Variable elimination for influence diagrams with super-value nodes

Manuel Luque and Francisco Javier Díez
Departamento de Inteligencia Artificial. UNED
28040, Madrid, Spain

Abstract

In the original formulation of influence diagrams, each model contained exacly one utility node. Tatman and Shachter (1990) introduced the possibility of having super-value nodes that represent the sum or the product of their parents' utility functions. However the algorithm they proposed for dealing with super-value nodes has two shortcomings: it requires dividing potentials when reversing arcs, and it tends to introduce unnecessary variables in the resulting policies. In this paper we propose a new algorithm for influence diagrams with super-value nodes that avoids these shortcomings and will be in general much more efficient than their arc-reversal algorithm.

1 INTRODUCTION

In the original proposal by Howard and Matheson (1984), each influence diagram (ID) had only one utility node, whose parents were necessarily random nodes or decision nodes. Later, Tatman and Shachter (1990) proposed the inclusion of super-value nodes, which are utility nodes whose parents are utility nodes, and adapted the arc-reversal algorithm (Olmsted, 1983; Shachter, 1986) to cope with super-value nodes of type sum and product. Other algorithms, which evaluate an ID by recursively eliminating its variables (Shenoy, 1992; Jensen et al., 1994), are in general more efficient than arc reversal because they do not need to divide potentials. They permit that the ID to contain several utility nodes (the global utility will be the sum of all of them) but do not admit explicit super-value nodes. All these algorithms try to keep the separability of the utility function as long as possible during the evaluation of the ID, not only for the sake of efficiency, but also to avoid the introduction of redundant variables in the resulting policies. However, all of them may introduce redundant variables, and for this reason some authors have proposed other algorithms that analyze the graph in order to detect those actually required (Faguiouli and Zaffalon, 1998; Shachter, 1998; Nielsen and Jensen, 1999; Nilsson and Lauritzen, 2000; Vomlelova and Jensen, 2002).

In this paper we will try to join the advantages of all the previous algorithms in a new one that (1) does not require the reversal of arcs, (2) admits super-value nodes, and (3) keeps the policy domains as small as possible without the need of auxiliary algorithms for eliminating redundant variables. The process consists in transforming the utility function before eliminating each variable, in order to keep its separability as long as possible.

The remainder of this paper is structured as follows. In Section 1.1 we introduce some basic definitions. In Section 2 we present a new algorithm by explaining how to eliminate chance variables (Sec. 2.1) and decision variables (Sec. 2.2). We discuss related work and future research lines in Section 3, and conclude in Section 4.

1.1 DEFINITIONS

An ID with super-value nodes is an acyclic directed graph that consists of three disjoint sets of nodes: decision nodes $\mathbf{V}_D$, chance nodes $\mathbf{V}_C$, and utility nodes $\mathbf{V}_U$. Given that each node represents a variable, we will use indifferently the terms variable and node. Chance nodes/variables represent events that are not under the direct control of the decision maker. The decision nodes correspond to actions under the direct control of the decision maker. We suppose that there is a total ordering among the decisions, which indicates the order in which the decisions are made.

We differentiate two types of utility nodes: ordinary, whose parents are decision and/or chance nodes, and super-value, whose parents are utility nodes, and may in turn by of two types, sum and product. We assume that there is a utility node U_0 having no children.

An arc from decision D_i to decision D_j means that D_i is made before D_j. We assume that there is a total ordering of the decisions. An arc from a chance node X_i to a decision node D_j means that the value of variable X_i is known when the decision is made. We assume the non-forgetting hypothesis, which means that a variable X_i known for a decision D_j is also known for any posterior decision D_k, even if there is not an explicit link $X_i \rightarrow D_k$. A chance or decision node without descendants is said to be barren.

The quantitative information that defines an ID is given by assigning to each random node X_i a probability distribution $p(X_i|pa(X_i))$ for each configuration of its parents, $pa(X_i)$, and assigning to each ordinary utility node U_j a function $\psi_j(pa(U_j))$ that maps the configurations of its parents onto the real numbers. The utility associated to a super-value node of type sum/product is the sum/product of the utility functions of its parents (Tatman and Shachter, 1990).

