Generating Dependence Structure of Multiply Sectioned Bayesian Networks

Y. Xiang and X. An
Dept. of Computing and Information Science
University of Guelph, Guelph, Ontario, Canada N1G 2W1
yxiang@snowhite.cis.uoguelph.ca, xan@uoguelph.ca

Abstract

Multiply sectioned Bayesian networks (MSBNs) provide a general and exact framework for multi-agent distributed interpretation. To investigate algorithms for inference and other operations , experimental MSBNs are necessary. However, it is very time consuming and tedious to construct MSBNs manually. In this work, we investigate pseduo-random generation of MSBNs. Our focus is on the generation of MSBN structures.

Pseduo-random generation of MSBN structures can be performed by a generate-and-test approach. We expect such approach to have a very low probability of generating legal MSBN structures that satisfy all the technical constraints, and hence will be inefficient. We propose a set of algorithms that always generates legal MSBN dependence structures.

1 Introduction

Multiply sectioned Bayesian networks (MSBNs) provides a coherent framework for probabilistic inference in a large domain (XPB93). It can be applied under a single agent paradigm (XPE+93) or a cooperative multi-agent paradigm (Xia96). It supports hierarchical model-based reasoning (SRA90) and object-oriented inference (KP97). To investigate issues involved in the verification (Xia98), compilation (Xia01), inference (XJ99; Xia00), and other operations, and test algorithms for resolving such issues, experimental MSBNs are necessary. For example, to compare the performance of the inference algorithms in (XJ99) and (Xia00), it is desirable to run the algorithms in a number of MSBNs of different structures. Since MSBNs are intended for very large domains and a legal MSBN must satisfy a set of technical conditions, it is time consuming and tedious to construct MSBNs manually. We investigate pseudo-random generation of MSBNs. Our focus is on the generation of MSBN structures.

Pseudo-random generation of MSBN structures can be performed by a generate-and-test approach. Due to the large number of variables to be generated and the technical conditions to be satisfied simultaneously, we expect such approach to have a very low probability of generating legal MSBN structures. We take the same approach in (XM99) where an algorithm that guarantees the generation of a legal Bayesian network structure is presented. We propose a set of algorithms for pseudo-random generation that when terminates will produce legal MSBN structures. Inference computation can be effectively performed only for sparse Bayesian networks. The same is true for MSBNs. Our algorithms can produce MSBNs which may or may not be sparse, although parameters in our algorithms can be adjusted to produce only sparse MSBNs.

2 Overview of MSBNs

A BN (Pea88) S is a triplet (N, D, P) where N is a set of domain variables, D is a DAG whose nodes are labeled by elements of N, and P is a joint probability distribution (jpd) over N. A MSBN (XPB93; Xia96) M is a collection of Bayesian subnets that together defines a BN. These subnets are required to satisfy certain conditions. One condition requires that nodes shared by different subnets form a *d-sepset*, as defined below.

Let $G_i = (N_i, E_i)$ $(i = 0, 1)$ be two graphs. The graph $G = (N_0 \cup N_1, E_0 \cup E_1)$ is referred to as the *union* of G_0 and G_1, denoted by $G = G_0 \sqcup G_1$.

Definition 1 *Let $D_i = (N_i, E_i)$ $(i = 0, 1)$ be two DAGs such that $D = D_0 \sqcup D_1$ is a DAG. The intersection $I = N_0 \cap N_1$ is a* `d-sepset` *between D_0 and D_1 if for every $x \in I$ with its parents π in D, either $\pi \subseteq N_0$ or $\pi \subseteq N_1$. Each $x \in I$ is called a* `d-sepnode`.

The structure of a MSBN is a multiply sectioned DAG (MSDAG) with a hypertree organization:

Definition 2 *A* `hypertree MSDAG` *$\mathcal{D} = \bigsqcup_i D_i$, where each D_i is a DAG, is a connected DAG constructible by the following procedure:*

Start with an empty graph (no node). Recursively add a DAG D_k, called a `hypernode`*, to the existing MSDAG $\bigsqcup_{i=0}^{k-1} D_i$ subject to the constraints:*

[d-sepset] For each D_j $(j < k)$, $I_{jk} = N_j \cap N_k$ is a d-sepset when the two DAGs are isolated.

[Local covering] There exists D_i $(i < k)$ such that, for each D_j $(j < k; j \neq i)$, we have $I_{jk} \subseteq N_i$. For an arbitrarily chosen such D_i, I_{ik} is the `hyperlink` *between D_i and D_k which are said to be* `adjacent`.

