A Comparison of two Approaches for Solving Unconstrained Influence Diagrams

Kristian S. Ahlmann-Ohlsen, Finn V. Jensen, Thomas D. Nielsen [*], Ole Pedersen [1], Marta Vomlelová [b]

Dept. of Computer Science, Aalborg University, Denmark,
{ahlmann, fvj, tdn, ole}@cs.aau.dk

[b]*Faculty of Mathematics, Charles University, Prague, Czech Rep.,*
marta@kti.mff.cuni.cz

Abstract

Influence diagrams and decision trees represent the two most common frameworks for specifying and solving decision problems. As modeling languages, both of these frameworks require that the decision analyst specifies all possible sequences of observations and decisions (in influence diagrams, this requirement corresponds to the constraint that the decisions should be temporarily linearly ordered). Recently the unconstrained influence diagram was proposed to address this drawback. In this framework, we may have a partial ordering of the decisions, and a solution to the decision problem therefore consists not only of a decision policy for the various decisions, but also of a conditional specification of what to do next. Relative to the complexity of solving an influence diagram, finding a solution to an unconstrained influence diagram may be computationally very demanding w.r.t. both time and space. Hence, there is a need for efficient algorithms that can deal with (and take advantage of) the idiosyncrasies of the language. In this paper we propose two such solution algorithms. One resembles the variable elimination technique from influence diagrams, whereas the other is based on conditioning and supports any-space inference. Finally we present an empirical comparison of the proposed methods.

Key words: Unconstrained influence diagrams, solution algorithms, variable elimination, any-space inference.

[1] Current email address: oleslir@gmail.com
[*] Corresponding author.

Preprint submitted to Elsevier *11 August 2008*

1 Introduction

The two must common languages for graphical representation of decision problems are decision trees and influence diagrams (Howard and Matheson, 1981; Shachter, 1986). Both of these languages share the virtues that they have a very simple syntax and semantics, making them well suited for human specification as well as computer calculation. Decision trees have a very high expressive power. However, as the size grows exponentially with the number of decision and observation variables, they are badly suited for representing complex decision problems. Influence diagrams on the other hand give a very compact representation, which does not increase more than quadratically in the number of variables in the decision problem. To be able to represent a decision problem as an influence diagram, the problem has to meet certain constraints, and the most restrictive is that the problem must be *symmetric*. In short, this means that all decision variables are considered and all observations are taken, and the observations and decisions are temporarily linearly ordered.

Unconstrained influence diagrams (Jensen and Vomlelova, 2002) is a language very close to influence diagrams. The main difference is that they do not require a linear temporal order of the decisions and observations. We shall use the abbreviation ID for influence diagrams and accordingly UID for unconstrained influence diagrams. UIDs represent a language, which in a very compact way can model scenarios with a partial temporal ordering of decisions and observations. Although the representation is compact, a solution to the decision problem may be very complex. Contrary to IDs (Jensen et al., 1994; Cano et al., 2006), a solution for a UID not only includes an optimal policy for the various decisions, but also a conditional specification of what decision to choose next.

The basic computational structure for establishing an optimal strategy is a so-called *normal form S-DAG*, which is a structure supporting a large set of strategies among which at least one is optimal. Given the normal form S-DAG, there are basically two approaches to finding an optimal strategy. One is to perform variable elimination similarly to the solution methods for IDs, and the other is to simultaneously solve and unfold the S-DAG into a decision tree. We will show how the latter method supports an any-space algorithm, and we present experimental results comparing both approaches.

2 The Representation Language

The following two sections are extracts of what can be read elsewhere, e.g. (Jensen and Nielsen, 2007).

We start by considering an everyday situation (Howard, 1962): we would like to buy a used car, but only if it is in a satisfactory condition. We cannot observe the real state of the car (C) directly, but we can get some information I by just looking at the car. We may also obtain additional information by performing some tests (t_A and t_B) before we decide whether to buy. For simplicity, we assume all variables to be binary.

The tests can be performed in any order, so to represent this decision problem as an influence diagram we need to introduce two (artificial) decision variables, $Test_1$ and $Test_2$, representing the decisions on the first and the second test, respectively. Both variables have the states t_A, t_B, and *no-test*. Similarly, we introduce two chance nodes O_1 and O_2 to represent the outcomes of the two tests (see Figure 1).

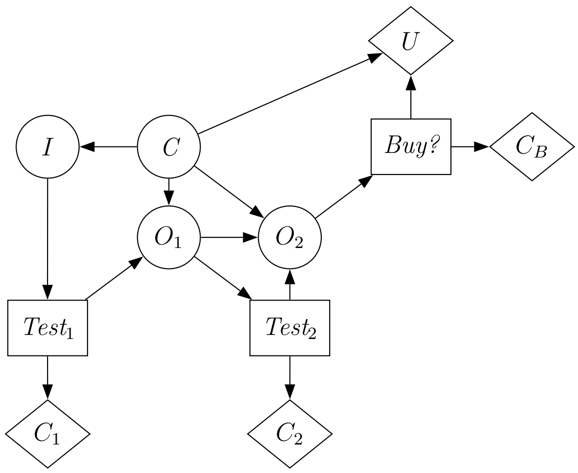


Fig. 1. An influence diagram representation of the buy-a-used-car example. The *Test*-nodes have three options, t_A, t_B and *no-test*. The O nodes have five states, $pos_A, pos_B, neg_A, neg_B$, and *no-test*. The arc $O_1 \rightarrow O_2$ indicates that repeating a test will give identical results. No forgetting is assumed. For example, I, $Test_1$, are known together with O_1 when deciding on $Test_2$.

In the used car buyer problem, you have the freedom of choosing the order of the two decisions. However, as the ID framework does not support this freedom, it is hidden in the specification of the state spaces. A more natural representation would be to have the two test options represented directly. This is done in Figure 2, where we furthermore represent the observable chance nodes explicitly: A double-circled chance node indicates that the node can be observed when all its preceding decisions are taken. For example, I does not have any preceding decisions so it may be observed at any point in time. But e.g. O_A is preceded by $Test_A$? so it can be observed only after $Test_A$? has been decided upon. In this model we have dropped the requirement of the decisions being temporarily linearly ordered and left the sequencing as a part of the solution to the problem.

Looking at Figure 2 it may seem that we have to analyze all possible se-

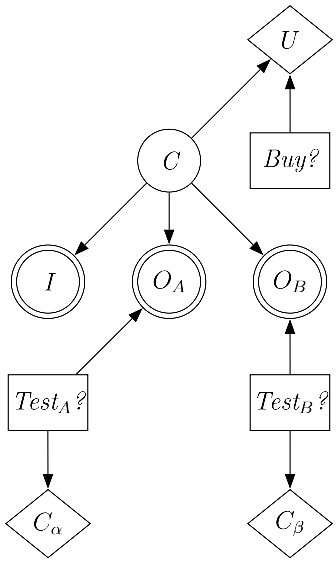


Fig. 2. An unconstrained influence diagram representation of the buy-a-used-car example. A double circled chance node can be observed when its preceding decision nodes have been decided upon. $Test_A$? has the states *yes* and *no*, and O_A has the states pos_A, neg_A, and *no-test*. Similarly for $Test_B$? and O_B.

quences of observations and decisions in order to establish a solution. However, as the expected utility (EU) can never increase by delaying a "cost free" observation, it is sufficient to consider sequences starting with observing I. For the same reason, O_A and O_B shall be observed immediately after the corresponding test decision has been taken. Finally, consider the sequence $< I, Test_A?, O_A, Buy?, Test_B?, O_B >$. As Buy? and $Test_B$? can be commuted without affecting the expected utility, it is equivalent to considering the sequence $< I, Test_A?, O_A, Test_B?, Buy?, O_B >$. Again, EU is not decreased if O_B is moved immediately to the right of $Test_B$?, and we get a sequence ending with Buy?. In summary, our sequencing options have been reduced to the ones illustrated in Figure 3.

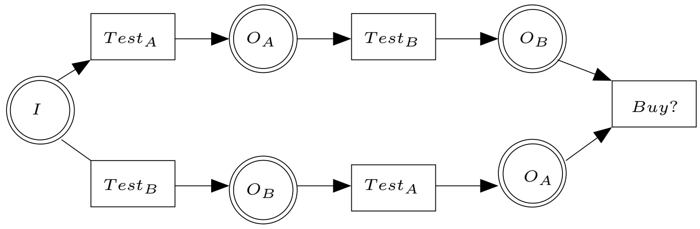


Fig. 3. A directed graph representing the possible optimal temporal sequences of observations and decisions for the used can buyer example. After observing I we should decide on the initial test, and in the end we must decide on Buy?.

From the structure of Figure 3 we can infer that solving the decision problem boils down to solving two influence diagrams, one for each of the two paths. The only thing left to resolve is to determine which path to follow as a function of the value of I. This is done as follows: Let $\sigma_A(I)$ and $\sigma_B(I)$ be the optimal policies for the first decision in the two influence diagrams, and let $EU_A(I)$ and $EU_B(I)$ be the expected utilities of following these policies. For any state i of I we compare $EU_A(i)$ and $EU_B(i)$ and choose the initial action of the model with maximal expected utility.

Next, consider a slightly more complex example. A patient may suffer from two different diseases (D). There are two possible tests, T_A and T_B, and each disease has a specific treatment Tr_1 and Tr_2. After each treatment, the new state of the disease is observed (the O-nodes). In Figure 4 the decision problem is represented graphically.

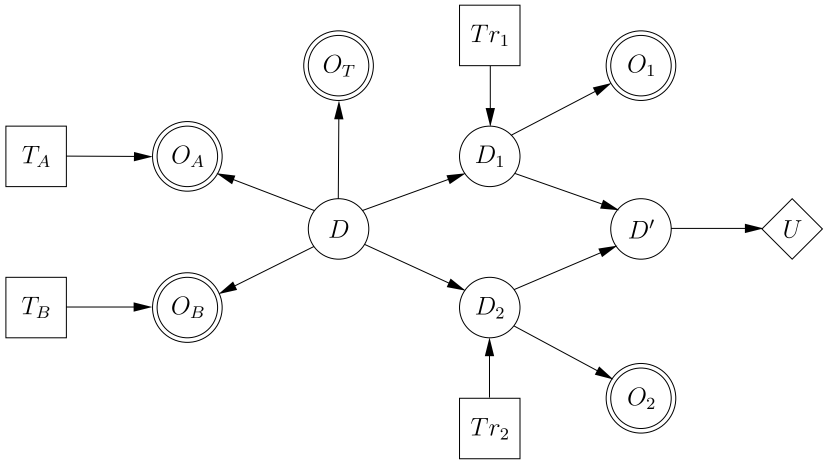


Fig. 4. An unconstrained influence diagram for a problem with two tests and two treatments. D_1 and D_2 represent the disease-situation after treatments, and D' represents the final situation.

Even for a simple decision problem as above it is extremely cumbersome to draw a decision tree, and as the problem does not include information about temporal orders it is rather tricky to squeeze the decision problem into the ID straight jacket. All possible sequences would have to be represented explicitly in the influence diagram.

Definition 1 *An* unconstrained influence diagram *(UID) is a directed acyclic graph (DAG) over decision nodes (rectangular shaped), chance nodes (circular shaped), and utility nodes (diamond shaped). Utility nodes have no children. There are two types of chance nodes,* observables *(doubly circled) and* non-observables *(singly circled). Non-observable chance nodes do not have decision nodes as children.*

The quantitative specification for a UID is similar to the specification for IDs: conditional probabilities and utility functions. We add the convention that each decision variable D has a cost. If this cost only depends on D, it need not be represented graphically, and the cost function is then attached to D. We say that a UID is *instantiated* when the structure has been extended with the required quantitative specification, i.e., probabilities and utilities.

The semantics of a UID is similar to the semantics of an ID. A link into a decision node represents temporal precedence; a link into a chance node represents causal influence; a link into a utility node represents functional dependence. We assume *no-forgetting*: at each point in the decision process the decision maker remembers all previous decisions and observations.

