9] P. Spirtes, C. Glymour, and R. Scheines. *Causation, prediction, and Search*. Springer-Verlag, 1993.

[10] Y. Xiang. A probabilistic framework for cooperative multi-agent distributed interpretation and optimization of communication. *Artificial Intelligence*, 87(1-2):295–342, 1996.

[11] Y. Xiang, S.K.M. Wong, and N. Cercone. A 'microscopic' study of minimum entropy search in learning decomposable Markov networks. *Machine Learning*, 26(1):65–92, 1997.