The *matrix* of an ID ψ is defined by

$$\psi = \left(\prod_i p(X_i|pa(X_i)) \right) \psi_0 \qquad (1)$$

The total ordering of the decisions $\{D_1, \ldots, D_n\}$ induces a partition of the chance variables $\{\mathbf{C}_0, \mathbf{C}_1, ..., \mathbf{C}_n\}$, where $\mathbf{C}_i$ is the set of variables known for D_i and unknown for D_{i+1}.

The *maximum expected utility* of an ID whose chance and decision variables are all discrete is defined by

$$MEU(\Delta^*) = \sum_{\mathbf{c}_0} \max_{d_1} \ldots \sum_{\mathbf{c}_{n-1}} \max_{d_n} \sum_{\mathbf{c}_n} \psi \qquad (2)$$

An *optimal policy* δ_{D_i} is a function that maps each configuration of the variables at the left of D_i in the above expression onto the value d_i of D_i (more exactly, *one* of the values of D_i) that maximize(s) the expression at the right of D_i:

$$\begin{aligned} &\delta_{D_i}(\mathbf{c}_0, d_1, \ldots, d_{i-1}, \mathbf{c}_{i-1}) \\ &\quad = \arg\max_{d_i \in D_i} \sum_{\mathbf{c}_i} \max_{d_{i+1}} \ldots \sum_{\mathbf{c}_{n-1}} \max_{d_n} \sum_{\mathbf{c}_n} \psi \end{aligned} \qquad (3)$$

However, in many cases δ_{D_i} does not depend on some of the variables in $\{\mathbf{C}_0, D_1, \ldots, D_{i-1}, \mathbf{C}_{i-1}\}$, which are then called *redundant variables*. When a variable is redundant as a consequence of the form of the graph, it is said to be *structurally redundant*.

For instance, for the graph given in Figure 1,

$$\begin{aligned} MEU(\Delta^*) = \sum_B \max_D \sum_A P(a) \cdot P(b) \cdot \\ (U_1(a) + (U_2(a,d) + U_3(b))) \end{aligned}$$

In principle, the domain of the policy δ_D is $\{B\}$, but we will later see that, as a consequence of the separability of the utility function, $dom(\delta_D) = \emptyset$, i.e., variable B is structurally redundant for the decision D.

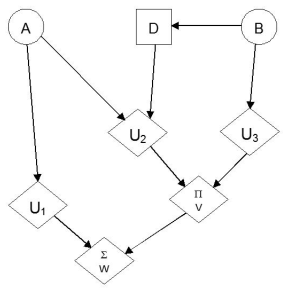

Figure 1: Graph of a small influence diagram containing two super-value nodes.

The evaluation of an ID consists in finding its *MEU* and a policy for each variable. The computational complexity of performing the summation on $\mathbf{C}_i$ in Equation 2 grows exponentially with the number of variables in $\mathbf{C}_i$. Therefore, it is in general more efficient to sum out its variables one by one. The cost of this recursive elimination depends on the form of the functions that define the matrix ψ (see Eq. 1) and on the order in which the variables are eliminated. The determination of the optimal elimination sequence is a NP-complete problem that we will not address in this paper. Our work focuses on how to eliminate a variable, either by summation or by maximization, with a double goal: to eliminate it efficiently and to preserve the separability of the matrix as much as possible.

2 VARIABLE-ELIMINATION ON A TREE OF POTENTIALS

The basic idea of our algorithm consists in representing the matrix of an influence diagram as a tree of potentials (ToP), in which terminal nodes represent probability potentials ϕ_i or utility potentials ψ_j, and non-terminal nodes represent either the sum or the product of the potentials represented by their children.

The construction of the ToP proceeds as follows. The root will always be a non-terminal node of type product. Each probability potential of the ID is added a child of the root. Then, we examine the bottom node of the ID, U_0. If it is an ordinary utility node or a super-value node of type sum, it is also added as a child of the root. On the other hand, if U_0 is a super-value node of type product, its parents in the ID are added as children of the root in the ToP. All the other utility nodes in the ID must be added analogously, so that the ToP reproduces the tree of utility nodes in the ID, although upside-down, together with the probability potentials. This way the ToP represents the matrix of the ID.