An observable can be observed when all its antecedent decision variables have been decided upon. In that case we say that the observable is *free*, and we *release* an observable when the last decision in its ancestral set is taken. Notice that observing a free variable is cost free. This does not cause a problem; if an observation has a cost, observing it needs to be decided, and it then has to be modeled as a test decision.

The structural specification yields a partial temporal order. The partial temporal ordering for the previous example was very simple, but this is rarely the case. For illustration we will in the following use the partial temporal ordering given in Figure 5.

If a partial ordering is extended to a linear ordering we get an influence diagram. Such an extension is called an *admissible ordering*.

3 Strategies and S-DAGs

A strategy for a UID is more complex than in the case of IDs. In principle we look for a set of rules telling us what to do given the current information, where "what to do" is to choose the next action as well as choosing a decision option if the next action is a decision.

A solution method could be to unfold the UID to a decision tree and compute an optimal strategy. However, this is unnecessarily complex. Structural analysis as performed above on the buy-a-used-car example can reduce the number of scenarios to consider substantially. For this purpose we construct a *strategy-DAG* (S-DAG). An S-DAG is a directed acyclic graph representing possible conditional orderings of the variables, including the optimal one. The S-DAG may then be unfolded to a decision tree or it may be used for variable elimination, analogous to variable elimination for IDs.

Notation Let Γ be a UID. The set of decision variables is denoted $\mathcal{D}_\Gamma$, the set of observables is denoted $\mathcal{O}_\Gamma$. Let $\mathcal{X} \subseteq \mathcal{D}_\Gamma \cup \mathcal{O}_\Gamma$ be a set of variables; $\mathrm{sp}(\mathcal{X})$ denotes the set of configurations over $\mathcal{X}$ (ignoring order). The partial temporal order induced by Γ is denoted $\prec_\Gamma$. When obvious from the context we avoid the subscript.

Definition 2 *Let* Γ *be a UID. An S-DAG is a directed acyclic graph* Σ*. The nodes are labeled with variables from* $\mathcal{D}_\Gamma \cup \mathcal{O}_\Gamma$ *such that each maximal directed path in* Σ *represents an admissible ordering of* $\mathcal{D}_\Gamma \cup \mathcal{O}_\Gamma$*. For notational convenience we add two unary nodes labeled source and sink. source is the only node with no parents and sink is the only node with no children. Source contains all the initially free chance variables, and sink contains all the chance variables which are never observed (both nodes may be empty).*

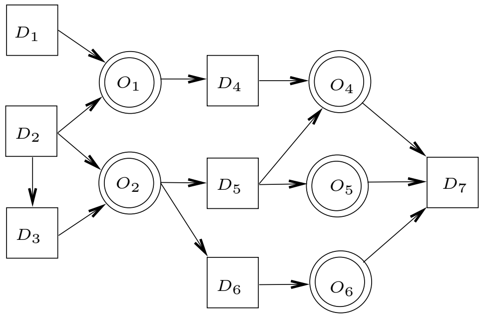


Fig. 5. An example partial temporal ordering from a UID.

Consider the partial temporal order in Figure 5. Figure 6 shows an S-DAG for a UID with this partial order.

Since an S-DAG encodes the possible orderings to consider, we define a strategy relative to an S-DAG. Let X be a node in an S-DAG Σ. The *past* of X (denoted $\mathrm{Past}(X)$) is the union of labels of X and its ancestors. The union of labels of X's children is denoted $\mathrm{ch}(X)$. A *step-policy* for X is a function $\sigma : \mathrm{sp}(\mathrm{Past}(X)) \to \mathrm{ch}(X)$, where $\mathrm{sp}(\mathrm{Past}(X))$ denotes all possible configurations of the variables in $\mathrm{Past}(X)$. If X has only a single child, then the step-policy is trivial. For example, a step-policy for the O_2 node between "Step 5" and "Step 4" in Figure 6 could be "if $O_2 = o_1^2$ then D_6 else D_5".

A *step-strategy* for Γ is a couple $(\Sigma, \mathcal{S})$, where Σ is an S-DAG for Γ and $\mathcal{S}$ is a set of step-policies, one for each node in Σ (except for *sink*).

A *policy* for X is an extension of a step-policy, such that whenever the step-policy yields a decision variable D, then the policy yields a state of D. A *strategy* for Γ is an S-DAG together with a policy for each node. A policy for the O_2 example above could be "if $O_2 = o_1^2$ then D_6, and if $O_1 = o_3^1$ then $D_6 = d_2^6$, else ...".

In order to specify an optimal strategy, we need to define the *expected utility* (EU) of a strategy for a UID. As a precise definition is unnecessarily complex we shall take the easy way: any strategy Δ for a UID can be folded out to a *strategy tree.* Following the policies in Δ we construct a tree, where all root-leaf paths represent admissible orderings. The expected utility of a strategy

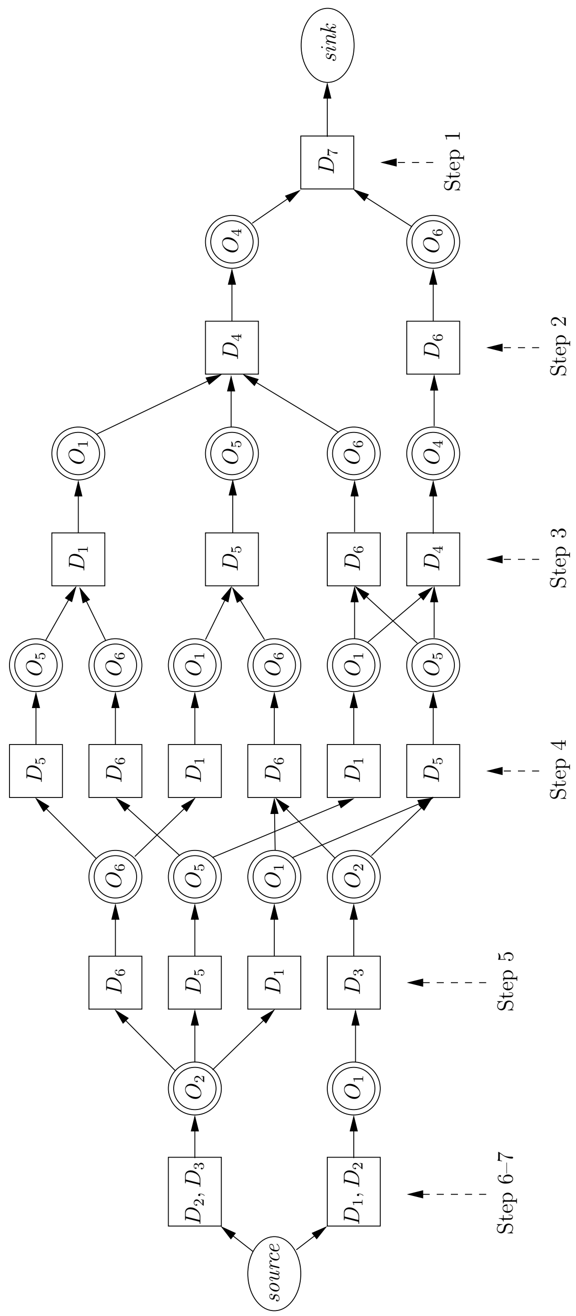


Fig. 6. An S-DAG for the partial order in Figure 5. The steps are explained in Section 4.

tree is defined as for decision trees, and the expected utility of a strategy is the expected utility of the corresponding strategy tree.

A *solution* to a UID is a strategy of maximal EU. Such a strategy is called *optimal*.

4 The Normal Form S-DAG

We wish to construct an S-DAG which is guaranteed to contain an S-DAG for an optimal strategy. To be on the safe side, you could let the S-DAG contain all possible admissible orderings. However, this will most likely be intractable. Instead you can reduce the S-DAG by exploiting the following two observations

(1) The expected utility can never increase by delaying an observation.[2] So, we need not have a path on which a decision variable is placed before an already free observation.
(2) As two maximizations (summations) over finite variables are commutable, a sequence of variables of the same type can be commuted without changing the EU. So, a sequence of consecutive variables of the same type can be characterized as a set rather than a sequence.

Due to observation 2 we let the labels of the nodes be sets (represented by calligraphic letters, e.g. $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$) rather than single variables. When it causes no confusion we will not distinguish between a node and its label, and when the label consists of one variable, we avoid talking about it in set terms. In general, we will use calligraphic letters to denote nodes in an S-DAG. With this convention, the labels of nodes are sets of variables of the same type, and we classify them as *decision nodes* and *observation nodes*.

Definition 3 *The* set of admissible orderings of an S-DAG *is the set of sequences of variables that can be obtained by following the directed paths from source to sink.*

Two admissible orderings yield the same EU if they only differ in the order of neighboring nodes of the same type. We shall call such orderings equivalent, and our concern is the set of equivalence classes represented in an S-DAG. In order not to obscure terminology, we do not distinguish between an admissible order and its equivalence class.

Definition 4 *Two S-DAGs are* equivalent *if their corresponding sets of admissible orderings are identical.*

By letting sequences of variables of the same type be represented as sets, it is sometimes possible to merge nodes in an S-DAG, thereby obtaining a smaller equivalent representation. Examples are given in Figure 7.

A minor matter should be resolved at this point. It may happen that there is a temporal link between two decision variables (you cannot take out the spark plugs before you have opened the motor hood). In this case we allow the two

[2] Recall that observations are cost free.

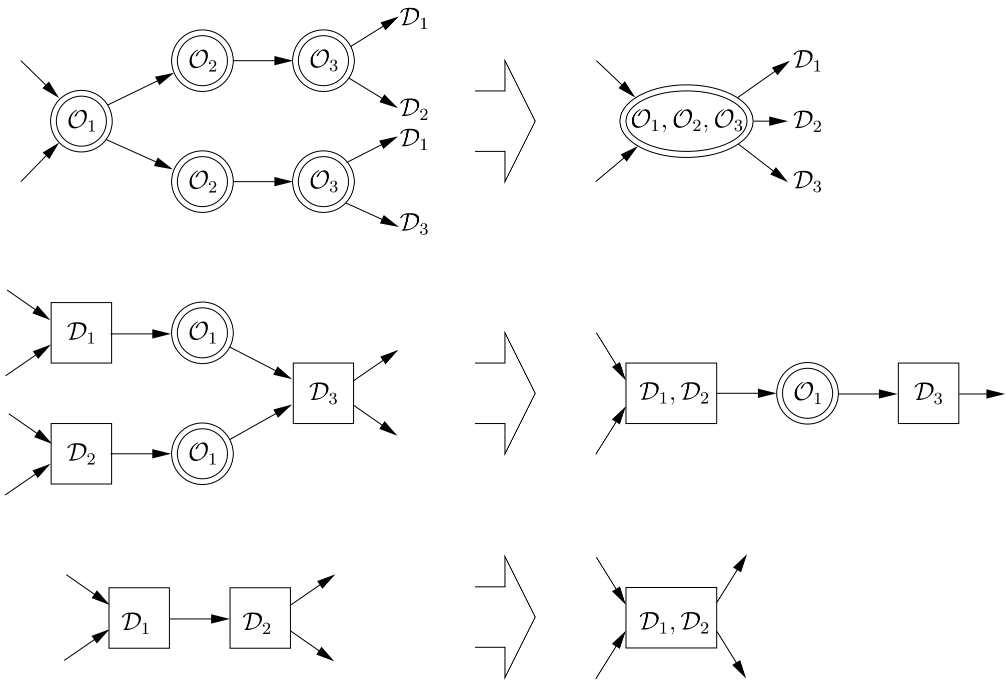


Fig. 7. Examples of merging nodes in an S-DAG.

decision variables to be placed in the same node, and since it does not affect the expected utility we also allow them to be commuted.

Definition 5 *An S-DAG is* minimal *if no consecutive nodes can be merged without changing the set of admissible orderings.*

Definition 6 *Let $\prec$ be a partial order induced by the UID Γ, let σ be an admissible ordering, let O be an observation variable, and let D be O's immediate decision predecessor in σ. O is* misplaced *if $D \not\prec O$.*

Note that due to observation 1 any UID has an optimal strategy without misplaced observations. We aim at constructing an S-DAG containing exactly all admissible orderings without misplaced observations.

