Embedded Bayesian Networks: Anyspace, Anytime Probabilistic Inference

Fabio T. Ramos, Fabio G. Cozman and Jaime S. Ide
Depto de Engenharia Mecatrônica
Escola Politécnica, University of São Paulo
Av. Prof. Mello Moraes, 2231, 05508-900, São Paulo, SP - Brasil
fabioram@usp.br, fgcozman@usp.br, jaime.ide@poli.usp.br

Abstract

An important aspect of probabilistic inference in embedded real-time systems is flexibility to handle changes and limitations in space and time resources. We present algorithms for probabilistic inference that focus on simultaneous adaptation with respect to these resources. We discuss techniques to reduce memory consumption in Bayesian network inference, and then develop *adaptive conditioning*, an anyspace anytime algorithm that decomposes networks and applies various algorithms at once to guarantee a level of performance. We briefly describe *adaptive variable elimination*, an anyspace algorithm derived from variable elimination. We present tests and applications with personal digital assistants and industrial controllers.

Introduction

What are the basic techniques that would allow embedded real-time systems to engage in probabilistic reasoning? In this paper we provide an answer to this question by presenting Bayesian network algorithms that adapt to varying memory and time resources.

The hallmark of probabilistic inference algorithms for embedded real-time systems should be *flexibility*: not just to be able to produce a solution at any given stopping time, but also to be able to plan ahead how to use available, possibly scarce, resources, and to quickly respond to changes or failures in resource allocation; not just to adapt to changes in memory *or* time resources, but to adapt to changes in memory *and* time resources. To explore these trade-offs, in this paper we focus on methods that combine conditioning operations with exact and approximate inference, producing algorithms that are anyspace *and* anytime.

We start by discussing techniques that reduce memory consumption in the variable elimination algorithm. Our purpose is both to show what can be done with standard methods and to discuss some of our assumptions concerning embedded systems. We also describe the EmBayes system, a very compact inference engine that we have used in real applications.

We then investigate new ways of thinking about embedded Bayesian networks, either for real-time systems and for non-real-time systems. Our algorithms attempt to first guarantee execution with a given memory limit, and then to explore approximate inference algorithms when time constraints are imposed. We present *adaptive conditioning*, an algorithm that can work with any given memory constraints by decomposing networks and conditioning variables. Adaptive conditioning splits a network into sub-networks and selects appropriate algorithms for each sub-network. We also describe *adaptive variable elimination*, an algorithm that complements adaptive conditioning by allowing an embedded system to adapt to varying memory constraints during inference.

The distinguishing characteristic of adaptive conditioning is that it divides a network into smaller pieces that can be processed by *different* algorithms. Although the idea of using more than one algorithm in a single inference run is not new (Kjaerulff 1995), we take it a more general approach than previously done, and explore the consequences for space and time constraints. Our approach provides a general framework for probabilistic inference; the vantage point of embedded systems provides an excellent panoramic view of inference algorithms and how the various techniques relate and compare to each other.

We finish by presenting tests and applications with personal digital assistants and industrial controllers. In the last section we summarize our results and discuss parallelization and hardware-based systems.

Embedded and Real-Time Bayesian Networks

Embedded systems are present in devices ranging from mobile phones to industrial controllers. We differentiate between embedded systems that have real-time requirements and systems that do not (a real-time system fails if applications do not finish within a prescribed time). We can view embedded systems as little agents, possibly connected into a large community.

Today, embedded systems usually have a modest amount of memory at their disposal; this situation is changing dramatically, both due to hardware and software developments. We can expect that one thing will always be true regarding "intelligent" embedded systems: computational resources vary during operation. Memory may be removed due to hardware failures or maintenance; and memory may be added physically or through network connections. Embedded systems may also be highly connected and able to share resources, or very isolated and unable to communicate with other devices. Finally, timing constraints can also vary from one inference to the next.