A MSBN is defined as follows:

Definition 3 *An MSBN M is a triplet $(\mathcal{N}, \mathcal{D}, \mathcal{P})$. $\mathcal{N} = \bigcup_i N_i$ is the* `total universe` *where each N_i is a set of variables. $\mathcal{D} = \bigsqcup_i D_i$ (a hypertree MSDAG) is the* `structure` *where nodes of each DAG D_i are labeled by elements of N_i. Let x be a variable and $\pi(x)$*

be all parents of x in $\mathcal{D}$. For each x, exactly one of its occurrences (in a D_i containing $\{x\} \cup \pi(x)$) is assigned $P(x|\pi(x))$, and each occurrence in other DAGs is assigned a uniform distribution. $\mathcal{P} = \prod_i P_{D_i}$ is the `jpd`*, where each P_{D_i} is the product of the probability tables associated with nodes in D_i. A triplet $S_i = (N_i, D_i, P_{D_i})$ is called a* `subnet` *of M. Two subnets S_i and S_j are said to be adjacent if D_i and D_j are adjacent.*

Each hypernode forms the basic local knowledge representation of an agent. The agents cooperate to interpret their local and global environment (such as in trouble-shooting a complex equipment, or in evaluating a distributed design) and to act based on the interpretation.

3 Generating hypertree MSDAG

Pseudo-random generation of MSBN dependence structure should satisfy a set of technical constraints. A *legal* structure should have the following properties:

- A tree organization for the hypernodes.
- Each hypernode is a connected DAG.
- The union of all DAGs is also a connected DAG.
- The shared nodes by each pair of DAGs form a d-sepset.
- The shared nodes by hypernodes in the hypertree satisfy local covering.

Furthermore, as the sparseness is for Bayesian networks, to allow effective inference with a MSBN, its dependence structure should be sparse. This means not only a sparse DAG structure at each hypernode, but also a sparse structure of each d-sepset (a hyperlink in the hypertree).

Finally, generated structures should display varieties of legal structures. That is, if a MSBN dependence structure satisfies all the above properties, then the generator should be able to generate it with a non-zero probability. For instance, a pair of adjacent hypernodes share a set I of d-sepnodes. It is possible that I is shared by no other hypernode. But it is also legal that I or a subset of I is shared by additional hypernodes. It is desirable for the generator to be able to produce both type of d-sepsets.

We aim at designing generation algorithms that do not backtrack. That is, the algorithms do not make mistakes for each decision made in the generation process. A *mistake* is considered as a generation decision/action that will result in an illegal generated structure no matter how the subsequent decisions are made. To ensure the generation algorithms backtrack-free, we adopt a top-down approach: That is, we first generate the *macro* aspect of the structure and add *micro* details stepwise.

We allow the user to specify the following input parameters.

- n: the total number of hypernodes ($n \geq 3$).
- d: the maximum degree of each hypernode ($d \geq 2$), where the *degree* of a hypernode D_i is the number of its adjacent hypernodes on the hypertree.

We propose the following top level algorithm for the generator:

Algorithm 1

1 create a hypertree topology;
2 create a sparse structure for each hyperlink;
3 create a junction tree for each hypernode;
4 convert each junction tree to a chordal graph;
5 convert each chordal graph to a DAG;

The hypertree topology is determined at the onset without specifying the details of any hypernodes and hyperlinks. Next, a sparse structure is created for each hyperlink. The structure is *secondary* in that it is not a directed graph but rather is a junction tree (called a *linkage tree*) where each cluster is a subset of shared variables (by the corresponding pair of hypernodes). In step 3, a sparse structure is created for each hypernode. Again, the structure is secondary. This structure is made consistent with the structures of all the hyperlinks that are incident to the hypernode. All the macro features have thus been determined. In step 4, each hypernode structure is converted into an undirected graph structure and in step 5 further converted into a DAG. The dependence structure of a MSBN is then completed.

In subsequent sections, we present and illustrate the algorithm for each step. Due to the space limit, we present the formal justification of the algorithms elsewhere.

4 Creating hypertree topology

The following algorithm generates a tree such that each node has a bounded degree. It is used to create the hypertree topology.