Definition 7 *Let Γ be a UID. A minimal S-DAG containing exactly all admissible orderings without misplaced observations is a* normal form *S-DAG (NFS-DAG) for Γ.*

The S-DAG in Figure 6 is a normal form S-DAG for a UID with the partial ordering in Figure 5.

Proposition 4.1 *Let G be an S-DAG for the UID Γ, and let G' be a normal form S-DAG for Γ. Then the expected utility of an optimal strategy for G' is not smaller than the expected utility of an optimal strategy for G.*

Proof If the optimal strategy for G has no misplaced observations, it is a strategy in G'. If it has misplaced observations, it can be improved. $\square$

We state the following proposition without proof:

Proposition 4.2 *Any normal form S-DAG has the following properties*

(1) All children (different from sink) of observation nodes are decision nodes.
(2) All children (different from sink) of decision nodes are observation nodes.
(3) A decision node has exactly one child.
(4) An observation node (different from source) has exactly one parent.
(5) Let the observation node $\mathcal{O}$ be a child of $\mathcal{D}$; let $D \in \mathcal{D}$ and $O \in \mathcal{O}$. Then $D \prec O$.

Lemma 1 *Let $\mathcal{D}$ be a decision node in an S-DAG, and let $\mathcal{O}_1$ and $\mathcal{O}_2$ be parents of $\mathcal{D}$. If $\mathcal{O}_1$ is a proper subset of $\mathcal{O}_2$, then any path from source to $\mathcal{O}_2$ contains a misplaced observation.*

Proof

Let $\mathcal{D}$ have the parents $\mathcal{O}_1$ and $\mathcal{O}_2$ such that $\mathcal{O}_1 \subset \mathcal{O}_2$, and let $\mathcal{D}_1$ and $\mathcal{D}_2$ be their parents, respectively. If $\mathcal{D}_1 \subset \mathcal{D}_2$, then *any* $O \in \mathcal{O}_2 \setminus \mathcal{O}_1$ would be misplaced with respect to any $D \in \mathcal{D}_1$ in $\mathcal{D}_2$. If $\mathcal{D}_1 \not\subseteq \mathcal{D}_2$, consider $D_1 \in \mathcal{D}_1 \setminus \mathcal{D}_2$. Then D_1 must appear in a $\mathcal{D}_3$ on any path from source to $\mathcal{O}_2$ (see Figure 8). Consider $O_3 \in \mathcal{O}_3$. Since $O_3 \notin \mathcal{O}_1$ (as $\mathcal{O}_1 \subset O_2$), we have $D_1 \not\prec O_3$, and O_3 is misplaced. Finally, if $\mathcal{D}_1 = \mathcal{D}_2$, then $O \in \mathcal{O}_2 \setminus \mathcal{O}_1$ is misplaced.

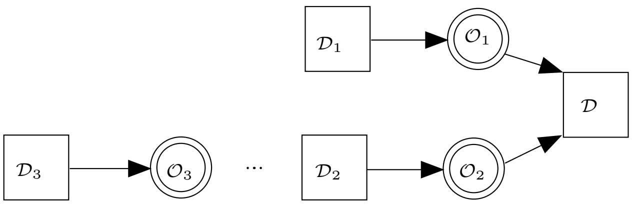


Fig. 8. The situation with $\mathcal{O}_1 \subset \mathcal{O}_2$ and $\mathcal{D}_2 \subseteq \mathcal{D}_1$.

$\square$

Proposition 4.3 *Let $\mathcal{D}$ be a decision node in a normal form S-DAG, and let $\mathcal{O}_1$ and $\mathcal{O}_2$ be parents of $\mathcal{D}$. Then $\mathcal{O}_1 \not\subseteq \mathcal{O}_2$.*

Proof If $\mathcal{O}_1 = \mathcal{O}_2$, then they can be merged, and if $\mathcal{O}_1 \subset \mathcal{O}_2$, then according to Lemma 1 the S-DAG contains a misplaced observation and cannot be minimal. $\square$

Let $\mathcal{D}$ be a decision node with child $\mathcal{O}$. The *decision children* of $\mathcal{D}$ are the children of $\mathcal{O}$.

5 Construction of Normal Form S-DAGs

To exploit Proposition 4.3, we construct a normal form S-DAG by starting with *sink* and move backwards in time. If the UID has n decision variables, the construction is performed in up to $n+1$ steps starting by determining the set of possible last decisions. At each step, we first identify the set of possible preceding decisions, and afterwards the sets of observations released by these decisions are found and links are entered.

We shall illustrate the method by using a UID with the partial temporal ordering illustrated in Figure 5.

We start the construction by introducing the node *sink* and entering all non-observables to *sink*. Next, consider which decisions may be the last decision. This is the decision variable D_7. As D_7 does not release any observations, we enter the link $(D_7, sink)$. Next, we determine the set of decisions that may precede D_7. These are D_4, D_5, and D_6. These decisions release the observations O_4, $\{O_4, O_5\}$, and O_6, respectively. As the observations released by D_5 contain the observations released by D_4, Proposition 4.3 yields that D_5 is not really an option. Hence, the situation is as illustrated in Figure 9.

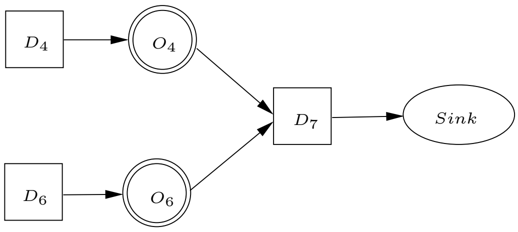


Fig. 9. The construction after Step 2.

The nodes D_4 and D_6 are called the top nodes of Step 2, and the set of decision variables on the path starting in a top node T and ending in *sink* are called the future of T.

In Step 3 we investigate whether top nodes from Step 2 have identical future. This is not the case, and we determine the possible decision parents for each top node. After determining the released observations and entering links, we get the situation in Figure 10.

In Step 4 we realize that the paths from the top nodes D_6 and D_4 to *sink* contain the same variables. The past of D_6 and D_4 must then also contain the same variables. This means that the decision options from source to D_4 and D_6, respectively, are identical. In short, they have identical past, and they shall be given the same parents. We determine the possible decision parents for each top node, determine released observations, enter links, and we get the situation in Figure 6 indicated with 'Step 4'.

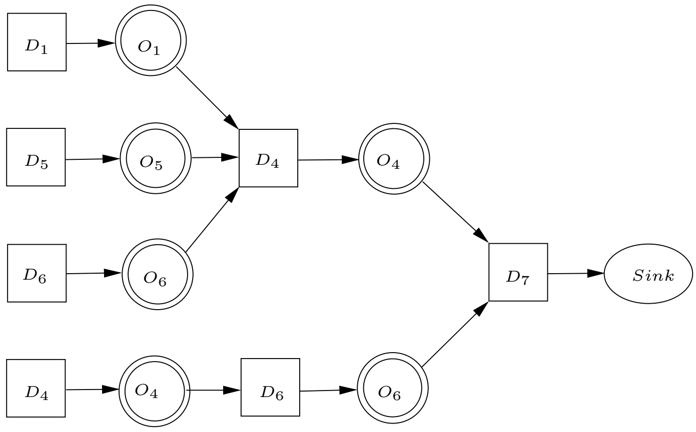


Fig. 10. The construction after Step 3.

In Step 5 we determine that the upper D_5 and the upper D_1 have common future, etc. and we get the situation indicated with 'Step 5'. In Step 6, the three upper top nodes have common future, and as $D_2 \prec D_3$, the parent is D_3. For the top node D_3, the possible parents D_1 and D_2 release the same observations, and the two decisions are merged into one node (thereby Step 7 has also been performed for the lower branch). In Step 7 we ignore $D_2 \prec D_3$ and merge. The construction is finished by adding possible free observations to *source*, and we get the S-DAG in Figure 6.

Proposition 5.1 *The construction results in a normal form S-DAG.*

Proof We shall prove that (1) no nodes can be merged, (2) there are no misplaced observations, and (3) all admissible orderings without misplaced observations are represented.

(1) No consecutive nodes are of the same type.
(2) When an observation node $\mathcal{O}$ with parent $\mathcal{D}$ is entered, none of $\mathcal{O}'s$ observations are misplaced. However, if $\mathcal{D}$ in the next step is extended with a new decision variable D (that is, D does not release any observations at that step), then some elements of $\mathcal{O}$ may be misplaced with respect to D. As D in step $i+1$ does not release observations, then its released observations must be a subset of $\mathcal{O}$. In that case we could in step i have branched to D as well as $\mathcal{D}$, but since the observations released by D form a proper subset of $\mathcal{O}$, then branching to $\mathcal{D}$ would have been refuted by Proposition 4.3.
(3) Let $\sigma = < x_1, \ldots, x_m >$ be an admissible ordering without misplaced observations, and assume that the subsequence $< x_i, \ldots, x_m >$ is represented, where x_i is a decision variable. Let x_j be the immediate preceding decision variable, and let the subsequence start in the node $\mathcal{D}$ from the S-DAG with decision parents $\mathcal{D}_1, \ldots, \mathcal{D}_k$ (see Figure 11).

    As all temporal descendants of x_j are included in the structure at $\mathcal{D}'s$ position, x_j has been considered at that place during the construction.

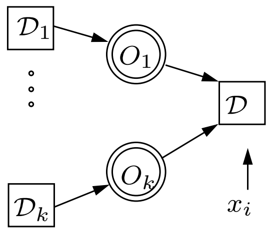


Fig. 11. The situation for the proof of Proposition 3.

If $x_j \in \mathcal{D}_1$, say, then the elements of $\mathcal{O}_1$ would be misplaced if they occur after x_i in σ, and they cannot occur before x_j. Hence the variables between x_j and x_i are exactly $\mathcal{O}_1$, and the subsequence $< x_j, \ldots x_m >$ is represented. If x_j is not a member of any $\mathcal{D}_q$, then it has been refused in the construction due to Proposition 4.3, and from the proof we infer that σ must have a misplaced observation.

□

6 Solving an S-DAG with Variable Elimination

An S-DAG is solved in almost the same manner as an influence diagram (Shachter, 1986; Shenoy, 1992; Jensen et al., 1994; Shenoy, 1994; Madsen and Jensen, 1999). Variables are eliminated in reverse order and when a branching point is met, the elimination is branched out. When several paths meet, the probability potentials are the same, and the utility potentials are unified through maximization.

Using the method of Jensen and Vomlelova (2002) this results in the following operations; all potentials are initially divided into two sets, one that includes the probability potentials, Φ, and one that includes the utility potentials, Ψ.

When eliminating a variable, N, the potential sets are modified in the following way. First we identify the sets

$$\Phi_N \leftarrow \{\phi \in \Phi | N \in \mathrm{dom}(\phi)\};$$
$$\Psi_N \leftarrow \{\psi \in \Psi | N \in \mathrm{dom}(\psi)\}.$$

It is from these sets that N is eliminated resulting in the potentials ϕ_N and ψ_N. If N is a chance variable, then

$$
\begin{aligned}
\phi_N &\leftarrow \sum_N \prod \Phi_N; \\
\psi_N &\leftarrow \sum_N \prod \Phi_N \left(\sum \Psi_N\right),
\end{aligned}
\tag{1}
$$

where the notations $\prod \Phi$ and $\sum \Psi$ denote the product of all probability potentials in Φ and the sum of all utility potentials in Ψ, respectively.

