Node Aggregation for Distributed Inference in Bayesian Networks

Kuo-Chu Chang and Robert Fung
Advanced Decision Systmes
1500 Plymouth Street
Mountain View, California 94043-1230

Abstract

This study describes a general framework and several algorithms for reducing Bayesian networks with loops (i.e., undirected cycles) into equivalent networks which are singly connected. The purpose of this conversion is to take advantage of a distributed inference algorithm |6|. The framework and algorithms center around one basic operation, node aggregation. In this operation, a cluster of nodes in a network is replaced with a single node without changing the underlying joint distribution of the network. The framework for us ing this operation includes a node aggregation theorem which describes whether a cluster of nodes can be combined, and a complexity analysis which estimates the computational require ments for the resulting networks. The algorithms described include a heuristic search algorithm which finds the set of node aggregations that makes a network singly connected and allows inference to execute in minimum time, and a "graph-directed" algorithm which is guaranteed to find a feasible but not necessary optimal solution and with less computation than the search algorithm.

1 Introduction

This study describes a general framework and several algorithms which use that framework for converting Bayesian networks with loops (i.e., undirected cycles) into equivalent singly-connected networks. The purpose of this conversion is to take advantage of a distributed inference algorithm [6]. The framework and algorithms center around one basic operation, node aggregation. In this operation, a cluster of nodes in a network is replaced with a single node without changing the underlying joint distribution of the network.

Like its predecessor technology, decision/risk tree technology, Bayesian Networks (a.k.a. influence diagrams) [3], is a technology for representing and making inferences about beliefs and decisions. A probabilistic Bayesian Network is a directed, acyclic graph (DAG) in which the nodes represent random variables, and the arcs between the nodes represent possible probabilistic dependence between the variables. A network as a whole represents the joint probability distribution between the random variables. The representation has proved to be an improvement over the older tree technologies for several reasons including increased functionality, compactness, and intuitiveness to users.

While a fast distributed inference algorithm exists for singly-connected networks, it has been proved [l] that no algorithm can be efficient on all Bayesian networks with loops. However, there appears to be much room for expanding the set of graph topologies which can be addressed in a computationally efficient manner. Other than the node aggregation approach presented in this study, several other approaches have been proposed to address the inference problem for arbitrarily-connected networks. These approaches include: conditioning |7|, cliques [4], using the influence diagram operations such as link reversal and node removal [8], and stochastic sim ulation [7].

In this paper, we have chosen the node aggregation method to handle the inference problem in an arbitrary network. For all node aggregation methods, the first and defining step is to reduce the graph using node aggregation into a singly connected graph. This step needs only occur once. In the second step, the distributed infer ence algorithm is applied to the reduced graph to calculate the posterior distributions of each node. Since the aggregated nodes in the reduced graph may consist of more than one original node, the third step calculates the posteriors of the original nodes by marginalizing the posterior probabilities of the aggregated nodes.

The paper is organized as follows. Section 2 describes the definitions and theorems which make up the framework. Of principal interest is a node combinability theorem which determines if a set of nodes can be aggregated. The effects of a node aggregation on a graph are then described. A computational complexity measure is also presented in Section 2. Given a graph's topology and the state space size for each node in the graph, this measure calculates the approximate computation time required for each update in inference. In Section 3, an A* search algorithm to find the optimal node aggregation partition is developed based on the performance criterion obtained in Section 2. This algorithm utilizes pruning techniques that can substantially reduce the search space and the optimal solution is guaranteed to be retained. A much

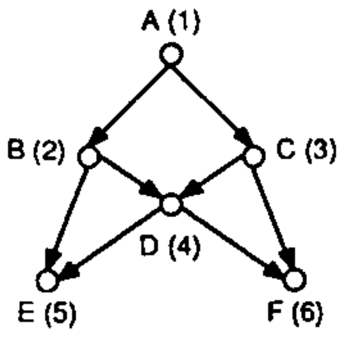

Figure 1: An Example Network

simpler heuristic approach (graph-directed decomposition) for finding a suitable partition is also presented in Section 3. Finally, some conclusions and directions for future research are discussed in Section 4.