Algorithm 2 (GetBoundedTree)

input: the number $n \geq 3$ of nodes and the
maximum degree $d \geq 2$ of each node.
begin
  create a set V of n nodes;
  associate each node $v \in V$ with a variable $b_v = d$;
  pick $u \in V$ randomly, set $V = V \setminus \{u\}$, and
    create a graph $G = (\{u\}, \emptyset)$;
  while V is not empty, do
    pick $x \in V$ randomly and set $V = V \setminus \{x\}$;
    pick a node y randomly from G where $b_y > 0$;
    add x to G and connect x with y;
    $b_x = b_x - 1$ and $b_y = b_y - 1$;
  return G;
end

In Figure 1(a), V initially contains 5 nodes with the maximum degree $d = 3$. The node h_2 is used to create the first node of the graph G (b). In (c), the node h_0 is removed from V and connected with h_2. Afterwards,

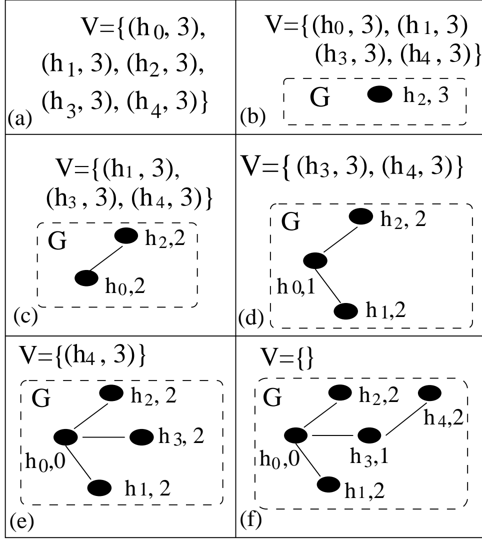

Figure 1: Illustration of GetBoundedTree

we have $b_{h_2} = 2$ and $b_{h_0} = 2$. In (d), h_0 is selected to connect with the new node h_1 resulting in $b_{h_0} = 1$ and $b_{h_1} = 2$. In (e), h_0 is selected to connect with the new node h_3 resulting in $b_{h_0} = 0$ and $b_{h_3} = 2$. Finally, h_4 is added to the tree and the result G is shown in (f).

The graph G will be used as the topology of the hypertree MSDAG.

5 Expanding a junction tree

We adopt the strategy of generating hyperlinks first and then hypernodes in order to ensure the local covering condition. Therefore, we need to construct hyperlink structures that are consistent with each other even though the hypernodes to which they are incident have not been determined yet. To do that, we use the following idea: If a linkage tree L_1 has been generated and another linkage tree L_2 is to be generated such that it shares some variables with L_1, then the graph separation relations in L_1 should not be invalidated. We therefore randomly take a subtree (possibly null) of L_1 and expand it into L_2 with the graphical separation in L_1 preserved. The expansion is defined in Algorithm 3.

The first part of the algorithm (the *if* section) produces a single cluster junction tree when the input T is null. The remaining part of the algorithm expands T by either enlarging an existing cluster or adding a new cluster.

In Figure 2, a set V of variables used to expand a junction tree T is shown in (a). In (b), a subset $\{g, h\}$ of V is removed to expand cluster c_1. In (c), i is removed from V to expand cluster c_2 and in (d), j is used to expand c_3. In (e), k is used to form a new cluster c_4. Note that a random separator between c_2 and c_4 is selected. In (f), the remaining variables form a new cluster c_5 in T.

An important property of the algorithm is that it preserves the graph separation defined by the input junction tree T.

Algorithm 3 (ExpandJunctionTree)
input: a set $V \neq \emptyset$ *and a junction tree* T *over*
    a set U *such that* $U \cap V = \emptyset$.
begin
  if T *is null*
    randomly select $C \subseteq V$ $(C \neq \emptyset)$;
    $V = V \setminus C$;
    set T *to have the single cluster* C;
  while V *is not empty, do*
    randomly select $X \subseteq V$ $(X \neq \emptyset)$;
    $V = V \setminus X$;
    randomly select a cluster Q *in* T;
    randomly select a boolean value b;
    if $b = true$, `expand` *cluster* $Q = Q \cup X$;
    else
      randomly select separator $S \subset Q$ $(S \neq \emptyset)$;
      `create` *a new cluster* $X \cup S$;
      connect the new cluster with Q *in* T;
  return T;
end

