Searching Bayes Nets by Exploiting Task Priority

E. Tick
Dept. of Computer Science
University of Oregon

Abstract

Bayes networks are directed acyclic graphs where nodes represent events and edges represent probablistic dependencies among events. Associated with each node are conditional probabilities of the associated event triggering if its ancestor nodes trigger. The total probability mass of a leaf node triggering can be computed from simple probability theory, albeit the number of minterms in the formula is exponential in the number of ancestor nodes of that leaf. It is a well-known result that for certain biased networks, a number of minterms only linear in the number of ancestor nodes contributes about 67% of the total probability mass. The problem of Bayes net search is to generate only these high-mass minterms. A concurrent algorithm for attempting this is introduced, based on converting the net into a concurrent process network. Each parent node sends messages containing partial minterms to child nodes. The novel idea is to prioritize these messages to give higher weight to partial terms that are likely candidates for inclusion in the final high-mass minterms. The KL1 implementation of this algorithm and its performance attributes are discussed.

1 Introduction

Bayes networks are directed acyclic graphs where nodes represent events and edges represent probablistic dependencies among events [4]. Associated with each node are conditional probabilities of this event triggering given ancestor nodes trigger. The total chance of a leaf node triggering can be computed from simple probability theory, albeit the number of minterms in the formula is exponential in the number of ancestor nodes.

To illustrate, Figure 1 (left) shows a simple four-node Bayes net. Node A is the only root, and node D is the sole leaf in this diagram. The diagram states that the probability distribution for node D is conditional on *both* nodes B and C, and that each of these nodes in turn is conditional on node A. Significantly, the diagram encodes the statement

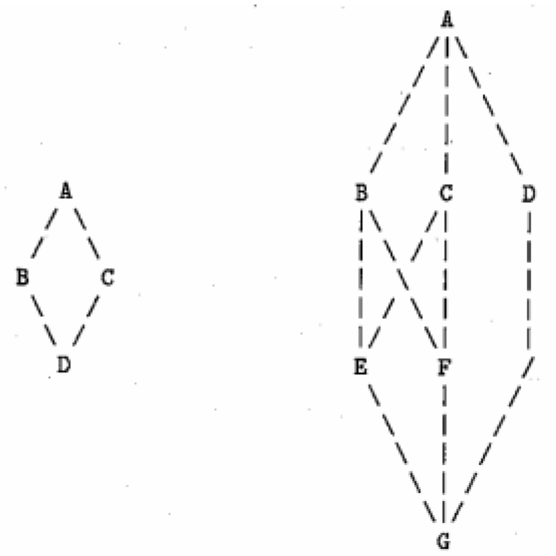

Figure 1: Two Simple Bayes Nets: Downwards Flow

that the distribution for node D is independent of the value taken by A once the values of both B and C are known. This can be formulated as:

$$P(D) = \sum_{A,B,C} P(D \mid BC)P(B \mid A)P(C \mid A)P(A)$$

This formulation can be expanded by expliciting enumerating the truth values of the events represented by nodes $\{A, B, C, D\}$. The expanded form is:

$$\begin{aligned} P(d) \;=\; & P(d \mid bc)P(c \mid a)P(b \mid a)P(a) + P(d \mid \bar{b}c)P(c \mid a)P(\bar{b} \mid a)P(a) + \\ & P(d \mid \bar{b}\bar{c})P(\bar{c} \mid a)P(\bar{b} \mid a)P(a) + P(d \mid \bar{b}\bar{c})P(\bar{c} \mid \bar{a})P(\bar{b} \mid \bar{a})P(\bar{a}) + \\ & P(d \mid b\bar{c})P(\bar{c} \mid \bar{a})P(b \mid \bar{a})P(\bar{a}) + P(d \mid b\bar{c})P(\bar{c} \mid a)P(b \mid a)P(a) + \\ & P(d \mid bc)P(c \mid \bar{a})P(b \mid \bar{a})P(\bar{a}) + P(d \mid \bar{b}c)P(c \mid \bar{a})P(\bar{b} \mid \bar{a})P(\bar{a}) \end{aligned}$$

There are $2^3 = 8$ minterms in this equation, i.e., it is exponential in the three ancestors of the exit port. There is an analogous equation for $P(\bar{d})$ with eight minterms.

It is a known result [2] that for a certain class of biased Bayes nets, 67% of total probability mass of a given node can be computed with a number of minterms linear in the number of its ancestor nodes. Thus for the previous example, if the conditional probabilities showed the proper bias, then it is known that only three minterms are needed to compute within 67% of the exact answer.