The ToP for the ID in Figure 1, which represents the potential $P(a) \cdot P(b) \cdot [U_1(a) + U_2(a,d) \cdot U_3(b)]$, will consist of a product node with three children, a sum node with two children, and a product node with two children.

A super-value node in an ID is *redundant* if it is of the same type (either sum or product) as its child. A non-terminal node in the ToP is *redundant* if it is of the same type as its parent. Therefore the ToP will be free of redundancies if and only if the ID was so. However a redundant node in a ToP can be removed by transferring its children to its parent.

In the context of trees of potentials, we will sometimes use indifferently the terms node and potential.

We describe in the next two subsections how to eliminate chance and decision variables from a ToP by applying the sum and max operators, respectively. We will assume that the ToP does not contain redundant nodes.

2.1 ELIMINATION OF A CHANCE VARIABLE

The elimination of a chance variable A consists in applying the operator $\sum_A$ to the ToP. We divide this process in two phases: we first unfork the ToP, and then eliminate A in the leaves of the new ToP. The following definitions will help us to explain the algorithm.

Definition 1 *A node of type product n is* forked with respect to (wrt) a variable A *if A belongs to the domain of some of its descendant leaves.*

Definition 2 *A ToP is* forked wrt A *if at least one of its (product) nodes is forked wrt $\dot{A}$. Otherwise, it is* non-forked.

2.1.1 Algorithm for eliminating forked nodes

Each node in a ToP may be implemented as an object having a boolean property, *forkedTree*, which is initialized to true for non-terminal nodes (which means that the tree rooted at this node may be forked wrt A) and to false for terminal nodes.

The class ToPNode implements a method, *unfork*, which takes variable A as a parameter and returns a boolean value. The purpose of this method is to unfork the node n_i receiving the message and all its descendants. The method returns true if the potential ψ_i depends on A; otherwise, it returns false.

If n_i is a leaf node, it is already unforked, and the method can immediately return true or false. If n_i is a non-terminal node, it sends the message *unfork* to all its children in order to unfork its subtrees and to know how many of its children depend on A. Then n_1 *compacts* its leaves, i.e., multiplies together all its children that are terminal and dependant on A, and replaces them by their product . If no children of n_1 depend on A, then the property *forkedTree* is set to false and the method returns false. If n_i is a sum node or if exactly one child depends on A, then *forkedTree* is set to false and the method returns true. It n_i is a product node and two or more children depend on A, then n_i is forked and must be unforked by iteratively distributing some of its potentials, as follows.

Let n_1 and n_2 be two of the children of n_i depending on A. Given that the tree has no redundant nodes, n_1 and n_2 must be either terminal or sum nodes, and since the leaves of n_i have been compacted, at least one of the two —say n_1— must be of type sum. Then n_2 will be distributed wrt the summands of n_1.

Let us assume that n_1 has j terminal children and $k-j$ non-terminal children, as shown in Figure 2. Each terminal child $n_{1,l}$ will be replaced by a product node having $n'_{1,l}$ (see Fig. 3). If the potential $\psi_{1,l}$ depends on A, we will mark the new non-terminal node $n'_{1,l}$ as *unforkedTree=false*[1], and $n'_{1,l}$ will receive the message *unfork*. Analogously, each non-terminal child $n_{1,l}$ —which must be of type product, because the tree has no redundant nodes— will add n_2 to its children, as shown in Figure 3. Again, if the potential $\psi_{1,l}$ depended on A before adding n_2, then $n_{1,l}$ must be marked as *unforkedTree=false* and receive again the message *unfork*. Then, we must mark n_1 as *unforkedTree=false* so that n_1 will receive the message *unfork*.

As a result of the distribution, the number of children of n depending on A has decreased by one. If n is still forked, it will be necessary

[1]The purpose of the boolean property *forkedTree*, whose purpose is to avoid the examination of subtrees already unforked, must be set to true for $n_{1,l}$ so that the *fork* method can process them again. However, the subtrees of $n_{1,l}$, which are already marked as *forkedTree* will not be processed again, unless it is required by a subsequent distribution of n_2.