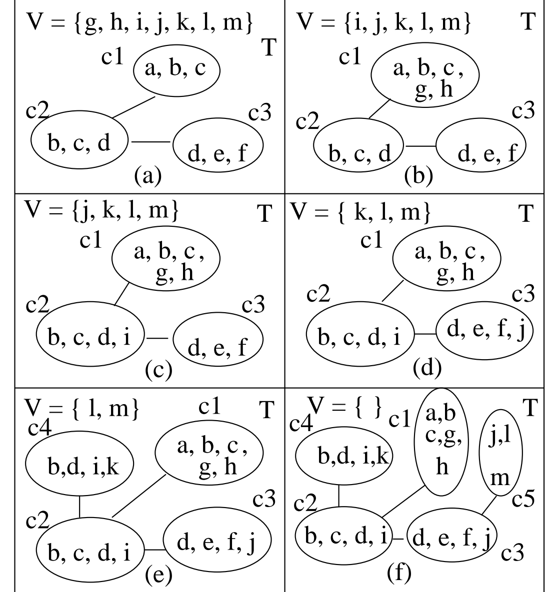

Figure 2: Illustration of ExpandJunctionTree

The linkage trees (corresponding to the hyperlinks in the hypertree MSDAG) are created using the above algorithm as follows: Consider Figure 1 (f) as the hypertree topology. The hypertree is traversed breadth-first starting at an arbitrary hyperlink, say, $\{h_0, h_2\}$. A linkage tree will be created using ExpandJunctionTree with a set V and a null T. Afterwards, two subtrees will be selected from the resultant T. One is expanded into the linkage tree for hyperlink $\{h_0, h_1\}$. The other is expanded for the linkage tree of $\{h_0, h_3\}$, followed by the selection of its subtree which is expanded into the linkage tree for $\{h_3, h_4\}$.

6 Merging two junction trees

Once the linkage trees are created, we create a junction tree for each hypernode H by merging the linkage trees

from all hyperlinks incident to H and expanding the result. We present the merging algorithm below. The key requirement of the algorithm is to preserve graph separation relations in all linkage trees merged. We first introduce some terminology to facilitate the description of the algorithm.

Let T be a junction tree, T' a (possibly null) expansion of T, R a (possibly null) subtree of T, and T^* an expansion of R. We shall call T a *source* JT, T' an *expansion* of T, and T^* a *partial expansion* of T. We can associate a boolean flag *marked* with each cluster in a junction tree. A cluster is said to be *marked* if the flag is true, otherwise it is *unmarked*. Let Y' be an expansion (possibily partial) of a junction tree Y. We mark the clusters in Y' as follows: If a cluster C in Y' is *expanded* from a cluster Q in Y, C will be marked and Q is called the *source* cluster of C. If a cluster in Y' is *created*, it will be unmarked. With clusters thus marked, we say that Y' is a *marked* expansion of Y. Note that there is an one-to-one mapping between marked clusters in Y' and clusters in Y. Algorithm 4 defines the merging operation.

Algorithm 4 (MergeJunctionTree)

input: A source junction tree T, a marked expansion T' of T and a marked partial expansion T^* of T.
begin
  search T^* for a marked cluster;
  if no such cluster is found, do
    pick randomly a cluster C in T^* and Q in T';
    randomly select $S_C \subset C$ and $S_Q \subset Q$;
    expand C with S_Q and Q with S_C;
    merge T^* into T' by connecting C to Q;
  else
    search T^* for a marked cluster C with a single adjacent marked cluster C';
    while C is found, do
      delete the link between C and C', which split T^* into two subtrees;
      denote the subtree rooted at C by R_C and denote the other subtree by T^*;
      search T' for a cluster Q that has the same source cluster with C;
      if $Q \supseteq C$, remove each subtree rooted at C from R_C and connect it to Q;
      else if $Q \subset C$, merge R_C into T' by replacing Q with C;
      else merge R_C into T' by connecting C to Q;
      search T^* for a marked cluster C with a single adjacent marked cluster C';
    if C is not found, do
      denote the single marked cluster in T^* by C;
      search T' for a cluster Q that has the same source cluster with C;
      if $Q \supseteq C$, merge T^* into T by connecting each subtree of T^* rooted at C to Q;
      else if $Q \subset C$, merge T^* into T' by replacing Q with C;
      else merge T^* into T' by connecting C to Q;
  return T';
end

The if section of the algorithm processes the case where T^* contains no marked clusters. Merging is performed by expanding a pair of clusters one from each JT to form a common separator, and connecting the two clusters.

