Belief propagation in a bucket-tree
Handouts, 275B Fall-2000

Rina Dechter

November 1, 2000

1 From bucket-elimination to tree-propagation

The bucket-elimination algorithm, elim-bel, for belief updating can be viewed as one phase message propagation from leaves to root along a bucket-tree. The algorithm computes the belief of the first node in the ordering given all the evidence. Often, it is desirable to get the belief of every variable in the network. A brute-force approach will require running elim-bel n times, each time with a different variable initiating the order. We will show next that this is unnecessary; by viewing bucket-elimination as function computation (by variable elimination) from leaves to rrot along a rooted *bucket-tree*, a second pass along the tree, from root to leaves will be equivalent to running the algorithm for each node as a root. This yields a two-phase variable elimination algorithm up and down the bucket-tree, which can also be viewed as two-phase message passing (or propagation) along the tree (i.e., λ and π messages) in a way that resembles closely Pearl's poly-tree propagation.

Let (G, P) be a Bayesian network, d an ordering of its variables $X_1, ..., X_n$ and $B_1, ..., B_n$ the final buckets created when running bucket-elimination along d. We will denote by B_i both the functional content of bucket i as well as the set of variables appearing in bucket i at the time of processing.

A *bucket-tree* of an ordered Bayesian network has buckets as its nodes and each bucket B_i is connected to B_j if the function created at B_i is placed in B_j. It is easy to see that this definition is equivalent to the following graphical definition.

Definition 1 (bucket tree) *Let G_d be the induced moral graph along d of a Bayesian network graph G. Each variable X and its earlier neighbors in the induced-graph is a bucket-cluster, denoted B_X. Each node B_X points to B_Y (or, B_Y is the parent of B_X) if Y is the latest earlier neighbor of X in G_d (namely it is the closest earlier neighbor to X). If B_Y is the parent of B_X in the bucket-tree, then the separator of X and Y, denoted S_{XY} is the set of variables appearing in $B_X \cap B_Y$.*

Therefore, in a bucket tree, every node B_X has one parent node B_Y and several child nodes $B_{Z_1}, ...B_{Z_t}$. Algorithm *bucket-tree-elimination (BTE)* for computing the belief of every node is described in Figure 1. In the top down phase, each bucket receives λ messages from its children and sends a λ message to it parent. The correctness of the algorithm follows from the fact that

Theorem 1 *The bucket-tree of a Bayesian network G is an i-map of G.*

Definition 2 *A scope of a function is the set of variables in its arguments.*

Theorem 2 *Algorithm BTE is sound. When terminates, each B_X has λ_j^X received from each child Z_j in the tree, its own original P functions and the π_Y^X sent from its parent Y. Then,*

$$P(B_X, e) = \alpha \prod_k P_k \cdot \prod_j \lambda_j \cdot \pi_Y^X$$

Proof: follows from the bucket-tree i-mapness (to complete).

Example 1 *The network in Figure 2a can express causal relationship between 'Season' (A), 'The configuration of an automatic sprinkler system,' (B), 'The amount of rain expected' (C), 'The wetness of the pavement' (F) whether or not the pavement is slippery (G) and 'the amount of manual watering necessary' (D). Figure 3 shows the initial buckets, the λ and π messages created after two passes, using complete bucket ordering. Figure 4 shows the execution relative to a partial order that correspond to the bucket-tree. In that picture the π and λ are viewed as messages placed on the outgoing arcs.*

The π functions computed in the up phase are:
$\pi_A^B(a) = P(a)$
$\pi_B^C(c, a) = P(b|a)\lambda_D(a, b)\pi_A^B(a)$
$\pi_B^D(a, b) = P(b|a)\lambda_C(a, b)\pi_A^B(a)$
$\pi_C^F(c, b) = \sum_a P(c|a)\pi_B^C(a, b)$
$\pi_F^G(f) = \sum_{b,c} P(f|b, c)\pi_C^F(c, b)$

1.1 Bucket-tree propagation, an asynchronous version

The BTE algorithm can be described in an asynchronous manner when viewing the bucket-tree as an undirected tree and passing only one type of messages. Each bucket receives λ messages from each of its neighbors and each sends a λ message to every neighbors. The algorithm executed by node B_X is described next. We distinguish between the original P functions placed in bucket B_i and the messages that its received from its neighbors. The algorithm is described in Figure 5.