A critical aspect of "intelligent" embedded systems is thus bound to be the flexibility of reasoning algorithms. We can classify changes in resources in two categories. We can have *offline* changes, in which the inference engine is informed about the resources before inference starts. We can also have *online* changes, in which the inference engine is informed about changes in resources during an inference run.

There is a natural tension between space and time constraints; no algorithm can be expected to produce the same answers under arbitrary space *and* time constraints. We suggest that the most relevant approach for embedded systems is to require that strict space and time constraints are fixed before any inference; these constraints *must* be met, possibly with some degradation in answers when time constraints are too tight, and with the expectation that if more time is available, answers will improve in quality.

In this paper we assume that memory and time are scarce and that algorithms must adapt to variations in these resources. The same type of assumption is taken by anytime algorithms. We suggest that the following definitions are relevant in our context (note that the first definition is slightly different from the usual meaning of anytime algorithms):

Definition 1 *An algorithm is* anytime *if it can produce a solution in a given time T, and the quality of solutions improve with time after T.*

Definition 2 *An algorithm is* anyspace *if it can improve its performance with increasing space, assuming that the available space is larger than some minimal amount.*

We focus on anyspace algorithms that improve running *time* performance with increasing space. Our interest then is to extend existing trade-offs between space and time (Darwiche 2001; Dechter 1996b) to anyspace anytime methods according to Definitions 1 and 2.

In this paper we focus on probabilistic reasoning conducted through Bayesian networks. A Bayesian network represents a joint distribution over a set of variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ through a directed acyclic graph with n nodes (Pearl 1988). Each node represents a variable X_i and is associated with a conditional distribution $p(X_i|\Pi_i)$, where Π_i is the set of parents of X_i in the graph. The joint distribution represented by the network factorizes as $\prod_i p(X_i|\Pi_i)$. An *inference* is obtained when we compute $p(\mathbf{Q}|\mathbf{E})$ for *query* variables $\mathbf{Q}$ and *evidence* variables $\mathbf{E}$.

Variable Elimination in Embedded Applications

We began our study of embedded systems when we attempted to incorporate probabilistic reasoning into programmable logic controllers for commercial applications (we later describe one of these applications). Our first reaction was to use existing methods with relatively small Bayesian networks. We assumed that our target systems would have minimal ability to communicate, so the inference engine would have to be small (so as to allow downloads and remote operations). We also assumed that our inference engine would be a general purpose system, as opposed to a very specialized one (Darviche & Provan 1996).

We produced a generic, small, efficient, and easy to port inference engine by implementing the variable elimination algorithm (reviewed in (Cozman 2000), also known as bucket elimination (Dechter 1996a) and peeling (Cannings, Thompson, & Skolnick 1978)). Contrary to many implementations, we do not store junction trees or similar structures in our engine; we use d-separation to discard unnecessary variables and compute an ordering for variables at each inference request. We have concluded that computing orderings and triangulations with existing heuristics is an *extremely* fast procedure that can be repeated at almost no cost. The simplifications obtained with d-separation are more effective than the pre-computation of orderings.

We have coded variable elimination so that every intermediate result can stay in memory only for the strictly necessary time; in this manner, memory can be exhausted only if the size of separators produced by variable elimination are too large. The algorithm was coded in Java, as this language can be easily ported among embedded applications. The resulting system is called EmBayes, and is extremely compact, occupying less than 30KBytes of space. The engine is based on the JavaBayes system, as both use the same coding interfaces; consequently, networks can be built and visualized in a friendly graphical interface using the JavaBayes system (freely distributed at http://www.cs.cmu.edu/˜javabayes). The EmBayes system is now used in a variety of locations as an effective method to embed probabilistic reasoning. From our experience with the EmBayes system, we decided to look for more flexible inference algorithms. The results of our investigation in this direction are described in the remainder of this paper.

Adaptive Conditioning

The algorithm described in this section, which we call adaptive conditioning, attains a high level of control regarding space and time constraints. As mentioned previously, our goal is to guarantee execution given hard limits on space and time, and *also* guarantees anytime behavior if necessary. The algorithm can trade time and space in several dimensions, essentially by interleaving conditioning operations with standard inference methods.

