Variable elimination for influence diagrams with super-value nodes

Manuel Luque and Francisco Javier Díez
Dept. Inteligencia Artificial. UNED
Juan del Rosal, 16. 28040 Madrid. Spain
{mluque,fjdiez}@dia.uned.es

Abstract

In the original formulation of influence diagrams (IDs), each model contained exactly one utility node. Tatman and Shachter (1990), introduced the possibility of having super-value nodes that represent a combination of their parents' utility functions. They also proposed an arc reversal algorithm for IDs with super-value nodes, which has two shortcomings: it requires dividing potentials when reversing arcs, and it tends to introduce redundant (i.e., unnecessary) variables in the resulting policies. In this paper we propose a variable-elimination algorithm for influence diagrams with super-value nodes that in general introduces fewer redundant variables, is faster, requires less memory, may simplify sensitivity analysis, and can speed-up inference in IDs containing canonical models, such as the noisy OR.

Contents

1 Introduction

1.1 Influence diagrams

An influence diagram (ID) [10] is a probabilistic graphical model for decision analysis, having three kinds of nodes: chance, decision, and utility—see Section 1.2 for a formal definition. The goal of evaluating an ID is to obtain the expected utility and an optimal strategy, which consists of a policy for each decision. The first algorithm for evaluating IDs proceeded by expanding and evaluating an equivalent decision tree [10]. Later, Olmsted [22] proposed the arc reversal (AR) algorithm, which evaluates the ID recursively by eliminating its nodes and inverting arcs when necessary—see also [25].

In the original proposal [10], each influence diagram (ID) had only one utility node. A node like this, whose parents are chance nodes or decision nodes, is called nowadays an *ordinary utility node*, in contrast with *super-value nodes* (SVNs), whose parents are other utility nodes; a SVN represents a utility that is a combination of the utilities of their parents. SVNs were introduced in 1990 by Tatman and Shachter [29], who also extended the AR algorithm to cope with SVNs of type sum and product.

In the next decade, several variable-elimination algorithms were proposed for IDs [3, 4, 11, 27], which are in general more efficient than AR because they do not need to divide potentials. They permit that the ID contains several ordinary utility nodes, under the assumption that the global utility is the sum of all of them, but none of those algorithms can deal with SVNs.

Our interest on SVNs arose during the construction of a decision-support system for the mediastinal staging of non-small cell lung cancer [17], whose utilities combine additively and multiplicatively, as shown in Figure 1. In order to evaluate this ID, we wished to have an algorithm for IDs with SVNs, such that:[1]

1. is faster than AR,
2. requires less memory,
3. avoids redundant variables,
4. simplifies sensitivity analysis, and
5. can be integrated with state-of-the-art algorithms for inference in IDs containing canonical models.

[1] In fact, the fourth objective was not set at the beginning of our study, but emerged as a possibility during the design of the algorithm.

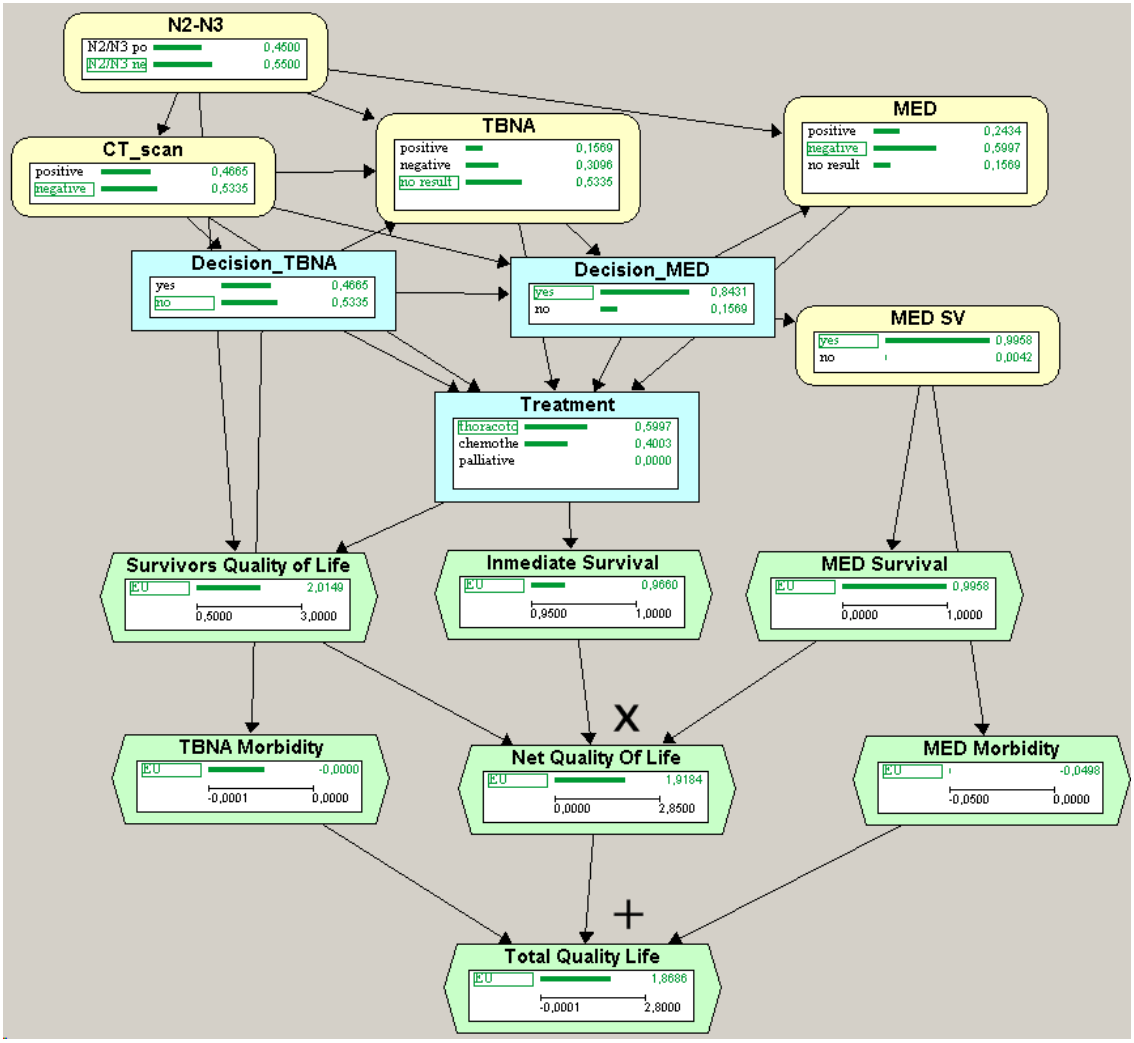

Figure 1: Decision-support system for the mediastinal staging of non-small lung cancer.

The first two objectives are obvious. We conjectured that a variable elimination algorithm for IDs with SVNs might fulfill them because, unlike AR, it does not need to divide potentials and the number of potentials stored in the working memory is smaller.

The second objective refers to redundant variables, i.e., those whose value is known when making a decision but do not affect the optimal policy—see Section 1.3. As the complexity of a policy grows exponentially with the number of variables in its domain, it is desirable to remove as many redundant variables as possible, not only for reducing the storage space but, more importantly, for communicating the policy to a human being. In fact, the explanation of reasoning is a crucial issue for building and deploying decision-support systems because it helps to debug the model and to convince the user that the results are correct, and also for educational purposes [13, 14, 15]. Policies containing redundant variables are more difficult to understand and to debug. Even worst, the inclusion of structurally-redundant variables (cf. Sec 1.3), i.e., those that can not affect the policy due to the nature of the causal relations involved in the problem, undermines the user's confidence in the policies recommended by the expert system.

The fourth objective refers to sensitivity analysis, which consists in studying how the expected utility and the optimal strategy vary as a consequence of changes in the model [2, 20]. Parametric

sensitivity analysis in IDs is performed by assigning a range of variation or a probability distribution to (some of) the utilities and conditional probabilities that define the ID, or by finding out the thresholds that determine a change of the optimal policies. However, in some cases the structure of the graph that defines an ID implies that the values of a certain potential (namely, a conditional probability table or a utility function) do not affect the expected utility or the optimal policies of some decisions. In that case, it is not necessary to analyze such parameters, thus saving computations and simplifying the report of the results. Section 2.2.1 shows an example of this.

Finally, the fifth objective has to do with canonical models, which are probabilistic relations defined by some constrains inspired on causal properties [5]. They are called "canonical" because they can be used as elementary blocks that combine to build up more sophisticated probabilistic models [23]. In particular, the relation between a node and its parents in a Bayesian network (or a chance node and its parents in an ID), which in the case of discrete variables takes the form of a conditional probability table (CPT), can sometimes be represented by a particular canonical model, while other CPTs in the same network might be based on different models. The canonical models that appear more often in practice are the noisy OR and its extension, the noisy MAX. Canonical models do not only simplify the process of building CPTs, but may also lead to drastic computational savings in both memory and time. For instance, CPCS [24] is a large medical Bayesian network that for many years resisted to exact inference algorithms, because all of them ran out of memory when trying to compute some marginal queries—see the references in [28]. Even most of the approximate algorithms converged very slowly when the evidence introduced was very unlikely [1]. However, in 1999 Takikawa and D'Ambrosio [28] proposed a new factorization of the noisy MAX that was able to do exact inference on that network very efficiently: the more complex query took only 0.29 seconds; the factorization by Díez and Galán [6] further reduced that time to 0.05 seconds. Those factorizations can be integrated with both variable elimination and clustering algorithms, but not with arc reversal. That was an additional reason for developing a variable elimination algorithm for IDs with SVNs, in order to obtain similar savings to those of Bayesian networks.

Our algorithm is an extension of variable-elimination algorithms for IDs [3, 4, 11, 12, 27]; in fact, when the ID has no SVN, it perfoms essentially the same operations as them. The main difference is that it represents the utility function of the ID in the form of a tree, and in a refined version of the algorithm, in the form of an acyclic directed graph (ADG). The algorithm usually transforms that tree or ADG before eliminating each variable, trying to preserve its separability as long as possible, in order to reduce the space complexity and to avoid redundant variables in the policies.

The remainder of this paper is structured as follows. Section 1.2 presents the basic definitions for IDs and Section 1.3 analyzes the problem of redundant variables. Section 2 presents a new algorithm for eliminating chance variables (Sec. 2.1) and decision variables (Sec. 2.2) from a tree of potentials (ToP) (and also exposes a variable-elimination algorithm for IDs with SV nodes on a ToP). Section 3 improves the previous algorithm by using an acyclic directed graph of potentials (ADGoP) instead of a ToP. Section 4 proposes three variations of that algorithm that in some cases may lead to more efficient computations. Section 5 describes the empirical evaluation of different versions of our algorithm, comparing them with arc-reversal. We discuss related work and future research lines in Section 6, and conclude in Section 7.

1.2 Basic definitions

An ID is a probabilistic graphical model that consists of three disjoint sets of nodes: decision nodes $\mathbf{V}_D$, chance nodes $\mathbf{V}_C$, and utility nodes $\mathbf{V}_U$. Chance nodes represent events that are not under the direct control of the decision maker. Decision nodes correspond to actions under the direct control of the decision maker. Given that each chance or decision node represents a variable, we will use indifferently the terms variable and node. IDs assume that there is a total ordering of the decisions, which indicates the order in which the decisions are made.

We distinguish two types of utility nodes: *ordinary*, whose parents are decision and/or chance nodes, and *super-value*, whose parents are utility nodes. We assume that there is a utility node

U_0 that is a descendant of all the other utility nodes, and therefore has no children.[2]

The meaning of an arc in an ID depends on the type of nodes that it links. An arc from a decision D_i to a decision D_j means that D_i is made before D_j. An arc from a chance node C to a decision node D_j means that the value of variable C is known when making decision D_j. We assume the *non-forgetting hypothesis*, which means that a variable C known for a decision D_j is also known for any posterior decision D_k, even if there is not an explicit link $C \rightarrow D_k$ in the graph.

A *potential* is a real-valued function over a domain of finite variables. The quantitative information that defines an ID is given by (1) assigning to each random node C a conditional probability potential $p(C|pa(C))$ for each configuration of its parents, $pa(C)$[3], (2) assigning to each ordinary utility node U a potential $\psi_U(pa(U))$ that maps each configuration of its parents onto a real number, and (3) assigning a utility-combination function to each super-value node. The domain of each function U is given by its *functional predecessors*, $FPred(U)$; thus, the functional precedessors of an ordinary utility node are its parents, $FPred(U) = Pa(U)$, and the functional predecessors of a super-value node are all the functional predecessors of its parents: $FPred(U) = \bigcup'_U \in Pa(U)FPred(U')$. The algorithms described in this paper assume that all the super-value nodes in the ID are either of type sum or product.[4]

The *matrix* of an ID ψ, is defined by

$$\psi(\mathbf{V}_C, \mathbf{V}_D) = \left(\prod_{C \in \mathbf{V}_C} P(C|pa(C)) \right) \psi_{U_0}(FPred(U_0)) \; . \tag{1}$$

The total ordering of the decisions $\{D_1, \ldots, D_n\}$ induces a partition of the chance variables $\{\mathbf{C}_0, \mathbf{C}_1, \ldots, \mathbf{C}_n\}$, where $\mathbf{C}_i$ is the set of variables unknown for D_i and known for D_{i+1}. The set of variables known to the decision maker when deciding on D_i is called the *informational predecessors* of D_i and denoted by $IPred(D_i)$. Consequently, $IPred(D_i) = \mathbf{C}_0 \cup \{D_0\} \cup \mathbf{C}_1 \cup \ldots \cup \{D_{i-1}\} \cup \mathbf{C}_i = IPred(D_{i-1}) \cup \{D_{i-1}\} \cup \mathbf{C}_i$.

The *maximum expected utility* (MEU) of an ID whose chance and decision variables are all discrete is

$$MEU = \sum_{\mathbf{c}_0} \max_{d_1} \sum_{\mathbf{c}_1} \ldots \sum_{\mathbf{c}_{n-1}} \max_{d_n} \sum_{\mathbf{c}_n} \psi(\mathbf{v}_C, \mathbf{v}_D) \; . \tag{2}$$

An *optimal policy* δ_{D_i} is a function that maps each configuration of the variables in $IPred(D_{i-1})$, i.e., those at the left of D_i in the above expression, onto the value d_i of D_i that maximizes the expression at the right of D_i (in the case of a tie, any of the values of D_i that maximize that expression can be chosen arbitrarily):

$$\delta_{D_i}(IPred(D_i)) = \arg\max_{d_i \in D_i} \sum_{\mathbf{c}_i} \max_{d_{i+1}} \ldots \sum_{\mathbf{c}_{n-1}} \max_{d_n} \sum_{\mathbf{c}_n} \psi(\mathbf{v}_C, \mathbf{v}_D) \; . \tag{3}$$

For instance, for the graph given in Figure 2,

$$MEU = \sum_b \max_d \sum_a P(a) \cdot P(b) \cdot [U_1(a) + (U_2(a,d) * U_3(b))] \tag{4}$$

and

$$\delta_D(b) = \arg\max_{d \in D} \sum_a P(a) \cdot P(b) \cdot [U_1(a) + (U_2(a,d) * U_3(b))] \tag{5}$$

[2]Clearly, an ID having only one utility node satisfies this condition by identifying such a node with U_0. An ID having several utility nodes assumes that the global utility is their sum, and can be modified to fulfill that condition by adding a new node U_0, of type sum, whose parents are the original utility nodes. Therefore, this assumption does not restrict the types of IDs that our algorithm can solve.

[3]$Pa(X)$ is the set of parents of X, and $pa(X)$ is a configuration of X.

[4]A super-value node U_i representing a combination function other than the sum or the product can be transformed into an ordinary utility node as follows: if U_j is a parent of U_i, we remove U_j from the ID and add its parents as new parents of U_i, and proceed recursively until no parent of U_i is a utility node. The new utility function for U_i derives from the original utility function of U_i and from those of its utility ancestors in the original ID. This transformation is necessary for both our algorithm and arc reversal [29].