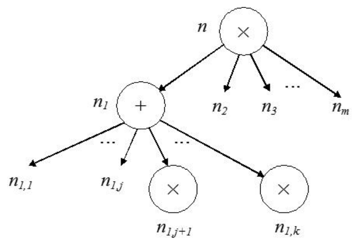

Figure 2: A tree of potentials (ToP) We assume that both n_1 and n_2 depend on the chance variable to be eliminated, A.

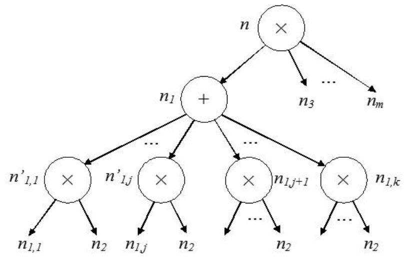

Figure 3: A ToP equivalent to the previous one, in which n_2 has been distributed with respect to n_1.

to distribute other of the child nodes that depend on A —say n_3— until n becomes unforked. Then the property *forkedTree* is set to false and the method returns true (because n depends on A).

The algorithm for the method *unfork* may be summarized as follows:

Algorithm 3 (unfork)

if forkedTree=true
  send the message unfork *to the children of* n_i
  if n_i *is of type product then*
    compact *its leaves;*
    while n_i *is forked* {
      distribute n_2 *wrt the sum node* n_1*;*
      send again the message unfork *to* n_1*;*
      }*;*
  forkedTree:=false*;*
if ψ_i *depends on A then*
  return true
else
  return false;

It is clear that both the compaction of the leaves and the distribution of a potential preserve the value of the potential, because

$$\psi_2 \times \sum_{l=1}^{k} \psi_{1,l} = \sum_{l=1}^{k} \psi_2 \times \psi_{1,l} \qquad (4)$$

Then, in order to guarantee the correctness of the method, it suffices to prove that the algorithm terminates. We prove it by induction on the number of summands that would result in the expansion of the tree.

Definition 4 *The number of summands of the expansion of a ToP rooted at node n, denoted by $s(n)$, is defined recursively as follows. If n is a terminal node, then $s(n) = 1$. If n has m children, $n_1, \ldots, n_m$, and n is of type sum, then $s(n) = \sum_{i=1}^{m} s(n_i)$; if n is of type product, $s(n) = \prod_{i=1}^{m} s(n_i)$.*

Lemma 5 *Given the distribution operation explained above (see Figures 2 and 3), $s(n'_{1,l}) < s(n_i)$.*

Proof. We have that $s(n_i) = s(n_1) \cdot \ldots \cdot s(n_m)$, which implies that $s(n_i) \geq s(n_1)$. Given that n_1 has more than one child and $s(n_1) = \sum_{l=1}^{k} s(n_{1,l})$, then $s(n_1) > s(n_{1,l})$ for all l, $s(n_1) > 1$, and $s(n_i) > s(n_2)$.

If $n_{1,l}$ was a leaf node, then $s(n'_{1,l}) = s(n_2)$ and $s(n'_{1,l}) < s(n_i)$. If $n_{1,l}$ was a non-terminal node then $s(n'_{1,l}) = s(n_{1,l}) \cdot s(n_2) < s(n_1) \cdot s(n_2) \leq s(n_1) \cdot \ldots \cdot s(n_m) = s(n_i)$, which proves the lemma. ■

Theorem 6 *For every ToP, the algorithm* fork *terminates in a finite number of steps.*

Proof. We prove it by induction on the number of summands of the root of the tree, $s(r)$, taking into account that the number of children of every node is finite.

If $s(r) = 1$ then the tree has only one terminal node or one product node having a finite number of leaves, and clearly the algorithm terminates.

Let us now assume that the theorem holds for all the trees such that $s(r) \leq n$ and let us examine a tree such that $s(r) = n + 1 \geq 2$. If r is of type sum, then each subtree of r has at most n summands (because r has at least two children), and therefore the *fork* method terminates for each child of r and for r itself. If r is of type product, then at least one of the children of r, say n_i, must be of type sum (otherwise $s(r)$ would be 1). Therefore, the number of summands for the other children of r is at most n, which means that *fork* terminates. Similarly, the number of summands of each child of n_i is at most n, which means that the algorithm terminates for each child of n_i and for n_i. When all the children of r have responded to the *fork* message, it may happen that two them, n_1 and n_2, depend on A. It is then necessary to distribute one of them, say n_2, wrt the other, as shown in Figures 2 and 3, and to send again the message *fork* to n_1. Since the lemma above states that $s(n'_{1,l})$ is at most n, the *fork* method terminates for the children of n_1 and, consequently, for n_1 itself. If n_i has still other children that depend on A, they must also be distributed wrt n_1, but the process terminates for each node, and given that the number of children of n_i is finite, the whole process terminates. ■