The basic idea of adaptive conditioning is to divide a network in several sub-networks so as to enforce strict memory constraints, and to run *different* algorithms in each network so as to attend to timing constraints. Networks are decomposed so that separators are guaranteed to be smaller than some given limit, and inference algorithms are distributed among sub-networks so as to obtain anytime behavior when needed. Decomposition of networks is accomplished by conditioning variables.

Many algorithms have used conditioning to trade space and time. For example, conditioning is employed in the super-bucket and the conditioning-plus-elimination algorithms by Dechter (Dechter 1996b); we actually use the ideas of conditioning-plus-elimination (described later), when we discuss the possibility of online changes in resources. The algorithm that is most similar to adaptive conditioning is recursive conditioning, where a network is recursively split until single-node networks are processed (Darwiche 2001). One obvious difference between recursive and adaptive conditioning is that the latter does not attempt to reach single-node networks, instead leaving the option of selecting algorithms for multi-node sub-networks. We could understand adaptive conditioning as an algorithm that constructs a partial dtree[1] over sub-netwoks and runs different inference algorithms in each sub-network. Even though this mental picture may be helpful, the algorithms are different both in purpose and in design. Not only we want to have anytime inference when necessary, but our goal is to guarantee execution under hard memory constraints; consequently, adaptive conditioning focuses on restricting separator size, not on obtaining balanced execution with respect to time. The dtree that we could build on top of our sub-networks certainly does not have minimum height, nor is it required; the rationale is that a network is split only when memory is exhausted, and in those circumstances the caching facilities provided by dtrees are not meaningful. While recursive conditioning always tries to find a "balanced" decomposition that guarantees worst-case $O(n \exp(w \log n))$ time, adaptive conditioning only decomposes when necessary, reaching worst-case $O(n \exp(n))$ time (where n is the number of variables and w is the width of the elimination ordering). Note that adaptive conditioning is more stringent regarding memory, and degrades to a brute-force approach in the worst-case. If time constraints are required, adaptive conditioning resorts to anytime behavior.

Active conditioning receives a request for an inference and a description of available resources: maximum allowable memory and maximum time spent to produce first results (with an understanding that, if more time is available, quality of results will improve). We assume that space constraints are specified in terms of the maximum allowable separator size, as this is the critical value in inference. Adaptive conditioning is designed to operate in two phases. First the algorithm plans the operations that will be executed. The second phase is then the actual execution of inference operations. We present the offline planning phase in the next subsection, and then describe the execution phase. In the last subsection we discuss methods that allow the algorithm to run the online phase even when the available memory decreases during execution.

[1]The binary tree that represents the recursive splits in recursive conditioning.

Planning Phase

The planning phase of adaptive conditioning receives resource constraints and must: (1) decide where to split networks (in cases when memory is scarce), by conditioning variables; (2) distribute inference algorithms among the sub-networks (in cases when time constraints are present); (3) organize caching of intermediate results in cases where a network split leaves memory unused.

The first goal of the planning phase is to choose the best decomposition of the network given constraints in separator size. The problem of finding the best decomposition is known to be NP-hard (Wen 1990). An heuristic method that conforms to our requirements is: build a bucket tree (or a junction tree) and split the clusters of variables at those separators that exceed given space constraints. The variables in these separators are conditioned, and consequently the original network is divided into smaller pieces. Figure 1 shows a junction tree and a division in three sub-networks.

In general, conditioning variables need not be separators. We say that conditioning variables form *cutsets*; after all sub-networks are generated, adaptive conditioning can find arbitrarye cutsets for the sub-networks (Darwiche 2001). A cutset for variables $\mathbf{X}'$ is a set of variables $\mathbf{C}$ such that if every $X_i \in \mathbf{C}$ is observed, variables $\mathbf{X}'$ are independent of all other variables in the network. Cutsets contain the parents of variables in $\mathbf{X}'$ that are not in $\mathbf{X}'$.