Clearly, algorithm BTP is guaranteed to converge, and when converged each B_X and its incoming messages will have the same content as bucket B_X in BTE.

Theorem 3 *The time and space complexity of BTE and BTP is time and space exponential in the induced-width of the corresponding ordered induced-graph.*

Algorithm bucket-tree elimination (BTE)
Input: A bucket-tree $B_1, ...B_n$ for Bayesian network (G, P) along the ordering d, where each function is placed in the latest bucket that mention a variable in its scope.
output: Buckets containing the original functions and all the π and λ functions received from neighbors in the bucket-tree. Marginal probabilities over the bucket's variables.

1. Top-down phase: (bucket-elimination)
For $i = n$ to 1, process bucket B_i:
Let B_X be the i^{th} processed bucket. Let $\lambda_1, ...\lambda_j$ be all the functions in B_X at the time of processing including the original P functions. The function λ_X^Y sent from X to its parent Y, is computed by taking the product of all the functions in its bucket, and summing over X. The scope of this function is S_{XY}, the separator between X and its parent Y. Namely:

$$\lambda_X^Y(S_{XY}) = \sum_X \prod_j \lambda_j$$

2. Bottom-to-top (variable-elimination from root to leaves):
For $i = 1$ to n, process up bucket B_i:
Let X be the current processed bucket. Let $\lambda_1, ...\lambda_j$ be all the functions in B_X at the time of processing including the original P functions. X takes the π message received from its parent Y, π_Y^X, and computes a message $\pi_X^{Z_j}$ for each child bucket Z_j by taking the product of all the functions in B_X excluding $\lambda_j = \lambda_{Z_j}^X$, (the lambda function received from Z_j), and summing over the variables in B_X which are not in the separator S_{XZ_j}. Namely, let $U_X = B_X - S_{XZ_j}$.

$$\pi_X^{Z_j}(S_{XZ_j}) = \sum_{U_X} \pi_Y^X \cdot \prod_{\{i|i \neq j\}} \lambda_i$$

3. Deriving beliefs
The joint probabilities $P(X, S_X, E = e)$ in B_X is computed by taking the product of all the functions in B_X (the original Ps, the λ functions and π function): Namely, given the functions $f_1, ..., f_t$ in B_X at termination,

$$P(B_X) = \alpha \prod_i f_i$$

and the belief of X is computed by

$$Bel(x) = \alpha \sum_{B_X - \{X\}} \prod_j f_j$$

Figure 1: Algorithm Bucket-tree Elimination

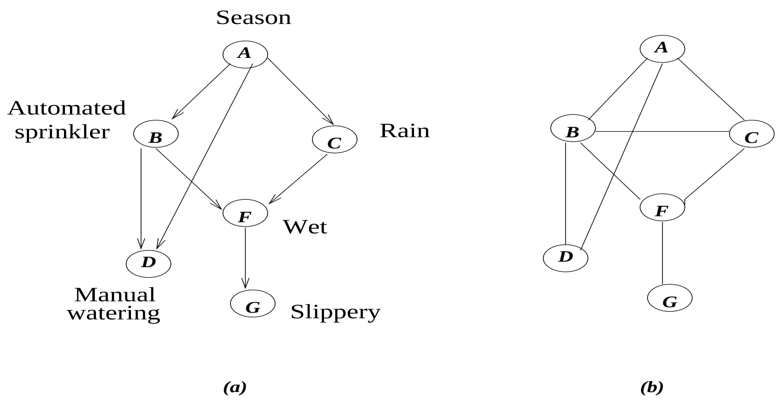

Figure 2: belief network $P(g,f,d,c,b,a) = P(g|f)P(f|c,b)P(d|b,a)P(b|a)P(c|a)P(a)$

2 From buckets to super-buckets to join-trees

The BTE and BTP algorithms are special cases of a wide class of algorithms all based on an underlying cluster-tree. We saw that in a chordal graph the maximal cliques form a tree which obeys the *running intersection property (r.i.p)*. This property guarantees that the clique-tree is an i-map of the original Bayes network.