The rest of the algorithm processes the case where T^* contains marked clusters. The *while* section handles the case where there are more than one marked cluster. The marked clusters form a connected subtree in T^*. The algorithm recursively removes a *terminal* cluster C from this subtree, together with the subtree rooted at C that contains no marked clusters. The subtree is connected to its source cluster Q in three possible ways (the $if/else/else$ section), depending on whether C is not expanded with new elements ($Q \supseteq C$), or C is expanded but Q is not ($Q \subset C$), or both are expanded. When the 'marked subtree' is reduced to one marked cluster, the *if* section that follows the *while* section will be entered. The subtree rooted at the last marked cluster will be merged into T similarly.

Figure 3 and 4 illustrate the algorithm MergeJunctionTree:

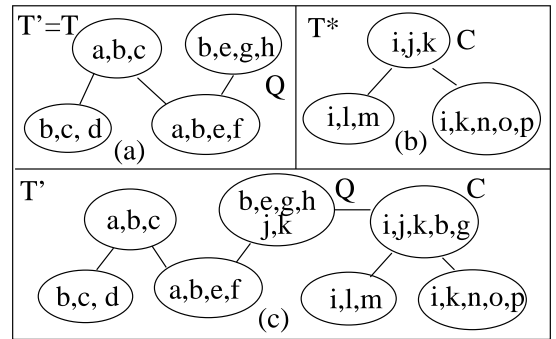

Figure 3: Illustration of MergeJunctionTree where T^* contains no marked cluster.

In Figure 3, the junction tree T' is the same as the source tree T and the junction tree T^* is an expansion of a null subtree of T. A cluster Q in T' and another cluster C in T^* are randomly selected to connect with each other and merge T' and T^*. The separator is constructed by randomly selecting a subset $\{j, k\}$ from C and a subset $\{b, g\}$ from Q.

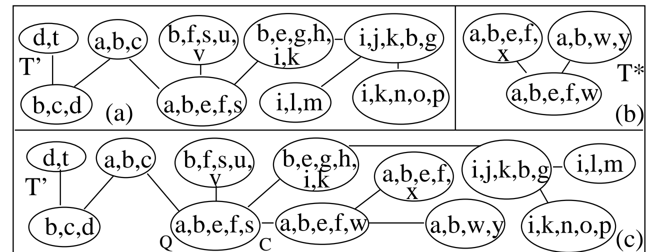

Figure 4: Illustration of MergeJunctionTree where clusters C and Q are incomparable

In Figure 4, an expansion T' (a) of T (as in Fig. 3 (a)) and a partial expansion T^* (b) of T are shown. In (c), a cluster C in T^* and its corresponding cluster Q in T' are selected. Because Q and C are incomparable, T^* is merged into T' by connecting C to Q.

After a junction tree for each hypernode is created, a linked junction forest is defined. Next, we convert each junction tree into a DAG.

7 Converting junction forest into MSDAG

To convert the junction tree at each hypernode into a DAG, we first convert the junction tree into its equivalent chordal graph. This operation is trivial. We then convert the chordal graph into a DAG by orienting the links into arcs with the possibility of removal of some links. The criteria for this step is the following:

1. The resultant graph (a local graph) at each hypernode is acyclic.
2. The union of local graphs is acyclic.
3. The d-sepset condition is maintained.
4. The set of all local graphs will compile to the same linked junction forest.

These conditions together ensure the resultant is a hypertree MSDAG and respects all the graphical separations defined by the previous operations.

The link orientation operation is performed by traversing the hypertree in a breadth-first fashion starting at an arbitrary hypernode. The operation at the first hypernode is performed by calling the following recursive algorithm DirectArc at a terminal cluster of the local junction tree.

Algorithm 5 (DirectArc) *Let Q be a cluster in a JT H and G be the corresponding chordal graph of H. A caller is either an adjacent cluster or H. When the caller calls in Q, it does the following:*

1. *If caller is a cluster, denote caller by C, else denote the unique adjacent cluster of Q by C.*
2. *Direct the link from each node in $Q \cap C$ to each node in $Q \setminus C$ in G.*
3. *Pick up randomly $z \in Q \setminus C$.*
4. *Direct the link from each node in $Q \setminus (C \cup \{z\})$ to z in G.*
5. *Remove randomly the remaining links among nodes in Q.*
6. *For each node $x \in Q$ with unoriented links, direct the links to x with x selected in random order.*
7. *Q calls DirectArc in each adjacent cluster in H except caller.*