It is then necessary to assign inference algorithms to each one of these sub-networks. We divide algorithms in two classes, exact and approximate. Exact algorithms could be further divided into clustering-based methods and conditioning-based methods. Approximate algorithms can involve Monte Carlo simulation (importance sampling, Gibbs sampling) or various deterministic techniques (structural simplifications, partial summations). Several approximation algorithms have an anytime character; we have opted to always use Gibbs sampling in our implementation. A more sophisticated implementation could be to select among approximation algorithms; it is interesting to note that Monte Carlo methods usually work well when probabilities are not extreme, while deterministic approximations tend to work better when probabilities are extreme — this

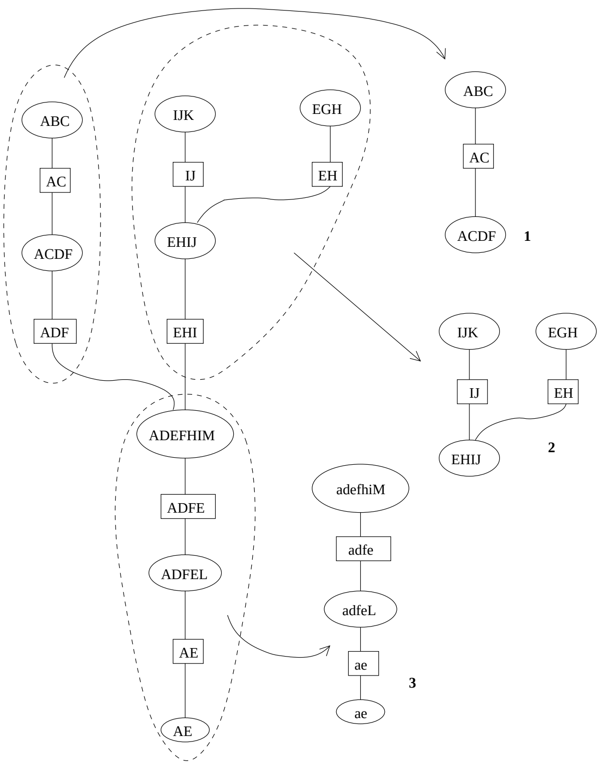

Figure 1: Decomposition of a junction tree in three sub-networks. Clusters are ovals; separators are rectangles. Conditioning variables are not capitalized.

criterion can guide the choice of approximations in sub-networks.

We chose to use an exact method that combines clustering and conditioning. The last subsection briefly describes adaptive variable elimination, an algorithm that uses previous ideas to guarantee inference results even when space constraints change during execution. Every sub-network runs adaptive variable elimination if time constraints are not imposed. To be able to select algorithms with time constraints, it is necessary to obtain an estimate of running times. For each sub-network, we run adaptive variable elimination *once* (that is, for one of the values of the conditioning variables). We can easily compute how many times each sub-network must be processed, and so we obtain an estimate of total running time. When the estimated running time exceeds the required limits, we switch to approximations. Here a decision must be made, because we can either run exact inference for some sub-networks and approximate inference for others, or we can run exact and approximate inferences for different values of the conditioning variables. In our implementation we always assign a single algorithm to each sub-network; the strategy is to assign exact algorithms to as many sub-networks as possible, selecting first the sub-networks that are "closer" to the query variables. We expect "closer" sub-networks to be more influential, as suggested by the analysis of loopy-propagation algorithms (Weiss & Freeman 1999).

After we process each sub-network once, we can examine the new separators and check whether we have available memory. It is possible that a split reduces drastically the size of separators and then space may become abundant. When this happens, the free memory can be used for caching results so as to reduce time spent in inferences. The same caching techniques developed for recursive conditioning can then be employed (Darwiche 2001).

Execution Phase

A network may be split in several sub-networks during the planning phase. We have to process each one of these sub-networks in the execution phase, when inferences are actually produced; sub-networks are processed conditioned on their cutsets. The following theorem shows the operations that must be performed.