The problem is how to find these magic minterms without computing all! There have been sequential algorithms proposed for doing this [2, 5], but to date no concurrent algorithms. The hypothesized advantage of a concurrent algorithm is to exploit multiprocessor parallelism to gain performance. The danger of a concurrent algorithm that exploits *speculative* parallelism is that a large number of minterms is generated in any case. Thus we risk increasing complexity to gain parallel execution.

This paper proposes a new concurrent algorithm to solve this problem. The basis of the algorithm is to generate partial minterms, such as $P(b \mid \bar{a})P(\bar{a})$ above, prioritized by their own values. Generation proceeds in parallel within a process network constructed directly from the Bayes net. The result is fast computation of heavier terms and delayed computation of lighter terms. The algorithm is dynamically self-balancing, i.e., as partial minterms are combined, their priorities change, affecting how they are scheduled within the remainder of the net.

The paper is organized as follows. Section 2 describes the proposed concurrent algorithm and gives an example to illustrate its execution. Section 3 describes an implementation in KL1. The performance of the algorithm is discussed in Section 4. Conclusions and future work are summarized in Section 5.

2 Concurrent Algorithm

For a full literature review of previous sequential algorithms for searching Bayes nets, see Tick and D'Ambrosio [9]. Figure 1 (right) illustrates a more complex example that helps motivate the concurrent algorithm. The probability mass at the exit port G is:

$$P(G) = \sum_{A,B...F} P(G \mid DEF)P(F \mid BC)P(E \mid BC)P(D \mid A)P(C \mid A)P(B \mid A)P(A)$$

Unlike the previous example, this net has reconverging edges, e.g., at nodes E, F and G. This creates a problem in formulating a concurrent algorithm because it is critically important to join together only those partial minterms that have the *same truth value assignment.*

Initially, the Bayes net is converted into a *process network*, i.e., a set of "object-oriented" procedures that actively send and receive messages on their edges. Edges are unidirectional corresponding to message *streams*. For example, nodes B and E communicate over a dedicated stream. This stream is *merged* at E with a stream originating from node C.

Conceptually a node contains the following static information: the set of probabilities conditional on its parent nodes. For example, node E contains $P(E \mid ABC)$ for all truth assignments of variables $\{A, B, C, E\}$, i.e., 16 values. This is formalized as follows.

> *Definition:* A *partial truth assignment* is a subset of all variables in the net, each with an indicated polarity (true or false). For example, a partial truth assignment for the net in Figure 1 is $\{a, b, c, \bar{e}, \bar{f}, \bar{g}\}$. □

Definition: A *key* is a pair of bit vectors (B^{pos}, B^{neg}). $B_i^{pos} = 1$ indicates that variable i is true (positive polarity). $B_i^{neg} = 1$ indicates that variable i is false (negative polarity). $B_i = 0$ does not indicate anything about the truth assignment for variable i. For example, the key (1110000,0000111) corresponds to the truth assignment $\{a, b, c, \bar{e}, \bar{f}, \bar{g}\}$. □

Definition: The function *conv* takes a set of truth assignments and returns a key. The function $conv^{-1}$ takes a key and returns a set of truth assignments. □

Definition: The function $cond(T_{node}, T_{anc})$, where T_{node} is a truth assignment for a node and T_{anc} is a partial truth assignment of the ancestors of that node, returns $P(T_{node} \mid T_{anc})$. □

The information in *cond* may be stored within each node, or combined in a global table. In the implementation, the information is distributed among the nodes. Note that there is no dynamic state required in a node. However, one might consider the *matching queue* associated with each node as holding the dynamic state of that node. The matching queue is defined in Section 3.1.

2.1 Message Definition

A node sends and receives messages consisting of a tuple $< Prob, Key, Priority >$. These are defined as follows:

- *Prob* is a probability between zero and one. Messages sent to *a selected* child of a parent node are assigned *Prob* based on probabilities from a set of incoming messages. Messages sent to all other children are assigned $Prob = 1$. This effectively creates a spanning tree, as is discussed in Section 2.4.

- *Key* is a partial truth assignment key as previously defined. The keys are used to match incoming messages in the matching queue. When the keys of an incoming message and previously enqueued message match, the two messages are combined. Two keys (P_1, N_1) and (P_2, N_2) match iff $P_1 \wedge N_2 = N_1 \wedge P_2 = 0$, i.e., the keys are consistent.

- *Priority* is a set of partial minterms whose product, suitably scaled, is a value in the priority system of the underlying implementation. For example, in the KL1 system (PDSS) used in the implementation [3], absolute priorities ranging from 0 to 4095 are acceptable. The *components* of the priority are saved, rather than the priority *value* itself, in order to *compose* priorities. This is discussed later in this section.