Figure 5 illustrates algorithm DirectArc. H and G are shown in (a) and (b). DirectArc is called on the cluster $\{a, b, c\}$. Its separator with the cluster $\{b, c, d, e, f, g\}$ is $\{b, c\}$. In (c), links in cluster $\{a, b, c\}$ are directed away from b and c. The remaining links are randomly removed. In this case the only link $\{b, c\}$ is removed. Afterwards, the processing is moved to the cluster $\{b, c, d, e, f, g\}$. Again, links are directed away from b and c. This leaves links among d, e, f, g undirected. In (d), d is selected as the common child of e, f, g. In (e), the link $\{e, g\}$ is removed, and the remaining links are oriented as shown in (f). In (g), the processing is shifted to the last cluster. Links are directed away from $\{c, d, e\}$. Finally, the remaining one link is oriented as in (h).

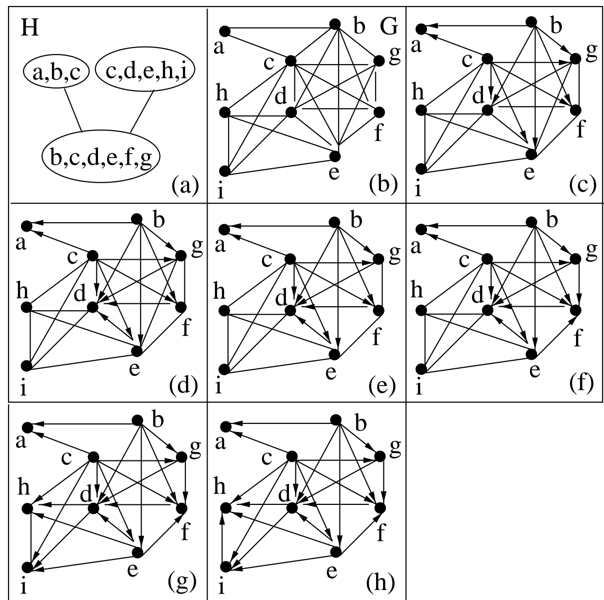

Figure 5: Illustration of DirectArc

After the first local graph is directed, the remaining local graphs are processed traversing the hypertree breath-first in a caller/callee fashion similar to the algorithm DirectArc (but at the hypertree level). Denote the caller by (H', G') and the callee by (H, G). When called by (H', G'), (H, G) executes the following algorithm DirectHnode. We define the *peer* concept to facilitate the description of the algorithm:

G and G' share a set of nodes. A cluster Q in H may contain one or more shared nodes. We call the set of shared nodes contained in Q the *peer set* of Q which we denote as $peer(Q)$. We say that Q has a *maximum peer* if no other cluster Q' satisfies $peer(Q') \supset peer(Q)$.

Algorithm 6 calls the algorithm DirectArc-d which is similar to DirectArc except that when a cluster Q with a maximum peer is called, it does nothing locally but calls the next adjacent cluster.

Since G' is a directed graph, after the first step of the algorithm, for each cluster, links among d-sepnodes are all directed. In the first statement of the *for* loop, each non-d-sepnode is made as the child of all the d-sepnodes of the cluster. In the rest of the *for* loop, some links among these non-d-sepnodes are deleted and the remaining are oriented.

Algorithm 6 (DirectHnode)
When DirectHnode is called in a hypernode (H, G) by an adjacent hypernode (H', G'), the following is performed at (H, G):

begin
copy all arcs among shared nodes between (H, G) and (H', G') from G';
remove each undirected link among shared nodes
for each cluster Q in H that has a maximum peer $C = peer(Q)$, do
for each non-shared node in Q, make it the child of each shared node in Q;
pick randomly $z \in Q \setminus C$;
direct the link from each node in $Q \setminus (C \cup \{z\})$ to z in G;
remove randomly the remaining links among nodes in Q;
for each node $x \in Q$ with unoriented links, do direct links to x with x selected in random order;
call DirectArc-d on a cluster Q with maximum peer;
end