In fact, given a chordal graph embedding of the original Bayesian networks' dag, (which can be obtained by generating the induced graph along an ordering, an operation known also as triangulation), there are many cluster-trees that have the desired running intersection properties. The join-tree is the most popular tree used in inference. The bucket-tree is another candidate. We next define *legal* cluster-trees as an extension that captures all viable cluster-trees that support tree propagation. A *cluster tree* is a set of subsets of variables connected by a tree structure.

Definition 3 *A legal tree of a Bayesian network is any cluster tree that satisfies the following two properties:*
1. Every family (CPT) has at least one cluster that contains its scope.
2. The cluster-tree obeys the running intersection property.

Join-trees and bucket-trees are legal trees. One way of structuring legal trees is to generate a bucket-tree from an induced graph, and then create subsequent trees by merging adjacent clusters.

Proposition 4 *If T is a legal cluster tree, then any tree obtained by merging adjacent clusters is legal.*

Proof: Exercise.

We can start from the bucket-tree and merge adjacent buckets, yielding *super-buckets*. The maximal cliques are special types of super-buckets. Clearly, the bucket-tree contains all the maximal cliques. If each maximal clique absorbs adjacent subsumed cliques we get the clique-tree or the *join-tree* (see Figure 6).

$$
\begin{aligned}
bucket_G = &\quad P(g|f), g = 1\\
bucket_F = &\quad P(f|b,c)\\
bucket_D = &\quad P(d|b,a)\\
bucket_C = &\quad P(c|a)\\
bucket_B = &\quad P(b|a)\\
bucket_A = &\quad P(a)
\end{aligned}
$$

Buckets after top-down message passing:

$$
\begin{aligned}
bucket_G = &\quad P(g|f), g = 1\\
bucket_F = &\quad P(f|b,c) || \ \lambda_G(f)\\
bucket_D = &\quad P(d|b,a)\\
bucket_C = &\quad P(c|a) || \ \ \lambda_F(b,c)\\
bucket_B = &\quad P(b|a) \ || \ \ \lambda_D(a,b)\\
bucket_A = &\quad P(a) \ || \ \lambda_B(a)
\end{aligned}
$$

Buckets after bottom-up message passing:

$$
\begin{aligned}
bucket_G = &\quad P(g|f), g = 1, \ || \ \pi_F^G(f)\\
bucket_F = &\quad P(f|b,c) || \ \lambda_G(f), \ \pi_C^F(b,c)\\
bucket_D = &\quad P(d|b,a) \ || \ \pi_B^D(a,b)\\
bucket_C = &\quad P(c|a) || \ \ \lambda_F(b,c), \pi_B^C(a,b)\\
bucket_B = &\quad P(b|a) \ || \ \ \lambda_D(a,b) \ , \pi_A^B(a)\\
bucket_A = &\quad P(a) \ || \ \lambda_B(a)
\end{aligned}
$$