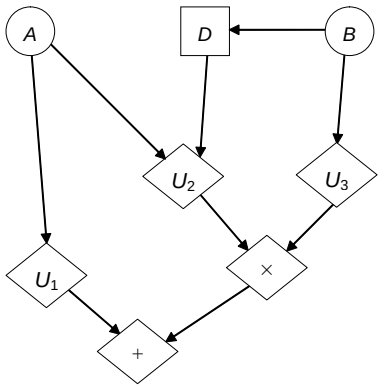

Figure 2: Graph of a small ID containing two super-value nodes: one of them is of type product, and the other of type sum.

1.3 Redundant variables

According with Equation 3, in principle, the domain of a policy consists of all the variables whose value is known when making that decision: $dom(\delta_{D_i}) = IPred(D_i)$. However, in some cases the policy δ_{D_i} does not depend on a particular variable X of $IPred(D_i)$; we then say that X is redundant. The formal definition is as follows.

Definition 1 *Let D be a decision variable in an ID and X an informational predecessor of D: $X \in IPred(D)$. Variable X is said to be* redundant *for D if and only if*

$$\forall x, \forall x', \forall \mathbf{y}, \;\; \delta_D(x, \mathbf{y}) = \delta_D(x', \mathbf{y})$$

where x and x' are values of X and $\mathbf{y}$ is a configuration of the other informational predecessors of D: $\mathbf{Y} = IPred(D) \setminus \{X\}$.

Shachter [26] distinguished two types of redundant variables, under the names of "irrelevant" and "probabilistically irrelevant". Following partially the terminology of Faguiouli and Zaffalon [8], we prefer to use the terms "structurally redundant" and "numerically redundant", which are defined as follows.

Definition 2 *A redundant variable for decision D in an ID I is* structurally redundant *if and only if it is redundant for all the IDs having the same graph as I. Otherwise, it is* numerically redundant.

Therefore, the structural redundancy only depends on the graph of the ID, while numerical redundancy depends on the assignment of probability and utility potentials. Several algorithms have been proposed in the literature for detecting structurally redundant variables by analyzing the graph [8, 19, 21, 26, 31], but none of them can cope with SVNs. In a future paper we will propose a new algorithm that solves this problem, but in our opinion this is a not crucial issue, as the variable elimination algorithm that we describe in this paper rarely includes structurally redundant variables—see the experiments in Section 5. However, we might always apply the redundancy-detection algorithm if redundant variables were a relevant problem in our domain of application.

An additional advantage of our algorithm, related to this topic, is that it usually avoids the introduction of quasi-structurally redundant variables, which we define as follows:

Definition 3 *An ordinary utility node in an ID is* monotonic *if its utility function contains only non-positive or non-negative values.*

Definition 4 *A variable X in an ID is* quasi-structurally redundant *for a decision D with respect to a subset of utility nodes if the monotonicity of all those nodes implies that X is redundant for D.*

Please note that quasi-structural redundancy is related with numerical redundancy, because it depends on the values of some parameters in the ID, but on the other hand, such variable will be redundant in all the IDs having the same graph and satisfying that condition, which implies that quasi-structural redundancy is a property of the graph, not of a particular ID—hence the name "quasi-structural".

For instance, for the graph given in Figure 2 the optimal policy for decision D, given by Equation 5, depends on B. That equation can be rewritten as

$$\delta_D(b) = \arg\max_{d \in D} P(b) \cdot [u'_1 + U'_2(d) * U_3(b)] \tag{6}$$

where $u'_1 = \sum_a P(a) \cdot U_1(a)$ and $U'_2(d) = \sum_a P(a) \cdot U_2(a, d)$. Given that $P(b)$ is always non-negative and u'_1 is a constant,

$$\delta_D(b) = \arg\max_{d \in D} U'_2(d) * U_3(b) \tag{7}$$

If the utility node U_3 is monotonic, then its values are either all non-negative or all non-positive. In the former case,

$$\forall b,\ U_3(b) \geq 0 \implies \forall b,\ \max_{d \in D} U'_2(d) * U_3(b) = U_3(b) * \max_{d \in D} U'_2(d) \tag{8}$$

$$\implies \forall b,\ \delta_D(b) = \arg\max_{d \in D} U'_2(d) \tag{9}$$

and in the latter

$$\forall b,\ U_3(b) \leq 0 \implies \forall b,\ \max_{d \in D} U'_2(d) * U_3(b) = U_3(b) * \min_{d \in D} U'_2(d) \tag{10}$$

$$\implies \forall b,\ \delta_D(b) = \arg\max_{d \in D} -U'_2(d) \tag{11}$$

In both cases $\delta_D(b)$ is independent of B, i.e., B is quasi-structurally redundant for decision D with respect to the subset of utility nodes $\{U_3\}$.

2 Variable-elimination on a tree of potentials

The basic idea of our algorithm consists in representing the matrix of an influence diagram, defined in Equation 1, as a tree of potentials (ToP), whose leaves (also called *terminal nodes*) represent probability potentials ϕ_i or utility potentials ψ_j, and each non-terminal node indicates either the sum or the product of the potentials represented by its children.

For instance, Figure 3 shows the ToP for the ID in Figure 2, whose matrix is $P(a) \cdot P(b) \cdot [U_1(a) + U_2(a, d) \cdot U_3(b)]$.

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 as a child of the root. If the bottom node of the ID, U_0, 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 in the same way. As a result, the ToP represents the matrix of the ID, i.e., the tree of utility nodes in the ID, although upside-down, together with the probability potentials.

A non-terminal node in a ToP is said to be *duplicated* if it is of the same type as its parent. A duplicated node in a ToP can be removed by transferring its children to its parent.

We describe in the next two subsections how the elimination of chance and decision variables in an ID is handled by applying the sum and max operators, respectively, to the ToP. We will assume that the ToP does not contain duplicated nodes.

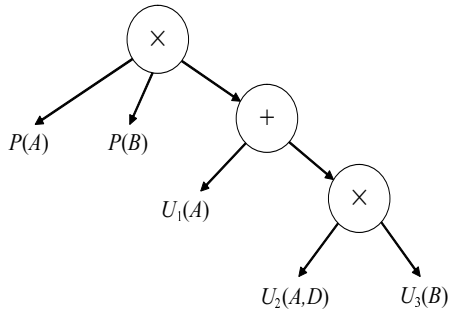

Figure 3: Tree of potentials (ToP) for the ID of Figure 2.

2.1 Elimination of a chance variable on a ToP

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 5 *A variable X* appears *in a ToP t if it belongs to its domain, i.e., if it belongs to the domain of some of the terminal nodes of t.*

Definition 6 *A node n of type product* is forked with respect to (wrt) a variable A *if A appears in more than one of the branches of n.*

Definition 7 *A ToP* is forked wrt A *if at least one of its product nodes is forked wrt A. Otherwise, it is* non-forked.

For example, variable A appears in the ToP of Figure 3. The root node is forked wrt A because A appears in two of its three branches. Consequently, the ToP in that figure is forked wrt variable A. In contrast, the subtree rooted at the sum node is not forked wrt A because it only contains one product node, which is not forked.

2.1.1 Algorithm for eliminating forked nodes

Using an object-oriented programming representation, each node in a ToP may be implemented as an object of class *ToP-node* having three properties:

- *dependsOnVariable,* a Boolean value that indicates whether the node depends on the variable A to be eliminated;
- *dependentChildren,* a list of the children of the node that depend on variable A;
- *mayBeForked*, a Boolean value used by the method *unfork* to avoid visiting each subtree several times; it is initialized to `true` for all nodes and is also set to `true` when the method *unfork* has to visit the same subtree again for a different variable.

The class *ToP-node* has a main method, *unfork*, which uses two auxiliary methods: *distribute* and *compact*. The method *distribute* transforms the tree in Figure 4.a, in which both siblings n_1 and n_2 depend on A, into the tree in Figure 4.b. Nodes n_1 and n_2 are children of a product node forked wrt A. The procedure *distribute* is described by Algorithm 1—see also Figure 4.

Under the conditions of the method *distribute*, illustrated in Figure 4, it is clear that the potential obtained after distributing n_2 is equivalent to the original potential, because

$$\psi_2 \times \sum_{l=1}^{k} \psi_{1,l} = \sum_{l=1}^{k} \psi_2 \times \psi_{1,l} \ . \tag{12}$$

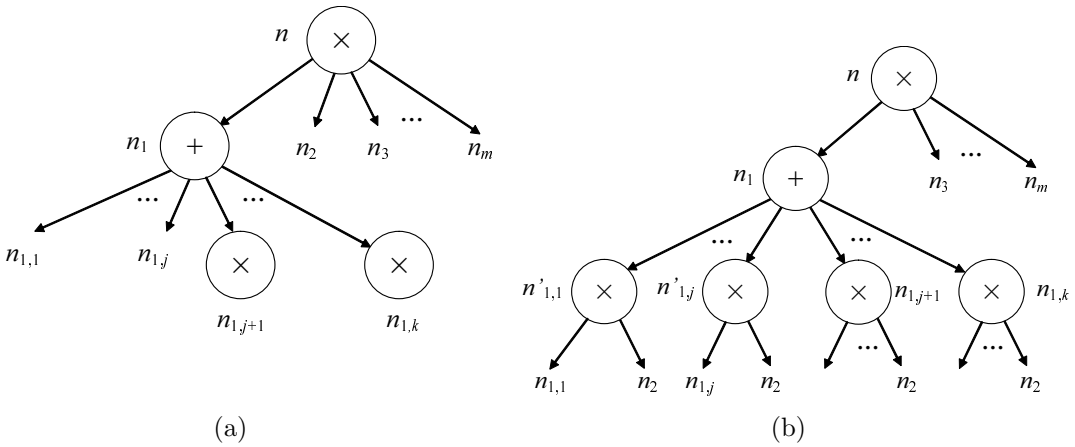

Figure 4: (a) A ToP, where we assume that both n_1 and n_2 depend on the chance variable to be eliminated, A. (b) A ToP equivalent to (a), in which n_2 has been distributed with respect to n_1.

For instance, in the example in Figure 3, 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, the new potential will be $P(b)\cdot[P(a)\cdot U_1(a)+P(a)\cdot U_2(a,d)\cdot U_3(b)]$ (see Figure 5.a).

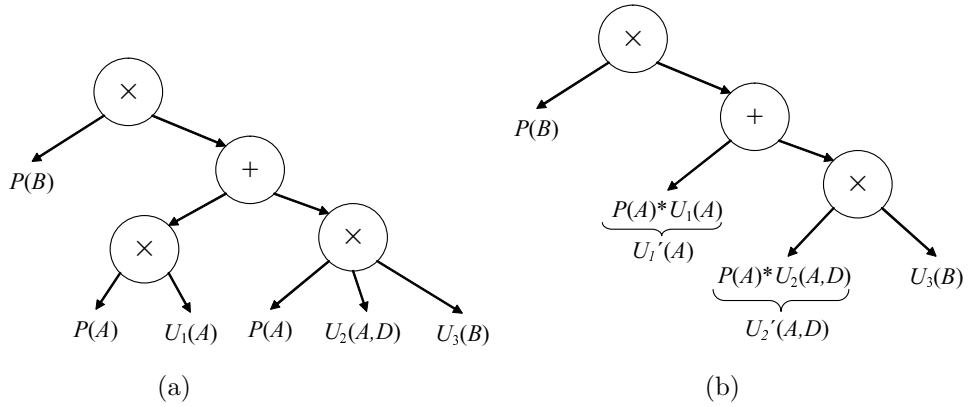

Figure 5: (a) ToP equivalent to that in Figure 2.b, in which $P(A)$ has been distributed with respect to the sum node. (b) ToP equivalent to (a), in which the leaves dependent on A have been compacted and replaced by two new potentials, $U_1'(A)$ and $U_2'(A,D)$.

Please note that the product node that represents $P(a)\cdot U_1(a)$ in Figure 5.a is forked, but since $P(a)$ and $U_1(a)$ are terminal nodes, it can be unforked by multiplying its children. This process is performed by the method *compact*, shown as Algorithm 2.

Figure 5.b shows the result of applying the method *compact* to the product nodes of the ToP of Figure 5.a, when A is the variable to be eliminated.

Finally, the method *unfork*, invoked as $n.unfork(A)$, transforms the subtree under node n into a new subtree representing an equivalent potential, but unforked wrt A. The method *unfork* is described by Algorithm 3.

Please note that the while loop in this algorithm executes only when at least two children of n depend on A, and at least one of them—which we have called n_1—is of type sum, because after

Algorithm 1 distribute (for a ToP)

Input: n_1: a sum node depending on A, child of a product node n forked wrt A;
n_2: another node depending on A, sibling of n_1;
A: a chance variable.
Effects: the subtrees under n_1 are multiplied by n_2.

```
1: for all n_{1,i} ∈ children(n_1) do
2:   if n_{1,i} is a terminal node then
3:     remove n_{1,i} as a child of n_1;
4:     add a new product node n'_{1,i} as a child of n_1;
5:     add both n_{1,i} and n_2 as children of n'_{1,i};
6:     n'_{1,i}.mayBeForked := true;
7:   else
8:     // n_{1,i} is a product node
9:     add n_2 as a child of n_{1,i};
10:    n_{1,i}.mayBeForked := true;
11:  end if
12: end for
13: n_1.mayBeForked := true;
```

Algorithm 2 Compact

Input: n: a product node, whose children depending on A are all terminal nodes.
Effects: the children of n that depend on A are replaced by their product.

```
1: remove from n.dependentChildren and n.children all the leaves that depend on A
   and remove them as children of n;
2: add the product of all to n.children and to n.dependentChildren;
3: if n has only one child (say n_1) then
4:   replace n with n_1 in the tree;
5: end if
```

executing *compact*(n) node n cannot have two leaf children depending on A.

Theorem 8 *For every ToP, the algorithm* unfork *terminates in a finite number of steps, returning a non-forked ToP.*

The proof can be found in Appendix A.1.

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

When a tree is non-forked, the process of eliminating a chance variable A can be understood as "transferring" the $\sum_A$ operator from the root of the ToP down to the leaves that depend on A, according with the following theorem.

Theorem 9 *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 with the potential $\sum_A \psi_i$.*

The proof can be found in Appendix A.2.

Coming back to the example in Figure 2, the potential $P(b) \cdot [U'_1(a) + U'_2(a,d) \cdot U_3(b)]$ was represented by the tree in Figure 5.b, which is non-forked wrt A. The elimination of chance variable A is performed by replacing $U'_1(a)$ with the constant $u'_1 = \sum_a U'_1(a)$, and replacing $U'_2(a,d)$ with $U'_2(d) = \sum_a U'_2(a,d)$. The result is $\psi = P(b) \cdot [u'_1 + U'_2(d) \cdot U_3(b)]$.

Algorithm 3 Unfork

Input: n (object receiving the message): a node in the ToP;
A: a chance variable.
Effects: n is unforked wrt A, its attribute *mayBeForked* is set to `false` and the attribute *dependsOnVariable* indicates whether n depends on A.

if $n.mayBeForked =$ `true` **then**
  if n is a terminal node **then**
    if the potential of n depends on A **then**
      $n.dependsOnVariable :=$ `true`;
    else
      $n.dependsOnVariable :=$ `false`;
    end if
  else
    // n is a non-terminal node
    for all $n_i \in children(n)$ **do**
      $n_i.unfork(A)$;
    end for
    $dependentChildren :=$ children n_i of n such that $n_i.dependsOnVariable =$ `true`;
    if $(size(dependentChildren) > 0)$ **then**
      $n.dependsOnVariable :=$ `true`;
    else
      $n.dependsOnVariable :=$ `false`;
    end if
    if n is of type product **then**
      $compact(n)$;
      while $(size(dependentChildren) > 1)$ **do**
        $n_1 :=$ a sum node in *dependentChildren*;
        $n_2 :=$ other node in *dependentChildren*;
        $distribute(n_1, n_2)$;
        $n_1.unfork(A)$;
      end while
    end if
  end if
  $n_1.mayBeForked :=$ `false`;
end if

2.2 Elimination of a decision variable on a ToP

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 that depends on D is also immediate. Let us assume that ψ is represented by a ToP, whose root node r is not terminal, and ψ_i is the potential represented by the i-th child of r.