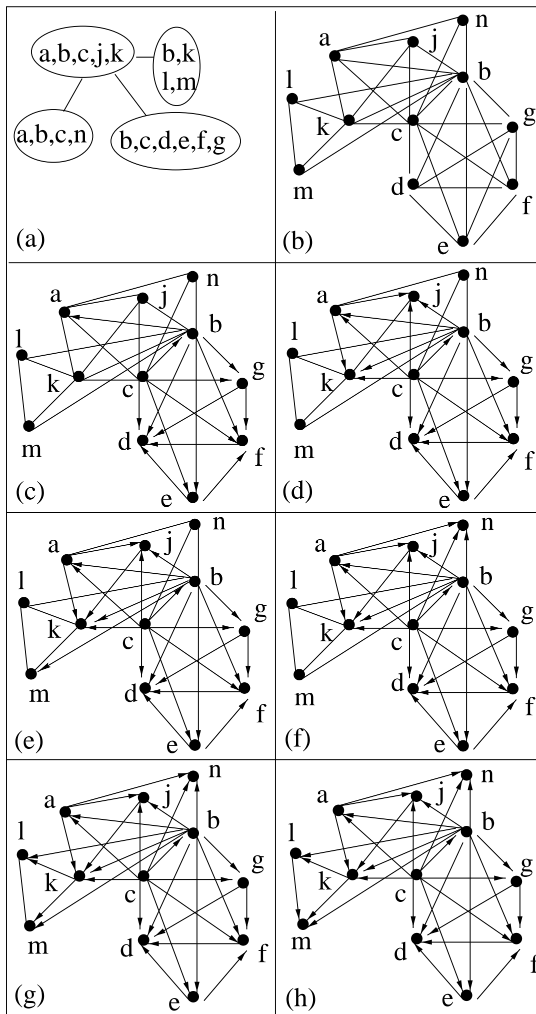

Figure 6: Illustration of DirectHnode

Figure 6 illustrates algorithm DirectHnode where the d-sepset is $\{a, b, c, d, e, f, g\}$. Clusters $\{a, b, c, j, k\}$, $\{a, b, c, n\}$ and $\{b, c, d, e, f, g\}$ each has a maximum peer. H and G are shown in (a) and (b). Arc copying from G' is shown in (c). Cluster $\{b, c, d, e, f, g\}$ requires no further processing. The processing of cluster $\{a, b, c, j, k\}$ is shown in (d) and (e). The processing of cluster $\{a, b, c, n\}$ is shown in (f). After DirectArc-d is called on, say, $\{a, b, c, n\}$, eventually, the cluster $\{b, k, l, m\}$ is processed, which is shown in (g) and (h).

References

[KP97] D. Koller and A. Pfeffer. Object-oriented Bayesian networks. In D. Geiger and P.P. Shenoy, editors, *Proc. 13th Conf. on Uncertainty in Artificial Intelligence*, pages 302–313, Providence, Rhode Island, 1997.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[SRA90] S. Srinivas, S. Russell, and A. Agogino. Automated construction of sparse Bayesian networks for unstructured probabilistic models and domain information. In M. Henrion, R.D. Shachter, L.N. Kanal, and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence 5*, pages 295–308. North-Holland, 1990.

[Xia96] Y. Xiang. A probabilistic framework for cooperative multi-agent distributed interpretation and optimization of communication. *Artificial Intelligence*, 87(1-2):295–342, 1996.

[Xia98] Y. Xiang. Verification of dag structures in cooperative belief network based multi-agent systems. *Networks*, 31:183–191, 1998.

[Xia00] Y. Xiang. Belief updating in multiply sectioned Bayesian networks without repeated local propagations. *Inter. J. Approximate Reasoning*, 23:1–21, 2000.

[Xia01] Y. Xiang. Cooperative triangulation in MSBNs without revealing subnet structures. *Networks*, 37(1):53–65, 2001.

[XJ99] Y. Xiang and F.V. Jensen. Inference in multiply sectioned Bayesian networks with extended Shafer-Shenoy and lazy propagation. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence*, pages 680–687, Stockholm, 1999.

[XM99] Y. Xiang and T. Miller. A well-behaved algorithm for simulating dependence structures of Bayesian networks. *Inter. J. Applied Mathematics*, 1(8):923–932, 1999.

[XPB93] Y. Xiang, D. Poole, and M. P. Beddoes. Multiply sectioned Bayesian networks and junction forests for large knowledge based systems. *Computational Intelligence*, 9(2):171–220, 1993.

[XPE+93] Y. Xiang, B. Pant, A. Eisen, M. P. Beddoes, and D. Poole. Multiply sectioned Bayesian networks for neuromuscular diagnosis. *Artificial Intelligence in Medicine*, 5:293–314, 1993.