2 Framework

In this section we define the basic definitions and present theorems which make up the framework. As directed acyclic graphs, there are two major topological classes of Bayesian Networks, graphs which are singly-connected and graphs which contain undirected cycles (i.e., loops). For graphs with loops, the distributed inference algorithm |6J does not apply since the loops cause the local propagation schemes to circulate messages indefinitely around these loops. In order to keep the advantages of the distributed inference scheme, several methods have been proposed to deal with this problem. These include *aggregation, conditioning* , and *stochastic simulation.* In this study, we focus on the aggregation method.

In order to aggregate nodes in a loop, first we need to be able to identify all nodes involved in loops for a given acyclic graph. This can be done by using well-known graph-theoretic algorithms. There may be more than one loop in a graph. In that case, we need to identify all the loops and group them into independent loop set, where two loops are independent if they don't share any node. For each independent loop set, the nodes involved in the loops are partitioned into clusters which are then aggregated into *macro* nodes. Note that not all sets of nodes can be aggregated. In particular a set of nodes cannot be aggregated if it creates a cycle in the network. In this section, we will describe the basic node aggregation theorem and illustrate the results of aggregation with several simple examples.

2.1 Node Aggregation Theorem

First a *path* is defined for a directed graph in the usual way. For example, in Figure 1, *(ACD)* is a path between *A* and D, but *(ACFD)* is not a path. Second, a pair of nodes is called *combinable* if there is no path between the two nodes which contains a third node. Similarly, a group of nodes is called *combinable* if for every pair of nodes in the group there exists no path between such pair which contains a node outside the group.

With the above definitions, we have the following lemma.

Lemma 1 : A pair of nodes in an acyclic graph can be combined into a *macro* node so that the resulting graph remains acyclic if and only if the node pair is combinable.

Proof : The proof is given in Appendix A.

With the above theorem, we also have the following node combiliability theorem.

Theorem 1 : A group of nodes in an acyclic graph can be combined into a *macro* node so that the resulting graph remains acyclic if and only if the group is combinable.

Proof : The proof of this theorem is similar to the proof of the above lemma.

2.2 The Results of Combining

When a macro node is created, the predecessors and successors of the node as well as its conditional probability requires definition. A macro node's predecessors are the union of its component nodes' predecessors, and its successors are the union of its component nodes' successors. For the conditional probability distribution of macro nodes, we have the following lemmas.

Lemma 2 : The conditional probability of the macro node given its predecessors is equal to the product of all component node's conditional probabilities. For the example in Figure 1, if *(B,C,D)* are combined into a macro node *M,* then the conditional probability of *M* given the predecessor *A* is equal to:

$$P(M|A) = P(B,C,D|A) = P(B|A)P(C|A)P(D|B,C). \quad (1)$$

Lemma 3 : The conditional probability of a macro node's successor is equal to the conditional probability of the successor given all the component nodes in the macro node, except for those component nodes which are not linked directly to the successor, in which case, they are irrelevant and will not affect the conditional probability other than increasing the dimension of the conditional probability matrix. For the same example as above, where *(B,C,D)* are combined into a macro node *M,* then the conditional probability of *E* given *M* is equal to:

$$P(E|M) = P(E|B,C,D) \quad (2)$$

Since *C* is not linked directly to *E,* the conditional probability only depends on *B* and D, namely, *P(E\B,* D), which is already available. Therefore, one only needs to fill up the matrix with appropriate entries, i.e., $P(E_i|B_j,C_k,D_l) = P(E_i|B_j,D_l)$

Proof : The proofs of Lemmas 2 and 3 are straightforward and will not be carried out here.