In the above example, whose potential was $P(a) \cdot P(b) \cdot [U_1(a) + U_2(a, d) \cdot U_3(b)]$, after distributing $P(a)$ with respect to the sum node and compacting the leaves, the new potential will be $P(b) \cdot [U'_1(a) + U'_2(a, d) \cdot U_3(b)]$, where $U'_1(a) = P(a) \cdot U_1(a)$ and $U'_2(a, d) = P(a) \cdot U_2(a, d)$.

2.1.2 Elimination of a chance variable from a non-forked tree

When the tree is not forked, the process of eliminating a chance variable A can be understood as "transferring" the $\sum_A$ operator from the root of the ToP to the leaves that depend on A, according with the following theorem.

Theorem 7 *Let t be a ToP non-forked wrt A representing the potential ψ. The potential $\sum_A \psi$ is equivalent to the potential represented by the ToP t' obtained by replacing in t each terminal node ψ_i depending on A in its domain with the potential $\sum_A \psi_i$.*

Proof. We prove the theorem by induction on the depth of the ToP, d. When $d = 1$, the tree has only one node, and the potential of the tree is the same as that of the node. If ψ depends on A, the theorem holds trivially. If ψ

does not depend on A, then $\sum_A \psi = \psi$, and no substitution is necessary. ■

Theorem 8 ***Proof.*** *Let us assume that the theorem holds for any tree whose depth is not greater than h and that there is a tree t of depth $d+1$, whose root r is necessarily an operator node having m children, $t_1, ..., t_m$, such that each tree t_i represents a potential ψ_i.*

If r is a sum node, the potential represented by the tree t is the sum of the ψ_i's:

$$\psi = \psi_1 + ... + \psi_m \tag{5}$$

Therefore,

$$\sum_A \psi = \sum_A \psi_1 + ... + \sum_A \psi_m \tag{6}$$

and, according with the induction hypothesis, each potential $\sum_A \psi_i$ can be obtained by summing out A on the terminal nodes that depend of A.

If r is a product node, at most one of its children will depend on A. If none of them depends on A, then $\sum_A \psi = \psi$ and the theorem holds. If one potential, say ψ_j, depends on A, then

$$\sum_A \psi = \sum_A \prod_{i=1}^{m} \psi_i = \left(\prod_{i \neq j} \psi_i \right) \sum_A \psi_j \tag{7}$$

Since the depth of t_j is d, the theorem holds because of the induction hypothesis. ■

In the above example, whose unforked tree represented the potential $P(b) \cdot [U'_1(a) + U'_2(a,d) \cdot U_3(b)]$, $U'_1(a)$ must be replaced with the constant $u_1 = \sum_a U'_1(a)$, and $U'_2(a,d)$ with $U_2(d) = \sum_a U_2(a,d)$.

2.2 ELIMINATION OF A DECISION VARIABLE

The elimination of a decision variable D from a potential ψ that does not depend on D is trivial, because $\max_D \psi = \psi$. The elimination from a terminal potential is immediate. The elimination of D from a potential ψ represented by a ToP whose root is of type sum and only one of its children ψ_j depends on D can be simplified to its elimination from ψ_i because

$$\max_D \psi = \max_D \sum_i \psi_i = \max_D \psi_j + \sum_{i \neq j} \psi_i \tag{8}$$

However, when there are more potentials, say $\{\psi_i\}_{i \in J}$, that depend on D, we can only apply that

$$\max_D \psi = \max_D \left(\sum_{j \in J} \psi_j \right) + \sum_{i \notin J} \psi_i \tag{9}$$

If a potential ψ is given by the product of several potentials, the equation

$$\max_D \psi = \max_D \prod_{i=1}^{m} \psi_i = \left(\prod_{i \neq j} \psi_i \right) \max_D \psi_j \tag{10}$$

can be applied only if all the ψ_i's other than ψ_j are non-negative and independent of D. In the rest of this section we will assume that all the potentials that make part of a product are non-negative in order to be able to apply the above equation.[2]