We analyze first the case in which r is of type sum. If more than one of the ψ_i's depend on D, it is not correct to eliminate D by replacing each ψ_i with $\max_D \psi_i$, because $\max_D(\psi_i + \psi_{i'})$ may be different from $\max_D \psi_i + \max_D \psi_{i'}$—please note the contrast with Theorem 9. The right procedure when r is of type sum is to add all the potentials that depend on D before eliminating D; the rest of the potentials are not modified. Formally, if J is the set of subindices such that ψ_j does not depend on D, and K contains the other subindices, then:

$$\max_D \psi = \max_D \sum_i \psi_i = \sum_{j \in J} \psi_j + \max_D \underbrace{\sum_{k \in K} \psi_k}_{\psi_D} . \tag{13}$$

In case that r is of type product, we define J as the set of subindices such that ψ_j is monotonic

and does not depend on D, and K as its complementary. We also define m as the number of indices in J such that ψ_j has at least one negative value (the other values of ψ_j must be either negative or null, because ψ_j is monotonic). If m is even, then $\prod_{j\in J}\psi_j$ is non-negative, and consequently,

$$\max_D \psi = \max_D \prod_i \psi_i = \prod_{j\in J}\psi_j \cdot \max_D \underbrace{\prod_{k\in K}\psi_k}_{\psi_D} . \tag{14}$$

If it is odd, then $-\prod_{j\in J}\psi_j$ is non-negative and

$$\max_D \psi = (-1)\cdot \prod_{j\in J}\psi_j \cdot \max_D \underbrace{-\prod_{k\in K}\psi_k}_{\psi_D} . \tag{15}$$

This analysis leads to Algorithm 4. The method for changing the sign of a subtree rooted at n, used in step 23, is given by Algorithm 5. Please note that when only one of the children of n depends on D, namely $n_{k'}$, then the algorithm is invoked recursively on $n_{k'}$, preserving the structure of that subtree (steps 11 and 25). On the contrary, when more than one children depend on D, those branches collapse into a potential ψ_D (steps 13 and 14), which after maximizing on D, is reinserted as a terminal node (step 36).

For example, we have already shown that, for the ID in Figure 2 (see also Fig. 5), after eliminating A the matrix is $\psi = P(b)\cdot[u'_1 + U'_2(d)\cdot U_3(b)]$. When eliminating D, we have $\max_d \psi = P(b)\cdot[u'_1 + \max_d(U'_2(d)\cdot U_3(b))]$. In general,

$$\delta_D(b) = \arg\max_{d\in D} \left(U'_2(d)\cdot U_3(b)\right) . \tag{16}$$

However, if U_3 is non-negative, then $\max_d \psi = P(b)\cdot[u'_1 + u'_2\cdot U_3(b)]$, where $u'_2 = \max_d U'_2(d)$, and the optimal policy is

$$\delta_D = \arg\max_{d\in D} \; U'_2(d) , \tag{17}$$

which does not depend on B. If U_3 is non-positive, then $\max_d \psi = P(b)\cdot[u'_1 + (-1)\cdot u'_2\cdot U_3(b)]$, where $u'_2 = \max_d(-U'_2(d))$, and the optimal policy is

$$\delta_D = \arg\max_{d\in D} \; -U'_2(d) , \tag{18}$$

which does not depend on B, either. Therefore, when $U_3(b)$ is monotonic, our algorithm does not include in the domain of δ_D the quasi-structurally redundant variable B. In contrast, the arc-reversal algorithm [29] would collapse all the utility nodes into a single node when eliminating A; as the parents of the new utility node are B and D, the elimination of D will always include B in the domain of δ_D.

2.2.1 Sensitivity analysis

Another advantage of our algorithm, closely related with the attempt to avoid redundant variables, is the possibility of simplifying sensitivity analysis. For instance, in the above example the policy for D was given by Equation 16, where $U'_2(d) = \sum_a P(a)\cdot U_2(a,d)$. Consequently, if we perform a sensitivity analysis for variable D in order to determine which variations in the parameters of the ID may lead to a different policy, we only need to examine the probabilities in $P(a)$ and the utilities in $U_2(a,d)$ and $U_3(b)$, because δ_D does not depend at all on the other parameters, namely those in $P(b)$ and $U_1(a)$. This may lead to a significant simplification of sensitivity analysis.

Additional simplifications occur when $U_3(b)$ is monotonic. In this case, Equations 17 and 18 tell us that the policy only depends on the values in $P(a)$ and $U_2(a,d)$, and on the sign of the values in $U_3(b)$. This is important because usually human experts have uncertainty about the exact value of a parameter, but not about its sign.

Algorithm 4 eliminateDecision

Input: n: a node in the ToP;
D: a decision variable.
Effects: the decision variable D is eliminated from the tree rooted at n.
Output: an optimal policy δ_D for D.

if n depends on D **then**
  if n is a terminal node with associate potential ψ **then**
    replace ψ with $\max_D \psi$;
    return the optimal policy, $\delta_D := \arg\max_D \psi$;
  else
    // n is non-terminal; ψ_i is the potential represented by n_i, the i-th child of n
    if n is of type sum **then**
      $K :=$ set of subindices such that ψ_k depends on D;
      if $|K| = 1$ **then**
        // K contains only one index, k'
        return *eliminateDecision*($n_{k'}$, D);
      else
        $\psi_D := \sum_{k \in K} \psi_k$; // cf. Equation 13
      end if
    else
      // n is of type product
      $J :=$ set of subindices such that ψ_j is monotonic and does not depend on D;
      $K :=$ set of subindices complementary of J;
      $m :=$ number of subindices in J such that ψ_j has at least one negative value;
      if only one potential, $\psi_{k'}$, depends on D **then**
        if m is odd **then**
          add a node with the constant potential -1 as a child of n;
          changeSign($n_{k'}$);
        end if
        return *eliminateDecision*($n_{k'}$, D);
      else
        // several potentials depend on D
        $\psi_D := \prod_{k \in K} \psi_k$; // cf. Equation 14
        if m is odd **then**
          add a node with the constant potential -1 as a child of n;
          $\psi_D := -\psi_D$; // cf. Equation 15
        end if
      end if
    end if
    for every $k \in K$, remove the k-th child of n;
    add a node with the potential $\max_D \psi_D$ as a child of n;
    return the optimal policy $\delta_D := \arg\max_D \psi_D$;
  end if
end if

Algorithm 5 changeSign

Input: n: a node in the ToP, representing the potential ψ.
Effects: The subtree rooted at n is modified to represent the potential $-\psi$.

```
1: if n is a terminal node then
2:    replace ψ with −ψ;
3: else
4:    // n has several children, {n_i}_{i∈I}
5:    if n is of type sum then
6:       for all i ∈ I do
7:          changeSign(n_i);
8:       end for;
9:    else
10:      // n is of type product;
11:      add a node with the constant potential −1 as a child of n;
12:   end if
13: end if
```

In contrast, in this example arc reversal [29] would eliminate D by maximizing on a potential derived from all the potentials that define the ID, namely $P(a)$, $P(b)$, $U_1(a)$, $U_2(a,d)$, and $U_3(b)$, which seems to indicate that every parameter in ID might affect the policy δ_D.

In summary, our variable-elimination algorithm may simplify sensitivity analysis if we keep track (for instance, by mantaining a set of pointers) of the ID potentials that have been involved in the computation of each potential at the ToP.

2.2.2 Inclusion of redundant variables

Although the distribution of potentials in general avoids the inclusion of redundant variables in the policies, as we have seen in the previous examples (see also the experiments in Section 5.2), it may fail to avoid it in some cases. The following example explains why.

Let us assume that we are interested in computing $\max_d \sum_a \psi$, where $\psi = [\psi_1(a) + \psi_2(b)] \cdot [\psi_3(a) + \psi_4(d)]$. When eliminating A, the node that represents ψ is forked wrt A. If the algorithm takes $[\psi_1(a) + \psi_2(b)]$ as n_1 and $[\psi_3(a) + \psi_4(d)]$ as n_2, the result of the distribution is $\psi = \psi_1(a) \cdot [\psi_3(a) + \psi_4(d)] + \psi_2(b) \cdot [\psi_3(a) + \psi_4(d)]$. The first summand is represented by a product node, which is still forked. A new distribution leads to $\psi = \psi_1(a) \cdot \psi_3(a) + \psi_1(a) \cdot \psi_4(d) + \psi_2(b) \cdot [\psi_3(a) + \psi_4(d)]$ and $\sum_a \psi = \psi_{13} + \psi'_1 \cdot \psi_4(d) + \psi_2(b) \cdot [\psi'_3 + \psi_4(d)]$, where ψ_{13}, ψ'_1, and ψ'_3 are the constant potentials that result from summing out A from the terminal leaves of the previous potential. Then, the elimination of D will explicitly compute $\psi'_1 \cdot \psi_4(d) + \psi_2(b) \cdot [\psi'_3 + \psi_4(d)]$, which yields a terminal potential that depends on both B and D:

$$\max_d \sum_a \psi = \max_d \underbrace{\psi'_1 \cdot \psi_4(d) + \psi_2(b) \cdot [\psi'_3 + \psi_4(d)]}_{\psi(b,d)} .$$

Therefore the algorithm will include B in the policy δ_D.

However, the algorithm would have been able to detect that B is quasi-structurally redundant (wrt ψ'_1 and ψ_2) if it had distributed the top factors in a different way:

$$\begin{aligned}
\max_d \sum_a \psi &= \max_d \sum_a \underbrace{[\psi_1(a) + \psi_2(b)]}_{n_2} \cdot \underbrace{[\psi_3(a) + \psi_4(d)]}_{n_1} \\
&= \max_d \sum_a \{[\psi_1(a) + \psi_2(b)] \cdot \psi_3(a) + [\psi_1(a) + \psi_2(b)] \cdot \psi_4(d)\} \\
&= \max_d \{\psi_{13} + \psi_2(b) \cdot \psi'_3 + [\psi'_1 + \psi_2(b)] \cdot \psi_4(d)\} \\
&= \psi_{13} + \psi_2(b) \cdot \psi'_3 + [\psi'_1 + \psi_2(b)] \cdot \max_d \psi_4(d) .
\end{aligned}$$

As we have seen, the first distribution performed in this example failed to detect that ψ_4 is a common factor for ψ'_1 and ψ_2.

This example underlies the importance of deciding which candidates for a distribution (i.e., those factors of type sum depending on the variable to be eliminated) should be chosen as n_1 and n_2. The problem is that our algorithm performs myopically, in the sense that when eliminating a variable it does not take into account the effect that it will have in the posterior elimination of other variables. The refinement of our algorithm in order to avoid redundant variables is an open problem, as mentioned in Section 6.[5]

2.3 Summary: Variable elimination algorithm using a ToP

Finally, Algorithm 6 integrates all the steps described so far. Please note that the input of this algorithm is not only an ID, but also an elimination order for the variables in $\mathbf{V}_C \cup \mathbf{V}_D$. This order has to be a *legal elimination sequence* [19], which means that must eliminate first the variables in $\mathbf{C}_n$, then D_n, then those in $\mathbf{C}_{n-1}$, and so on (see Equation 2). However, this condition only imposes a partial ordering on the variables of $\mathbf{V}_C \cup \mathbf{V}_D$: it is still necessary to order the variables inside each $\mathbf{C}_i$. The similarity of this problem with others in Bayesian networks make us conjecture that finding an optimal elimination sequency for our algorithms is NP-complete. Finding near-optimal orderings is an open issue, that we will discuss in Section 6.

Algorithm 6 Variable elimination for IDs with SVNs on a ToP

Input: I: an ID that may contain SVNs;
O: a legal sequence elimination for the variables in $\mathbf{V}_C \cup \mathbf{V}_D$;
Output: the *MEU* of the ID and an optimal policy δ_D for each decision variable D.

```
1: construct the ToP t of I;
2: remove the duplicate nodes from t;   // see Section 2
3: for all V ∈ O do
4:    if V ∈ V_C then
5:       unfork t wrt V;   // Algorithm 3;
6:       for each terminal node n_i in t depending on V, replace that node with Σ_v ψ_i;
7:    else
8:       // V ∈ V_D
9:       eliminateDecision(r, V);   // Algorithm 4;
10:   end if
11: end for
12: MEU := numerical value of the potential at t (a constant);
```

The correctness of Algorithm 6 is ensured given that the elimination of chance and decisions variables are performed according to a legal elimination sequence, and each transformation of the ToP preserves the *MEU* and the optimal policies.

3 Variable elimination on an ADG of potentials

In Section 2, the matrix of an ID was represented as a ToP. However, the matrix can also be represented as an acyclic directed graph of potentials (ADGoP). The main advantage of this representation is twofold: the saving of space in memory when a subtree appears more than once in a tree, and the saving of computational time when distributing a potential and when eliminating a variable. Another advantage is the ability to cope with IDs in which a utility node can have two or more super-value children, as shown in Figure 6.

[5] It is noteworthy that in this example our algorithm can obtain the right domain for decision D if it selects the right distribution of potentials, while arc reversal [29] would always include the redundant variable B in the policy of D. However, the experiments in Section 5 show that in some exceptional cases arc reversal may include fewer redundant variables than ours—an issue that deserves further investigation.

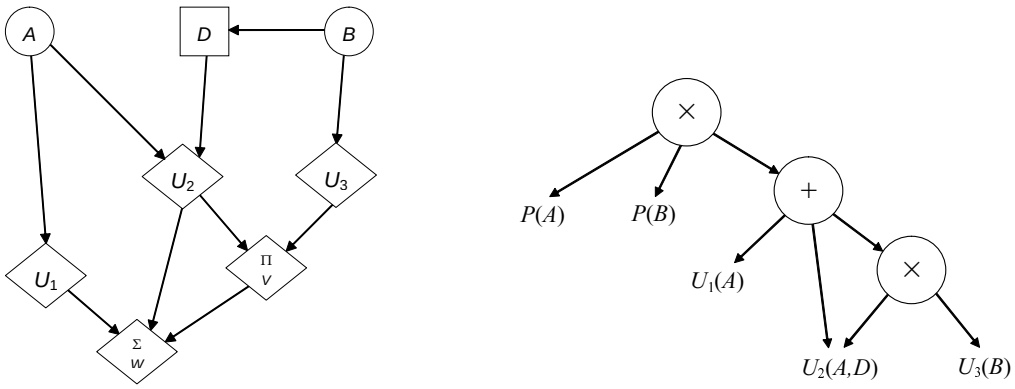

Figure 6: (a) Graph of an ID with super-value nodes, in which U_2 has two children. The global utility potential is $\psi = U_1 + U_2 + (U_2 \cdot U_3)$. (b) Acyclic directed graph of potentials (ADGoP) for this ID.

The construction of the ADGoP from an ID is very similar to that of the ToP. The next two subsections explain how to eliminate chance and decision variables from an ADGoP, under the assumption that redundant nodes have already been removed—see Section 2.

3.1 Elimination of a chance variable on an ADGoP

The elimination of a chance variable A consists in applying the operator $\sum_a$ to the ADGoP. This process is divided in two phases: unforking the ADGoP, and eliminating A from the leaves of the new ADGoP.

3.1.1 Algorithm for eliminating forked nodes

The process of unforking the ADGoP is similar to that of the ToP. This way, each node in a ADGoP may be implemented as an object of class *ADGoP-node*, which has the same properties and methods as *ToP-node* (see Sec. 2.1.1). The method *unfork* is identical, but *distribute* and *compact* are slightly different in both classes, because when a node compacts its leaves, their children having other parents can not be removed from the ADGoP.

For instance, in Figure 7.a, the node n_2 is forked wrt A. When n_2 compacts its leaves $\phi_1(A)$ and $\phi_2(A)$, the leaf $\phi_1(A)$ can not be removed from the ADGoP because it is also a child of n_3. The link $n_2 \to \phi_1(A)$ will be removed and $\phi_1(A)$ will be multiplied by $\phi_2(A)$, but link $n_3 \to \phi_1(A)$ will remain, as shown in Figure 7.b.

In turn, the method *distribute* differs in that instead of creating several copies of n_2, as we did in the case of a ToP (see Fig. 4.b), we will draw several links from the children of n_1 to n_2 (see Fig. 8).