According to the above theorem and lemmas, for a graph which is not singly connected, one may *partition* each independent loop set in the graph into several *clusters* and aggregate the nodes in each cluster into a macro node so that the graph can be reduced to a singly-connected graph and the distributed inference algorithm [6*]* can be applied in processing. For the example given in Figure 1, there are several ways of aggregating nodes and reducing the original graph into a singly-connected graph. Each one of these is called a *feasible partition.*

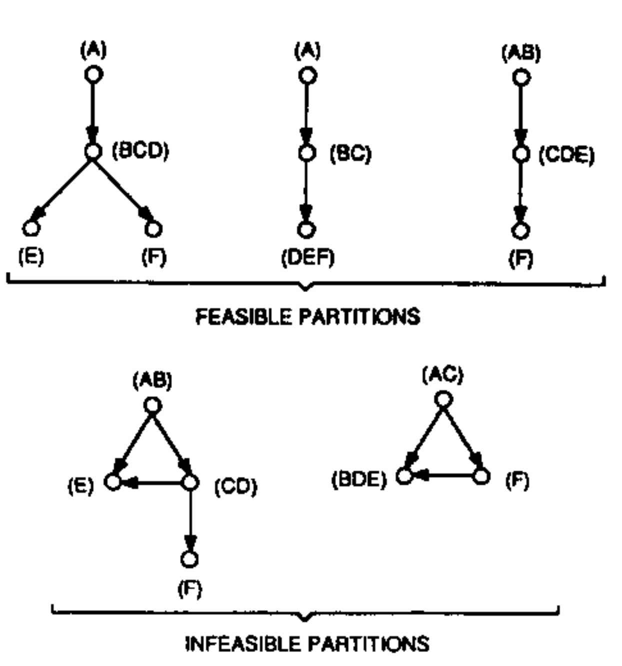

Figure 2: Feasible and Infeasible Partitions

Some of the examples with feasible partition or infeasible partition (with loops) are given in Figure 2. Obviously, certain partitions are better than others in computation requirements in the resulting networks. In order to distinguish them, a criterion of performance needs to be defined. In the next subsection, we develop the computational requirements for each macro node and define it as the performance measure to be considered.

2.3 Computational Complexity

In order to determine the computational efficiency for different network structures, we first need to be able to quantify the computational requirements for various configurations. Due the characteristics of the distributed algorithm [6], the computational requirements for each node depend not only on the size of state space of the node itself, but also on the number of its predecessors and successors as well as their state space sizes. For each node in a singly connected network, there are four modules in the algorithm, two of them for upward propagation and two of them for downward propagation. In a distributed processing environment, each processing node may propagate the data upward and downward simultaneously, however, due to the interaction between the four processing modules, calculating the exact processing time needed for each node is not trivial. We may approximate the computation requirements by considering the number of multiplications that have to be performed in all four modules for each update.

As carried out in Appendix B, for a node X with M predecessors and TV successors, the total number of multiplications of all modules for each update is,

$$n(1+M+N)+Mn\prod_{i=1}^{M}n_i^P+$$
$$\sum_{j=1}^{M}\left\{(M-1)n\prod_{i=1,\dots,M,\ except j}n_i^P+nn_j^P\right\}$$

where n and n^{pi} are the state space sizes of node X and its i:-th predecessor Xpi respectively. In a tree structure, each node can have at most one predecessor, i.e., M — 1, therefore the number of multiplications reduced to 2n + nN | 2n^2, where npi is assumed to be equal to n. This coincides with the calculations given in [6j.

In the above calculations, it was assumed that the enlarged conditional probability matrices as obtained in equations (1) and (2) are stored in their corresponding macro node at the time the graph was reduced and "initialized". Therefore, in normal processing when new evidence is added, only the computational requirements involved in standard propagation as described above need be considered. Alternatively the conditional probabilities can also be calculated from the component nodes whenever needed instead of storing the matrix at the macro nodes. This alternative is preferable when the matrix is very large and a large amount of memory is required. Nevertheless, more computational resources are necessary in this approach since the conditional probabilities need to be re-calculated each time they are needed. This kind of trade-off should be considered in choosing aggregation partition for various network configurations. In the next section, several heuristic algorithms are proposed for finding suitable partitions.