Upon receipt of a set of messages (with matching keys) from the matching queues, a node N compresses the set into two output messages, corresponding to both polarities of N. Consider an incoming message set from k parents:

$$\{< Prob_i, Key_i, Priority_i > \mid 1 \le i \le k\}$$

Consider one of the output messages (say for negative polarity of N): $< Prob', Key', Priority' >$, computed as follows:

- Key' is computed from the incoming keys and the appropriate assignment of N, e.g.,

$$\begin{aligned} Key &= (\bigvee_{i=1}^{k} P_i, \bigvee_{i=1}^{k} N_i) \\ Key' &= conv(conv^{-1}(Key) \cup \{\bar{N}\}) \end{aligned}$$

- $Prob'$ is the product of the incoming probabilities and the local conditional probability corresponding to the truth assignment (determined from the incoming keys):

$$\begin{aligned} Q &= cond(\{\bar{N}\}, conv(Key)) \\ Prob' &= Q \prod_{i=1}^{k} Prob_i \end{aligned}$$

  Again, $Prob'$ is sent only to one of the children and all others are given $Prob' = 1$.

- $Priority'$ is computed from the set of matching incoming message priorities. The union of the incoming priority sets is computed and the product of these elements and the local conditional probability corresponding to the truth assignment is taken:

$$Priority' = Q \prod \bigcup_{i=1}^{k} Priority_i$$

  When sending a message, *each* child receives a copy with the same priority. This allows us to separate dynamically load-balancing from correct computation of probability masses. The reason for keeping a priority *set* rather than a priority *value* is to prevent "double counting."

2.2 Example of Message Combination

To illustrate message receipt, consider node F in the previous example. Suppose the messages it receives from parent node B are:

$$< P(b \mid a), (1100000, 0000000), \{P(a), P(b \mid a)\} >$$

$$< P(\bar{b} \mid a), (1000000, 0100000), \{P(a), P(\bar{b} \mid a)\} >$$
$$< P(b \mid \bar{a}), (0100000, 1000000), \{P(\bar{a}), P(b \mid \bar{a})\} >$$
$$< P(\bar{b} \mid \bar{a}), (0000000, 1100000), \{P(\bar{a}), P(\bar{b} \mid \bar{a})\} >$$

The keys have all possible assignments for A and B only. Consider messages from node C that are similar:

$$< P(c \mid a), (1010000, 0000000), \{P(a), P(c \mid a)\} >$$
$$< P(\bar{c} \mid a), (1000000, 0010000), \{P(a), P(\bar{c} \mid a)\} >$$
$$< P(c \mid \bar{a}), (0010000, 1000000), \{P(\bar{a}), P(c \mid \bar{a})\} >$$
$$< P(\bar{c} \mid \bar{a}), (0000000, 1010000), \{P(\bar{a}), P(\bar{c} \mid \bar{a})\} >$$

Let's observe how two of these messages are combined, from B and C respectively:

$$< P(b \mid a), (1100000, 0000000), \{P(a), P(b \mid a)\} >$$
$$< P(c \mid a), (1010000, 0000000), \{P(a), P(c \mid a)\} >$$

The keys match and the messages are combined, producing two new messages emanating from node F:

$$< P(f \mid abc)P(c \mid a)P(b \mid a), (1110010, 0000000), \{P(a), P(b \mid a), P(c \mid a), P(f \mid abc)\} >$$
$$< P(\bar{f} \mid abc)P(c \mid a)P(b \mid a), (1110000, 0000010), \{P(a), P(b \mid a), P(c \mid a), P(\bar{f} \mid abc)\} >$$

A subtle point is that the message probability is not necessarily equal to the product of terms in its priority set. In this example, the difference is $P(a)$ which is a value incorporated in the probability of the right child of node A, not here.

Consider attempting to combine the following two messages from nodes B and C respectively:

$$< P(b \mid \bar{a}), (0100000, 1000000), \{P(\bar{a}), P(b \mid \bar{a})\} >$$
$$< P(c \mid a), (1010000, 0000000), \{P(a), P(c \mid a)\} >$$

They should not be combined because each assumes a truth assignment with opposite polarity for A. The keys do not match ($P_C \wedge N_B \neq 0$), so combination is avoided.

2.3 Message Spawning and Priorities

The algorithm reduces the computation needed to produce a final probability mass estimate if messages are delivered in a schedule related to their priorities. In languages such as KL1, procedure invocations can be assigned priorities for scheduling. Although these priorities are not *guaranteed*, in a multiprocessor implementation scheduling does follow priorities as best it can [7]. A key design issue is how to convert a *message priority* into a *process priority*.