Theorem 1 (Adaptive Conditioning) *Let* $\mathbf{C}$ *be a set of variables over network* $\mathcal{N}$ *that, when observed, make the network disconnected into sub-networks* $\mathcal{N}_i$ *with* $i = 1, 2, ..., n$*, and let* $\mathbf{Q}$ *be a set of variables in* $\mathcal{N}$ *for which we want to compute marginals. Then,*

$$Pr^{\mathcal{N}}(\mathbf{Q}) = \sum_{\mathbf{C}} \prod_i Pr^{\mathcal{N}_i}(\mathbf{Q}_i, \mathbf{C}_i). \tag{1}$$

where $Pr^{\mathcal{N}}(\mathbf{Q})$ *is the probability of* $\mathbf{Q}$ *in the network* $\mathcal{N}$*,* $\mathbf{Q}_i$ *is a subset of* $\mathbf{Q}$ *that is in* $\mathcal{N}_i$ *and* $\mathbf{C}_i$ *is a subset of* $\mathbf{C}$ *that is in* $\mathcal{N}_i$*.*

The theorem is an immediate generalization of the basic theorem of recursive conditioning (Darwiche 2001); the difference is that recursive conditioning always splits in two sub-networks at a time, and consequently keeps only two terms in the summations. Adaptive conditioning may instead divide a network at once in many sub-networks. The proof of the theorem follows from the fact that $Pr^{\mathcal{N}}(\mathbf{Q}) = \sum_{\mathbf{C}} Pr^{\mathcal{N}}(\mathbf{Q}, \mathbf{C})$, and the fact that the cutset $\mathbf{C}$ leads to a decomposition of the form $\prod_i Pr^{\mathcal{N}_i}(\mathbf{Q}_i, \mathbf{C}_i)$.

So, we have to compute $Pr^{\mathcal{N}_i}(\mathbf{Q}_i, \mathbf{C}_i)$ for each network $\mathcal{N}_i$ and for each different instantiation of variables in $\mathbf{C}$. Note that the number of inferences grows exponentially with the number of variables in $\mathbf{C}$.

As an example, consider the network $\mathcal{N}$ in Figure 2. This network is decomposed into three sub-networks by conditioning on C and B. The Figure 3 shows the result of this decomposition. In this figure the dashed nodes represent conditioned variables that were added in each sub-network. In this example, if we want the marginal probabilities of E and F, we have to compute the following probabilities: $Pr^{\mathcal{N}_1}(C \mid B' = b')$, $Pr^{\mathcal{N}_2}(E, F \mid C' = c', B' = b')$ and $Pr^{\mathcal{N}_3}(B)$. Suppose all variables were binary. We have to compute $Pr^{\mathcal{N}_1}(C \mid B' = b')$ twice, we have to compute $Pr^{N_2}(E, F \mid C' = c', B' = b')$ four times and we have to compute $Pr^{N_3}(B)$ only once.

Figure 4 presents the steps that are executed to produce inferences, following the operations in Theorem

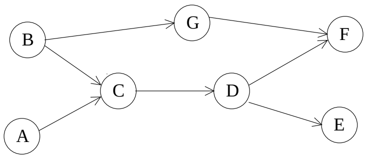

Figure 2: A Simple Bayesian Network $\mathcal{N}$.

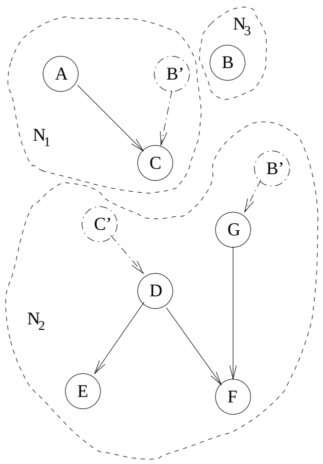

Figure 3: The Bayesian Network $\mathcal{N}$ decomposed into three sub-networks.