3 Aggregation Algorithms

All aggregation algorithms face the basic fact that the number of possible partitions of a graph with n nodes is $B(n)$, where $B(n)$ is given by the recursive formula:

$$B(n+1)=\sum_{k=0}^{n}\binom{n}{k}B(k) \tag{4}$$

This sequence of numbers is called the Bell sequence [2], and grows exponentially. The first few elements of this series are: 1,1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975. Thus for a graph with 10 nodes there are about 116 thousands possible partitions.

However, many of these partitions (i.e., reduced graphs) are not feasible since they have undirected or directed cycles. Although it is possible for certain specialized structures to find the optimal partition through direct inspection of the graph, we have as yet found no such general algorithm for the general case. Instead the algorithms proposed in this paper, either by searching through the space of partitions or by inspection of the graph, find a feasible but not necessary optimal partition. In this paper, we have identified several methods to use the special (i.e., directed, acyclic) structure of Bayesian Networks in order to cut down the number of partitions examined in the search for the optimal partition. In this section, we will use the example graph shown in Figure J to illustrate each concept presented.

3.1 Heuristic Search

In this subsection, an aggregation algorithm is proposed which formulates the aggregation problem as a search problem and guarantees the identification of the optimal partition. In this algorithm, the nodes in each dependent loop set are first ordered, and then partial partitions are generated recursively, with pruning of infeasible partitions occurring at each level.

It is well known that acyclic graphs can be numbered such that every node has a larger number than all of its predecessors, and a smaller number than any of its successors. The numbers shown in Figure 1 are obtained by this scheme. It should be noted that some of the decisions are arbitrary, (e.g., B could switch order with

Given the node ordering scheme above, the full set oi partitions for a dependent loop set can be identified by the following recursive algorithm:

```
1. Initialize the old-partition-list to 0.
2. For node i from 1 to n
      2a. Start a new-partition-list
      2b. For each partition in old-partition-list
            2bl. For each cluster in partition
            2b2. Make a new partition by adding node i
                 to that cluster
            2b3. Add that new partition to new-partition-
                 list
            End For (2b1)
      End For (2b)
3. Set the old-partition-list to the new-partition-list
End For (2)
```

An illustration of this expansion algorithm for the example graph is shown in Figure 3 up to the fourth node (i.e.D).

Without pruning, this expansion algorithm for an independent loo]) set with *n* nodes will produce *B(n)* number of complete partitions. Two pruning operations have been identified to manage the search space without removing any feasible partitions. The first pruning operation removes any partition which has a node cluster which is not combinable. The partial partition (AD,), BO) (see Figure 3) is one such partition since the nodes AD are not combinable. The second pruning operation removes any partition which contains a loop. The partial partition (A,B,C,D) is one such partition since the nodes *ABCD* form a loop. The claim that these pruning operations do not remove any feasible partitions from consideration is proven below.

Theorem 2 : With pruning of *infeasible* partial partitions which contain at least one loop) or at least one node cluster which is not combinable, all feasible *complete* partitions are still reachable.

Proof : Because of the ordering scheme, if a node cluster in a partial partition is not combinable, the addition of any new node cannot make that node cluster combinable. Also if a partial partition contains a loop, the addition of any new node will not remove the loop from that partition. Thus any successor partition of any infeasible partial partition will be infeasible. Thus pruning an infeasible partial partition does not prune any feasible complete partition.

In order to use heuristic search techniques, an evaluation function must be defined. In particular for the *A** algorithm, the cost functions g(p) and *h(p)* must be defined where *p* is a partial partition. The function g*(p)* is an estimate of the cost from the start node to the the partial partition. This can be calculated using the computational complexity results derived in Sec 2.3. The function h*(p)* is an estimate of the additional cost of getting to the final complete partition and must be less than or equal to the "actual" cost of reaching the final partition. For *h(p)* we compute the additional cost of completing the partition by assuming the rest of the nodes are added singly since this is the minimum cost solution. The cost functions *g(p)* and *h(p)* are combined into the evaluation function f(p) where:

$$f(p) - g(p) + h(p) \tag{5}$$

While we have chosen to discuss the *A** algorithm [5] here, other heuristic search techniques could be applied as well.