Figure 3: Prpagation up an down the buckets using $d_1 = A, B, C, F, D, G$

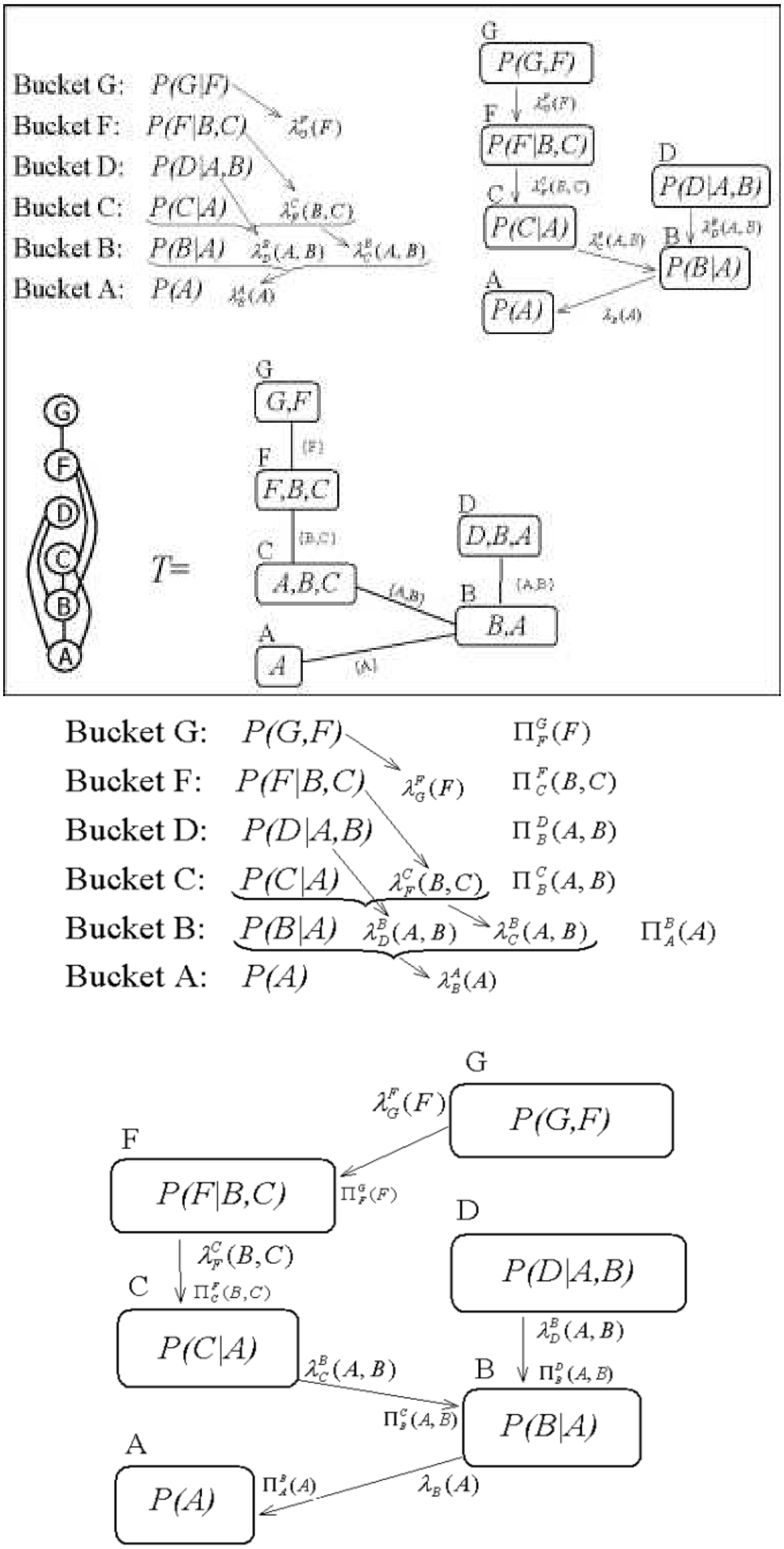

Figure 4: propagation of πs and λs along the bucket-tree

Bucket-tree Propagation (BTP)
Input: For each node X, its bucket B_X and its neighboring buckets.
Output: As in BTE.
Initialize: make all initial messages constant 1.
For every bucket B_X do:
Let $\{P_i\}, i = 1, ...j$ be the original functions in B_X, let $Y_1, ...Y_k$ its neighbors, Let $\lambda_j = \lambda_{Y_j}^X$ be the message sent to X from its neighbor Y_i.
The message X sends to a neighbor Y_j is:

$$\lambda_X^{Y_j}(S_{XY_j}) = \sum_{B_X - S_{XY_j}} (\prod_i P_i) \cdot (\prod_{i \neq j} \lambda_i)$$