If N is a decision variable, then [3]

$$
\begin{aligned}
\phi_N &\leftarrow \max_N \prod \Phi_N; \\
\psi_N &\leftarrow \max_N \prod \Phi_N \left(\sum \Psi_N\right).
\end{aligned}
\tag{2}
$$

Finally, the potentials that did not include N in their domains are joined with the potentials resulting from the elimination:

$$
\begin{aligned}
\Phi &\leftarrow (\Phi \backslash \Phi_N) \cup \{\phi_N\}; \\
\Psi &\leftarrow (\Psi \backslash \Psi_N) \cup \left\{\frac{\psi_N}{\phi_N}\right\}.
\end{aligned}
\tag{3}
$$

With the above procedure for eliminating a variable, it is now possible to eliminate variables along the branches of an S-DAG; at branching points Φ and Ψ are copied, so that the variable elimination can continue along each branch, and at points where branches meet, the potentials in Ψ are unified by means of max-combination:

$$\Psi \leftarrow \left\{\max\left(\sum \Psi^1, \ldots, \sum \Psi^n\right)\right\},$$

where n is the number of branches that connect at a specified node in the S-DAG and Ψ^i is the set Ψ of branch i. There is no need to join sets of probability potentials, as their products are identical for each branch that meet in a point. This is due to the fact that they are all the result of sum-marginalizing out the same set of variables from the same set of potentials. As sum-marginalizations can be commuted, all the branches must give the same result, and an arbitrary set can be used for the following marginalizations.

The procedure above determines the maximal expected utility of an optimal strategy, but is easily modified to also specify an optimal strategy.

[3] The potentials in Φ_N are constant over N and the first max-operation in (2) can therefore be replaced by any other operator fixing the state of N.

7 Solving an S-DAG by Conditioning

The variable elimination (VE) algorithm solves a UID by eliminating the variables in reverse temporal order as specified by the S-DAG. This approach may give rise to (at least) two potential problems. First of all, if the decision maker stops the algorithm prematurely (due to time constraints), she would most likely be interested in a solution that at least covers the first decision in the UID. Unfortunately, the first decision in the UID is also the last decision being visited by the VE algorithm. Secondly, if a large part of the past is relevant for a particular decision we may end up with intermediate potentials that are intractably large.

To address these problems, we may look for alternative solution methods. For example, rather than going in the reverse temporal order, the UID may be solved by recursively "unfolding" the model into a decision tree using a depth-first search procedure. When a "call" returns to a chance node we simply add the returned value to any previous value associated with that node, and when a "call" returns to a decision node we keep the existing value if it is larger than the returned value, otherwise we keep the returned value. The probabilities for the decision tree may be found from the probability model embedded in the UID; for the purpose of calculating the probabilities required by the decision tree, the UID can be treated as a Bayesian network.

This algorithm has two potential advantages over VE: the algorithm starts with the first decision, hence by devising an appropriate heuristic function for estimating the expected utility at any node in the tree, we may get an any-time algorithm that always returns a solution for the first decision. [4] Moreover, due to the way in which the tree is simultaneously expanded and solved, we only need to store a single number for each variable in the UID including branch point decisions (disregarding the space required for calculating the probabilities). Thus, we have a linear space algorithm for solving UIDs. The drawback of this procedure is that the time complexity is exponential in the number of variables. In particular, by using the decision tree framework as the underlying computational structure we are faced with complexity problems of at least the same magnitude as when influence diagrams are unfolded into decision trees. This problem points towards another framework for organizing the calculations: instead of working on the decision tree representation of the UID, we take outset in the corresponding S-DAG and perform the search/conditioning in this structure; by conditioning we refer to the instantiation of the past of a node in the S-DAG.

In what follows we describe a linear-space solution procedure, called S-DAG

[4] Devising such a heuristic function is a subject of ongoing research.

conditioning (SC), which follows the approach outlined above. The algorithm also incorporates a space efficient algorithm for calculating the required probabilities, and it will subsequently be extended with a cache that will effectively make it an any-space algorithm. [5]

7.1 Initializing the S-DAG Structure

When performing S-DAG conditioning we first initialize the S-DAG with the probability and utility potentials from the UID: by traversing all paths from *sink* to *source*, a pointer from a node $\mathcal{X}$ to a potential is introduced if *i*) the potential contains a variable from $\mathcal{X}$ in its domain, and *ii*) it has not been assigned to another node on the path between $\mathcal{X}$ and *sink*. We shall use $\Phi_{\mathcal{X}}$ and $\Psi_{\mathcal{X}}$ to denote the probability potentials and the utility potentials assigned to node $\mathcal{X}$; for each $\phi \in \Phi_{\mathcal{X}}$ we have $\mathrm{dom}(\phi) \subseteq \mathrm{Past}(\mathcal{X}) \cup \{\mathcal{X}\}$ (similarly for each $\psi \in \Psi_{\mathcal{X}}$).

Note that based on the initialized S-DAG structure it is easy to verify that the maximum expected utility $\rho_{\mathcal{D}_k}$ for any decision node $\mathcal{D}_k$ (as also calculated by the VE algorithm) can be written as (see Figure 12)

$$\rho_{\mathcal{D}_k}(\mathrm{Past}(\mathcal{D}_k)) = \max_{\mathcal{D}_k}\left(\sum \Psi_{\mathcal{D}_k} + \sum_{\mathcal{C}_k} P(\mathcal{C}_k \mid \mathrm{Past}(\mathcal{D}_k), \mathcal{D}_k)\left[\sum \Psi_{\mathcal{C}_k} + \rho_{\mathcal{D}_{k+1}}\right]\right), \tag{4}$$

where $\rho_{\mathcal{D}_{k+1}}$ is the maximum over the expected utility potentials $\{\rho_{\mathcal{D}^1_{k+1}}, \rho_{\mathcal{D}^2_{k+1}}, \ldots, \rho_{\mathcal{D}^l_{k+1}}\}$ for the children of $\mathcal{C}_k$.

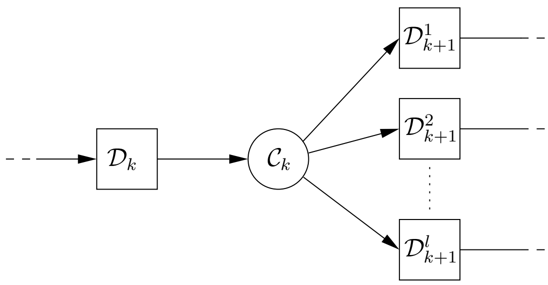


Fig. 12. A partial S-DAG.

7.2 Performing S-DAG Conditioning

In S-DAG conditioning we start in *source* and move towards *sink* following a depth first search. Every time a new node $\mathcal{X}$ is reached we iteratively consider

[5] The algorithm is inspired by the version of recursive conditioning for Bayesian networks introduced by Darwiche (2001).

all configurations over the variables in $\mathcal{X}$ and calculate their contribution to the expected utility.

Assume now that the recursion has reached a decision node $\mathcal{D}_k$ and that the variables in $\text{Past}(\mathcal{D}_k)$ are instantiated to $\text{past}(\mathcal{D}_k)$. In order to calculate the expected utility at $\mathcal{D}_k$ we should evaluate

$$\sum \Psi_{\mathcal{D}_k} + \sum_{\mathcal{C}_k} P(\mathcal{C}_k \mid \text{past}(\mathcal{D}_k), \mathbf{d}_k) \cdot \left(\sum \Psi_{\mathcal{C}_k} + \rho_{\mathcal{D}_{k+1}}\right) \tag{5}$$

for each instantiation $\mathbf{d}_k$ of the variables in $\mathcal{D}_k$ and select the maximum value (see Equation 4).

For the first part of the expression above, we have that all variables down to $\mathcal{D}_k$ are instantiated, and $\sum \Psi_{\mathcal{D}_k}$ is therefore reduced from a sum of potentials to a sum of terms from the potentials in $\Psi_{\mathcal{D}_k}$. For the second part, consider the chance node $\mathcal{C}_k$ following $\mathcal{D}_k$. The chance variables in this node are marginalized out by evaluating (and summing together) the following expression for each instantiation of $\mathcal{C}_k$:

$$P(\mathbf{c}_k \mid \text{past}(\mathcal{D}_k), \mathbf{d}_k) \cdot \left(\sum \Psi_{\mathcal{C}_k} + \rho_{\mathcal{D}_{k+1}}\right). \tag{6}$$

As in (5), the operation $\sum \Psi_{\mathcal{C}_k}$ is a sum of lookups in the original potentials. Moreover, the expected utility $\rho_{\mathcal{D}_{k+1}}$ is found by taking the maximum of $\rho_{\mathcal{D}^1_{k+1}}$, $\ldots, \rho_{\mathcal{D}^l_{k+1}}$, where $\mathcal{D}^1_{k+1}, \ldots \mathcal{D}^l_{k+1}$ are the children of $\mathcal{C}_k$; each of these terms are evaluated in the same way as (5).

The recursion stops at *sink*, and when the procedure returns to *source* we have the maximum expected utility of the UID. From the description above we also see that we only need to store a single number for each node being visited, hence the expression in Equation 4 is reduced from (space consuming) operations on potentials to operations working on the "entry-level" of the potentials. Thus, we have a linear space algorithm for solving UIDs (not including the space used for representing the UID and the normal form S-DAG). [6]

7.3 Calculation of Probabilities

The calculations above require the conditional probabilities $P(\mathbf{c}_k \mid \text{past}(\mathcal{C}_k))$, for all instantiations $\mathbf{c}_k$ of $\mathcal{C}_k$ and for all k. These probabilities can be calculated as

$$P(\mathbf{c}_k \mid \text{past}(\mathcal{C}_k)) = \frac{P(\mathbf{c}_k, \text{past}(\mathcal{C}_k))}{\sum_{\mathbf{c}_k} P(\mathbf{c}_k, \text{past}(\mathcal{C}_k))}.$$

[6] It is still an open problem how to avoid holding the full normal form S-DAG in memory.

For the last chance node in an S-DAG the probability is given by

$$P(\mathbf{c}_n \mid \text{past}(\mathcal{C}_n)) = \frac{\prod \Phi_{\mathcal{C}_n}(\mathbf{x}_n)}{\sum_{\mathcal{C}_n} \prod \Phi_{\mathcal{C}_n}(\mathbf{x}_n)}, \tag{7}$$

where $\mathbf{x}_n$ is the recorded instantiation of the potentials in $\Phi_{\mathcal{C}_n}$. By iterating over all the configurations of $\mathcal{C}_n$ and calculating the numerator, the denominator can be found as the sum of the numerators. Thus, we do not need to calculate the denominator separately. For the second last chance node $\mathcal{C}_{n-1}$ the probabilities are given by

$$P(\mathbf{c}_{n-1} \mid \text{past}(\mathcal{C}_{n-1})) = \frac{\prod \Phi_{n-1}(\mathbf{x}_{n-1}) \left[\max_{\mathcal{D}_n} \sum_{\mathcal{C}_n} \prod \Phi_n(\mathbf{x}_n)\right]}{\sum_{\mathcal{C}_{n-1}} \prod \Phi_{n-1}(\mathbf{x}_{n-1}) \left[\max_{\mathcal{D}_n} \sum_{\mathcal{C}_n} \prod \Phi_n(\mathbf{x}_n)\right]}.$$

It is worth emphasizing that these calculations simply mimics the probability calculations done by variable elimination. Note also that the maximization is vacuous, since the involved potentials are constant over the decision variables in question (see Equation 2).

In general, the probabilities for $\mathcal{C}_k$ can be calculated (recursively) from the potentials assigned to $\mathcal{C}_k$ together with the probability (the denominator) calculated at the previous step:

$$P(\mathbf{c}_k \mid \text{past}(\mathcal{C}_k)) = \frac{\prod \Phi_k(\mathbf{x}_k)\ p_{k+1}}{\sum_{\mathcal{C}_k} \prod \Phi_k(\mathbf{x}_k)\ p_{k+1}} \tag{8}$$

where

$$p_{k+1} = \max_{D_{k+1}} \sum_{\mathcal{C}_{k+1}} \prod \Phi_{k+1}(\mathbf{x}_{k+1}) \cdots \max_{D_n} \sum_{\mathcal{C}_n} \prod \Phi_n(\mathbf{x}_n). \tag{9}$$

Observe that p_{k+1} is the potential/value obtained by eliminating all variables following $\mathcal{C}_k$; in that way the procedure above resembles updating steps 1 and 3 in the VE algorithm (except that this calculation should be done for all chance variables due to the recursion).

From the above expression we see that the space required for calculating the necessary probabilities is linear in the number of variables.