One method for doing this conversion is to spawn a `send` procedure for any message to be sent by a node. It is this `send` procedure *itself* that is given the priority *value* for the message, i.e., the product of the terms in the priority set. The `send` procedure copies the message to issue down the streams to individual child processes. It ensures that one and only one child gets the actual probability value and all others get a value of one. The scheduler sorts `send` goals by their priorities, effectively suspending low mass messages. Thus speculative parallelism is throttled in proportion to the progress of the computation.

In the steady state one expects a large number of `send` goals waiting for their prioritized turn to be executed. It won't matter where in the network these messages correspond — resulting in a balanced execution wherever it is most profitable. This can be subverted if later nodes (closer to the exit port) turn out to have such low conditional probabilities that previously assumed high-priority computations turn out to unnecessary. This is the prevalent danger in any such distributed speculation scheme. It is unprofitable to derive analytic complexity measures for such nondeterminate algorithms: empirical performance measurements of its performance is shown in Section 3.

Mapping the partial probabilities onto priorities is more of an art than a science. A logarithmic mapping was chosen here:

$$Priority = 4095 - min(4095, S \cdot log_{10}(Prob))$$

where 4095 is the highest priority, S is a suitable scale factor, and $0 < Prob \leq 1$. For example, if one wishes to break the priority range into five logarithmic decades, choose $S = 4096/5$. A problem arises when attempting to choose S to be effective when collecting low mass, such as 0.7, as well as high mass, such as 0.98. Furthermore, within a given search, the optimal S changes as the mass collects. Developing a more sophisticated, dynamic mapping function is a topic of future research.

2.4 Correctness

In this section a correctness proof of the algorithm is sketched. For this purpose one can safely ignore the priorities and their affect on scheduling because this is orthogonal to computation of the probability mass. The full evaluation of a Bayes net of n variables consists of computing the sum of 2^n minterms, each containing n partial minterms (conditional probabilities). To prove correctness, one must prove that every truth assignment is covered and that within a given minterm, each partial minterm appears once and only once.

Consider our technique of sending a message from a parent node to its children. One selected child is sent the "real" probability mass and the others are sent a mass of one. The effect of receiving a message with mass one is as if the receiving node is a root of the graph (again, ignoring priorities). Thus the scheme effectively removes all edges but one between any parent and its children. It is not difficult to see that the resulting graph is a spanning tree of the original graph, i.e., there is only one path from any node to the exit node. Thus a spanning tree is guaranteed to have minterms wherein each conditional probability appears once and only once.

To see that every truth assignment is considered, recall that *two* messages are sent from a node N, corresponding to the conditional probability of N given its parent's truth assignment, for both polarities of N. Thus by induction on the nodes in the spanning tree, one can prove that every possible truth assignment is represented in the totality of messages received at the exit port.

3 Implementation

This section describes the key parts of the implementation: how the matching queue is designed, how messages are copied, how termination works, how priorities are implemented, and how the process network is specified.

3.1 Matching Queue

The matching queue is a set of queues that accept input messages arriving from the parents of a given node, and combine messages that have consistent keys. It is required that one message from each parent be combined before the combination can be processed and propagated in the network. What makes the matching queue difficult to design is the capability of quickly matching messages.

In our prototype there were two competing designs for the matching queue:

- Copying Method: streams from all parents of a given node are merged into a single stream. Messages from that stream traverse the queue, checking if they match any entry already enqueued. If so, they are combined. If no match is found, a new entry is created. This scheme implies that *every* message must be assigned a unique truth assignment from among the common ancestors of the child node. Thus messages need to be reproduced, at the parent, with alternative keys covering the truth assignment space. Although this method is conceptually simple, copying can be exponential in the number of ancestors. In practice, such behavior was observed and so it was abandoned for performance reasons.

- Active Queue: Instead of merging all input streams to a given node, each stream leads to its own queue at the child node. A queue manager is responsible for routing messages through one queue after the other, combining messages once per queue. The final output stream from this chain of queues holds fully combined messages ready for processing and propagation to other parts of the network. The key insight here is that *no copying is necessary.* In practice this proved to have satisfactory performance and to be quite elegant to implement in a concurrent language.

An expanded description of the active queue, which was adopted for the prototype, is now given. Suppose there are k parents of a given node and thus $k-1$ queues. The parent with the greatest number of ancestors is called the *lead parent* or *leader.* Matching entails combining k messages: one from the lead parent, and $k-1$ from each of the queues. Matching does not mean *exact* matching (as in the Copying Method above), since keys in messages other than the lead queue are incomplete. Nonconflicting key matches are sufficient.

At a given node, the $k-1$ queues are linked and called *followers* because they "follow" the leader. A message that arrives from the leader and is subsequently combined is called a *packet.* A packet is passed from one follower to the next, combining it with some message from each follower. Combining entails key matching, followed by adding information to the packet; however, the follower message is *never removed.* Thus followers fill up and their memory is not reclaimed. If the packet successfully passes through all followers, it is finally processed by the node itself.