3.1.2 Elimination of a chance variable from a non-forked ADGoP

When the ADGoP is non-forked, the process of eliminating a chance variable A is performed as in a ToP, i.e., the $\sum_a$ operator is "transferred down" from the root of the ADGoP to the leaves that depend on A. Even if a leaf has several parents, the $\sum_a$ operator is applied to it only once, thus saving time with respect to the case of a ToP.

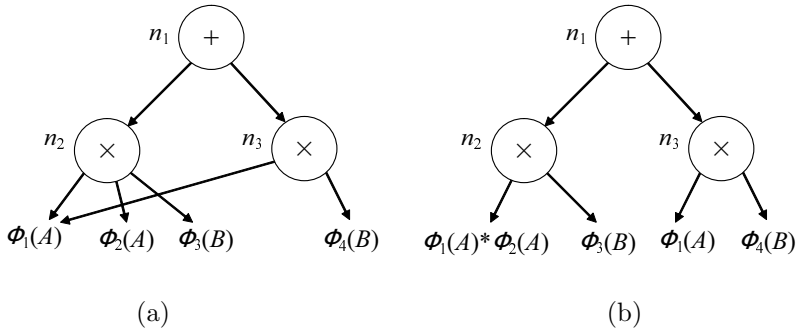

Figure 7: (a) An ADGoP, in which n_2 is forked wrt A. (b) ADGoP after the node n_2 in (a) compacts its leaves dependent on A.

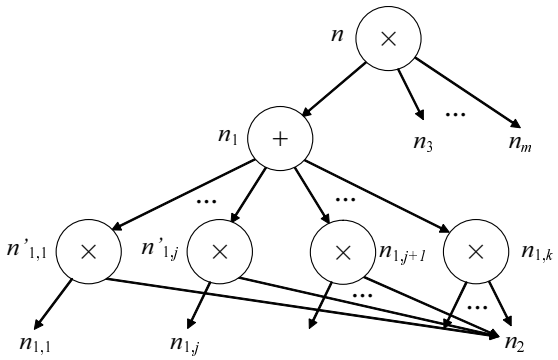

Figure 8: An ADGoP equivalent to the potential in Figure 4.a, in which n_2 has been distributed with respect to n_1.

3.2 Elimination of a decision variable on an ADGoP

The elimination of a decision variable D from an ADGoP is similar to its elimination from a ToP (cf. Algorithm 4, in Section 2.2): Algorithm 4 can also be applied when eliminating a decision variable in an ADGoP, but when a node compacts its leaves, as required by some steps of Algorithm 4, their children having other parents can not be removed from the ADGoP. Then, if a child n_1 of n has other parents, the link $n \rightarrow n_1$ must be removed, but n_1 cannot be eliminated from the graph. This is the same situation that appears when compacting the leaves of a node in an ADGoP before eliminating a chance variable.

3.3 Summary: algorithm VE

The variable-elimination algorithm on an ADGoP is performed as in a ToP (see Algorithm 6), with the only difference in step 1 we construct an ADGoP instead of a ToP, and then we perform all the operations of Algorithm 6 in the ADGoP as we have described in this section, i.e., by adapting the corresponding algorithms.

4 Variations of the algorithm

The previous section has presented the basic algorithm of variable elimination (VE) on an ADGoP. We discuss now three variations of that algorithm that may lead to more efficient computations. In Section 5 we will compare empirically these versions with the standard VE.

4.1 Division of potentials (algorithm VE-D)

The VE algorithm presented above does not distinguish between probability and utility potentials. In this respect, it is similar to some variable-elimination algorithms for IDs without SVNs [3, 4, 27]. On the contrary, the variable-elimination algorithm proposed in [12] for IDs without super-value nodes differentiates both types of potentials and, when eliminating a chance variable, normalizes the probability potentials by means of a *division*. The main advantage of this process is that the utility potentials obtained after multiplication by the normalized potentials represent the utilities associated with different scenarios, which may be useful for explaining the decision process to the user [14].

Similarly, it is possible to design a new version of our VE algorithm for ID with SVNs, called VE-D (where "D" stands for divisions), which instead of storing all the potentials in the same ADGoP, handles a *list of probability potentials* (LoPP) and an *ADG of utility potentials* (ADGoUP). Their product represents the matrix of the ID. The construction of the ADGoUP for an ID is identical to that of the ADGoP (cf. Secs. 2 and 3), with the only difference that it does not include the probability potentials.

The procedure of VE-D is described by Algorithm 7. This algorithm substitutes the variable elimination algorithm on an ADGoP (Algorithm 6 adapted for using an ADGoP instead of a ToP). The main difference between both algorithms is that Algorithm 7 keeps the probability potentials in a LoPP, separated from the utility potentials contained in the ADGoP. The LoPP and the ADGoP are modified when a variable V is eliminated as follows. The LoPP is updated by removing the probality potentials dependent of V and adding to it their marginalization. The ADGoP is prepared to proceed to the elimination of V, which is performed in the ADGoP as described in Sections 3.1 and 3.2.

In Algorithm 7, the operator project_V in step 16 only makes sense when applied to a potential that does not depend on V, i.e., a potential whose value is the same for all the configurations having the same value of V. For instance, given a potential $\phi(v_1, v_2)$ such that $\phi(+v_1, +v_2) = \phi(\neg v_1, +v_2) = 0.9$ and $\phi(+v_1, \neg v_2) = \phi(\neg v_1, \neg v_2) = 0.4$, which does not depend on V_1, the operator project_{V_1} gives a new potential $\phi'(v_2) = \text{project}_{V_1}\phi(v_1, v_2)$ such that $\phi'(+v_2) = 0.9$ and $\phi'(\neg v_2) = 0.4$.

In Appendix A.3 we prove the correctness of VE-D, including the fact that when Algorithm 7 applies the operator project_V ϕ_V, this potential does not depend on V.

Example 10 *For the ID in Figure 9, we have*

$$MEU = \sum_b \max_{d_1} \sum_c \max_{d_2} \sum_a P(a) \cdot P(b|a) \cdot P(c|a, d_1) \cdot \psi(a, d_2) \; . \tag{19}$$

When eliminating A we have $\phi_A(a, b, c, d_1) = P(a) \cdot P(b|a) \cdot P(c|a, d_1)$, $\phi^*_A(b, c, d_1) = \sum_a \phi_A(a, b, c, d_1)$, and

$$MEU = \sum_b \max_{d_1} \sum_c \max_{d_2} \; \phi^*_A(b, c, d_1) \underbrace{\sum_a \frac{\phi_A(a, b, c, d_1)}{\phi^*_A(b, c, d_1)} \cdot \psi(a, d_2)}_{\psi(b,c,d_1,d_2)} \; , \tag{20}$$

When eliminating D_2 there is no probability potential depending on D_2. Therefore, it is not necessary to perform any multiplication nor any projection:

$$MEU = \sum_b \max_{d_1} \sum_c \phi^*_A(b, c, d_1) \cdot \underbrace{\max_{d_2} \; \psi(b, c, d_1, d_2)}_{\psi(b,c,d_1)} \; . \tag{21}$$

Algorithm 7 Algorithm VE-D (variable elimination with divisions)

Input: I: an ID that may contain SVNs;
O: a legal sequence elimination for the variables in $\mathbf{V}_C \cup \mathbf{V}_D$;
Output: the *MEU* of the ID and an optimal policy δ_D for each decision variable D.

```
construct the LoPP l and the ADGoP g of I;
remove the duplicate nodes from g;
for all V ∈ O do
    remove from l all the potentials that depend on V;
    let φ_V be the product of all of them;
    if V ∈ V_C then
        φ*_V := Σ_V φ_V;
        // multiply g by φ_V/φ*_V
        if the root of g (say r) is of type product then
            add φ_V/φ*_V to the children of r;
        else
            replace r by a product node r', and add r and φ_V/φ*_V as children of r';
        end if
    else
        // V is a decision
        φ*_V := project_V φ_V;
    end if
    add φ*_V to l;
    eliminate V from g, as explained in Sections 3.1 and 3.2;
end for
MEU := numerical value of the potential at g (a constant);
```

When eliminating C only one probability potential depends on this variable, namely ϕ_A^*. Therefore, $\phi_C(b,c,d_1) = \phi_A^*(b,c,d_1)$, $\phi_C^*(b,d_1) = \sum_c \phi_C(b,c,d_1)$, and

$$MEU = \sum_b \max_{d_1} \phi_C^*(b,d_1) \underbrace{\sum_c \frac{\phi_C(b,c,d_1)}{\phi_C^*(b,d_1)} \cdot \psi(b,c,d_1)}_{\psi(b,d_1)} . \tag{22}$$

When eliminating D_1 only one probability potential depends (apparently) on this variable: $\phi_{D_1}(b,d_1) = \phi_C^*(b,d_1)$. However, Lemma 13 (cf. Appendix A.3) states that $\phi_{D_1}(b,d_1)$ does not depend on d_1. We then have $\phi_{D_1}^*(b) = \text{project}_{D_1}\ \phi_{D_1}(b,d_1)$. Then,

$$MEU = \sum_b \phi_{D_1}^*(b) \cdot \underbrace{\max_{d_1} \psi(b,d_1)}_{\psi(b)} . \tag{23}$$

Finally, when eliminating B we have $\phi_B(b) = \phi_{D_1}^*(b)$, $\phi_B^* = \sum_b \phi_B(b)$, and

$$MEU = \phi_B^* \sum_b \phi_B(b) \cdot \psi(b) . \tag{24}$$

When the algorithm VE-D eliminates a decision D_i, the ADPoUP represents a potential that depends on the informational predecessors of D_i: $\psi_i(\mathbf{c}_0, d_1, \ldots, d_{i-1}, \mathbf{c}_{i-1}, d_i)$. It is possible to show that the value of ψ_i is the utility of the scenario in which (1) the variables in $\mathbf{C}_j$ $(0 \leq j \leq i)$ take the values dictated by the configuration $\mathbf{c}_j$, (2) the decision maker chooses option d_k for each decision D_k $(1 \leq k \leq i)$ and (3) chooses the best option for the decisions after D_i.[6] This is the main reason for dividing the probability potentials: when a user

[6]The proof is similar to that offered in [12] for the variable-elimination algorithm for IDs without supervalue nodes: the proof remains valid if ψ is given by an ADGoUP instead of a sum of utility potentials.

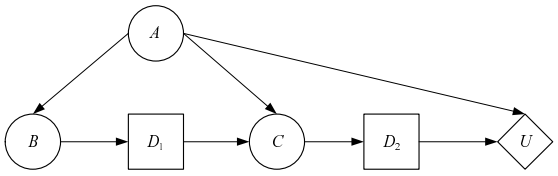

Figure 9: Influence diagram whose variable-elimination evaluation is detailed in Example 10.

of the decision-support system is often interested in knowing why the system recommends option d_i for scenario $(\mathbf{c}_0, d_1, \ldots, d_{i-1}, \mathbf{c}_{i-1})$ rather than option d'_i, it is useful to display the values $\psi_i(\mathbf{c}_0, d_1, \ldots, d_{i-1}, \mathbf{c}_{i-1}, d_i)$ and $\psi_i(\mathbf{c}_0, d_1, \ldots, d_{i-1}, \mathbf{c}_{i-1}, d'_i)$, i.e., the utilities of the two options [16]. When evaluating an ID with the arc-reversal algorithm, these values can be read directly from the utility table of the ID before eliminating D_i. However, in general VE (variable elimination without divisions) can not show these utilities, and this is the main reason for using VE-D instead of VE.

4.2 Subset rule

Tatman and Shachter [29] proposed a heuristic, called the *subset rule*, for reducing the storage space required by their arc reversal algorithm: if two utility nodes U_1 and U_2 have the same successor U, being a super-value node of type sum/product, and $Pa(U_2) \subseteq Pa(U_1)$, it is possible to replace them by a new node U', such that [1] $Pa(U') = Pa(U_1)$, [2] U' is a parent of U, and [3] $U' = U_1 + U_2$ or $U' = U_1 \times U_2$, respectively. This replacement seems to be advantageous in general because it does not increase the size of any operation necessary to solve the ID, and may simplify subsequent combinations of potentials.

The subset rule can also be introduced in the algorithms VE and VE-D: when two leaves n_1 and n_2 representing potentials ϕ_1 and ϕ_2, respectively, have the same parent in the ToP of in the AGDoUP and $dom(\phi_1) \subseteq dom(\phi_2)$, then compacting n_1 and n_2 liberates storage space and may simplify the next operations.

However, the application of the subset rule needs to check if $dom(\phi_1) \subseteq dom(\phi_2)$ for every pair of children of each potential, which has a certain computational cost, thus this rule is counterproductive in some cases. In Section 5.2.2 we analyze empirically the changes in the time and space required when the subset rule is applied.

4.3 Unity potentials

The efficiency of the algorithms VE and VE-D can be improved by avoiding certain computations. For example, when eliminating a barren node X_i,[7] the ADGoP (for VE) or the LoPP (for VE-D) contains a potential $P(X_i|pa(X_i))$. The elimination of X_i implies computing $\sum_{x_i} P(x_i|pa(x_i))$, which is 1. Similarly, if $\phi_X = P(x|pa(X))$ and X is a chance variable, the VE-D algorithm will compute $\phi'_X = \sum_x \phi_X$, which is also 1. In other cases, it will be necessary to perform the marginalization $\sum_x [\phi_X / \phi^*_X]$, which is equal to 1 even if ϕ_X was not a conditional probability, because $\phi^*_X = \sum_x \phi_X$—see Algorithm 7, Step 7.

If the algorithm recognizes these situations, it can save the computational cost of summing out X. More importantly, the algorithm will be able to replace $\sum_x P(x|pa(X))$ with 1—a constant, while the computation of $\sum_x P(x|pa(x))$ may return a potential whose domain is apparently $pa(X)$, and this may lead to including redundant variables in the policies.

An open line for future research is how to integrate in our variable elimination algorithm the recent improvements for the lazy evaluation of IDs [18, 31], whose purpose is to avoid the operations that yield unity potentials.

[7] According with [25], a chance or decision node without descendants is said to be barren.

5 Empirical evaluation

We have performed a series of experiments for assessing the efficiency of the variable elimination algorithm proposed in this paper. The first problem we faced is that we only have a few real-world examples of IDs with SV nodes, and these are not complex enough for comparing the different versions of our algorithm between themselves and with the arc reversal (AR) algorithm of Tatman and Shachter. The repositories of graphical probabilistic models available on Internet do not contain IDs with SV nodes. For this reason, we have run the experiments on randomly generated IDs.

5.1 Algorithm for generating IDs randomly

Vomlelova [30] proposed an algorithm for randomly generating IDs with several (ordinary) utility nodes. We have adapted and extended it in order to generate IDs with SV nodes. The parameters of the algorithm are: *nNodes*, the total number of chance and decision nodes; *decisionRatio*, the probability that a node is a decision (otherwise, it is a chance node); N, the number of iterations, each adding or deleting an arc; *nParents*, the maximum number of parents for a chance or decision node; *nUtil*, the number of ordinary utility nodes; and *nParentsUtil*, the number of parents per utility node. The procedure for generating an ID is described by Algorithm 8.

We must note in step 13 that the fact that i and j are not ordered (step 12) implies that there is no path from one node to the other. Therefore, drawing a link between them (step 13) can not create a loop nor a cycle.

We have assigned random non-negative values to the potentials, with the only restriction that probabilities must be normalized.

5.2 Experimental results

We executed the above algorithm with *decisionRatio* = 0.3, $N = 300$ (additions or removals of arcs), *nParents* = 3, *nUtil* = 7, and *nParentsUtil* = 7. The number of nodes, *nNodes*, varied from 5 to 24. We generated 100 influence diagrams for each number of nodes, which amounts to a total of 2,000 IDs. Figure 10 displays the graph of one of the influence diagrams generated.