Then, the elimination of a decision variable D is algorithmically more simple —although computationally more expensive— than the elimination of a chance node: when a node at a ToP has more than one children that depend on D, all these children must be reduced into a unique terminal node before eliminating D. We will reduce first the lower nodes by performing a depth first search. The resulting tree will contain just one terminal potential depending on D, say ψ_D. The elimination of D just amounts to replacing ψ_D in the potential with a new potential

$$\psi'_D = \max_D \psi_D \tag{11}$$

which does not depend on D. The optimal policy for decision D is

$$\delta_D = \arg\max_{d \in D} \psi_D \tag{12}$$

Given a decision D_i, the domain of δ_{D_i} will then be $dom(\psi_{D_i}) \backslash \{D_i\}$, which is a subset of the variables in $\{\mathbf{C}_0, D_1, \ldots, D_{i-1}, \mathbf{C}_{i-1}\}$, because the rest of the variables have been eliminated before D_i. In practice, $dom(\delta_{D_i})$ will be a

[2] We believe that it is a reasonable assumption, because in our experience in building influence diagrams for medical applications we have often encountered negative utilities, but never as multiplicative factors of other utilities. In any case, the algorithm should check it before applying Equation 10.

proper subset of such variables, because the application of Equations 8 to 10 prevents that the variables that do not belong to the ψ_j's make part of the domain of ψ_D.

In the above example, after eliminating A the potential represented by the ToP is $U_1 + U_2(d) \cdot U_3(b)$. The maximization of this potential leads to $u_1 + u_2 \cdot U_3(b)$, where $u_2 = \max_d U(d)$. The optimal policy is $\delta_D() = \arg\max_{d \in D} U(d)$, and its domain is empty, as mentioned above. This way, our algorithm has not included the structurally redundant variable B, without needing to analyze the graph of the ID with an auxiliary algorithm.

However, it is possible that the distribution of a potential n_2 during the elimination of a chance variable A may duplicate a potential depending on A, say ψ_i, on different branches of the tree. This potential may be multiplied by the potentials that depend on D, thus adding the variables in ψ_i —other than A, which has already been eliminated— to the domain of ψ_D, even if ψ_i were a common factor that could have been taken out by applying Equation 10. An issue that remains to be analyzed is whether this hypothetical situation may actually occur, and if so, how to detect the common factors, in order to guarantee that our algorithm does not include structurally redundant variables in the returned policies.

3 RELATED WORK AND FUTURE RESEARCH

The algorithm that we have presented in the previous sections preserves the separability of the utility function in many situations in which other algorithms would join several potentials. For instance, in the above example, the algorithm by Tatman and Shachter (1990) would join U_1, U_2, and U_3 into a single potential before eliminating A. This has two shortcomings. The first one is the burden of operating with bigger potentials. The second one is that after eliminating A, B is still a parent of D, and consequently the policy δ_D returned by this algorithm would depend on B, even though this variable is structurally redundant. An additional shortcoming of the algorithm by Tatman and Shachter is the need to divide potentials when reversing an arc. For this reason variable-elimination algorithms are in general more efficient than arc-reversal (Bielza and Shenoy, 1999).[3]

However, variable-elimination algorithms developed up to date (Shenoy, 1992; Jensen et al., 1994; Jensen, 2001) were not able to deal with IDs having a structure of super-value nodes such as the one in our example. The algorithms for detecting structural redundancies (Faguiouli and Zaffalon, 1998; Shachter, 1998; Nielsen and Jensen, 1999; Nilsson and Lauritzen, 2000; Vomlelova and Jensen, 2002) have the same shortcoming, so they cannot help the Tatman-Shachter algorithm to remove redundant variables.

Even for some problems that could be solved by variable-elimination, standard algorithms will include redundant variables—see for instance the example in (Jensen, 2001, Figure 7.4)

An open question is: given an ID without product super-value nodes, is our algorithm more efficient than previous ones? We claim that in general it is, because the separability of the utility function, which our algorithm tries to keep as long as possible, leads to smaller potentials. However, the elimation of the next variable may join together some potentials that our algorithm has tried to keep separated, thus making some distributions of potentials unnecessary and counterproductive. It is necessary to carry out experiments in order to empirically compare the efficiency of the available algorithms.