If the packet does not match any message in a follower this indicates that a needed follower message has not yet arrived. It cannot be the case that all the follower's messages have arrived but none match. This suggests building *active follower queues* similar to the pipeline of prime filters in the Sieve of Erastothenes (e.g., [8]). When a message arrives at a follower from its parent, it is transformed into a new filter process and added to the end of the follower's pipeline.

If a packet fails to match any filter element in a follower, it will naturally suspend at the end of the follower's pipeline! A new message sent from this follower's parent will be transformed into a new filter process. This will cause resumption of the attempt to match the packet. Eventually either the match will succeed or the program will terminate.

The beauty of this scheme is that packets that are incompletely matched suspend at the precise spot where they need more information. Furthermore, such suspended packet

```
spawn( _, [], S0, S1 ) :- S1 = S0.
spawn( Stop, [ F | Fs ], S0, S2 ) :-
   follower( Stop, F, S0, S1 ),
   spawn( Stop, Fs, S1, S2 ).

follower( _, [], Pin, Pout ) :- Pout = Pin.
follower( Stop, [ M | Ms ], Pin, Pout ) :-
   M = packet( Key, [ Item ] ) |
   filter( Stop, Pin, Key, Item, S_fail, S_match ),
   follower( Stop, Ms, S_fail, Out ),
   merge( { S_match, Out }, Pout ).

filter( _, [], _, _, S_fail, S_match ) :-
   S_fail = [], S_match = [].
filter( Stop, [ P | Ps ], Key, Item, S_fail, S_match ) :-
   P = packet( Key0, Ms ),
   check_keys( Key0, Key, Status ),
   subfilter( Stop, Status, P, Ps, Key, Item, S_fail, S_match ).

subfilter( stop, _, _, _, _, _, S_fail, S_match ) :-
   S_fail = [], S_match = [].

alternatively.

subfilter( Stop, no, P, Ps, Key, Item, S_fail, S_match ) :-
   S_fail = [ P | Rest ],
   filter( Stop, Ps, Key, Item, Rest, S_match ).

subfilter( Stop, yes, _, Ps, Key, Item, S_fail, S_match ) :-
   combine( P, Key, Item, NewP ),
   S_match = [ NewP | Rest ],
   S_fail = [ P | Rest0 ],
   filter( Stop, Ps, Key, Item, Rest0, Rest ).
```

Figure 2: Active Queue in KL1 (Simplified for Illustration)

will not delay subsequent packets that can match.

A sketch of the active queue code, written in KL1, is shown in Figure 2. Procedure *spawn* creates a group of *follower* processes that send packets through a chained stream (variables S0, S1, S2). Each follower is issued an input stream F where it receives messages from its parent. All procedures share a common first argument: the global termination signal (see Section 3.2).

Process *follower*(*Stop*, M_{in}, P_{in}, P_{out}) has an input stream M_{in} of arriving messages from a non-lead parent. Input stream P_{in} has packets arriving from previous followers in the chain (or from the leader to the first follower). P_{out} is an output stream of packets to the next follower in the chain (or from the last follower to the receiving node). For each arriving message, *follower* spawns a *filter* process that will filter packets going through the chain.

Procedure *filter* checks if the packet's key matches the filter's key and acts on the

```
 from
 lead ----+       +------+
parent    |       |      |
          V       |      V
       +-----+    |   +-----+
       | M1  |-->O    | M4  |-->O-->
       |     |    |   |     |   |
       +-----+    |   +-----+   |
          |       |      |      |
          |       |      |      |
          V       |      V      |
       +-----+    |   +-----+   |
       | M2  |-->O    | M5  |-->O
       |     |    |   |     |   |
       +-----+    |   +-----+   |
          |       |             |
          |       |             |
          V       |
       +-----+    |
       | M3  |-->O
       |     |    |
       +-----+    |
                  |
```

Figure 3: Illustration of Active Queue

status of this check. If match fails then the original packet sent to the next filter (clause 2 of *subfilter*). This recursive call will suspend if there is no corresponding filter spawned yet: the filter *will* be spawned because all packets must either match eventually or be discarded during termination. If the match succeeds then the filter's message is combined with the packet and the combined packet is sent to the next follower (clause 3 *subfilter*). Critically, the original packet is also sent to the next filter within the current follower to ensure that all possible keys are matched. This "cascading" technique produces an exponential number of combinations if needed.