1, and possibly using caches to store intermediate results. As we can see in the pseudocode, we can attain linear space in the number of nodes if caching is not used. The computational complexity for this algorithm is affected by the number of inferences that we have to perform in each sub-network, which in turn depends on the number of variables in the cutset:

Theorem 2 *Given a Bayesian network with n variables and a cutset of width w_c that decomposes the network into w_n sub-networks, the number of inferences performed by adaptive conditioning is $O\left(w_n \cdot exp\left(w_c\right)\right)$.*

Adaptive Variable Elimination

We consider now the possibility that, *during* the execution phase, memory is suddenly reduced. What can be done in those sub-networks that are running exact inference? A simple solution is to run exact inference algorithms based on clustering, but resort to conditioning operations if memory problems occur — this is the idea of the conditioning-plus-elimination algorithms suggested by Dechter (Dechter 1996b). We adopt this idea.

In our implementation of adaptive conditioning, we use a conditioning-plus-elimination scheme for every sub-network that can run exact inference. We call the scheme *adaptive variable elimination*, as it is based on variable elimination. The idea is simple: if suddenly there is no more space to allocate separators, then variables in intermediate buckets are conditioned; computation will take more time as it uses less space.

Adaptive conditioning — Execution phase

```
AC(T)
result ← 0
for each instantiation of variables in cutset do
    result ← result + MULTI(T)
return result
MULTI(T)
result ← 1
for each sub-network in T do
    if cache[E] is nil then
        result ← result · inference(E)
        cache[E] ← inference(E)
    else
        result ← result·cache[E]
return result
```

Figure 4: Pseudocode for the execution phase of adaptive conditioning.

Experiments and Applications

In this section we show how adaptive conditioning handles realistic networks by analyzing different constraints on time and space. We also present a real application to ilustrate how adaptive conditioning can be used.

We have implemented adaptive conditioning in the Java language, using the EmBayes package (described previously) as the underlying inference engine for exact inference. When approximate inference is required, we use Gibbs sampling. We tested the algorithm with simulated and real networks; the simulated networks were generated using an iterative procedure for producing random directed acyclic with adequate properties graphs (Melançon, Dutour, & Bousquet-Melou 2001). Code for adaptive conditioning and for random network generation is available from the authors.

Tests with real networks

We describe here tests with two real networks: Alarm and Munin. We evaluate the decomposition of Alarm; that is, we evaluate how the actual maximum separators compare to the prescribed space limits. For this test we decomposed the Alarm network in sub-networks and performed inferences using adaptive variable elimination for each sub-network and for all variables. As we see in Figure 5, when we have a low memory constraint (the first column), the maximum separator weight produced was often smaller than the limit. This indicates that we still have memory to use, and so we could use caching. For the last column in this figure, the separators are very close to the limit in several inferences.

In the next test we have used the network Munin to evaluate how the time complexity increases with the size of cutsets. For the worst case, adaptive conditioning takes exponential time in the cutset width. Figure 6 shows the cutset width against the maximum separator weight allowed for the variable with the biggest separator, displaying the expected exponential behavior.

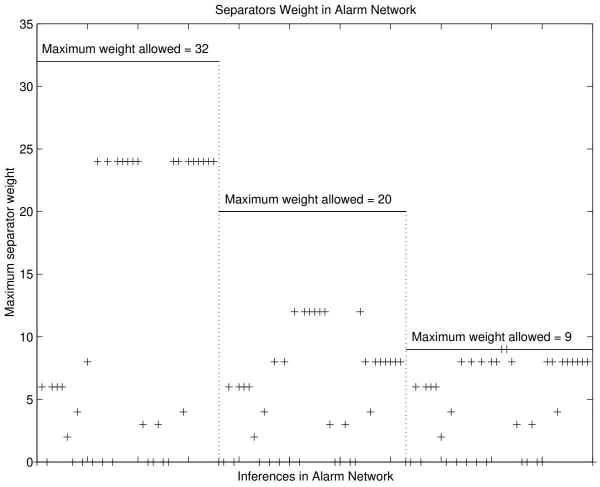

Figure 5: Results of the decomposition.