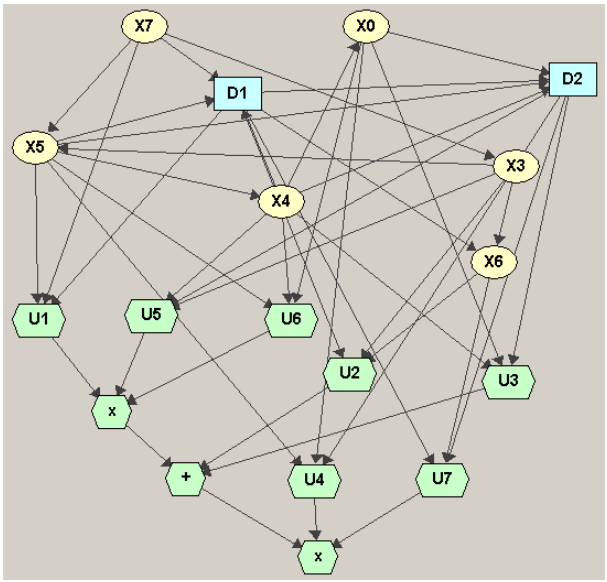

Figure 10: A random influence diagram generated by Algorithm 8, with $nNodes = 8$.

Algorithm 8 Generate a random ID

Input: *nNodes*, *decisionRatio*, *N*, *nParents*, *nUtil* and *nParentsUtil* (cf. Sec. 5.1).
Output: an influence diagram.

create a tree having *nNodes* nodes;
for each node, randomly decide whether it is a decision (with probability *decisionRatio*) or a chance node;
for $k = 1$ to N **do**
  select randomly a pair of distinct nodes i and j;
  if the arc $i \rightarrow j$ exists in the graph **then**
    if the graph remains connected, delete this arc;
  else
    if the graph remains acyclic and the number of parents of j does not exceed *nParents*, add the arc*;*
  end if
end for
while there are at least two decisions i and j such that $i \notin ancestors(j)$ and $j \notin ancestors(i)$ **do**
  select two distinct decision nodes i and j such that $i \notin ancestors(j)$ and $j \notin ancestors(i)$;
  randomly decide whether the arc $i \rightarrow j$ or $i \rightarrow j$ is added to the graph (with probability 0.5);
end while
generate *nUtil* ordinary utility nodes, each with *nParentsUtil* parents randomly selected among the decision and chance nodes;
while there are several utility nodes without descendants **do**
  if the number of utility nodes without descendants is greater than *nParentsUtil* **then**
    randomly select *nParentsUtil* utility nodes without descendants and add arcs from them to a new super-value node;
  else
    draw arcs from them to a new super-value node;
  end if
  randomly decide if the new super-value node is sum (with probability 0.5) or product;
end while
generate a probability table for each chance node;
generate a utility table for each ordinary utility node;

Each ID was evaluated with three algorithms: Tatman and Shachter's arc reversal (AR), variable-elimination without divisions (VE) and with divisions (VE-D). We only calculated the global utility of the ID and the optimal policy for each decision because VE can not compute the expected utility of each option.

All the algorithms employed the same elimination order of variables when evaluating each ID in order to compare them in the same conditions. It was determined previously by evaluating the ID qualitatively with Tatman and Shachter's algorithm.[8]

The algorithms were implemented in Java 6.0 with the Elvira software package.[9] The tests were run on an Intel Core 2 computer (2.4 GHz) with 2 GB of memory under Windows XP.

[8]The time necessary to evaluate an ID qualitatively with AR is negligible compared with the time required by a full evaluation. Therefore, the fact that VE and VE-D would need an additional amount of time to obtain the elimination order does not affect the results of our experiments—see also Section 6.

[9]The Elvira program was developed as a collaborative project of several Spanish universities [7]. The source code, a user manual, and other documents can be downloaded from `www.ia.uned.es/~elvira`.

5.2.1 Comparison of AR, VE, and VE-D (without the subset rule)

Time and space efficiency First, we have computed for each ID the ratio of the times required by AR and VE. Table 1 shows the results, grouped by the number of nodes. Given that the distribution is very skewed, we show in this table both the median and the mean, as well as some other percentiles.[10]

By observing the means (second column), we can see that on average VE is around 10 times faster that AR. The last column in this table tells us that, for one ID, AR was 339 times slower than VE—see also Figure 11.[11] The 95th percentile column shows that in around 5% of cases VE is at least 30 times faster than AR. On the contrary, the cases in which AR is faster than VE (those in which the ratio is smaller than 1) are unfrequent, as shown by the 5th percentile column.

The minimum displayed in the table means that in the most favorable case for AR, it was only 5 times faster than VE, and this difference occurred for an ID having only 6 nodes, for which the time spent by both algorithms is negligible. For bigger diagrams, AR could never be 2 times faster than VE. In contrast, Figure 11 shows that for several influence diagrams VE was 50 or even over 100 times faster than AR, with a maximum of around 340.

When comparing the storage space required by these algorithms, we observe that in general VE needs less memory than AR—see the "means" column in Table 2. In the case of large IDs, in which the limit of memory is a critical issue, AR requires on average around 3 or 4 times more space than VE, with a median ratio of almost 2. The 5th and 95th percentile columns in Table 2 also show the superiority of VE over AR in the case of large IDs: the former rarely needs twice more space than the latter (the maximum shown in the "min" column is $1/0.14 = 7$ times), while in 5% of the cases AR needs at least 10 times more space than VE (the maximum being almost 60). Correspondingly, in Figure 12 we can see that for several IDs AR needed 10, 20, and even 60 times more space than VE, which is a significant difference.

[10] The minimum, the median, and the maximum are the 0th, 50th, and 100th percentiles, respectively.

[11] In this figure we have used *boxplots*, which provide a graphical simple summary of a set of data. The top and bottom of each box represent the upper and lower quartiles of each group of data, and the line in the middle represents the median. Whiskers extend from each end of the box, and their extremes values are within 1.5 times the interquantile range from the ends of the box. Outliers, represented by red circles, are individuals whose value is higher or lower than the extremes of the whiskers.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	13.20	2.91	7.58	13.06	17.72	37.93
6	11.78	0.20	6.50	11.62	17.00	26.39
7	10.81	3.70	4.88	11.10	16.27	20.26
8	11.21	2.59	6.40	10.78	19.51	22.77
9	9.19	2.23	4.30	8.49	15.66	21.04
10	9.90	3.14	3.80	9.34	18.27	28.33
11	9.81	1.96	2.65	9.09	19.97	24.85
12	11.12	1.50	3.21	10.11	22.76	34.25
13	9.36	0.76	2.42	6.82	17.89	64.99
14	11.41	0.89	2.06	8.07	35.51	62.27
15	12.22	0.74	2.18	9.60	35.28	45.46
16	12.07	1.08	2.42	8.15	29.26	101.98
17	16.84	0.97	2.14	8.04	79.22	112.67
18	13.58	0.98	1.80	7.48	45.85	96.57
19	15.08	1.29	2.10	6.13	57.53	122.20
20	11.37	0.54	1.48	7.43	40.54	79.38
21	15.80	0.88	1.64	7.12	46.42	338.84
22	14.03	1.24	1.72	6.54	39.77	219.88
23	12.62	0.74	1.42	5.87	53.02	90.36
24	12.29	0.82	1.54	6.64	39.21	157.80
Total	**12.18**	**0.20**	**2.16**	**9.15**	**30.76**	**338.84**

Table 1: Ratio of the times required by AR and VE.

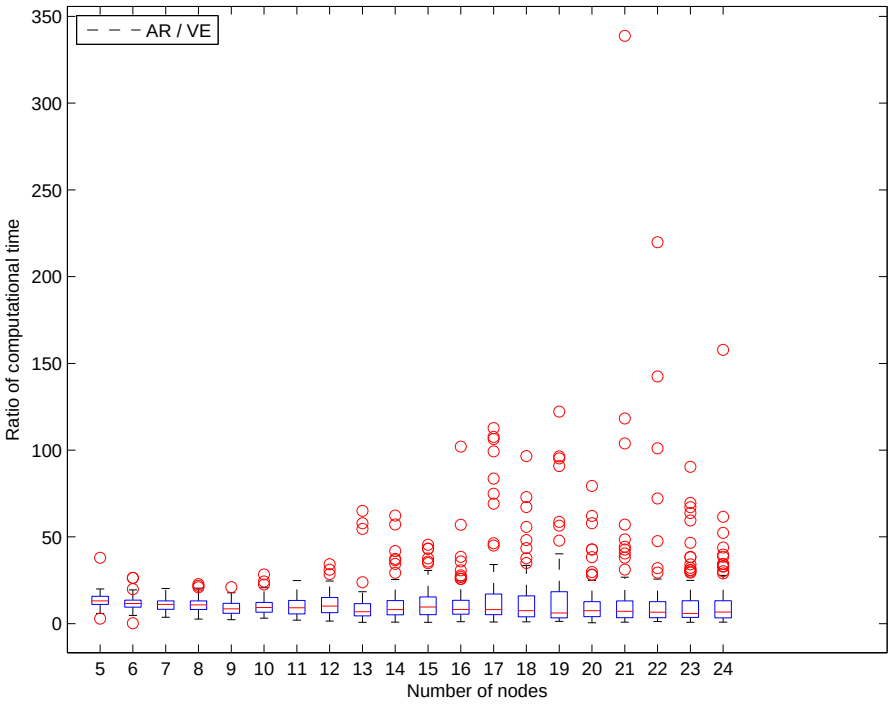

Figure 11: Ratio of the times required by AR and VE.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	0.96	0.51	0.69	1.00	1.05	1.16
6	0.91	0.37	0.61	1.00	1.00	1.00
7	0.88	0.31	0.41	0.99	1.17	1.27
8	1.10	0.34	0.67	1.00	1.72	2.00
9	1.24	0.35	0.56	1.14	2.22	3.97
10	1.43	0.43	0.51	1.14	3.25	4.76
11	1.67	0.39	0.50	1.23	4.27	6.28
12	1.99	0.39	0.56	1.67	4.56	8.13
13	1.76	0.14	0.56	1.32	3.71	11.81
14	2.31	0.27	0.47	1.59	7.00	16.19
15	2.80	0.22	0.45	1.88	8.96	19.49
16	3.14	0.24	0.67	1.97	8.85	33.68
17	3.26	0.27	0.64	1.94	10.49	22.42
18	2.81	0.17	0.39	1.69	8.26	12.03
19	4.02	0.34	0.53	1.91	13.84	51.36
20	3.28	0.25	0.52	1.92	10.14	25.03
21	4.18	0.42	0.68	1.96	14.41	58.86
22	3.74	0.24	0.61	1.88	12.18	58.74
23	4.28	0.33	0.41	1.98	16.25	59.89
24	3.26	0.33	0.56	1.88	11.58	16.01
Total	**2.45**	**0.14**	**0.55**	**1.29**	**7.75**	**59.89**

Table 2: Ratio of the maximum storage space required by AR and VE.

We obtained very similar results when comparing AR and VE-D, both with respect to time (Table 3 and Figure 13) and space (Table 4 and Figure 14). This result is coherent with the experimental evidence that the performances of VE and VE-D are very close, both in terms of time (Table 5) and space (Table 6). The cases in which VE is significantly more efficient than VE-D, or vice versa, are very rare. If time or space are a critical issue for a decision-support system based on an ID, it would be necessary to perform an ad-hoc comparison for that problem.

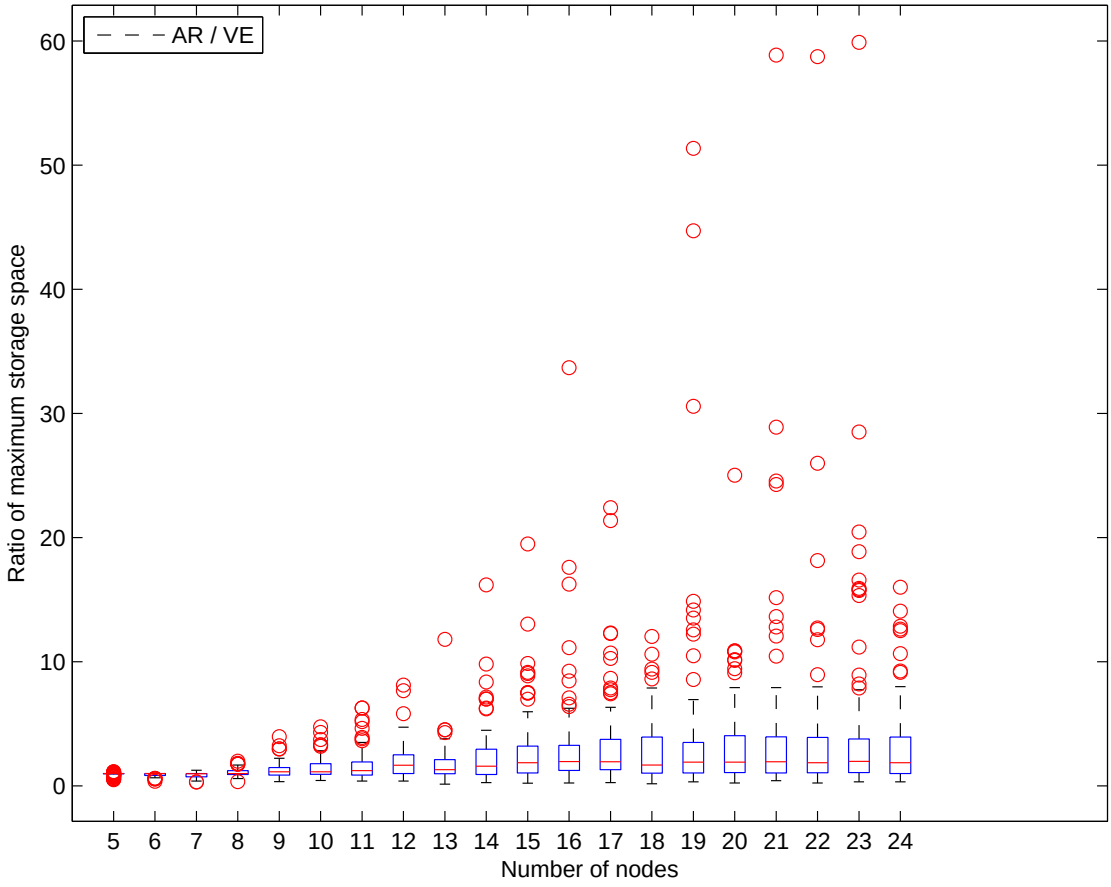

Figure 12: Ratio of the maximum storage space required by AR and VE.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	13.82	3.79	7.16	14.11	19.37	38.86
6	12.35	4.83	7.06	12.35	18.97	28.98
7	11.05	3.21	4.51	11.54	16.80	21.66
8	11.15	2.46	6.20	10.66	19.77	22.69
9	9.15	1.97	4.42	8.63	15.96	21.76
10	9.76	2.90	3.75	9.33	19.19	26.14
11	9.57	1.71	2.69	8.98	19.68	25.46
12	10.40	1.24	2.84	9.72	23.04	29.81
13	9.13	0.76	2.25	6.40	18.25	62.42
14	10.97	0.94	1.86	8.02	30.74	63.37
15	12.09	0.73	2.07	9.70	32.85	53.59
16	11.40	1.48	2.29	7.76	27.46	94.86
17	17.11	0.78	2.11	7.05	89.70	146.80
18	12.72	0.82	1.81	7.30	46.44	97.43
19	13.70	0.78	1.73	5.09	64.95	109.88
20	11.21	0.54	1.33	6.47	44.98	71.09
21	14.47	0.82	1.39	6.09	43.20	339.83
22	12.54	0.94	1.43	5.80	39.20	153.88
23	13.14	0.70	1.21	4.93	59.17	151.10
24	14.19	0.74	1.37	5.79	39.36	277.02
Total	**12.00**	**0.54**	**1.96**	**8.79**	**29.44**	**339.83**

Table 3: Ratio of the times required by AR and VE-D.