An example of an active queue process network is illustrated in Figure 3. An initial packet stream from the lead parent enters a chain of three filters comprising the first follower. These filters contain messages `M1`, `M2`, and `M3`. The outputs of these filters are *merged* into a stream feeding the second follower. Builtin mergers are denoted by "`O`". The second follower is composed of two filters holding `M4` and `M5`. Note that the *follower* process corresponding to each follower is actually suspended, waiting for more messages to arrive so that it can extend its filter chain. Any incomplete packets that require such a message will suspend on a call to the *follower* process. Thus incomplete packets are tucked out of the way of subsequent packets that can be completed. Furthermore, incomplete packets are automatically resumed once new filtering information is received at the required follower. Note the dangling stream pointers to the last builtin merger of each follower: this is perfectly acceptable and will not prevent messages from five joint streams from proceeding.

The matching queue processes are given the same system priority as the node processes (see Section 3.3). However, among the node and queue processes that might be available for scheduling at any time, ideally those processes corresponding to the *exit node* should have highest priority. This will facilitate eager termination, as is discussed in the next section. For other details of the queue construction, see the Appendix for the actual source code.

3.2 Termination

The exit node of the network plays a special role for termination. Messages combined at the exit queue have their probability masses accumulated. When the accumulated value reaches a certain threshold, a global termination signal will be set. Each process in the network kills itself when this signal is set. It is *critical* to performance that the global signal be applied to all frequently executed process goals, particularly `send` goals and matching queue goals. Without such termination, unnecessary messages will continue to be generated, and previously enqueued messages will continue to be processed, significantly increasing the time to termination.

Refering back to Figure 2, one can see how termination is performed once the stop signal is set. The first clause of *subfilter* is devoted to early termination. If the global signal is bound to the atom `stop`, *subfilter* terminates itself after closing its output streams. Even if the *subfilter* task was originally suspended, it will be resumed and then terminated. The KL1 control construct `alternatively` guarantees that the first clause is attempted *first* during every invocation (without such a guarantee, the first clause might never be tried at all). It is possible to add early termination clauses to *filter* and *follower*; however, since these procedure invoke *subfilter*, not much would be gained.

Note that termination by the technique of "short circuiting" messages (e.g., [8]) is not needed because the termination condition is determined at a central location.

3.3 Prototype

The proposed algorithm was implemented in KL1 and executed on the PDSS pseudo-parallel runtime system [3]. The KL1 program consists of 785 source lines of code (including comments), not including the data description of the network. The most interesting part of the program is the definition of `send` for dispatching the messages. A node process invokes `send` as follows: `send(...,Outs,...)@priority(*,Rate)` where `Outs` is a list of output streams to child nodes, and `Rate` is the KL1 priority value computed from the priority set in the algorithm. The notation above indicates that `send` is assigned an *absolute* priority of Rate, which ranges from 0 to 4095. The other arguments indicate the messages to be sent. Children of send, which recurse on the Out list while doing the sending, inherit the Rate priority. Interesting, the node procedures retain their high default priority. A node process will nominally be suspended waiting for a message to appear from its message queues. Thus prioritized senders/producers control non-prioritized receivers/consumers by feeding or starving them of data.

4 Performance Evaluation

To introduce the evaluation methodology, consider the net shown in Figure 1. The performance of this net is labeled "example 1" in Table 1. Computing $P(g) + P(\bar{g})$, 0.7 mass is collected at node G in the first 12 complete minterms, generating a total of 50 partial minterms. Considering that to compute the full 1.0 mass, there are $2^7 = 128$ minterms exiting node G and 186 partial minterms generated in the net, the algorithm appears to be polynomial if not linear.

The other benchmarks in Table 1 have similar attributes. The table lists the benchmarks and their search performance for different captured masses. Each time (in msec) is the lowest observed execution time. This time includes both building and searching the net. The empty table entries indicate that the program exceeded its memory limitation and therefore did not terminate.

The execution complexity is highly variant on the net topology and its conditional probabilities. Thus it is difficult to discern a clear pattern among these benchmarks. However, it is clear that to collect 70% mass, the complexity is not growing exponentially with net size. The number of messages sent is a more fair indicator of complexity than time because the latter includes the complexity effects of accessing several data structures.

Additional experimentation was performed concerning setting a *threshold* to reduce message traffic. If a the mass of a partial minterm is less than the threshold, its message is discarded. This reduces network traffic that does not add to the solution. As the threshold is increased, the traffic decreases, until so many messages are discarded that the desired total mass cannot be collected at the exit node. This is illustrated for nets 4 & 5 in Figures 4 and 5 respectively. These programs were measured for PDSS on a Sparc-10. Notice the time reduction for greater thresholds, especially for large masses and complex networks.