As a consequence, another open issue is to develop criteria, at least of heuristic nature, for deciding if it is worthy in a certain situation

[3]If the purpose of the evaluation of an ID is just to obtain the global utility and the optimal policy for each decision (Eqs. 2 and 3) then variable-elimination algorithms do not need to divide potentials. However, if we are interested in knowing as well the utility corresponding to each option of a decision, then variable-elimination algorithms must differentiate probability potentials from utility potentials and normalize (wrt A, the chance variable to be eliminated) the probability potential that will be multiplied by the utility potential—see (Jensen, 2001) for the details.

to distribute potentials or to combine them, depending on the variables that will be eliminated afterwards.

Clearly, it is also very important to develop heuristics for finding close-to-optimal elimination orderings, given the impact that this ordering usually has on the efficiency of the algorithm. However, it is a difficult problem, given that the optimal ordering not only depends on the domains of the ordinary utility nodes, but also on how they are combined by the super-value nodes, taking into account that from the point of view of variable elimination a sum node behaves in a different way from a product node, and the elimination of a chance variable is very different from the elimination of a decision variable.

Finally, we have to study the issue mentioned in the last paragraph of Section 2.2, namely whether our algorithm may include structurally redundant variables in the policies, and if so, how to fix it in order to avoid this problem.

4 CONCLUSION

We have presented a new variable-elimination algorithm for evaluating influence diagrams with super-value nodes, which could not be evaluated with previous variable-elimination algorithms. Another advantage of our algorithm with respect to both variable-elimination methods and to arc reversal algorithms is that —at least in general— it does not include structurally redundant variables. An issue that must be studied is whether our algorithm may actually return policies with redundant variables, and if so, how to fix this shortcoming.

References

C. Bielza and P. P. Shenoy. 1999. A comparison of graphical techniques for asymmetric decision problems. *Management Science*, 45(11):1552–1569.

E. Faguiouli and M. Zaffalon. 1998. A note about redundancy in influence diagrams. *International Journal of Aproximate Reasoning*, 19(3-4):231–246.

R. A. Howard and J. E. Matheson. 1984. Influence diagrams. In R. A. Howard and J. E. Matheson, editors, *Readings on the Principles and Applications of Decision Analysis*, pages 719–762. Strategic Decisions Group, Menlo Park, CA.

F. Jensen, F. V. Jensen, and S. L. Dittmer. 1994. From influence diagrams to junction trees. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence (UAI'94)*, pages 367–373, San Francisco, CA. Morgan Kaufmann Publishers.

F. V. Jensen. 2001. *Bayesian Networks and Decision Graphs*. Springer-Verlag, New York.

T. D. Nielsen and F. V. Jensen. 1999. Welldefined decision scenarios. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 502–511, San Francisco, CA. Morgan Kaufmann Publishers.

D. Nilsson and S. Lauritzen. 2000. Evaluating influence diagrams using limids. In *Proceedings of the 16th Annual Conference on Uncertainty in Artificial Intelligence (UAI'00)*, pages 436–445, San Francisco, CA. Morgan Kaufmann Publishers.

S. M. Olmsted. 1983. *On Representing and Solving Decision Problems*. Ph.D. thesis, Dept. Engineering-Economic Systems, Stanford University, CA.

R. D. Shachter. 1986. Evaluating influence diagrams. *Operations Research*, 34:871–882.

R. D. Shachter. 1998. Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI'98)*, pages 480–487, San Francisco, CA. Morgan Kaufmann Publishers.

P. P. Shenoy. 1992. Valuation based systems for bayesian decision analysis. *Operations Research*, 40(3):463–484.

J. A. Tatman and R. D. Shachter. 1990. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(2):365–379.

M. Vomlelova and F. V. Jensen. 2002. An extension of lazy evaluation for influence diagrams avoiding redundant variables in the potentials. In *Proceedings of the First European Conference on Probabilistic Graphical Models*, pages 186–193. J. A. Gamez and A. Salmeron (eds.).