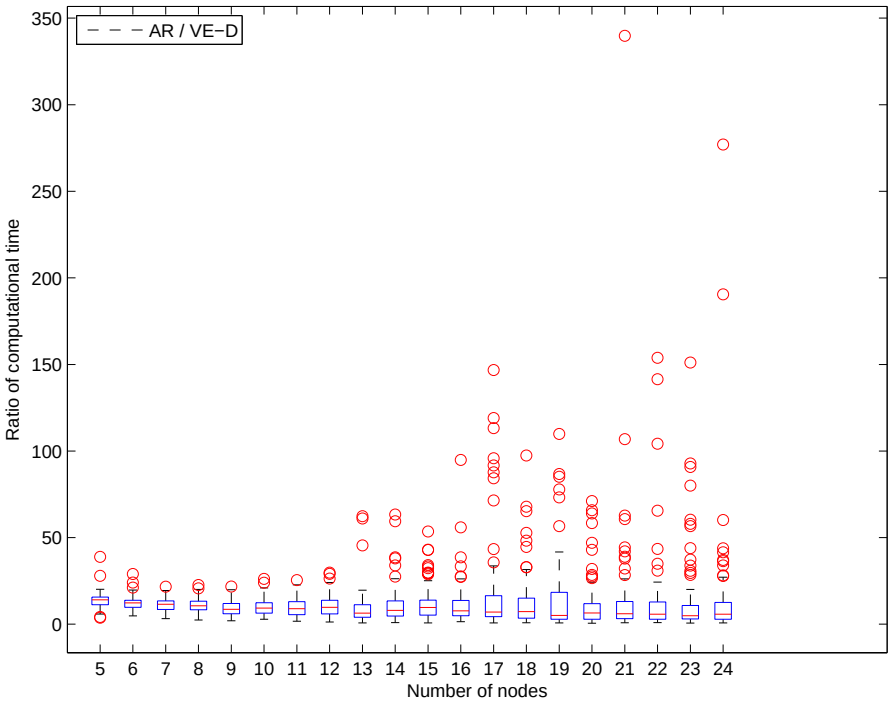

Figure 13: Ratio of the times required by AR and VE-D.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	0.96	0.49	0.70	1.00	1.05	1.16
6	0.91	0.36	0.61	1.00	1.00	1.00
7	0.88	0.30	0.41	1.00	1.16	1.26
8	1.09	0.34	0.66	1.00	1.68	2.03
9	1.23	0.35	0.56	1.13	2.22	3.97
10	1.43	0.44	0.54	1.14	3.26	4.80
11	1.70	0.39	0.50	1.23	4.56	6.28
12	1.95	0.39	0.53	1.60	4.57	8.13
13	1.78	0.14	0.56	1.34	3.70	11.30
14	2.32	0.26	0.47	1.65	6.91	16.25
15	2.85	0.22	0.51	1.92	8.98	19.58
16	3.18	0.27	0.67	1.97	9.81	33.68
17	3.26	0.26	0.57	1.93	10.49	22.48
18	2.84	0.17	0.39	1.71	8.29	12.03
19	3.96	0.08	0.46	1.79	14.22	50.10
20	3.53	0.25	0.51	1.91	11.11	31.48
21	4.00	0.43	0.66	1.95	13.91	40.44
22	3.77	0.23	0.62	1.85	11.68	58.74
23	4.68	0.33	0.41	1.97	17.72	58.20
24	3.66	0.33	0.57	1.87	12.47	48.24
Total	**2.50**	**0.08**	**0.54**	**1.28**	**7.75**	**58.74**

Table 4: Ratio of the maximum storage space required by AR and VE-D.

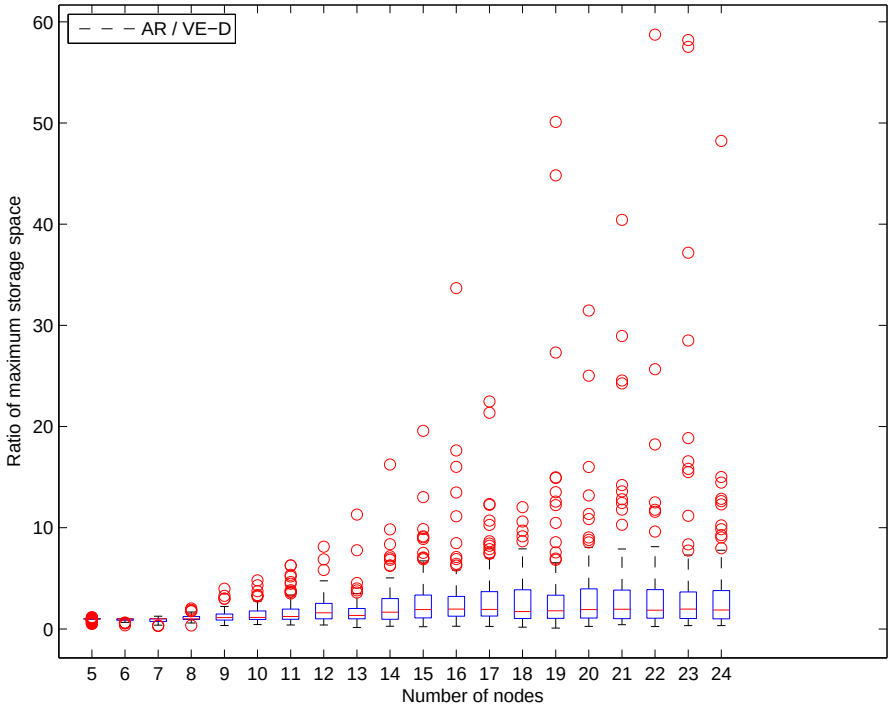

Figure 14: Ratio of the maximum storage space required by AR and VE-D.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	1.00	0.23	0.67	0.99	1.39	4.07
6	0.96	0.02	0.79	0.97	1.19	1.41
7	1.00	0.67	0.82	0.98	1.16	2.48
8	1.01	0.69	0.86	1.00	1.24	1.39
9	1.01	0.39	0.81	1.00	1.19	1.31
10	1.03	0.79	0.88	1.01	1.21	1.35
11	1.03	0.42	0.80	1.00	1.33	1.51
12	1.08	0.58	0.91	1.03	1.44	1.94
13	1.06	0.43	0.83	1.02	1.44	1.77
14	1.06	0.74	0.85	1.01	1.42	1.71
15	1.03	0.27	0.77	1.02	1.33	1.42
16	1.08	0.63	0.73	1.06	1.45	1.64
17	1.16	0.51	0.80	1.03	1.52	7.90
18	1.12	0.50	0.86	1.08	1.50	1.90
19	1.34	0.69	0.81	1.11	1.70	20.82
20	1.12	0.32	0.80	1.05	1.69	2.05
21	1.15	0.28	0.87	1.10	1.52	1.86
22	1.15	0.48	0.82	1.11	1.58	2.42
23	1.17	0.19	0.77	1.11	1.68	3.46
24	1.13	0.10	0.64	1.11	1.63	1.87
Total	**1.09**	**0.02**	**0.80**	**1.03**	**1.44**	**20.82**

Table 5: Ratio of the times required by VE-D and VE.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	1.00	0.99	0.99	1.00	1.00	1.08
6	1.00	0.99	0.99	1.00	1.00	1.03
7	1.00	0.99	0.99	1.00	1.08	1.11
8	1.01	0.70	0.99	1.00	1.10	1.27
9	1.01	0.99	0.99	1.00	1.10	1.22
10	1.00	0.68	0.96	1.00	1.09	1.19
11	0.99	0.50	0.87	1.00	1.08	1.20
12	1.02	0.49	0.93	1.00	1.22	1.43
13	1.00	0.40	0.74	1.00	1.17	1.44
14	1.00	0.73	0.81	1.00	1.07	1.14
15	0.98	0.30	0.86	1.00	1.08	1.12
16	1.00	0.69	0.76	1.00	1.11	1.21
17	1.05	0.66	0.87	1.00	1.18	4.72
18	0.99	0.34	0.70	1.00	1.18	1.28
19	1.16	0.50	0.86	1.00	1.18	13.65
20	1.01	0.34	0.81	1.00	1.18	1.49
21	1.03	0.49	0.80	1.00	1.15	3.28
22	1.00	0.51	0.76	1.00	1.15	1.48
23	1.07	0.28	0.77	1.00	1.24	5.21
24	1.00	0.16	0.85	1.00	1.17	1.25
Total	**1.02**	**0.16**	**0.89**	**1.00**	**1.12**	**13.65**

Table 6: Ratio of the spaces required by VE-D and VE.

Redundant variables We have also recorded the cases in which one algorithm included more redundant variables than the other: Table 7 shows that in 22.4% of cases (448 out of 2,000) VE

included fewer redundant variables than AR, while AR outperformed VE only in 8 cases, i.e., 0.4%. It means that for each case in which AR was superior, there were over 50 cases in which VE was better.

	AR	VE	VE-D	*Won*
AR	-	8 (0.4%)	5 (0.25%)	13 (0.33%)
VE	448 (22.4%)	-	10 (0.5%)	458 (11.45%)
VE-D	461 (23.05%)	58 (2.9%)	-	519 (12.98%)
Lost	909 (22.73%)	66 (1.65%)	15 (0.38%)	990 (8.25%)

Table 7: Comparison of the number of redundant variables between AR, VE and VE-D. Each cell (i, j) shows how many times the algorithm in the i-th row outperformed the algorithm in the j-th column. For instance, VE returned smaller policies than AR for 448 out of the 2,000 IDs (22.4%), while AR has beaten VE only in 8 cases. The *Won* column indicates how many times each algorithm beat each of the others. The percentages in this column are computed over 2,000×2=4,000 cases, because each algorithm is compared twice for each ID. The interpretation of the *Lost* column is similar. 990 is the number of cases in which there was not a tie.

Similarly, VE-D outperformed AR in 461 cases (23%), while the opposite happened only in 5 cases (0.25%); i.e., for each case in which AR returned fewer redundant variables, there were almost 100 cases in which it returned more.

VE-D was also superior to VE: the former performed better in 58 cases (2.9%), while the latter was superior in 10 cases (0.5%), a difference of almost 6 to 1. Therefore, if avoiding redundant variables is a priority, we should use VE-D. Given that VE and VE-D have a similar efficiency in time and space on average, we recommend VE-D as the default algorithm for evaluating IDs.

5.2.2 Effect of the subset rule

As mentioned in Section 4.2, Tatman and Shachter [29] proposed the *subset rule* (SR) as a heuristic for reducing the storage space required by their arc reversal algorithm, but they did not provide any empirical evidence of such saving of space. On the other hand, the application of the SR entails a computational cost, which might make it counterproductive, as we discussed in Section 4.2. For this reason, we thought it would be worthy to test empirically the utility of the SR.

First we analyzed the impact of that rule on the space requirements of AR. (We denote by AR-SR the version of arc reversal that uses the subset rule.) Contrary to the intuition by Tatman and Shachter, in most of the cases the SR did not save any space in general: if we measure the maximum storage space of both algorithms and compute the ratio $s_{\text{AR-SR}}/s_{\text{SR}}$, we see that the mean of ratios is 1.00, up to rounding errors, and the median is exactly 1—cf. Table 8. There were cases in which AR-SR saved space, but they were quite unfrequent and in general the difference was negligible; only in some exceptional cases the ratio reached values as low as 0.56 or 0.69, which means that the SR saved around half the storage space. More surprisingly, in other cases, also very exceptional, AR-SR required 1.82 or 2.29 times more space than AR. When we compared the computational times, we found slightly bigger differences (cf. Table 9): in one case AR-SR was almost twice faster, but in others AR was between 2 and 4 times faster. Given the scarce number of cases in which there was a significant difference and the opposite signs of the differences, the only conclusion that we can draw is that in general the SR has virtually no impact on the performance of the arc reversal algorithm.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	1.00	0.90	1.00	1.00	1.00	1.00
6	1.00	1.00	1.00	1.00	1.00	1.00
7	1.00	0.90	1.00	1.00	1.00	1.16
8	0.99	0.84	0.94	1.00	1.00	1.00
9	0.99	0.88	0.96	1.00	1.00	1.00
10	1.00	0.95	0.97	1.00	1.00	1.00
11	1.00	0.89	0.98	1.00	1.00	1.00
12	1.00	0.89	0.97	1.00	1.00	1.25
13	1.00	0.96	0.98	1.00	1.00	1.00
14	1.00	0.98	0.99	1.00	1.00	1.00
15	1.00	0.96	0.99	1.00	1.00	1.00
16	1.00	0.98	0.99	1.00	1.00	1.00
17	0.99	0.56	0.99	1.00	1.00	1.00
18	1.00	0.99	1.00	1.00	1.00	1.00
19	1.00	0.69	0.99	1.00	1.00	1.82
20	1.00	1.00	1.00	1.00	1.00	1.00
21	1.00	1.00	1.00	1.00	1.00	1.00
22	1.00	0.99	1.00	1.00	1.00	1.00
23	1.00	0.99	1.00	1.00	1.00	1.00
24	1.01	1.00	1.00	1.00	1.00	2.29
Total	**1.00**	**0.56**	**0.99**	**1.00**	**1.00**	**2.29**

Table 8: Ratio of the spaces required by AR-SR and AR.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	1.03	0.77	0.93	1.00	1.08	3.92
6	1.01	0.45	0.95	1.00	1.09	2.78
7	0.99	0.78	0.92	1.00	1.05	1.24
8	0.99	0.73	0.89	0.99	1.08	1.18
9	1.00	0.87	0.95	1.00	1.07	1.22
10	0.99	0.83	0.93	0.99	1.05	1.15
11	1.00	0.86	0.92	1.00	1.03	2.13
12	1.00	0.88	0.94	1.00	1.04	1.66
13	0.98	0.66	0.91	0.99	1.03	1.14
14	0.99	0.78	0.91	1.00	1.03	1.08
15	0.99	0.87	0.93	1.00	1.02	1.19
16	0.99	0.76	0.87	1.00	1.04	1.34
17	0.98	0.83	0.91	1.00	1.01	1.04
18	0.99	0.78	0.89	1.00	1.04	1.10
19	0.98	0.74	0.90	0.99	1.02	1.29
20	0.98	0.80	0.88	1.00	1.01	1.03
21	0.99	0.83	0.91	1.00	1.00	1.05
22	0.98	0.83	0.91	1.00	1.01	1.05
23	0.98	0.84	0.88	1.00	1.01	1.06
24	0.99	0.75	0.89	1.00	1.01	1.83
Total	**0.99**	**0.45**	**0.91**	**1.00**	**1.03**	**3.92**

Table 9: Ratio of the times required by AR-SR and AR.

We then studied the effect of that rule on our variable elimination algorithm. When we compared VE (without divisions and without the subset rule) with VE-SR (with the subset rule), we

found no consistent difference in the maximum storage space (cf. Table 10) nor in the computational time (cf. Table 11), even thought the differences seemed to be slightly higher than in the comparison of AR with AR-SR. We might be tempted to say, after looking at the "mean" columns of those tables, that the subset rule reduces slightly the maximum storage space for large IDs, on average, at the expense of increasing the time of computation, as we expected, but a look at the "median" column makes this conclusion doubtful.

Nodes	**Mean**	**Min**	**5th perc.**	**Median**	**95th perc.**	**Max**
5	0.97	0.68	0.84	1.00	1.00	1.06
6	0.97	0.73	0.87	1.00	1.00	1.05
7	0.98	0.68	0.89	0.99	1.02	1.28
8	0.97	0.75	0.86	0.98	1.04	1.49
9	0.94	0.50	0.69	0.97	1.00	1.18
10	0.94	0.42	0.70	0.97	1.05	1.19
11	0.94	0.53	0.64	0.98	1.02	1.53
12	0.96	0.54	0.76	0.98	1.02	1.60
13	0.94	0.37	0.67	0.99	1.02	1.61
14	0.97	0.52	0.65	0.99	1.06	3.59
15	0.96	0.39	0.77	0.99	1.00	1.89
16	0.95	0.64	0.68	0.99	1.04	1.92
17	0.96	0.45	0.72	0.99	1.02	1.99
18	0.95	0.40	0.67	0.99	1.10	1.50
19	0.99	0.44	0.61	1.00	1.26	2.30
20	0.95	0.38	0.67	1.00	1.00	1.17
21	0.95	0.39	0.68	1.00	1.00	1.02
22	0.92	0.43	0.53	1.00	1.00	1.59
23	0.96	0.36	0.68	1.00	1.00	1.76
24	0.93	0.52	0.67	1.00	1.00	1.24
Total	**0.95**	**0.36**	**0.68**	**0.99**	**1.01**	**3.59**

Table 10: Ratio of the spaces required by VE-SR and VE.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	0.96	0.52	0.70	0.95	1.26	2.28
6	0.97	0.02	0.77	0.98	1.16	1.54
7	0.97	0.66	0.72	0.99	1.15	1.32
8	1.00	0.70	0.82	0.99	1.20	2.10
9	1.00	0.62	0.71	0.99	1.24	2.91
10	0.99	0.53	0.73	0.99	1.24	1.58
11	1.04	0.35	0.71	1.01	1.52	2.22
12	1.06	0.67	0.81	1.03	1.31	2.36
13	1.00	0.59	0.74	1.00	1.26	1.81
14	1.09	0.53	0.77	1.02	1.33	6.50
15	1.05	0.54	0.74	1.04	1.43	1.94
16	1.04	0.52	0.71	1.02	1.48	2.06
17	1.07	0.64	0.76	1.04	1.48	1.90
18	1.06	0.59	0.78	1.04	1.33	1.85
19	1.09	0.60	0.75	1.04	1.62	2.00
20	1.06	0.39	0.78	1.04	1.40	1.69
21	1.04	0.42	0.79	1.03	1.32	1.73
22	1.05	0.59	0.68	1.04	1.49	2.20
23	1.06	0.43	0.75	1.04	1.34	1.68
24	1.00	0.25	0.71	1.00	1.30	1.80
Total	**1.03**	**0.02**	**0.74**	**1.02**	**1.34**	**6.50**

Table 11: Ratio of the times required by VE-SR and VE.