The current prototype breaks down for large nets when collecting high mass because of the nature of the data structures used. For example, the priority sets are implemented as ordered lists, over which set union is frequently performed. The logarithm function for the priority mapping is done by a simplified table lookup because floating point operations

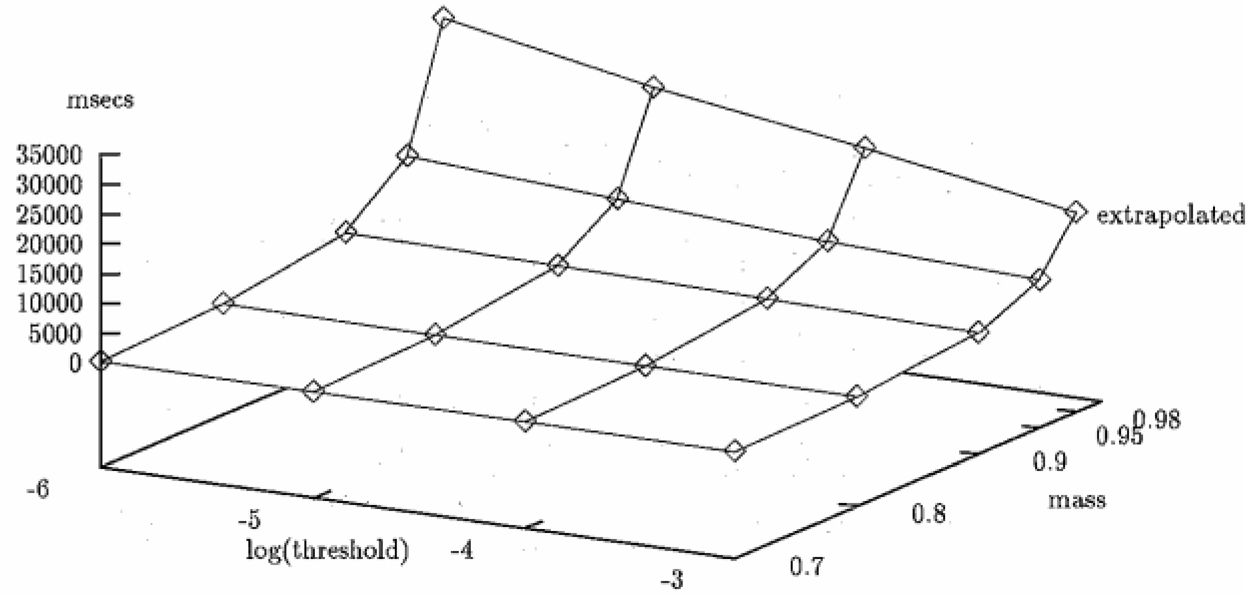

Figure 4: Threshold *vs.* Mass *vs.* Time (Example 4)

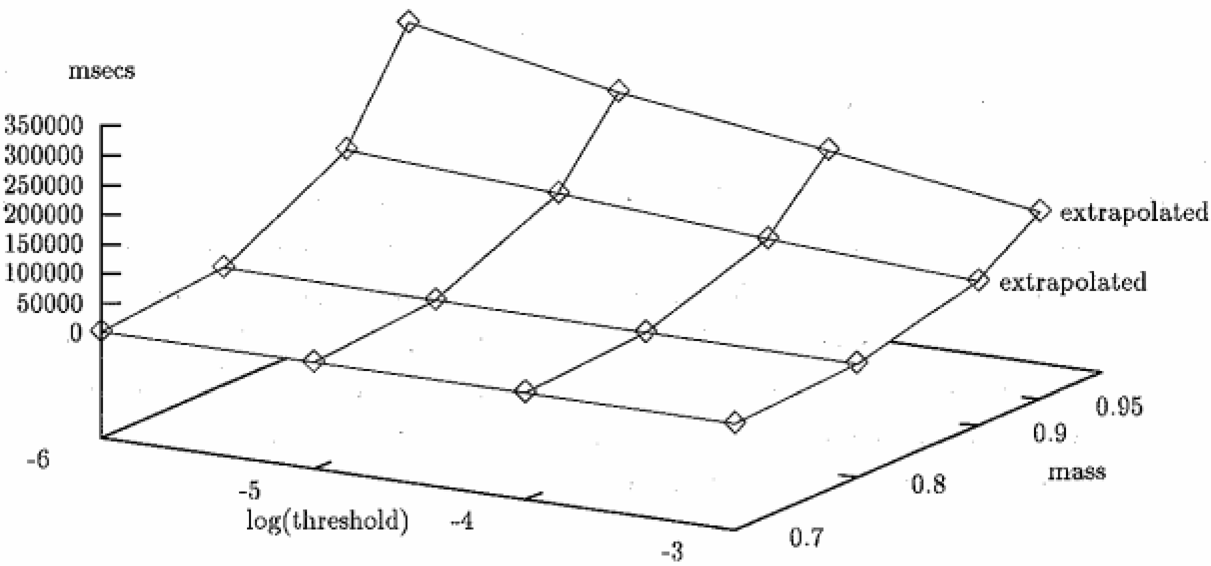

Figure 5: Threshold *vs.* Mass *vs.* Time (Example 5)