Figure 5: The Bucket-tree propagation (BTP) for X

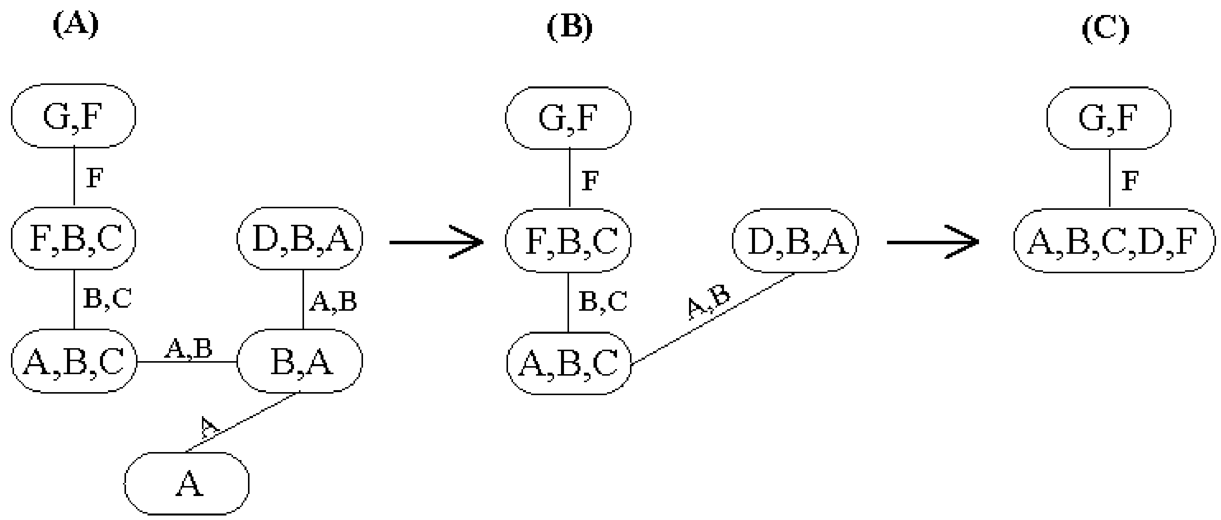

Figure 6: From a bucket-tree to join-tree to a super-bucket-tree

Algorithm join-tree clustering (JTC)
Input: we assume a legal tree T whose nodes are clusters of variables $C_1, ...C_t$ and each node has a collection of neighbors.
Output: Each cluster will have all input messages received from all its neighbors.
Initialize: Put each original function in *any* cluster that contains its scope.
The algorithm for cluster C_X is:
Let $\{P_i\}, i = 1, ...j$ be the original functions in C_X, let $Y_1, ...Y_k$ be its neighbors, let $\lambda_j = \lambda_{Y_j}^X$ be the message sent to C_X from its neighbor C_{Y_j}.
When C_X receives all messages from its neighbors except from Y_j, the message C_X sends to C_{Y_j} is:

$$\lambda_{C_X}^{C_{Y_j}}(S_{C_X C_{Y_j}}) = \sum_{C_X - S_{C_X C_{Y_j}}} (\prod_i P_i) \cdot (\prod_{i \neq j} \lambda_i) \tag{1}$$

Figure 7: Algorithm The Join-Tree Clustering (JTC)

Both BTE and BTP can be extended to any legal cluster-tree. When the tree is the join-tree, the algorithm is called join-tree clustering. Algorithm join-tree clustering (JTC) (also, called super-bucket propagation (SBP)) is presented in Figure 7. We could clearly define the algorithm for synchronous execution from leaves to the root and back.