Interestingly, in the case of the VE-D algorithm, the subset rule seems to save both time and space for large IDs, but the average difference is small and the medians show no difference in storage space (Table 12) nor in computational time (Table 13).

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	0.98	0.69	0.82	1.00	1.00	1.00
6	0.99	0.73	0.91	1.00	1.00	1.00
7	0.98	0.69	0.84	1.00	1.00	1.00
8	0.97	0.75	0.84	1.00	1.00	1.05
9	0.95	0.49	0.71	1.00	1.00	1.00
10	0.94	0.43	0.68	1.00	1.00	1.00
11	0.93	0.51	0.64	1.00	1.00	1.00
12	0.94	0.52	0.72	1.00	1.00	1.00
13	0.93	0.37	0.67	1.00	1.00	1.06
14	0.92	0.42	0.58	0.99	1.00	1.00
15	0.92	0.42	0.61	1.00	1.00	1.00
16	0.94	0.59	0.67	1.00	1.00	1.00
17	0.93	0.30	0.67	1.00	1.00	1.04
18	0.92	0.34	0.60	1.00	1.00	1.00
19	0.93	0.44	0.54	1.00	1.00	1.00
20	0.93	0.46	0.62	1.00	1.00	1.00
21	0.94	0.40	0.69	1.00	1.00	1.00
22	0.89	0.33	0.48	1.00	1.00	1.00
23	0.93	0.34	0.58	1.00	1.00	1.00
24	0.89	0.39	0.59	1.00	1.00	1.00
Total	**0.94**	**0.30**	**0.66**	**1.00**	**1.00**	**1.06**

Table 12: Ratio of the spaces required by VE-D-SR and VED.

Nodes	Mean	Min	5th perc.	Median	95th perc.	Max
5	1.11	0.50	0.68	0.95	1.15	18.16
6	0.98	0.60	0.76	0.98	1.22	1.85
7	0.98	0.34	0.74	0.98	1.24	1.99
8	0.99	0.59	0.75	0.98	1.23	1.93
9	0.93	0.57	0.67	0.97	1.08	1.29
10	0.97	0.52	0.72	0.97	1.15	2.53
11	0.96	0.49	0.73	0.97	1.18	1.72
12	0.96	0.40	0.74	0.98	1.13	1.47
13	0.97	0.50	0.74	0.99	1.09	1.93
14	0.95	0.44	0.67	0.99	1.15	1.24
15	0.96	0.52	0.73	0.98	1.12	1.28
16	0.97	0.49	0.70	1.00	1.15	1.76
17	0.97	0.48	0.72	1.00	1.13	1.26
18	0.94	0.50	0.65	0.98	1.13	1.29
19	0.95	0.48	0.60	0.99	1.13	1.26
20	0.97	0.55	0.67	1.00	1.15	1.20
21	0.96	0.44	0.74	1.00	1.14	1.26
22	0.93	0.46	0.65	0.97	1.09	1.26
23	0.96	0.43	0.68	1.00	1.13	1.22
24	0.93	0.52	0.60	0.95	1.09	1.16
Total	**0.97**	**0.34**	**0.70**	**0.99**	**1.14**	**18.16**

Table 13: Ratio of the times required by VE-D-SR and VED.

6 Related work and future research

There are several variable elimination algorithms for IDs proposed in the literature [3, 4, 11, 27, 12], but none of them can evaluate IDs with super-value nodes. The algorithm that we have presented in this paper can be seen as an extension of those methods, designed as an alternative to the arc reversal algorithm by Tatman and Shachter [29], the only one that could evaluate IDs super-value nodes.

A problem of all these algorithms, including ours, is that they occasionally introduce redundant variables—see Section 1.3. Several algorithms have been proposed in the literature for detecting structurally redundant variables by analyzing the graph [8, 19, 21, 26, 31], but none of them can analyze IDs with super-value nodes. One of the advantages of the algorithm that we have proposed is that it rarely introduces redundant variables (see the experimental results in Sec 5.2.1). However, in some real-world applications it might be desirable to ensure that the decision-support system does not include any redundant variable at all, and for this reason we will propose in a future work an algorithm for eliminating them in the case of IDs with super-value nodes. This algorithm must take into account the distinction between *structurally redundant* and *quasi-structurally redundant* variables (cf. Sec. 1.3), which is one of the contributions of this paper.

Another element of crucial importance for the efficiency of algorithms for IDs (as in the case of Bayesian networks) is finding an efficient elimination order. In the first experiments that we carried out, our algorithm randomly selected the elimination order inside each $\mathbf{C}_i$ (let us remember that $\mathbf{C}_i$ is the set of variables unknown for decision D_i and known for D_{i+1}). In those experiments our algorithm was faster than AR in general, but it usually required more memory. Then we realized that one of the advantages of AR is that it automatically detects sink nodes, i.e., nodes having no outgoing arcs towards chance or decision nodes [22, 25][12]. When we forced our VE algorithm to use the elimination order as AR, VE was able to outperform AR not only in time efficiency, but

[12] AR takes profit of sink nodes by eliminating them without performing any numerical computation. In turn, VE can take profit of them because they lead to unity potentials—see Sec. 4.3.

also by requiring less storage space, as shown in Section 5.2.[13]

However, it might be desirable to have a method for finding the optimal elimination order for VE, which is not necessarily optimal for AR. However, given that finding the optimal elimination ordering for a Bayesian network is NP-complete, we conjecture that finding the optimal elimination order for VE is also NP-complete. For this reason, we should concentrate our efforts on developing heuristics that return near-optimal orderings. This a very difficult issue even when the ID has only one utility node [9], and becomes much more complex when the utility function is given by an ADG of potentials, since the basic operations of our algorithm (distribution and variable elimination) treat sum nodes in a very different way from product nodes, and treat chance variables differently from decision variables. A solution to this problem might be to examine different orderings, as in [9]: we can operate on the ADGoP, but instead of perfoming the numerical computations, we estimate their computational cost by analyzing the size of the potentials.

There is another line of improvement for our algorithm. The reason why VE and VE-D are not always better than AR is that, even though they try to preserve the separability of the utility function when eliminating a variable, sometimes the subsequent elimination of other variables merges the potentials that we wished to keep separate. This way some distributions become counterproductive, first because they increase the storage space and second because they prevent the algorithm from detecting common factors, as shown in Section 2.2.2. Therefore, the methods *unfork* and *distribute*, which in the current version of the algorithm only focus on the variable to be eliminated, should be refined in order to take into account the effect of the next eliminations.

We might try to solve both problems at the same time: we might assess the cost of different elimination orderings and different distribution strategies—also analyzing the number of redundant variables introduced by each one of them—and then perform the numerical computations for the optimal combination. However, it would be necessary to prove empirically that the time spent in the qualitative evaluation of several possibilities is compensated by finding a more efficient path for the evaluation of the ID.

Finally, it would be interesting to investigate how the ideas exposed in [30] to extend lazy evaluation to IDs without super-value nodes could be integrated with our algorithm and applied to more general IDs. This might avoid unnecessary multiplications and subsequent divisions, and also avoid redundant variables in the policies.

7 Conclusion

As we said in the beginning, we wished to develop a variable-elimination (VE) algorithm for IDs with supervalue nodes having five advantages over the arc reversal (AR) algorithm by Tatman and Shacher [29]: is faster, requires less memory, avoids redundant variables, simplifies sensitivity analysis, and can take profit of canonical models.

We have conducted some experiments to find out if we have succeded in the first three objetives. The analysis of 2,000 IDs randomly generated shows, in the first place, that on average VE is around 10 times faster than AR, especially for large IDs and in 5% of cases it is at least 30 times faster; for some IDs, it was between 100 and 340 times faster. In contrast, the cases in which AR is faster than VE are unfrequent and the differences are much smaller: for IDs having more than 6 nodes, AR could never be twice faster than VE.

In the second place, AR requires on average 3 or 4 times more space than VE, with a median ratio of about 2. For 5% of the IDs, AR needs at least 10 times more space. In several cases, it needed between 20 and 60 times more space. On the contrary, the cases in which VE required more memory are unfrequent and the differences are much smaller.

Third, our experiments showed that for each case in which AR introduces fewer redundant

[13]Fortunately, it is possible to feed our algorithm with the elimination order used by AR without incurring in the cost of completely executing the algorithm by Tatman and Shachter: it suffices to reverse arcs and delete nodes from the graph without performing any numerical computation. Given that these operations only focus on the neighbors of each node, the cost of obtaining the elimination order is absolutely negligible compared to the cost of operating with potentials.

variables than VE, there are over 50 in which VE is superior. A version of VE with division of probability potentials (VE-D) is even better: for each case in which AR introduces fewer redundant variables, there were almost 100 in which VE-D beat AR. Given that the time and space efficiency of VE-D is not worse than VE, we recommend VE-D as the default algorithm for evaluating IDs with super-value nodes.

Forth, our algorithm can simplify sensitivity analysis by keeping track of which potentials have been involved in the computation of the (new) potentials on which maximizations are performed —see Section 2.2.1.

Fifth, we have not tested empirically the time and space savings that our variable elimination algorithm can provide for IDs containing canonical models, such as the noisy OR/MAX [5]. However, the important savings obtained by the integration of variable elimination and canonical models in the case of Bayesian networks (see Section 1.1 and [6]) indicate that similar savings might be obtained for IDs.

In summary, we conclude that we have attained, to different degrees, the five objectives set at the beginning of our research.

A minor contribution of this paper is the empirical evaluation of the *subset rule* [29], which—as far as we know—had never been tested before. Our experiments have shown that for most IDs the impact of this rule is null or almost null and, contrary to our expectations, it can either increase or decrease the time and space spent by the algorithms (see Section 5.2.2).

Three main issues must be investigated in order to refine our algorithm: how to avoid introducing redundant variables at all, how to find near-optimal elimination orderings, and how to combine our algorithm with the recent proposals for the lazy evaluation of traditional IDs, in order to reduce the computational cost of evaluating IDs with super-value nodes.

7.1 Acknowledgements

This work has been supported by the Spanish Ministry of Education and Science, under grant TIN-2006-11152. Manuel Luque has also been partially supported by Department of Education of the Comunidad de Madrid and the European Social Fund (ESF). We thank the reviewers of the Second European Workshop on Probabilistic Graphical Models (PGM-04) for their comments on a previous version of this paper. We also thank Marta Vomlelova and Thomas D. Nielsen for valuable email discussions.

A Appendices

A.1 Proof of Theorem 8

Before proving the theorem, we introduce a definition and a lemma.

Definition 11 *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 12 *When the method* distribute *(Algorithm 1) is applied to a node n having a child of type sum, n_1, then $s(n'_{1,l}) < s(n)$ for each child $n'_{1,l}$ of n_1 in the new ToP (see Figure 4).*

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

Proof of Theorem 8. 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 every tree such that $s(r) \leq k$ and let us examine a tree such that $s(r) = k+1$, where $k \geq 1$. If r is of type sum, then each subtree of r has at most k summands (because r has at least two children), and therefore the *unfork* 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 $(k+1)/2$, and $(k+1)/2 \leq k$, which means that *unfork* terminates for those children. Similarly, the number of summands of each child of n_i is at most k, which means that the algorithm terminates for each child of n_i and for n_i itself. When all the children of r have processed the *unfork* message, it may happen that two of them, say 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 Figure 4, and to send again the message *unfork* to n_1. Since the lemma above states that $s(n'_{1,l}) < s(n)$, then $s(n'_{1,l}) \leq k$, and the *unfork* method terminates for the children of n_1 and, consequently, for n_1 itself. If r has still other children that depend on A, they must also be distributed wrt n_1, but the process terminates for each of those other children, and given that the number of children of r is finite, the whole process terminates. ■

A.2 Proof of Theorem 9

Proof. We prove the theorem by induction on the depth of the ToP, d. Clearly, the theorem holds for $d = 1$. Let us assume that the theorem holds for any tree whose depth is not greater than d and that there is a tree t of depth $d+1$, whose root r has m children, $n_1, \ldots, n_m$, such that each node n_i represents a potential ψ_i. If r is a sum node, the potential represented by r is:

$$\psi = \psi_1 + \ldots + \psi_m \ .$$

Therefore,

$$\sum_A \psi = \sum_A \psi_1 + \ldots + \sum_A \psi_m \ ,$$

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, because the tree is non-forked. 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

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

Since the depth of the tree rooted at n_j is d, the theorem holds because of the induction hypothesis. ■

A.3 Correctness of the algorithm VE-D

We prove now the correctness of VE-D (cf. Section 4.1), which eliminates the variables by applying Algorithm 7 iteratively.

Given an ID, let $\{V_1, \ldots, V_n\}$ be a valid elimination sequence for that ID, i.e., a sequence in which the first variables are those in $\mathbf{C}_n$, then D_n, then those in $\mathbf{C}_{n-1}$, and so on; the last variables are D_1 and those in $\mathbf{C}_0$. Because of Equations 1 and 2,

$$MEU = \operatorname*{op}_{v_n} \ldots \operatorname*{op}_{v_1} P(\mathbf{v}_C : \mathbf{v}_D) \cdot \psi_{U_0}(fPred(U_0)) \ , \tag{25}$$

where op is an operator that depends on the type of variable to be eliminated,

$$\operatorname*{op}_{v_i} = \begin{cases} \sum_{v_i} & \text{if } V_i \in \mathbf{V}_C \\ \max_{v_i} & \text{if } V_i \in \mathbf{V}_D \ , \end{cases} \tag{26}$$

and $P(\mathbf{v}_C|\mathbf{v}_D)$ is a family of probability distributions defined as follows:

$$P(\mathbf{v}_C|\mathbf{v}_D) = \prod_{V_i \in \mathbf{V}_C} P(v_i|pa(V_i)) \tag{27}$$

i.e., for each configuration $\mathbf{v}_D$ we have a probability distribution defined on $\mathbf{V}_C$.

We define $\mathbf{V}^0$ as the set of all the variables, $\mathbf{V}^0 = \mathbf{V}_C \cup \mathbf{V}_D$, and $\mathbf{V}^i$ as the set of variables remaining after eliminating V_i:

$$\forall i,\, 1 \leq i \leq n, \quad \mathbf{V}^i = \mathbf{V}^{i-1} \backslash \{V_i\} \; . \tag{28}$$

Clearly, $\mathbf{V}^n = \varnothing$. Analogously, $\mathbf{V}_C^i$ is the set of chance variables remaining after eliminating V_i, $\mathbf{V}_C^i = \mathbf{V}^i \cap \mathbf{V}_C$, and $\mathbf{V}_D^i = \mathbf{V}^i \cap \mathbf{V}_D$. Therefore, $\mathbf{V}_C^{i-1}$ contains all the variables remaining when V_i is to be eliminated.