The low space complexity is achieved at the expense of the time complexity of the algorithm: For each node in the S-DAG and for each configuration of its past, we calculate both the expected utility and the probability for that configuration. Since these calculations are performed without reusing previous results, all the expected utility calculations involve iterating over all configurations and admissible orderings (bounded by the number of paths in the S-DAG) of the future variables for the node in question; the same holds for the probability calculations except that the ordering is irrelevant. An upper bound on the time complexity is therefore given by $O(n \cdot (\text{nPath} \cdot \exp(\text{nDec} + \text{nObs} + \text{nHid}) + \exp(\text{nObs} + \text{nHid}))$, where n is the number of nodes in the

S-DAG, nPath is the number of different directed paths from *source* to *sink*, nDec is the number of decision variables, nObs is the number of observable chance variables, and nHid is the number of unobservable chance variables.

7.4 Pseudo Code for the S-DAG Conditioning Algorithm

The full SC-algorithm (described above) is summarized in Algorithm 1 and Algorithm 2; Algorithm 1 handles the probability calculations and Algorithm 2 implements the expected utility calculations.

Algorithm 1 marginalizes out chance and decision variables from the probability potentials. The most basic function is LookUpProb, which performs a simple lookup in the relevant probability tables according to the recorded instantiation (globally defined). The functions SumMarProb and MaxMarProb correspond to steps 1 and 3 in the VE algorithm (i.e., the first part of (1) and (2), respectively). In SumMarProb we calculate the probability of the recorded past for chance node $\mathcal{C}$. The calculations depend on whether $\mathcal{C}$ has any temporal successors: if no temporal successors exists (i.e., $\mathcal{C}$ is *sink*), then we can directly marginalize out $\mathcal{C}$ (lines 5–6). Otherwise we first (recursively) eliminate the successor nodes before marginalizing out $\mathcal{C}$ (lines 8–9). In MaxMarProb we exploit that the past of a decision variable $\mathcal{D}$ is independent of $\mathcal{D}$. Thus, finding the probability $P(\text{past}(\mathcal{D}))$ basically amounts to picking an arbitrary state for $\mathcal{D}$ (line 1) and marginalizing out the variables in the future of $\mathcal{D}$ (lines 2-3).

Algorithm 2 consists of four sub-functions. SumMarUtil and MaxMarUtil perform the actual marginalizations of chance and decision variables, respectively.[7] The algorithm MaxCombUtil performs maximization at branching points (which, according to Proposition 4.2, are always chance nodes) and LookUpUtil performs a simple lookup in the relevant utility tables according to the recorded instantiation. Similar to SumMarProb, SumMarUtil considers two situations depending on whether $\mathcal{C}$ has any temporal successors. If $\mathcal{C}$ is *sink*, then for each state of $\mathcal{C}$ we find the relevant probability and utility by simply indexing the associated tables (lines 6–7). On the other hand, if $\mathcal{C}$ is not *sink*, then for each configuration of $\mathcal{C}$ we calculate the corresponding utility (line 9) by combining the associated utility with the expected utility of the recorded instantiation (calculated recursively using MaxCombUtil); similar calculations are performed when finding the associated probability (line 11). Finally, the expected utility of $\mathcal{C}$ is calculated (lines 12–13 and 15). The func-

[7] For notational convenience we assume that the algorithms are invoked on an NFS-DAG, which also guarantees that neighboring nodes will be of opposite type (disregarding *source* and *sink*). The proposed algorithm can, however, easily be adapted to handle general S-DAG structures.

tion MaxMarUtil marginalizes out decision variables by keeping track of the state having the highest expected utility (calculated recursively).

Algorithm 1 Calculates required probabilities

SumMarProb($\mathcal{C}$) - *Find* $P(\text{past}(\mathcal{C}))$

Input: $\mathcal{C}$ *- an S-DAG node containing chance variables*

$\mathbf{y} \leftarrow$ *the recorded instantiations*
$p \leftarrow 0$
for all *instantiations,* $\mathbf{c}$*, of* $\mathcal{C}$ **do**
  record instantiation $\mathbf{c}$
  if $\mathcal{C}$ *is sink* **then**
    $p \leftarrow p +$ LookupProb($\mathcal{C}$)
  else
    $\mathcal{D} \leftarrow$ *a child of* $\mathcal{C}$.
    $p \leftarrow p +$ LookupProb($\mathcal{C}$) $\cdot$ MaxMarProb($\mathcal{D}$)
  un-record instantiation $\mathbf{c}$
return p

MaxMarProb($\mathcal{D}$) - *Find* $P(\text{past}(\mathcal{D}))$

Input: $\mathcal{D}$ *- an S-DAG node containing decision variables*

Record any instantiation of the variables in $\mathcal{D}$
$\mathcal{C} \leftarrow$ *a child of* $\mathcal{D}$.
$p \leftarrow$ SumMarProb($\mathcal{C}$)
un-record the instantiation of variables in $\mathcal{D}$
return p

LookupProb($\mathcal{C}$)

Input: $\mathcal{C}$ *- an S-DAG node*

$\Phi_{\mathcal{C}} \leftarrow$ *the probability potentials associated with* $\mathcal{C}$.
$\mathbf{x} \leftarrow$ *the recorded instantiations of the potentials in* $\Phi_{\mathcal{C}}$.
return $\prod \Phi_{\mathcal{C}}(\mathbf{x})$

8 S-DAG Conditioning with Cache

During S-DAG conditioning, we may perform the same calculations several times. For example, the calculation of the maximum expected utility for a certain node $\mathcal{X}$ is independent of the ordering of the nodes in the past of $\mathcal{X}$. Thus, if there are several paths between *source* and $\mathcal{X}$, then the maximum expected utility for $\mathcal{X}$ will be the same for each path.

In this section we address this issue by extending the conditioning algorithm with a cache that allows calculations to be stored and reused, thus reducing the required number of recursions and thereby the runtime. Moreover, since we can control the amount of space reserved for cache, it is possible to make an explicit trade-off between time and space.

Algorithm 2 Determines the MEU for an S-DAG.

SumMarUtil($\mathcal{C}$)

Input: *$\mathcal{C}$ - an S-DAG node containing chance variables*

$\mathbf{y} \leftarrow$ *the recorded instantiations*
$eu, p, n, u \leftarrow 0$
for all *instantiations,* $\mathbf{c}$, *of* $\mathcal{C}$ **do**
  record instantiation $\mathbf{c}$
  if $\mathcal{C}$ *is sink* **then**
    $u \leftarrow$ LookupUtil($\mathcal{C}$)
    $p \leftarrow$ LookupProb($\mathcal{C}$)
  else
    $u \leftarrow$ LookupUtil($\mathcal{C}$) + MaxCombUtil($\mathcal{C}$)
    $\mathcal{D} \leftarrow$ *a child of* $\mathcal{C}$.
    $p \leftarrow$ LookupProb($\mathcal{C}$) $\cdot$ MaxMarProb($\mathcal{D}$)
  $eu \leftarrow eu + p \cdot u$
  $n \leftarrow n + p$.
  un-record instantiation $\mathbf{c}$
return eu/n

MaxMarUtil($\mathcal{D}$)

Input: *$\mathcal{D}$ - an S-DAG node containing decision variables*

$\mathbf{y} \leftarrow$ *the recorded instantiations*
$\mathcal{C} \leftarrow$ *the S-DAG node containing chance variables released by* $\mathcal{D}$
$v \leftarrow -\infty$
for all *instantiations,* $\mathbf{c}$, *of* $\mathcal{D}$ **do**
  record instantiation $\mathbf{c}$.
  $v \leftarrow \max\{v,$ LookupUtil($\mathcal{D}$) + SumMarUtil($\mathcal{C}$)$\}$
  un-record instantiation $\mathbf{c}$.
return v

MaxCombUtil($\mathcal{C}$)

Input: *$\mathcal{C}$ - an S-DAG node containing chance variables*

$v \leftarrow -\infty$
for all *child nodes,* $\mathcal{D}_i$, *of* $\mathcal{C}$ **do**
  $v \leftarrow \max\{v,$ MaxMarUtil($\mathcal{D}_i$)$\}$
return v

LookupUtil($\mathcal{X}$)

Input: *$\mathcal{X}$ - an S-DAG node*

$\Psi_{\mathcal{X}} \leftarrow$ *the utility potentials associated with* $\mathcal{X}$.
$\mathbf{x} \leftarrow$ *the recorded instantiations of the potentials in* $\Psi_{\mathcal{X}}$.
return $\sum \Psi_{\mathcal{X}}(\mathbf{x})$.

8.1 Caching Expected Utility Calculations

Consider a node $\mathcal{X}$ in an S-DAG, and let $\mathbf{x}_i$ and $\mathbf{x}_j$ be two instantiations of the past for $\mathcal{X}$ differing only on the state of a variable $Y \in \text{Past}(\mathcal{X})$. If the maximum expected utility for $\mathcal{X}$ is the same for both $\mathbf{x}_i$ and $\mathbf{x}_j$, then we can cache the result for $\mathbf{x}_i$ and reuse that result instead of invoking the algorithm on $\mathbf{x}_j$. To determine when calculations can be reused involves identifying the variables in the past of a node, say $\mathcal{X}$, that do not influence the expected utility at $\mathcal{X}$.

Definition 8 *Let X and Y be two variables in a UID and let $\prec$ be an admissible ordering with $Y \prec X$. Y is said to be* EU-relevant *for X w.r.t. $\prec$ if there exists a realization and a configuration $\mathbf{c} \in \mathrm{sp}(\mathrm{Past}(X)_{\prec} \setminus \{Y\})$ s.t.*

$$\rho_X^{\prec}(\mathbf{c}, y_i) \neq \rho_X^{\prec}(\mathbf{c}, y_j) \text{ for some } y_i, y_j \in \mathrm{sp}(Y),$$

where $\rho_X^{\prec}$ is the expected utility of X under the admissible ordering $\prec$.

We say that Y is EU-relevant for X if there exists an admissible ordering $\prec$ so that Y is EU-relevant for X w.r.t. $\prec$. The set of variables being EU-relevant for X under $\prec$ is called the *EU-Context* for X w.r.t. $\prec$ (denoted by $\mathrm{EU\text{-}Context}(X)^{\prec}$), and the set of EU-relevant variables is denoted $\mathrm{EU\text{-}Context}(X)$. Finally, if $\mathcal{X}$ is a set of variables, then Y is EU-relevant for $\mathcal{X}$ if Y is EU-relevant for some $X \in \mathcal{X}$. The configurations over the variables in $\mathrm{EU\text{-}Context}(\mathcal{X})$ therefore constitute the necessary and sufficient set of configurations over the past of $\mathcal{X}$ for which we need to calculate the expected utility.

In order to specify a syntactical characterization of the EU-Context, we note that if a past variable Y should influence the expected utility at $\mathcal{X}$ it should (loosely speaking) influence a utility function being relevant for either $\mathcal{X}$ or a variable in the future of $\mathcal{X}$. The set of variables influencing a utility function relevant for $\mathcal{X}$ is called the *required past* for $\mathcal{X}$ (denoted by $\mathrm{Req}(\mathcal{X})$), see below for details. Based on the notion of required past we have the following proposition.

Proposition 8.1 *Let I be a UID and let $\mathcal{X}$ be a node in an S-DAG representing I. Then*

$$\textit{EU-Context}(\mathcal{X}) = \left(\bigcup_{\mathcal{Y} \in \mathrm{Ftr}(\mathcal{X})} \mathrm{Req}(\mathcal{Y}) \right) \cap \mathrm{past}(\mathcal{X}), \tag{10}$$

where $\mathrm{Ftr}(\mathcal{X})$ are the nodes (including $\mathcal{X}$) on the path from $\mathcal{X}$ to sink.