An additional technique for reducing search is to devise an algorithm for initializing *f(p)* before any search starts. Such algorithms find a "good" feasible complete solution directly. Such an algorithm is discussed in the next subsection.

In summary, the heuristic search algorithm can be described as follows:

```
1. Order the nodes in the dependent loop set.
2. Initialize the old-partition-list to be 0.
3. Initialize the minimum cost by an initialization algo-
   rithm (e.g., Sec. 3.2)
4. Choose the partial partition $p_i$ with minimum $/(p_i,)$
   to expand.
    4a. Make new partitions by adding the $i + 1$ node
        to the partial partition $p_i$
    4b. Prune all partitions containing a "non-combin
        able'[1] node cluster.
    4c. Prune all partitions containing a loop.
    4d. Evaluate $j\{p_{i+1})$ for each new partial partition.
    4e. if there is a partial partition to expand, go to 4,
        else done.
```

3.2 Graph-Directed Algorithm

Another heuristic method developed based on the node aggregation theorem is the so called "graph-directed algorithm". This algorithm is "graph-directed" since the aggregation process follows the direction of the graph (i.e., from a graph's roots to its leaves). This algorithm consists of the following steps :

1. For each independent loop set, find the root nodes.
2. Identify all root nodes[1] successors which are within the loop set.
3. Group these successors into clusters such that any two nodes of different clusters do not share any predecessor.
4. If the nodes in a cluster are combinable, combine

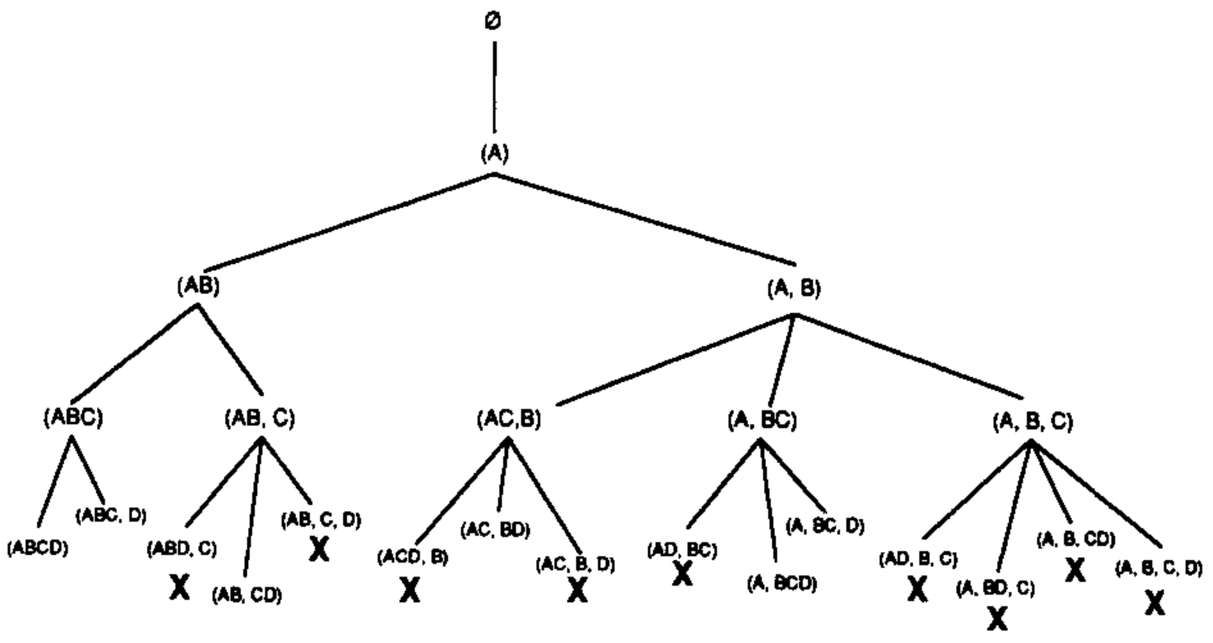

Figure 3: An Expansion Example

them into a macro node.