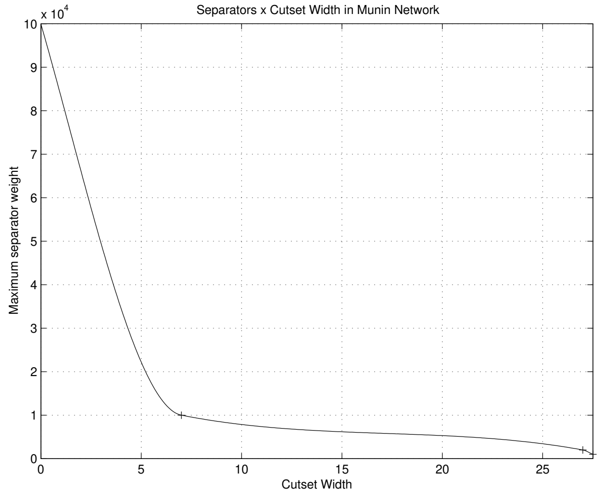

Figure 6: Time-space trade-off for exact inferences.

The car-wash maintenance system

We describe an application of embedded Bayesian networks that has employed the algorithms presented in this paper. Our application was the automatic diagnosis, for maintenance and fault detection purposes, of car-wash machines. A car-wash system is composed of three towers: the first tower showers the car with water and shampoo; the second brushes the car; the third tower dries the car. A PLC (Programable Logic Controller) is typically responsible for controlling these towers, turning electric motors and valves on and off, and at same time receiving information from sensors. The whole system contains about 20 devices, all of them exchanging information with the PLC. The system is simple when compared to a complete industrial plant, but it is difficult to conduct diagnosis on it — any device can fail and sensors are quite rough. We developed a Bayesian network for maintenance and fault detection

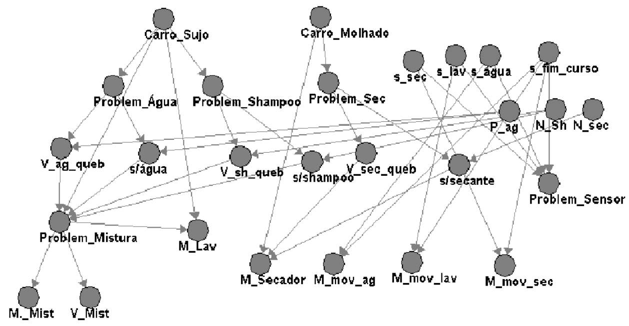

Figure 7: Network for the car-wash system.

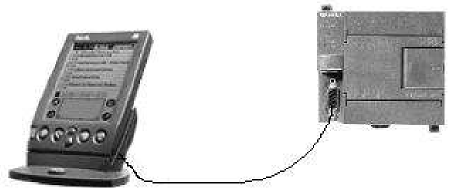

Figure 8: Palm getting information from PLC.

by analyzing the system's fail tree and studying its operating characteristics; the network contained 27 nodes and is shown in Figure 7.

Instead of storing the Bayesian network in the PLC, we decided to use a personal digital assistant (the simplest Palm IIIe) to conduct diagnosis. We built an interface between the Palm and the PLC (a model produced by Siemens AG), so that the data in the PLC could be transferred to the Palm. Figure 8 shows the Palm and PLC exchanging informations during the diagnosis process. Variables like water pressure, shampoo level or voltage applied in a motor are observed and then the inference engine indicates the most probable cause of failure. The inference system was easy to handle and made good use of minimal computing resources. Some of the inference requests demanded considerable processing time (sometimes reaching thirty seconds), but memory, the most scarce resource, was not exahusted. As industrial PLCs were not able to receive Java programs, we could not download the engine and run adaptive conditioning directly on the PLC. As PLCs become more sophisticated, we can imagine different configurations where the Palm only downloads networks for inference, or where the Palm and the PLC interact in a more intelligent and cooperative way. Adaptive conditioning could then be used directly in PLCs, where memory resources are minimal and can change during operation.