Theorem 5 *Algorithm join-tree clustering is sound. When it converges, each cluster's functions product provide the marginal probabilities of the cluster's variables joint with the evidence.*

Theorem 6 *The time complexity of JTC is exponential in the largest clique size. The space complexity of JTC is exponential in the separator sizes.*

proof: Clearly the time complexity is exponential in the cluster size since we consider all the assignments to all variables in each cluster, and there are $k^{|C|}$ tuples, when $|C|$ is the size of the cluster and k is the domain size for each variable. The space complexity however can be restricted to the output of the recorded function (the messages). For each tuple of the recorded function over a seperator we accumulate the sum of probabilities over all the rest of the variables, and each such probability (of a tuple) can be computed in linear time and space. This computation can be done by traversing the search space of all possible assignments in a depth-first manner.

In the bucket-tree the separator sizes where equal to the cluster sizes (minus 1) and therefore the time complexity and space complexity are the same. In general however, for the super-bucket-tree or, in particular, for the join-tree, the separators sizes may be far less than the maximal cliques sizes. Furthermore, it is sometimes worthwhile to combine two adjacent cliques having a *wide separator* in order to save space (see time-space paper).

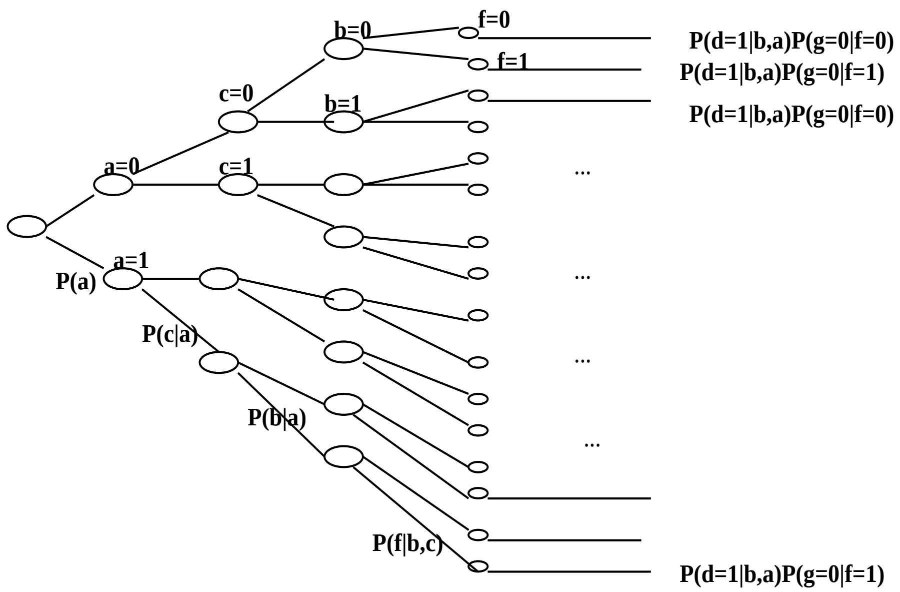

Figure 8: probability tree

3 Combining Elimination and Conditioning; the cycle-cutset method

A serious drawback of elimination and clustering algorithms is that they require considerable memory for recording the intermediate functions. Conditioning search, on the other hand, requires only linear space. By combining conditioning and elimination, we may be able to reduce the amount of memory needed while still having performance guarantee.

Full conditioning for probabilistic networks is search, namely, traversing the tree of partial value assignments and accumulating the appropriate sums of probabilities. (It can be viewed as an algorithm for processing the algebraic expressions from left to right, rather than from right to left as was demonstrated for elimination). For example, we can compute the expression for belief updating in the network of Figure 2:

$$Bel(A=a) = \alpha \sum_{c,b,f,d,g} P(g|f)P(f|b,c)P(d|a,b)P(c|a)P(b|a)P(a)$$

$$= \alpha P(a) \sum_c P(c|a) \sum_b P(b|a) \sum_f P(f|b,c) \sum_d P(d|b,a) \sum_g P(g|f), \qquad (2)$$

by traversing the tree in Figure 8, going along the ordering from first variable to last variable. The tree can be traversed in a depth-first manner. The sum can be accumulated for each value of variable A.