In order to establish a precise definition of the required past, consider a UID with a variable X. The set of variables required for X depends on the ordering of the variables succeeding X. Thus, each admissible ordering $\prec$ of the succeeding variables defines a set of required variables $\mathrm{Req}(X)^{\prec}$, and the union of these sets corresponds to the variables $\mathrm{Req}(X)$ required for X. Since a UID with an admissible ordering is basically an ID, we can use a specification of the required variables in IDs to characterize the required variables in UIDs: each path from a node $\mathcal{X}$ to *sink* represents an ordering, so to find the required past for $\mathcal{X}$ we identify the required past for $\mathcal{X}$ for each of these orderings. Precise definitions as well as algorithms for performing this type of analysis in IDs are described in (Nielsen and Jensen, 1999; Shachter, 1999; Lauritzen and Nilsson, 2001), and since they are easily adapted to UIDs we have chosen not to include their (slightly) modified versions in this paper. Note that the required variables are described for decision variables only, but the definitions can easily be generalized to chance variables.

8.2 Caching Probability Calculations

First of all, recall that Equation 8 specifies the calculation of the probability for chance node $\mathcal{C}_k$. In this expression the factor p_{k+1} is found according to Equation 9, which in turn corresponds to the denominator of Equation 8 calculated at $\mathcal{C}_{k+1}$. Thus by storing the denominator p_{k+1} found at $\mathcal{C}_{k+1}$ it will not be necessary to recalculate Equation 9 when the recursion returns to $\mathcal{C}_k$.

As for the calculation of expected utilities, the probabilities are calculated based on a specific instantiation of the past variables. However, for probability calculations, the set of variables defining the context for a node (called the P-Context) is slightly different from the EU-Context for that node. More precisely, the P-Context for a node $\mathcal{C}$ is the set of nodes in the past of $\mathcal{C}$ that may influence the conditional probability of either $\mathcal{C}$ or a descendant of $\mathcal{C}$. If $\mathrm{Req}_P(\mathcal{C}) \subseteq \mathrm{Past}(\mathcal{C})$ denotes the variables on which $\mathcal{C}$ is conditionally dependent given the remaining variables in its past, then we have:

$$\text{P-Context}(\mathcal{C}) = \left(\bigcup_{\mathcal{X} \in Ftr(\mathcal{C}) \,\wedge\, \mathcal{X} \text{ is a chance node}} \mathrm{Req}_P(\mathcal{X}) \right) \cap \mathrm{Past}(\mathcal{C}).$$

For the syntactical characterization of the variables being required for a chance variable we first note that this set is independent of the ordering of both the future and the past variables; all variables in the past are instantiated and the future chance variables may be commuted (no future decision can influence the probability of the node in question). Based on this observation we have that:

$$\mathrm{Req}_P(\mathcal{C}) = \{Y \,|\, Y \in \mathrm{Past}(\mathcal{C}) \text{ and } Y \text{ is d-connected to } \mathcal{C} \text{ given } \mathrm{Past}(\mathcal{C}) \setminus \{Y\}\}.$$

Finally, it should be noted that this characterization may include redundant variables. For example, $Y \in \text{P-Context}(X)$ is redundant if Y and X may be commuted without affecting the expected utility, or if Y is included in

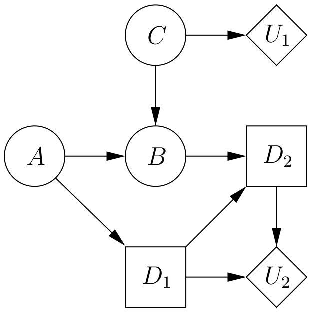


Fig. 13. A is required for C under $\{A\} \prec D_1 \prec \{B\} \prec D_2 \prec \{C\}$, but A is not required for C under the EU-equivalent ordering $\{A\} \prec D_1 \prec D_2 \prec \{B, C\}$.

P-Context(X) because $Y \in \text{Req}_P(Z)$ and Z is effectively barren (i.e., it can be marginalized out without affecting the expected utility).

Example 2 *A UID with an admissible ordering of the variables basically corresponds to an ID, hence we can consider the ID depicted in Figure 13 as a UID. This model specifies the ordering $\{A\} \prec D_1 \prec \{B\} \prec D_2 \prec \{C\}$, and according to the characterization above we find that $A \in$ P-Context(C). However, B and D_2 can be commuted since B is not required for D_2, in which case we get the ordering $\{A\} \prec D_1 \prec D_2 \prec \{B, C\}$, and for this ordering we find that $A \notin$ P-Context(C).*

8.3 Extending the conditioning algorithm with cache

The proposed conditioning algorithm can easily be changed to take advantage of a cache. Basically we simply need to check for existing cache entries in the beginning of each algorithm and, if required, add the result of the algorithm to the cache prior to returning. In Algorithm 2, the cache check is accomplished by including the following line of code between lines 1 and 2 in both MaxMarUtil($\mathcal{X}$) and SumMarUtil($\mathcal{X}$).

if EU-cache$_{\mathcal{X}}$ $[\mathbf{y}] \neq$ nil **then return** EU-cache$_{\mathcal{X}}$ $[\mathbf{y}]$

Any result calculated in MaxMarUtil($\mathcal{X}$) may be added to the cache by including the following line of code between lines 7 and 8 (EU-cache? $(\mathcal{X}, \mathbf{y})$ represents the caching strategy determining whether a result should be cached).

if EU-cache? $(\mathcal{X}, \mathbf{y})$ **then** EU-cache$_{\mathcal{X}}$ $[\mathbf{y}] \leftarrow v$

SumMarUtil($\mathcal{X}$) also includes the above command, but in addition it also includes the following command (between lines 14 and 15) that populates the probability cache (basically storing the value of the denominator from Equation 8 in the cache).

if P-cache? $(\mathcal{X}, \mathbf{y})$ **then** P-cache$_{\mathcal{X}}[\mathbf{y}] \leftarrow n$

For Algorithm 1, cache-handling is done by SumMarProp($\mathcal{X}$), and it needs only to implement a cache check since the probability cache is populated by SumMarUtil($\mathcal{X}$). In order to accomplish this, the following command should be included between lines 1 and 2.

if P-cache$_{\mathcal{X}}$ [$\mathbf{y}$] $\neq$ nil **then return** P-cache$_{\mathcal{X}}$ [$\mathbf{y}$]

8.4 Complexity of the SC Algorithm with Cache

An upper bound for the space complexity of the SC algorithm with full cache is the size of the contexts for all the S-DAG nodes. For the time complexity we note that the number of calls made to a given node is the sum of the number of calls made from each of its parents. For a parent node we make a call for each configuration of the variables in that node and its context. An upper bound on the time complexity is therefore $O(n \cdot |\text{nParents}|^* \cdot \exp(|\text{Context} + \text{Node}|^*))$, where $|\text{nParents}|^*$ is the maximum number of parents for an S-DAG node, and $|\text{Context} + \text{Node}|^*$ is the maximum number of variables in the context of the nodes and the variables in the nodes itself.

8.5 Caching Strategies

The EU-Context and the P-Context define the variable configurations for which calculations may be repeated and therefore the configurations for which results may be cached and possibly reused. Since not all results should necessarily be cached (e.g. due to memory constraints), we introduce a *caching function* specifying the proportion of the context for a given node that will have a cache attached; the notion of a caching function is inspired by Darwiche (2001).

Definition 9 *A* caching function *for an S-DAG is a function* cf *that for each node $\mathcal{X}$ in the S-DAG returns a number, $0 \leq cf(\mathcal{K}) \leq 1$. For a given node $\mathcal{K}$, the caching function specifies a* caching factor cf($\mathcal{K}$)*.*

Since we are dealing with two types of contexts, we use ucf and pcf to refer to the caching functions for the EU-Contexts and the P-Contexts, respectively. The caching factor ucf($\mathcal{X}$) specifies the fraction of $|\,\text{sp}(\text{EU-Context}(\mathcal{X}))|$, for which we allocate space in memory. Using EU-Context$(\mathcal{X})^{\#}$ as a shorthand for $|\,\text{sp}(\text{EU-Context}(\mathcal{X}))|$, we get the following for the total size of the cache for node $\mathcal{X}$

$$\begin{aligned} |\text{EU-cache}_{\mathcal{X}}| &= \text{ucf}(\mathcal{X}) \cdot \text{EU-Context}(\mathcal{X})^{\#}; \\ |\text{P-cache}_{\mathcal{X}}| &= \text{pcf}(\mathcal{X}) \cdot \text{P-Context}(\mathcal{X})^{\#}. \end{aligned} \tag{11}$$

We may employ different strategies for assigning cache (i.e., determining ucf and pcf), and in the following we will describe two such strategies.

8.5.1 Naive Caching

A simple caching strategy could be to give all nodes the same caching factor. That is, if Mem is the total amount of memory available, then the caching function is given by

$$\text{ucf}(\cdot) = \frac{\text{Mem}}{\sum_{\mathcal{X}} \text{EU-Context}(\mathcal{X})^{\#} + \sum_{\mathcal{X}} \text{P-Context}(\mathcal{X})^{\#}}.$$

This naive algorithm is introduced merely as a point of reference for the following algorithm.

8.5.2 Greedy Caching

The main aim of caching is to reduce the runtime by increasing the amount of space available to the conditioning algorithm. Since the runtime of the algorithm is proportional to the total number of lookups in the probability and utility potentials (see Section 9.1), we look for a caching strategy that seeks to minimize this number.

In this section we define a greedy caching strategy that repeatedly assigns a full cache to the node for which we obtain the greatest reduction in the total number of lookups relative to the size of the cache. [8]

In general, the total number of lookups is the number of lookups for the expected utility calculations and the number of lookups for the probability calculations:

$$\text{TotalReads}_{\text{ucf,pcf}} = \text{EU-TotalReads}_{\text{ucf}} + \text{P-TotalReads}_{\text{ucf,pcf}}. \tag{12}$$

The total number of lookups for the expected utility calculations is the sum of the number of lookups at each node:

$$\text{EU-TotalReads}_{\text{ucf}} = \sum_{\mathcal{X}} \text{EU-Reads}(\mathcal{X})_{\text{ucf}}.$$

[8] This algorithms is inspired by the greedy algorithm of Allen et al. (2004).

The value of EU-Reads$(\mathcal{X})_{\text{ucf}}$ is the sum of the number of lookups performed at each call to $\mathcal{X}$. A single call to $\mathcal{X}$ involves a cache check and if a cache-entry is not found the algorithm iterates over the configurations of the variables in $\mathcal{X}$; at each iteration we make a lookup in the potentials in $\Phi_{\mathcal{X}}$ and $\Psi_{\mathcal{X}}$. The proportion of the number of calls for which a cache-entry will not be found is $(1 - \text{ucf})$, and since the number of lookups required to fill the cache is $|\text{EU-cache}_{\mathcal{X}}| \cdot \mathcal{X}^{\#} \cdot (|\Phi_{\mathcal{X}}| + |\Psi_{\mathcal{X}}|)$ we get:

$$\begin{aligned}\text{EU-Reads}(\mathcal{X})_{\text{ucf}} = &\text{EU-CallTo}(\mathcal{X})_{\text{ucf}}(1 + (1 - \text{ucf})\mathcal{X}^{\#}(|\Phi_{\mathcal{X}}| + \\ &|\Psi_{\mathcal{X}}|)) + |\text{EU-cache}_{\mathcal{X}}|\mathcal{X}^{\#}(|\Phi_{\mathcal{X}}| + |\Psi_{\mathcal{X}}|).\end{aligned}$$

If $\mathcal{X}$ is not the source node, then the total number of calls to $\mathcal{X}$ is the sum of the number of calls to $\mathcal{X}$ made from each of its parents $\text{pa}(\mathcal{X})$ in the S-DAG. For each parent $\mathcal{P}$ we make $\mathcal{P}^{\#}$ calls for each cache entry at $\mathcal{P}$ together with $\mathcal{P}^{\#}$ calls for each call to $\mathcal{P}$ that is not covered by the cache (there will be $(1 - \text{ucf}(\mathcal{P}))\text{EU-CallTo}(\mathcal{P})_{\text{ucf}}$ such calls). Thus we have

$$\text{EU-CallTo}(\mathcal{X})_{\text{ucf}} = \begin{cases} 1 & \text{if } \mathcal{X} = \textit{source} \\ \sum_{\mathcal{P}\in\text{pa}(\mathcal{X})} \Big([1 - \text{ucf}(\mathcal{P})]\ \text{EU-CallTo}(\mathcal{P})_{\text{ucf}}\ \mathcal{P}^{\#} & \\ \qquad\qquad + |\text{EU-cache}_{\mathcal{P}}|\ \mathcal{P}^{\#}\Big) & \text{else.} \end{cases}$$