5. If the nodes in a cluster are not combinable, collect these nodes together with their direct successors within the loop set and go to step 3.
6. When all nodes involved in the loop set have been considered, done.

This algorithm reduces a graph with loops into a singly-connected one. It is not optimal but is simple and requires small amounts of computational resources. It can serve as a mechanism for initializing the search algorithm described in the previous subsection or can be used as a stand-alone algorithm for finding a partition for node aggregation. To illustrate the algorithm, Figure 4 shows several example graphs together with their resulting partitions obtained based on this algorithm.

4 Conclusions and Discussion

In this paper, we have presented a general framework and several algorithms for converting Baycsian networks with loops into equivalent singly-connected networks. The purpose of such conversions is to take advantage of the distributed algorithm [6] for inferencing in a singly-connected network. The framework and algorithms center around one basic operation, node aggregation. In this operation, a cluster of nodes in a network is replaced with a single node without changing the underlying joint distribution of the network. The framework consists of a node combinability theorem which determines if a set of nodes can be aggregated, a description of the results of node aggregation and a computational complexity measure associated with the aggregation.

A search algorithm to find an optimal node aggregation partition as well as a simpler heuristic approach for finding a suitable partition are then presented. While the algorithms described in this paper may not in the end prove to be the most efficient ones, all of the ba-

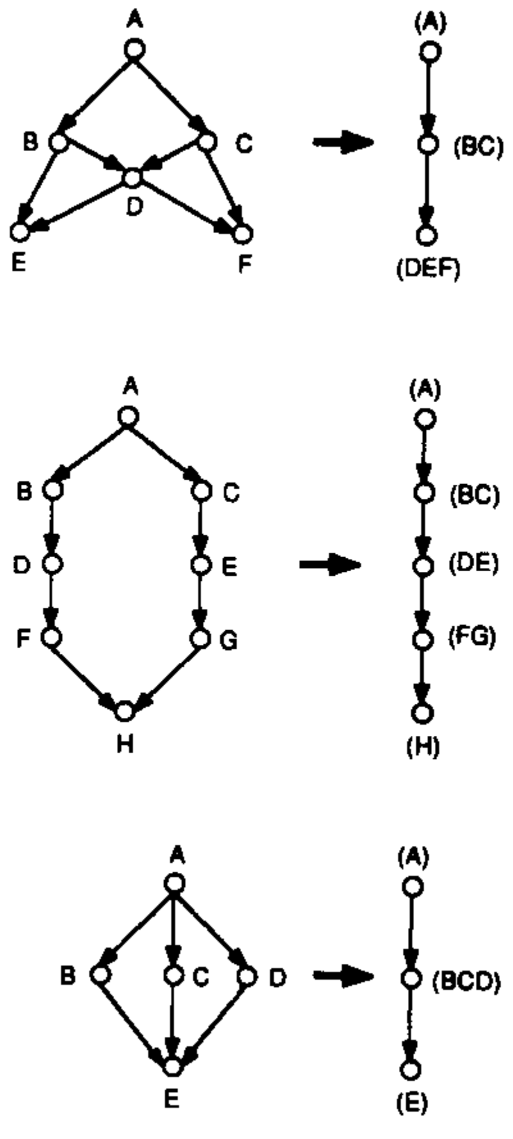

Figure 4: Partition Examples

sic concepts in the framework and those used in the algorithms (e.g., node combinability, computational requirements measure, node ordering, pruning operations) are fundamental and useful to all node aggregation algorithms.

There are at least two primary directions of research to pursue, algorithm development within the node aggregation framework and the comparison and integration of other methods for dealing with graphs with loops (e.g., conditioning, stochastic simulation).