Let X be a subset of variables and $V=v$ be a value assignment to V. $f(X)|_v$ denotes the function f where the arguments in $X \cap V$ are assigned the corresponding values in v.

Let C be a subset of conditioned variables, $C \subseteq X$, and $V = X - C$. We denote by v an

Algorithm elim-cond-bel
Input: A belief network $BN = \{P_1, ..., P_n\}$; an ordering of the variables, d; a subset C of conditioned variables; observations e.
Output: $Bel(A)$.
Initialize: $\lambda = 0$.

1. For every assignment $C = c$, do
   - $\lambda_1 \leftarrow$ The output of elim-bel with $c \cup e$ as observations.
   - $\lambda \leftarrow \lambda + \lambda_1$. (update the sum).
2. **Return** λ.

Figure 9: Algorithm *elim-cond-bel*

assignment to V and by c an assignment to C. Obviously,

$$\sum_x P(x, e) = \sum_c \sum_v P(c, v, e) = \sum_{c,v} \Pi_i P(x_i | x_{pa_i})|_{(c,v,e)}$$

Therefore, for every partial tuple c, we can compute $\sum_v P(v, c, e)$ using variable elimination, while treating the conditioned variables as observed variables. This basic computation will be enumerated for all value combinations of the conditioned variables, and the sum will be accumulated. This straightforward algorithm is presented in Figure 9.

Given a particular value assignment c, the time and space complexity of computing the probability over the rest of the variables is bounded exponentially by the induced width $w_d^*(e \cup c)$ of the ordered moral graph along d adjusted for both observed and conditioned nodes. Namely, the induced graph is generated without connecting earlier neighbors of both evidence and conditioned variables, and the adjusted induced-width is the width of the resulting ordered graph.

Theorem 7 *Given a set of conditioning variables, C, the space complexity of algorithm elim-cond-bel is $O(n \cdot exp(w_d^*(c \cup e))$, while its time complexity is $O(n \cdot exp(w_d^*(e \cup c) + |C|))$, where the induced width $w^*(d, c \cup e)$, is computed on the ordered moral graph that was adjusted relative to e and c.* $\Box$

When the variables in $e \cup c$ constitute a cycle-cutset of the graph, the graph can be ordered so that its adjusted induced width equals 1 and elim-cond-bel reduces to the known loop-cutset algorithm (see Pearl, chapter 4).

Definition 4 *Given an undirected graph, G a cycle-cutset is a subset of the nodes that breaks all its cycles. Namely, when removed, the graph has no cycles. A loop-cutset of a directed graph is a subset of nodes whose removal makes the graph a poly-tree.*

Corollary 1 *A Bayesian network having a loop-cutset of size r can be processed in time $O(n \cdot F \cdot exp(r))$, when n is the number of variables F is the size of the maximal consitional probability table in the input.*

In general Theorem 7 calls for a secondary optimization task on graphs:

Definition 5 (secondary-optimization task) *Given a graph $G = (V, E)$ and a constant r, find a smallest subset of nodes C_r, such that $G\prime = (V - C_r, E\prime)$, where $E\prime$ includes all the edgs in E that are not incident to nodes in C_r, has induced-width less or equal r.*

Clearly, the minimal cycle-cutset corresponds to the case where $r = 1$. The loop-cutset corresponds to the case when conditioning creates a poly-tree. The general task is clearly NP-complete.

Clearly, algorithm elim-cond-bel can be implemented more effectively if we take advantage of shared partial assignments to the conditioned variables. There are a variety of possible hybrids between conditioning and elimination that can refine this basic procedure. One method imposes an upper bound on the arity of functions recorded and decides dynamically, during processing, whether to process a bucket by elimination or by conditioning Another method which uses the super-bucket approach collects a set of consecutive buckets into one super-bucket that it processes by conditioning, thus avoiding recording some intermediate results