The derivations above do not cover the number of lookups needed to perform probability updating. Probability updating is always initiated by the expected utility calculations at a chance node. When calculating a probability at a chance node $\mathcal{C}$, the algorithm performs lookups at all nodes on the path between $\mathcal{C}$ and *sink*:

$$\text{EUtoP-Reads}(\mathcal{C})_{\text{pcf}} = \sum_{\mathcal{Y}\in\text{Path}(\mathcal{C},\textit{sink})} \text{P-Reads}(\mathcal{Y})_{\text{pcf}}, \tag{13}$$

where $\text{Path}(\mathcal{C}, \textit{sink})$ is any path from $\mathcal{C}$ to *sink*. The value of P-Reads$(\mathcal{Y})_{\text{pcf}}$ is determined by Algorithm 1. Recall that this algorithm is not responsible for filling the P-cache and, in particular, for the node in question the cache is filled before the first call to Algorithm 1. The number of lookups for a node $\mathcal{X}$ is therefore

$$\text{P-Reads}(\mathcal{X})_{\text{pcf}} = \begin{cases} 0 & \text{if } \mathcal{X} \text{ is a dec.} \\ \text{P-CallTo}(\mathcal{X})_{\text{pcf}} \cdot \Big(1 + [1 - \text{pcf}(\mathcal{X})]\ \mathcal{X}^{\#}\ |\Phi_{\mathcal{X}}|\Big) & \text{else.} \end{cases}$$

If $\mathcal{X}$ is the start node for the current recursive call (for probability calculations) initiated in Equation 13, then P-CallTo$(\mathcal{X})_{\text{pcf}}$ is set to 1. Otherwise, the number of calls to $\mathcal{X}$ is (the derivation follows the derivation for Equation 8)

$$\text{P-CallTo}(\mathcal{X})_{\text{pcf}} = \begin{cases} 1 & \text{if } \mathcal{X} = \textit{source} \\ \text{P-CallTo}(\mathcal{P})_{\text{pcf}} & \text{if } \mathcal{P} \text{ is a dec.} \\ [1 - \text{pcf}(\mathcal{P})]\ \text{P-CallTo}(\mathcal{P})_{\text{pcf}} \mathcal{P}^{\#} + & \\ |\text{P-cache}_{\mathcal{P}}|\mathcal{P}^{\#} & \text{else,} \end{cases}$$

where $\mathcal{P}$ is a parent of $\mathcal{X}$.

Finally, the total number of lookups for probability calculations is the number of lookups required to perform the expected utility calculations at all the chance nodes:

$$\begin{aligned}\text{P-TotalReads}_{\text{ucf,pcf}} = \sum_{\mathcal{C} \in \mathcal{V}_C} \Big(& |\text{EU-cache}_{\mathcal{C}}|\mathcal{C}^{\#}\text{EUtoP-Reads}(\mathcal{D}_{\mathcal{C}}) + \\ & (1 - \text{ucf}(\mathcal{C}))\text{EU-CallTo}(\mathcal{C})_{\text{ucf}}\mathcal{C}^{\#}\text{EUtoP-Reads}(\mathcal{D}_{\mathcal{C}})_{\text{pcf}}) \Big),\end{aligned}$$

where $\mathcal{V}_C$ are the chance nodes in the S-DAG and $\mathcal{D}_{\mathcal{C}}$ is any decision child of $\mathcal{C}$.

We now have a specification of the number of lookups performed by the algorithm using caching functions ucf and pcf. In order to identify the caching functions that minimize the total number of lookups using the least amount of memory, we employ a greedy strategy: myopically change ucf and pcf so that a full cache is assigned to the node that gives the largest reduction in the total number of lookups relative to the size of the context/cache. This corresponds to changing either ucf or pcf so that a full cache is assigned to the context with largest score ($\text{ucf}_{\mathcal{X}}$ represent the caching function obtained from ucf by assigning a full cache to $\mathcal{X}$; similar for $\text{pcf}_{\mathcal{X}}$):

$$\begin{aligned}\text{EU-Score}_{\text{ucf,pcf}}(\mathcal{X}) &= \frac{\text{TotalReads}_{\text{ucf,pcf}} - \text{TotalReads}_{\text{ucf}_{\mathcal{X}},\text{pcf}}}{\text{EU-Context}(\mathcal{X})^{\#}} \\ \text{P-Score}_{\text{ucf,pcf}}(\mathcal{X}) &= \frac{\text{TotalReads}_{\text{ucf,pcf}} - \text{TotalReads}_{\text{ucf,pcf}_{\mathcal{X}}}}{\text{P-Context}(\mathcal{X})^{\#}},\end{aligned}$$

where $\text{ucf}(\mathcal{X}) = 0$, $\text{ucf}_{\mathcal{X}}(\mathcal{X}) = 1$, $\text{pcf}(\mathcal{X}) = 0$, $\text{pcf}_{\mathcal{X}}(\mathcal{X}) = 1$,. Note that after a node has been assigned a cache, the scores need to be recalculated. However, we may exploit that only the future of the node in question are affected by the cache assignment.

The greedy caching strategy terminates when all available memory has been assigned or if all nodes have zero score. In the latter case, no reduction in runtime can be achieved by using more memory on cache. Note that some scores may be zero even if they have no cache assigned. This can happen if e.g. the context of a node corresponds to its parent node and the context of that node.

Finally, it should be noted that the calculation of the scores can be implemented using dynamic programing starting at *source* and going towards *sink*.

9 Empirical Results

The proposed algorithms have been tested on a collection of randomly generated UIDs. The UIDs were generated using a modified version of the algorithm described by Ide and Cozman (2002) for generating Bayesian networks.[9] The modified algorithm is specified in Algorithm 3, and the generated UIDs are summarized in Table 1, where the UIDs are grouped together in sets with similar structural properties. For example, TS3 consists of 20 randomly generated UIDs, each of which has 3 decision nodes, 16 observable chance nodes, 2 hidden chance nodes, and 16 utility nodes. All nodes have an in-degree of 2 and the corresponding normal form S-DAG has 3 paths between *source* and *sink*. The two singleton test sets represent handcrafted models: PigsLegs4 is shown in Figure 14, and TT33 is similar to Figure 4, but with an extra test and treatment decision added. Observe that the S-DAG for PigsLegs4 contains a single path, which implies that the UID basically corresponds to an ID.

Algorithm 3 Random generation of UIDs.

```
UIDGenerator(nObs, nHid, nDec, nUtil, nDeg, nIt)
  Output: A random UID.
  Create a path with nObs observable chance nodes, nHid unobservable chance
  nodes, and nDec decision nodes. In this path no edge may go from an unobserv-
  able chance node to a decision node.
  for i = 1 to nIt do
    Select two different nodes s and p.
    if there is an edge between s and p then
      remove it provided that the graph remains connected.
    else
      Otherwise add an edge from s to p, provided that: (a) it will not make
      the graph cyclic, (b) p is not an unobservable chance node and s is not
      an decision, and (c) the number of children of s and parents of p does not
      extend nDeg.
  Add nUtil utility nodes with nDeg randomly selected parents (ensure that all
  decision nodes have a utility node as child). Add randomly generated utility
  functions to the utility nodes, and randomly generated probability potentials to
  the chance nodes.
```

All tests have been performed using a Standard Java 2 implementation extending the Elvira framework and running on a Windows XP Pro SP2 PC with a 1.7GHz Pentium 4 CPU and 512MB RAM.

[9] Following Ide and Cozman (2002) we used $\mathrm{nIt} = 6 \cdot (\mathrm{nObs} + \mathrm{nHid} + \mathrm{nDec})^2$ in the algorithm.

Name	nUID	nDec	nObs	nHid	nUtil	nDeg	nPath
TS3	20	3	16	2	16	2	3
TS4	20	4	8	12	4	3	3
TS7	20	7	14	7	7	3	12
TS16	20	16	4	2	3	6	1
PigsLegs4	1	12	8	5	13	3	1
TT33	1	6	7	5	7	7	360

Table 1
The test sets used for the empirical tests. nUID is the number of UIDs in the test set.

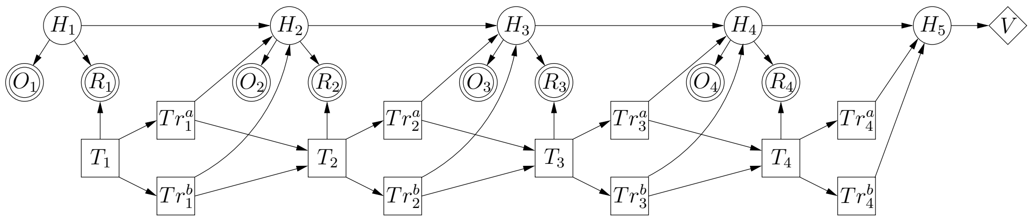


Fig. 14. A UID modeling test and treatment decisions for pigs with recurring leg problems. At time i, H_i represent the health state of the pig, O_i is a cost-free observation of symptoms, T_i is a test that produces the test result R_i, and Tr_i^a and Tr_i^b are possible treatments.

9.1 Estimating the Runtime of the SC Algorithm

The greedy caching algorithm is based on the assumption that the running time is proportional to the number to lookups in the probability and utility tables. To verify this, we plotted the running time and the number of lookups for each of the randomly generated UIDs. The results are shown in Figure 15 and confirm the assumption.

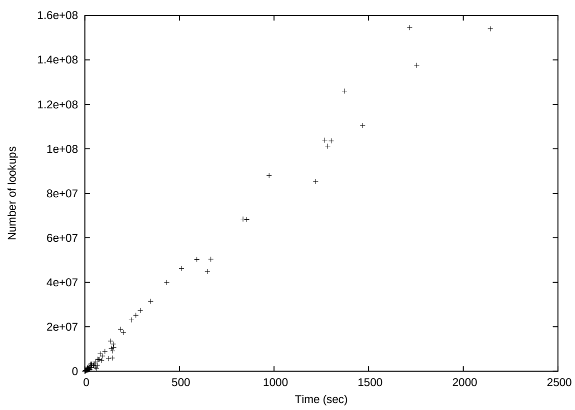


Fig. 15. The plot shows the running time versus the number of lookups in the probability and utility potentials for the randomly generated UIDs in Table 1.

As indicated by the plot, the number of lookups is a good predictor for the actual running time of the algorithm. For a given amount of memory we can therefore predict the running time of the algorithm, and thereby allow the user to make an explicit trade-off between time and space prior to running the algorithm.