With respect to algorithm development, the most promising avenue of research seems to be algorithms like the "graph-directed" algorithm which would utilize graph features such as connectivity and state space size, to directly identify a near optimal if not the optimal so lution. Search approaches seem less promising since even heavy pruning of infeasible and high cost alternatives can still be computationally costly for some graphs. Hybrid approaches which combine search and algorithms which generate an initial feasible solution may also be worth looking into.

For the integration of algorithms, it seems clear that with each method which has been suggested, there are certain graph topologies where that method provides the best solution. However, the same method may perform very poorly for other topologies. This suggests a direction of research in which the algorithms are compared and integrated based on their individual strength in dealing with various graph topologies.

A Proof of Combinability Theorem

The proof of the lemma follows,

Proof :

(⟶) If a cycle is created in the new graph due to the combination of the node pair, then there must exist at least one path between the two nodes which contains a third node. This violates the combinable node pair definition. Therefore, if a node pair is combinable, then the resulting graph (with the node pair combined into a macro node) remains acyclic.

(⟵) If a node pair is not combinable, then there exist at least one path between the two nodes which contains a third node. Now if the pair is combined into a macro node, that directed path will create a cycle and make the resulting graph cyclic. Therefore, in order to maintain the resulting graph acyclic, the node pair to be combined has to be combinable.

B Computational Complexity

For the four modules in the algorithm [6], the respective number of multiplications needed are,

Module I : Calculate $\lambda_X(X_i^P)$. According to the updating formula, there are two parts, first calculate the transition matrix, then multiply the matrix by the vector $\lambda(X)$. Since wc have M predecessors, the total number of multiplications needed is:

$$\sum_{j=1}^{M}\{(M-1)n \prod_{i=1,..,M;\ except\ j} n_i^P + nn_j^P\} \qquad (6)$$

where n is the state space size of node X, and n_i^P is the state space size of node X_i^P.

Module II : Calculate $\lambda(X)$. This term is a product of $\lambda_{X_i^S}(X)$ from its N successors, X_i^S, the number of multiplications needed is nN.

Module III : Calculate $\pi_X(X_i^P)$. For each predecessor, this term is a division (product) of $BEL(X_i^P)$ and $\lambda_X(X_i^P)$, therefore, the total number of multiplications needed is nM.

Module IV : Calculate $BEL(X)$. There are two parts in the calculation of this term, first calculate the transition vector by multiplying the transition matrix with the $\pi_X(X_i^P)$ vectors and summing over X_i^P, then do the vector multiplication between the resulting vector and $\lambda(X)$. The total number of multiplications needed in this module is:

$$Mn\prod_{i=1}^{M} n_i^P + n \qquad (7)$$

From the above, the total number of multiplications of all modules for each update is obtained as in (3).

References

[l] G. F. Cooper. Probabilistic inference using belief networks in np-hard. *Report KSL-81-21, Medical Computer Science Group, Stanford University,* 1987.

[2] 11. W. Gould. Bell and Catalan numbers: research bibliography of two special number sequences, *(Combinatorial Research Institute,* 1977.

[3] R.A. Howard and .J. E. Matheson. Influence diagrams. In P.A. Howard and J.E. Matheson, editors, The *Principles and Applications of Decision Analysts, vol. II,* Menlo Park: Strategic Decisions Group, 1981.

[4] S. L. Lauritzen and 1). J. Spiegelhalter. Local computations with probabilities on graphical structures and their application in expert systems. *Journal Royal Statistical Society B,* 50, 1988.

[5J P. E. Hart N. J. Nilsson and B. Raphael. A formal basis for the heuristic determination of minimum cost path. *IEEE Trans. Syst. Science and Cybernetics,* 4, 1968.

[G] Judea Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence,* 29, 1986.

[7] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, 1988.

[8] Ross D. Shaehter. Intelligent probabilistic inference. In L.N. Kanal and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence,* Amsterdam: North-Holland, 1986.