		mass captured				
example	nodes	70%	80%	90%	95%	98%
		minterms required				
1	7	12	16	24	37	49
2	16	3	4	4	4	10
3	16	15	21	28	31	51
4	19	5	9	29	51	126
5	24	12	26	82		
		messages sent				
1	7	50	54	70	85	99
2	16	45	46	46	46	68
3	16	67	76	86	91	160
4	19	51	83	166	219	393
5	24	116	235	343		
		msec				
1	7	340	370	540	640	740
2	16	350	350	340	350	1,860
3	16	510	570	630	670	10,520
4	19	1,000	2,910	17,100	54,180	151,210
5	24	4,960	48,510	339,670		

Table 1: Performance of Benchmark Nets (Sun Sparcstation-1)

are too expensive in PDSS (each operator spawns a process). All the nets use lists to implement the conditional probability tables. Only the 24-node net exploited vectors instead of lists for only two (the largest) of the 24 tables. The KL1 prototype is about as fast as it can be. One might imagine implementing the algorithm in an imperative, explicitly parallel language. However, the advantages of using a concurrent logic language are that implicit dataflow synchronization of active process networks and prioritized task scheduling are "free" to the programmer.

4.1 Multiprocessor Considerations

Future work entails porting the KL1 program to a multiprocessor implementation, e.g., PIM [7] or KLIC [1], to measure speedup. Ordinarily this would require no modification to the program because parallelism is implicit in the language. However, in our case, porting will require simulating global priorities within the PIM (or KLIC) systems, which support only local priority queues per processor. Preliminary experiments on PIM indicated that the algorithm is extremely sensitive to priorities, i.e., if local queues continue goal reductions without globally synchronizing the priorities, exponential explosion can occur. To solve this problem, two alternatives are being considered. One is to build a deamon that artificially synchronizes the local priority queues in a PIM cluster. This deamon must be scheduled fairly frequently to avoid having any queue "run away." Another is to build a meta-interpreter that simulates a global priority queue and then schedules the tasks on individual processors. K. Kumon has implemented both techniques and we are analyzing their behaviors.

5 Conclusions

A concurrent algorithm was introduced to evaluate a Bayes network. The key contribution is to convert the network into a concurrent process network and send partial minterms as messages. These messages are prioritized as a function of the mass they represent. To solve the potential problem of "double counting" partial minterms, the message priorities and masses are decoupled in a novel fashion. Furthermore an interesting "active queue" was developed to allow efficient combining of prioritized messages.

A sketch of a correctness proof for the algorithm was given, as well as a real implementation in KL1. The KL1 implementation, running under PDSS, a sequential emulator on a Sun Sparcstation, demonstrated the ability of the concurrent algorithm to achieve what appears to be polynomial reduction in work.

Future work involves further optimizations such as better priority assignment, as well as exploring multiprocessor implementations of global priority.

6 Acknowledgements

The author was supported by a grant from the Institute of New Generation Computer Technology (ICOT), and an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. I thank K. Kumon for interesting discussions about this research.

References

[1] T. Chikayama. A Portable and Efficient Implementation of KL1. In *International Symposium on Programming Language Implementation and Logic Programming*, Madrid, September 1994.

[2] B. D'Ambrosio. Incremental Probabilistic Inference. In D. Heckerman and A. Mamdani, editors, *Conference on Uncertainty in Artifical Intelligence*, pages 301–308. Morgan Kaufmann Publishers Inc., July 1993.

[3] ICOT. *PDSS Manual (Version 2.52e).* 21F Mita Kokusai Bldg, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, February 1989.

[4] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann Publishers Inc., Palo Alto, 1988.

[5] B. Poole. The Use of Conflicts in Searching Bayesian Networks. In *Conference on Uncertainty in Artifical Intelligence*, pages 359–367, July 1993.

[6] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[7] K. Taki. Parallel Inference Machine PIM. In *International Conference on Fifth Generation Computer Systems*, pages 50–72, Tokyo, June 1992. ICOT.

[8] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA., 1991.

[9] E. Tick and B. D'Ambrosio. Evaluating Bayes Nets with Concurrent Process Networks. Technical Report CIS-TR-94-15a, Dept. of Computer Science, University of Oregon, Eugene, OR 97403, September 1994.