Lemma 13 *If V_i is a decision, then $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D)$ does not depend on $\check{\mathbf{V}}_D^i$, where $\check{\mathbf{V}}_D^i = \mathbf{V}_D \backslash \mathbf{V}_D^i$.*

Please note that $\mathbf{V}_C^{i-1}$ is the set of chance variables remaining before eliminating V_i, $\mathbf{V}_D^i$ is the set of decisions remaining after eliminating V_i, and therefore $\check{\mathbf{V}}_D^i$ includes V_i and the decisions eliminated before V_i.

Before proving the lemma, we illustrate it with an example.

Example 14 *Coming back to the ID in Example 10 on page 18 (Fig. 9), Equation 27 tells us that*

$$P(\mathbf{v}_C|\mathbf{v}_D) = P(a, b, c, e|d_1, d_2) = P(a) \cdot P(b|a) \cdot P(c|a, d_1) \; . \tag{29}$$

The only valid elimination sequence is $\{V_1 = A, V_2 = D_2, V_3 = C, V_4 = D_1, V_5 = B\}$. The first decision eliminated is V_2 (D_2) and the second one is V_4 (D_1). Let us focus on the former. In this case, $i = 2$, $\mathbf{V}^{i-1} = \{D_2, C, D_1, B\}$, $\mathbf{V}_C^{i-1} = \{C, B\}$, and $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = P(c, b|d_1, d_2)$. We also have $\mathbf{V}_D^i = \{D_1\}$ and $\check{\mathbf{V}}_D^i = \{D_2\}$. The lemma states that $P(c, b|d_1, d_2)$ does not depend on d_2, which is obvious, because $P(c, b|d_1, d_2) = \sum_a P(a, b, c|d_1, d_2) = \sum_a P(a) \cdot P(b|a) \cdot P(c|a, d_1)$ and none of the factors inside the last summatory depends on d_2. Let us focus now on the second decision eliminated, D_1. Then, $i = 4$, $\mathbf{V}^{i-1} = \{D_1, B\}$, $\mathbf{V}_C^{i-1} = \{B\}$, and $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = P(b|d_1, d_2)$. We also have $\mathbf{V}_D^i = \varnothing$ and $\check{\mathbf{V}}_D^i = \{D_1, D_2\}$. The lemma states that $P(b|d_1, d_2)$ does not depend on d_1 nor on d_2. This result is not obvious, because apparently $P(b|d_1, d_2)$ depends on d_1:

$$P(b|d_1, d_2) = \sum_a \sum_c P(a, b, c, e|d_1, d_2) = \sum_a \sum_c P(a) \cdot P(b|a) \cdot P(c|a, d_1) \; . \tag{30}$$

Now, we prove the lemma.

Proof. We build a Bayesian network (BN) as follows: we create a chance node for each variable in $\mathbf{V}$. If V_i is a chance variable in the ID, we draw a link from each node that was a parent of V_i in the ID; the conditional probability distribution of V_i in the BN is the same as in the ID. If V_i is a decision in the ID, then V_i has no parents in the BN; we assign it an arbitrary distribution, for instance, a uniform probability. [The BN for the ID in Figure 9 is shown in Figure 15.] If $P_{BN}(\mathbf{v})$ is the join probability of the BN, then it follows from Equation 27 that

$$P(\mathbf{v}_C|\mathbf{v}_D) = P_{BN}(\mathbf{v}_C|\mathbf{v}_D) \tag{31}$$

and, consequently,

$$P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = P_{BN}(\mathbf{v}_C^{i-1}|\mathbf{v}_D) \; . \tag{32}$$

Now we focus on the decision V_i. In the BN, $\mathbf{V}_C^{i-1}$ is conditionally independent of $\check{\mathbf{V}}_D^i$ given $\mathbf{V}_D^i$, for the following reason: In the BN the nodes that correspond to decisions in the ID do not have parents. Therefore, any path departing from a decision V_j in $\check{\mathbf{V}}_D^i$ (V_j can be V_i itself) must pass through a child of V_j, that we call X, which was a chance variable in the ID. This node X and its descendants have been eliminated before V_i and before V_j, because a descendant of V_j

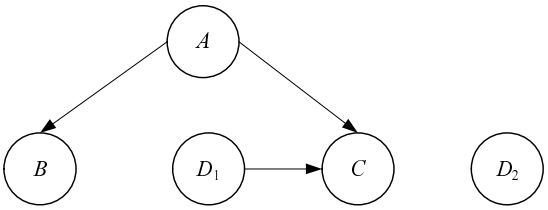

Figure 15: Bayesian network for the ID in Figure 9 (see the proof of Lemma 13).

can not be an informational predecessor of any of these decisions. Additionally, no node in $\mathbf{V}_D^i$ is a descendant of V_j. This implies that any path from any node in $\check{\mathbf{V}}_D^i$ to the any node in $\mathbf{V}_C^{i-1}$ is inactive given $\mathbf{V}_D^i$, in the sense of d-separation [23], and consequently $\mathbf{V}_C^{i-1}$ is conditionally independent of $\check{\mathbf{V}}_D^i$ given $\mathbf{V}_D^i$.[14] From the fact that $\mathbf{V}_C^{i-1}$ is conditionally independent of $\check{\mathbf{V}}_D^i$ given $\mathbf{V}_D^i$, we conclude thatp

$$P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = P_{BN}(\mathbf{v}_C^{i-1}|\underbrace{\mathbf{v}_D^i, \check{\mathbf{v}}_D^i}_{\mathbf{v}_D}) = P_{BN}(\mathbf{v}_C^{i-1}|\mathbf{v}_D^i) \ , \tag{33}$$

which proves that $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D)$ does not depend on $\check{\mathbf{V}}_D^i$. ■

We define ϕ^0 as the set of all the probability potentials, $\phi^0 = \{P(v_i|pa(V_i)) \ |V_i \in \mathbf{V}_C\}$, and ϕ^i as the list of probability potentials (LoPP) handled by VE-D *after* eliminating variable V_i. We denote by $\Pi\phi^i$ the product of all the potentials in ϕ^i.

Proposition 15 *At each step of algorithm VE-D, the list of probability potentials (LoPP) represents the probability of the chance variables that have not been eliminated yet:*

$$\forall i, \ 0 \leq i \leq n, \quad \Pi\phi^i = P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) \ . \tag{34}$$

Proof. We prove it by induction on i. When $i = 0$, i.e., before eliminating any variable, the LoPP contains all the conditional probability potentials that define the ID, i.e., ϕ^0; in this case, the proposition holds because of Equation 27. Let us assume that it holds for $i-1$:

$$\Pi\phi^{i-1} = P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) \ . \tag{35}$$

We divide ϕ^{i-1} in two sets: ϕ_+^{i-1} contains the potentials that depend on V_i and ϕ_-^{i-1} those that do not. If V_i is a chance variable, then $\mathbf{V}_C^i = \mathbf{V}_C^{i-1} \backslash \{V_i\}$ and

$$P(\mathbf{v}_C^i|\mathbf{v}_D) = \sum_{v_i} P(\underbrace{\mathbf{v}_C^i, v_i}_{\mathbf{v}_C^{i-1}}|\mathbf{v}_D) = \sum_{v_i} \Pi\phi^{i-1} = \Pi\phi_-^{i-1} \underbrace{\sum_{v_i} \Pi\phi_+^{i-1}}_{\phi^*_{V_i}} = \Pi\phi^i \ . \tag{36}$$

Algorithm 7 just implements this equation, because it leaves in the LoPP the potentials that do not depend on V_i, namely ϕ_-^{i-1}, and replaces those that depend on V_i with a new potential $\phi^*_{V_i}$ computed by multiplying all those potentials and summing out V_i. If V_i is a decision, then $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D)$ does not depend on V_i, because of Lemma 13. Given that $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = \Pi\phi_+^{i-1} \cdot \Pi\phi_-^{i-1}$ and no potential in ϕ_-^{i-1} depends on V_i, then $\Pi\phi_+^{i-1}$ can not depend on V_i. Therefore,

$$P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = \Pi\phi^{i-1} = \Pi\phi_-^{i-1} \cdot \underbrace{\text{project}_{V_i}\Pi\phi_+^{i-1}}_{\phi^*_{V_i}} = \Pi\phi^i \ . \tag{37}$$

[14] Coming back to Example 14, when eliminating D_2 ($i = 2$), we have $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = P(b, c|d_1, d_2) = P_{BN}(b, c|d_1, d_2)$. Because of d-separation (see Figure 15), $P_{BN}(b, c|d_1, d_2) = P_{BN}(b, c|d_1)$, which explains why $P(b, c|d_1, d_2)$ does not depend on D_2.

When eliminating D_1 ($i = 4$), we have $P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = P(b|d_1, d_2) = P_{BN}(b|d_1, d_2)$. Again, because of d-separation, $P_{BN}(b|d_1, d_2) = P_{BN}(b)$, which explains why $P(b|d_1, d_2)$ does not depend on D_1 nor on D_2.

Given that $\mathbf{V}^i = \mathbf{V}^{i-1} \backslash \{V_i\}$ and V_i is a decision, then $\mathbf{V}_C^i = \mathbf{V}_C^{i-1}$ and

$$P(\mathbf{v}_C^i|\mathbf{v}_D) = P(\mathbf{v}_C^{i-1}|\mathbf{v}_D) = \Pi\phi^i \ . \tag{38}$$

■

Theorem 16 *We have*

$$\forall i,\ 0 \leq i \leq n, \quad \text{MEU} = \operatorname*{op}_{v_n} \ldots \operatorname*{op}_{v_{i+1}} \Pi\phi^i \cdot \psi^i \ , \tag{39}$$

where ψ^i is the potential represented by the ADG of utility potentials (ADGoUP) after algorithm VE-D has eliminated variable V_i.

Proof. We prove it by induction on i. The theorem holds for $i = 0$ because $\Pi\phi^0 = P(\mathbf{v}_C|\mathbf{v}_D)$ and originally the ADGoUP represents the utility of the ID: $\psi^0 = \psi_{U_0}(fPred(U_0))$. Let us assume that it holds for $i-1$:

$$MEU = \operatorname*{op}_{v_n} \ldots \operatorname*{op}_{v_i} \Pi\phi^{i-1} \cdot \psi^{i-1} \ . \tag{40}$$

If V_i is a chance variable,

$$\operatorname*{op}_{v_i} \Pi\phi^{i-1} \cdot \psi^{i-1} = \sum_{v_i} \Pi\phi^{i-1} \cdot \psi^{i-1} \tag{41}$$

$$= \phi_-^{i-1} \sum_{v_i} \Pi\phi_+^{i-1} \cdot \psi^{i-1} \tag{42}$$

$$= \underbrace{\phi_-^{i-1} \cdot \phi_{V_i}^*}_{\phi^i} \underbrace{\sum_{v_i} \frac{\phi_{V_i}}{\phi_{V_i}^*} \cdot \psi^{i-1}}_{\psi^i} \ , \tag{43}$$

where ϕ_{V_i} and $\phi_{V_i}^*$ are defined as in Algorithm 7: $\phi_{V_i} = \Pi\phi_+^{i-1}$ and $\phi_{V_i}^* = \phi_{V_i}$. When comparing this equation with Algorithm 7, is is clear that

$$\operatorname*{op}_{v_i} \Pi\phi^{i-1} \cdot \psi^{i-1} = \phi^i \cdot \psi^i \tag{44}$$

If V_i is a decision, then $\Pi\phi^{i-1}$ does not depend on V_i and

$$\operatorname*{op}_{v_i} \Pi\phi^{i-1} \cdot \psi^{i-1} = \max_{v_i} \Pi\phi^{i-1} \cdot \psi^{i-1} \tag{45}$$

$$= \Pi\phi^{i-1} \max_{v_i} \psi^{i-1} \tag{46}$$

$$= \Pi\phi_-^{i-1} \cdot \underbrace{\Pi\phi_+^{i-1}}_{\phi_{V_i}^*} \cdot \underbrace{\max_{v_i} \psi^{i-1}}_{\psi^i} \ . \tag{47}$$

As $\Pi\phi_+^{i-1}$ does not depend on V_i, $\Pi\phi_+^{i-1} = \text{project}_{V_i}\Pi\phi_+^{i-1} = \phi_{V_i}^*$. On the other hand, $\psi^i = \max_{v_i} \psi^i$, which implies that Equation 44 also holds when V_i is a decision. This result, together with Equation 40, proves the theorem. ■

Corollary 17 *For every ID, Algorithm 7 returns the MEU and an optimal policy.*

References

[1] J. Cheng and M. J. Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000.

[2] R. T. Clemen and T. A. Reilly. *Making Hard Decisions*. Duxbury, Pacific Grove, CA, 2001.

[3] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag, New York, 1999.

[4] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI'96)*, pages 211–219, Portland, OR, 1996. Morgan Kaufmann, San Francisco, CA.

[5] F. J. Díez and M. J. Druzdzel. Canonical probabilistic models for knowledge engineering. Technical Report CISIAD-06-01, UNED, Madrid, Spain, 2006.

[6] F. J. Díez and S. F. Galán. Efficient computation for the noisy MAX. *International Journal of Approximate Reasoning*, 18:165–177, 2003.

[7] The Elvira Consortium. Elvira: An environment for creating and using probabilistic graphical models. In *Proceedings of the First European Workshop on Probabilistic Graphical Models (PGM'02)*, pages 1–11, Cuenca, Spain, 2002.

[8] E. Faguiouli and M. Zaffalon. A note about redundancy in influence diagrams. *International Journal of Aproximate Reasoning*, 19:231–246, 1998.

[9] M. Gómez and C. Bielza. Node deletion sequences in influence diagrams using genetic algorithms. *Statistics and Computing*, 14:181–198, 2004.

[10] R. A. Howard and J. E. Matheson. 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, 1984.

[11] F. Jensen, F. V. Jensen, and S. L. Dittmer. 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, 1994. Morgan Kaufmann.

[12] F. V. Jensen and T. D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, New York, second edition, 2007.

[13] C. Lacave and F. J. Díez. A review of explanation methods for Bayesian networks. *Knowledge Engineering Review*, 17:107–127, 2002.

[14] C. Lacave, M. Luque, and F. J. Díez. Explanation of Bayesian networks and influence diagrams in Elvira. *IEEE Transactions on Systems, Man and Cybernetics—Part B: Cybernetics*, 37:952–965, 2007. In press.

[15] C. Lacave, A. Oniśko, and F. J. Díez. Use of Elvira's explanation facilities for debugging probabilistic expert systems. *Knowledge-Based Systems*, 19:730–738, 2006.

[16] M. Luque and F. J. Díez. Decision analysis with influence diagrams using Elvira's explanation capabilities. In M. Studeny and J. Vomlel, editors, *Proceedings of the Third European Workshop on Probabilistic Graphical Models*, pages 179–186, 2006.

[17] M. Luque, F. J. Díez, and C. Disdier. Influence diagrams for medical decision problems: Some limitations and proposed solutions. In J. H. Holmes and N. Peek, editors, *Proceedings of the Workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP'05)*, pages 85–86, 2005.

[18] A. Madsen and F. V. Jensen. Lazy evaluation of symmetric Bayesian decision problems. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 382–390, San Francisco, CA, 1999. Morgan Kaufmann.

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

[20] T. D. Nielsen and F. V. Jensen. Sensitivity analysis in influence diagrams. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33:223–234, 2003.

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

[22] S. M. Olmsted. *On Representing and Solving Decision Problems.* PhD thesis, Dept. Engineering-Economic Systems, Stanford University, CA, 1983.

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

[24] M. Pradhan, G. Provan, B. Middleton, and M. Henrion. Knowledge engineering for large belief networks. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence (UAI'94)*, pages 484–490, Seattle, WA, 1994. Morgan Kaufmann, San Francisco, CA.

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

[26] R. D. Shachter. 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, 1998. Morgan Kaufmann.

[27] P. P. Shenoy. Valuation based systems for Bayesian decision analysis. *Operations Research*, 40:463–484, 1992.

[28] M. Takikawa and B. D'Ambrosio. Multiplicative factorization of the noisy-MAX. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 622–630, Stockholm, Sweden, 1999. Morgan Kaufmann, San Francisco, CA.

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

[30] M. Vomlelova. Unconstrained influence diagrams - experiments and heuristics. In *The Sixth Workshop on Uncertainty Processing WUPES'2003*, Hejnice, Czech Republic, 2003.

[31] M. Vomlelova and F. V. Jensen. 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.), 2002.