9.2 Time-Space Trade-off for the SC Algorithms

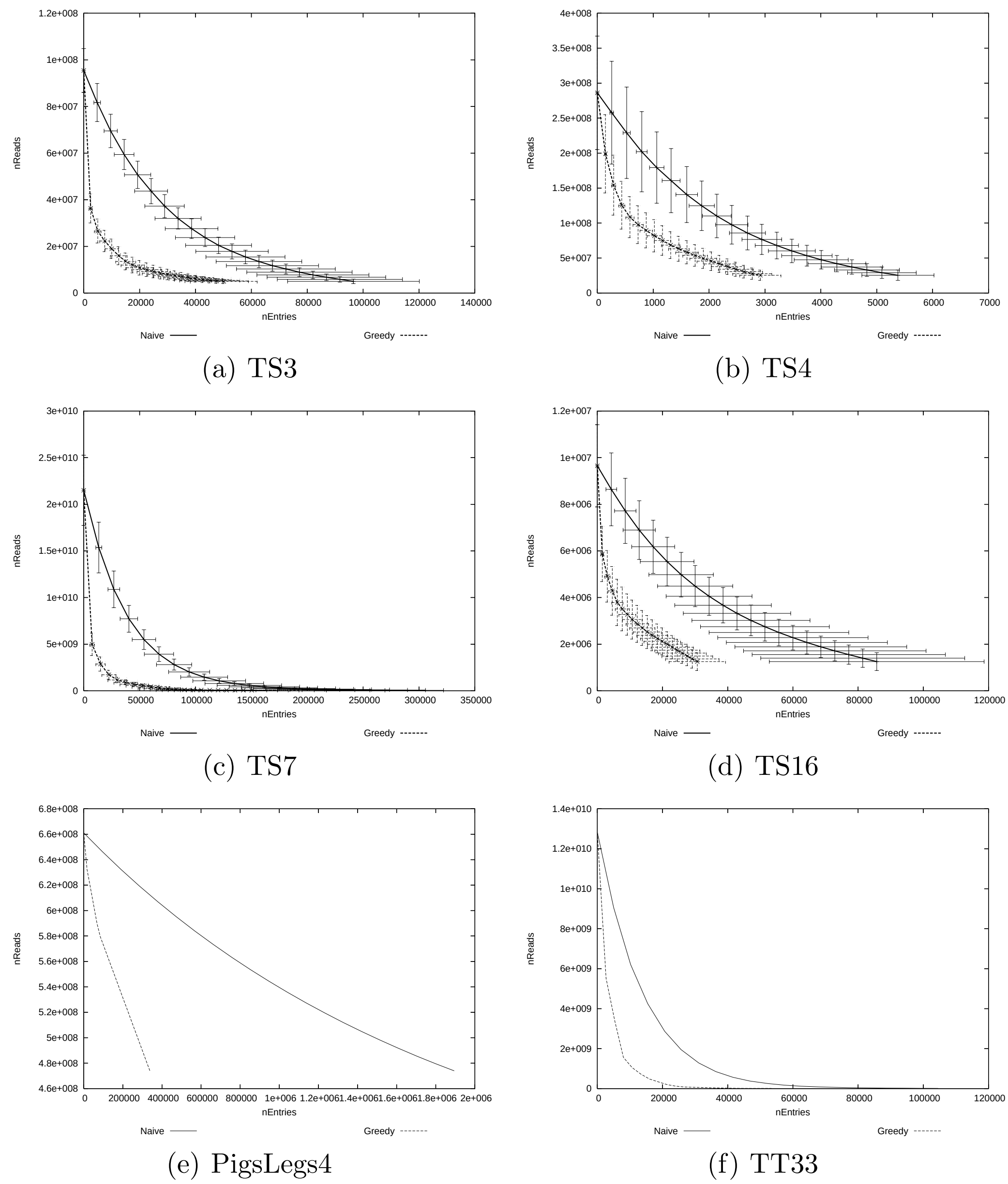
We have tested the performance of the greedy caching algorithm compared to the naive algorithm. Specifically, for each of the UIDs we have calculated the number of lookups as a function of the amount of memory available. This has been done for both algorithms, and the results are shown in Figures 16(a)-16(f). The graphs show that the greedy algorithm consistently outperforms the naive algorithm. The graphs also shows that the running time quickly drops, when you allow the algorithm to use just a small amount of space for caching.

9.3 VE vs. SC Using Full Cache

We have compared the variable elimination algorithm with the conditioning algorithm using full cache and for both types of caching strategies. The tests are intended to show to what extend the SC algorithms are competitive with variable elimination when memory usage is not a problem.

Table 2(a) shows the memory usage of the algorithms and Table 2(b) shows the time usage. The time reported is the time needed for calculating the MEU and the space is the maximum heap size allocated by Java during the calculation of the MEU. The latter includes the space for representing the UID, the normal form S-DAG, the potential sets Φ and Ψ, the temporary potentials produced by the variable eliminations (for the VE algorithm only), and the space used for cache (for the SC algorithm only). It should be noted that the SC algorithm (as described in Section 8.3) does not store the optimal policies, however, it is straight-forward to modify the algorithm so that space is reserved for a selected subset of these.

If we look at the memory usage of the algorithms we see that the greedy SC algorithm is at least as good as both the naive SC algorithm and the VE algorithm. The favorable space complexity of the SC algorithms (and the greedy algorithm in particular) can be explained by the removal of redundant variables during the structural analysis that establishes the contexts for the nodes; the contexts' sizes determine the maximum size of the cache. A similar analysis is not performed by the VE algorithm, and the VE algorithm may therefore produce intermediate potentials that include irrelevant variables.



(a) TS3 (b) TS4

(c) TS7 (d) TS16

(e) PigsLegs4 (f) TT33

Fig. 16. (nEntries, nReads)-graphs. The graphs shows the number of lookups in the probability and utility potentials as a function of the size of the cache.

Moreover, for VE the maximum size of a potential before eliminating an S-DAG node $\mathcal{X}$ is in the worst case $|\operatorname{Past}(\mathcal{X}) \cup \mathcal{X}|$. In comparison, for the SC algorithm the Context is in the worst case equal to $\operatorname{Past}(\mathcal{X})$. [10]

The UID PigsLegs4 is an example of a particularly difficult model, since the required past of a decision node equals the whole past for that node. This

[10] The problem of redundant variables is not specific for UIDs, but also occurs when solving e.g. influence diagrams using standard solution algorithms (see also Vomlelova and Jensen (2004)).

results in large potentials for the VE algorithm and large contexts for the SC algorithms; in general, the size of the contexts/potentials grows exponentially in the number of time slices in the model. However, it turns out that $\text{EU-Score}_{\text{ucf,pcf}}(\mathcal{X}) = 0$ for every node $\mathcal{X}$, which implies that every EU-cache is effectively "dead". Assigning cache to these nodes will therefore not reduce the running time (this is recognized and exploited in the greedy algorithm). In fact, in the PigsLegs4 example we got an `OutOfMemoryError` for the VE algorithm, and at the same time the greedy algorithm could find the MEU using only 0.243 Mb.

By considering the evaluation time we see that both SC algorithms perform almost equally, but the VE algorithm is significantly better than both of them (even though the theoretical time complexities are comparable). This is not surprising, since VE may exploit independences within the nodes in the S-DAG, and there is also a computational overhead involved in making the recursive calls performed by the SC algorithms.

10 Future Work

The proposed conditioning algorithms rely on a full specification of the S-DAG to be present in memory. For certain UIDs this may impose a memory problem in itself, since the size of the S-DAG may grow exponentially in the number of variables in the UID. An area of future research is to devise an algorithm for constructing the S-DAG from *source* to *sink*, thereby allowing the construction of the S-DAG to be interleaved with the SC-algorithm.

The current characterization of the P-Context (and the EU-Context for chance variables) is only unique up to the specified S-DAG. This also implies that in order to find a minimal P-Context we may need to investigate different S-DAG representations of the same model. The S-DAG definition imposes certain constraints on the ordering of the variables. For example, the S-DAG should not include misplaced variables. However, having misplaced variables does not necessarily influence the expected utility and this extra degree of freedom may provide a way to reduce the size of the contexts by allowing certain variables (of different types) to be commuted. It is a subject for future research to find a characterization of the contexts that do not include redundant variables.

Acknowledgments

We wish to thank the Machine Intelligence group at Aalborg University for fruitful discussions. We also thank the authors of the Elvira system for pro-

	Max heap size ($\cdot 10^6$ bytes)				Eval. time (sec.)			
Alg.	Avg.	Min.	Max.	Std.	Avg.	Min.	Max.	Std.
TS3								
VE	1.015	0.268	3.404	0.734	26.588	3.671	96.076	22.628
Naive	0.542	0.267	2.609	0.645	70.277	15.813	224.766	55.353
Greedy	0.318	0.267	0.751	0.133	67.416	14.766	204.234	51.981
TS4								
VE	0.219	0.207	0.360	0.034	0.860	0.063	5.688	1.258
Naive	0.209	0.206	0.212	0.001	278.244	0.734	1383.343	364.274
Greedy	0.209	0.206	0.212	0.001	278.129	0.687	1372.015	362.127
TS7								
VE	1.051	0.343	3.367	0.796	18.985	0.561	72.297	19.153
Naive	1.256	0.244	7.191	1.830	785.293	11.500	2114.672	699.235
Greedy	0.898	0.239	4.901	1.377	781.980	11.750	2143.922	701.438
TS16								
VE	0.663	0.189	3.793	0.902	8.878	0.047	84.501	19.949
Naive	0.716	0.189	4.368	1.122	32.086	0.703	145.828	40.501
Greedy	0.300	0.190	1.118	0.220	32.498	0.719	145.547	40.547
PigsLegs4								
VE	NA	-	-	-	NA	-	-	-
Naive	15.351	-	-	-	5080.219	-	-	-
Greedy	0.243	-	-	-	5003.500	-	-	-
TT33								
VE	2.688	-	-	-	19.597	-	-	-
Naive	1.570	-	-	-	127.282	-	-	-
Greedy	0.183	-	-	-	138.188	-	-	-
	(a)				(b)			

Table 2
Part (a) shows the maximum heap size used during the evaluation of the UIDs. The heap includes the memory for representing the UID, the normal form S-DAG, the potentials (for the VE algorithm), and the cache (for the SC algorithms: Naive and greedy). Part (b) shows the time (in seconds and milliseconds) for determining the MEU for the normal form S-DAGs. The time does not include the time for constructing the normal form S-DAG and the time for the determination of the caching factors for the SC algorithms.

viding the basis for our implementation.

References

Allen, D., Darwiche, A., Park, J. D., 2004. A greedy algorithm for time-space tradeoff in probabilistic inference. In: Proceedings of the Second European Workshop on Probabilistic Graphical Models. pp. 1–8.

Cano, A., Gómez, M., Moral, S., May 2006. A forward-backward Monte Carlo method for solving influence diagrams. International Journal of Approximate Reasoning 42 (1–2), 119–135.

Darwiche, A., 2001. Recursive conditioning. Artificial Intelligence Journal 125 (1–2), 5–41.

Howard, R. A., 1962. The used car buyer. In: Howard, R. A., Matheson, J. E. (Eds.), The Principles and Applications of Decision Analysis. Vol. 2. Strategic Decision Group, Ch. 36, pp. 691–718.

Howard, R. A., Matheson, J. E., 1981. Influence diagrams. In: Howard, R. A., Matheson, J. E. (Eds.), The Principles and Applications of Decision Analysis. Vol. 2. Strategic Decision Group, Ch. 37, pp. 721–762.

Ide, J. S., Cozman, F. G., 2002. Random generation of Bayesian networks. In: Brazilian Symposium on Artificial Intelligence. pp. 366–375.

Jensen, F., Jensen, F. V., Dittmer, S. L., 1994. From influence diagrams to junction trees. In: de Mantaras, R. L., Poole, D. (Eds.), Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers, pp. 367–373.

Jensen, F. V., Nielsen, T. D., 2007. Bayesian Networks and Decision Graphs, 2nd Edition. Springer-Verlag New York, ISBN: 0-387-68281-3.

Jensen, F. V., Vomlelova, M., 2002. Unconstrained influence diagrams. In: Darwiche, A., Friedman, N. (Eds.), Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers, pp. 234–241.

Lauritzen, S. L., Nilsson, D., 2001. Representing and solving decision problems with limited information. Management Science 47 (9), 1235–1251.

Madsen, A. L., Jensen, F. V., 1999. Lazy evaluation of symmetric Bayesian decision problems. In: Laskey, K. B., Prade, H. (Eds.), Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers, pp. 382–390.

Nielsen, T. D., Jensen, F. V., 1999. Well-defined decision scenarios. In: Laskey, K. B., Prade, H. (Eds.), Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI). Morgan Kaufmann Publishers, pp. 502–511.

Shachter, R. D., 1986. Evaluating influence diagrams. Operations Research 34 (6), 871–882.

Shachter, R. D., 1999. Efficient value of information computation. In: Laskey, K. B., Prade, H. (Eds.), Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers, pp. 594–601.

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

Shenoy, P. P., 1994. A comparison of graphical techniques for decision analysis. European Journal of Operational research 78 (1), 1–21.

Vomlelova, M., Jensen, F. V., 2004. An extension of lazy evaluation for influence diagrams avoiding redundant variables in the potentials. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 12, 1–17.