Conclusion

Adaptive conditioning is a combination of many ideas, with a design that is driven by a specific model of how probabilistic reasoning should be used in embedded systems. Adaptive conditioning is based on the desire to guarantee maximum flexibility in the specification of hard space and time constraints, while resorting to anytime behavior when needed. The distinguishing characteristic of the algorithm is the fact that various inference algorithms are distributed so as to satisfy various resource constraints. The algorithm can be presented as a convergence point where exact and approximate algorithms can coexist, while different types of time/space trade-offs can be formulated. For example, the algorithm suggests a simple way to combine Monte Carlo algorithms with exact inference — an idea that has been implemented somewhat differently previously, by applying Monte Carlo algorithms to clusters of variables (Kjaerulff 1995).

Some ideas used by adaptive conditioning are related to parallelization of Bayesian network inference, where decomposition into sub-networks is important (Kozlov & Singh 1996; Pennock 1998). Adaptive conditioning can be easily parallelized; in particular, it can be applied to embedded systems acting as a community and able to share resources.

The pursuit of embedded probabilistic reasoning certainly opens some interesting directions for research. Algorithms such as adaptive conditioning can be seen as a first step in letting automated Bayesian reasoning become prevalent in our surrounding environment. One of the most effective ways to allow devices, big and small, to be endowed with probabilistic reasoning would be to create hardware circuits that can perform Bayesian inference. In particular, approximate inference could be greatly facilitated by the introduction of random-number generation circuits. Such are the ideas that we intend to pursue in the future.

Acknowledgments

We thank Marsha Duro from HP Labs, Edson Nery from HP Brasil, and the Instituto de Pesquisas Eldorado for partially funding the research. The second author was partially funded by a grant from CNPq, and the third author was funded by a grant from FAPESP.

References

Cannings, C.; Thompson, E. A.; and Skolnick, M. H. 1978. Probability functions in complex pedigrees. *Advances in Applied Probability* 10:26–61.

Cozman, F. G. 2000. Generalizing variable elimination in Bayesian networks. In *Workshop on Probabilistic Reasoning in Artificial Intelligence*, 27–32. São Paulo: Tec Art.

Darviche, A., and Provan, G. 1996. Query dagss: A practical paradigm for implementing belief-network inference. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 203–210. San Francisco, California, United States: Morgan Kaufmann.

Darwiche, A. 2001. Recursive conditioning. *Artificial Intelligence* 126(1-2):5–41.

Dechter, R. 1996a. Bucket elimination: A unifying framework for probabilistic inference. In *XII Uncertainty in Artificial Intelligence Conference*, 211–219. San Francisco, California, United States: Morgan Kaufmann.

Dechter, R. 1996b. Topological parameters for time-space tradeoff. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 220–227. San Francisco, California, United States: Morgan Kaufmann.

Kjaerulff, U. 1995. Combining exact inference and gibbs sampling in junction trees. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. San Francisco, California, United States: Morgan Kaufmann.

Kozlov, A. V., and Singh, J. P. 1996. Parallel implementations of probabilistic inference. 29(12):33–40.

Melançon, G.; Dutour, I.; and Bousquet-Melou, M. 2001. Random generation of directed acyclic graphs. *Electronic Notes in Dicrete Mathematics, Euroconference on Combinatorics, Graph Theory and Applications special issue.*

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, California: Morgan Kaufmann.

Pennock, D. M. 1998. Logarithmic time parallel bayesian inference. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, 431–438. Morgan Kaufmann.

Weiss, Y., and Freeman, W. T. 1999. Correctness of belief propagation in gaussian graphical models of arbitrary topology. Tech Report at UC Berkeley TR UCB//CSD-99-1046.

Wen, W. X. 1990. Optimal decomposition of belief networks. In *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence*, 245–256. Morgan Kaufmann.