Flexible Policy Construction by Information Refinement

Michael C. Horsch
horsch@cs.ubc.ca

David Poole
poole@cs.ubc.ca

Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, B.C., Canada V6T 1Z4

Abstract

We report on work towards flexible algorithms for solving decision problems represented as influence diagrams. An algorithm is given to construct a tree structure for each decision node in an influence diagram. Each tree represents a decision function and is constructed incrementally. The decision maker can balance the cost of computing the next incremental improvement to a tree against the expected value of the improvement. The improvements to the tree converge to the optimal decision function computed by dynamic programming techniques, and the asymptotic behaviour is only a constant factor worse than dynamic programming techniques, counting the number of Bayesian network queries. Empirical results show that utility varies with the size of the tree and the number of Bayesian net calculations.

Keywords: influence diagrams, influence diagram evaluation, resource bounded computation.

1 Introduction

Influence diagrams provide expressive and intuitive representations for an important class of decision problems [4, 13, 9]. Small problems can be solved by finding a policy which maximizes the decision maker's expected utility without considering the cost of computation, but finding these solutions is an NP-hard problem [13, 12, 15, 1].

It is widely acknowledged that the assumption of negligible computational costs can be lifted, and that doing so may provide the leverage needed to address large problems. When the costs of computation are taken into account, the decision maker must reason, not only about the outcomes of acting in the world, but also about the outcome of computing on finite hardware while other processes in the world continue [3, 11].

We report a technique to compute policies for decision problems expressed as influence diagrams. For each decision node in the influence diagram, the technique builds a decision function in the form of a tree whose vertices are labelled with predecessors of the decision node, and whose leaf nodes are actions.

Our technique constructs a sequence of trees, the next being an incremental improvement to the previous, ending with a tree which represents the same "optimal" decision function that would be computed by traditional dynamic programming techniques [13].

This technique is a step towards flexible iterative refinement of policies for decision problems. *Flexible* means that policies are available in an anytime manner [3]. *Iterative* means that the next policy in the sequence is constructed by *refining* the previous policy. Each tree in the sequence represents a sub-optimal decision function, whose expected value to the decision maker is well defined. The computational effort to make an incremental improvement to a tree is known in advance. These two facts provide the basis of a flexible system in which the decision maker can explicitly balance the expected value of a policy against the cost of computing it.

For a decision node with n informational predecessors, each having at most b values, the sequence of improvements to the tree converges after $O(b^n)$ queries to a Bayesian network [12], only a constant fact worse than traditional dynamic programming techniques.

We demonstrate the preliminary empirical results of this approach to the sequence of decision trees, and discuss ways to order the sequence.

2 Influence diagrams

An influence diagram (ID) is a directed acyclic graph representing a sequential decision problem under uncertainty [4]. An ID models the subjective beliefs, preferences, and available actions from the perspective of a single decision maker.

Nodes in an ID are of three types. Circle shaped *chance* nodes represent random variables which the decision maker cannot control, square shaped *decision* nodes represent decisions, *i.e.*, sets of mutually exclusive actions which the decision maker can take. The diamond shaped *value* node represents the decision maker's preferences.

Arcs represent dependencies. A chance node is conditionally independent of its non-descendants given its direct predecessors. A decision maker will observe a value for each of a decision node's direct predecessors before an action must be taken. The decision maker's preferences are expressed as a function of the value node's direct predecessors.

In an ID, there is a conditional probability table associated with every chance node (unconditional, if it has no predecessors), and a value function associated with the value node.

For example, Figure 1(a) shows an augmented version of the well known *Weather* ID [12]. The ID represents the information relevant to a hypothetical decision maker, whose problem is to decide whether to take an umbrella to work. The goal is to maximize the decision maker's expected *Satisfaction*, which depends on the *Weather* and decision maker's decision to *Bring Umbrella?* The decision maker can choose to *Take Umbrella*, or *Leave Umbrella*, which are not explicit in the figure.

The decision maker has two sources of information: a *Radio Weather Report*, and the *View From Window*. These random variables are explicitly assumed to be independent given the weather, and both have three possible outcomes: *Sunny*, *Cloudy*, and *Rainy* (not explicit in the figure). The *Weather* is also a random variable, not directly observable at the time an action must be taken; it has two states: *Sun* and *Rain*, (not explicit in the figure).

For brevity, probability and utility information for this example has not been shown. However, conditional probability tables of the form P(*Weather*), P(*Radio Weather Report*|*Weather*), and P(*View From Window*|*Weather*) are necessary to complete the specification. The value function, *Satisfaction*(*Weather*, *Take Umbrella*) is also necessary.

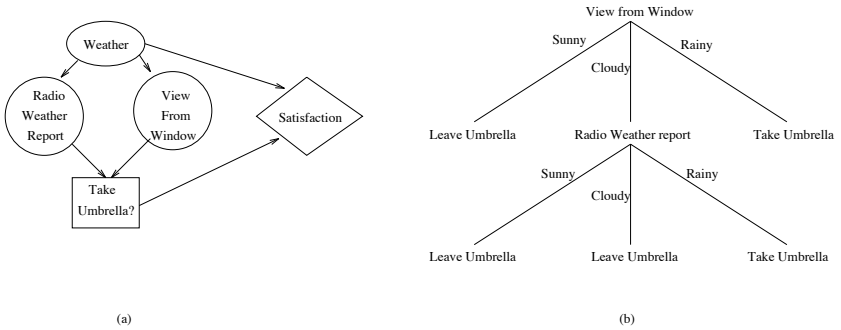

Figure 1: *(a) A simple ID. (b) A decision tree representation of a policy (see Section 3).*

A policy prescribes an action (or sequence of actions, if there are several decision nodes) for each possible combination of outcomes of the observable variables. In one of the possible policies for the above example, the decision maker always takes an umbrella, regardless of the information available. An optimal policy is the policy which maximizes the decision maker's expected *Satisfaction*, without regard to the cost of finding such a policy.

The goal of maximizing the decision maker's expected *Satisfaction* can be achieved by finding an optimal policy, if computational costs are assumed to be negligible. If computational costs are not negligible, the decision maker's expected utility might be maximized by a policy which is not optimal in the above sense.

In this paper, IDs are assumed to have chance and decision nodes with a finite number of discrete values. Furthermore, we limit the discussion to IDs with a single value node.

2.1 Terminology

Chance nodes are labelled $x, y, z, \ldots$. Decision nodes are labelled d, with subscripts if necessary to indicate the order the decision nodes. The value node, and its value function, will be labelled v.

The set of a node's direct predecessors is specified by Π subscripted by the node's label. The set of values (outcomes or actions) which can be taken by a node is specified by Ω, similarly subscripted. The set Ω_{Π_d} is the set of all possible combinations of values for decision node d's direct predecessors.

An element in this set will be called an *information state*.

A decision function for d is a mapping $\delta : \Omega_{\Pi_d} \rightarrow \Omega_d$. A policy for an ID is a set $\Delta = \{\delta_i, i = 1 \ldots n\}$ of decision functions, one for each of the decision nodes $d_i, i = 1 \ldots n$.

2.2 Related work

There are several techniques for solving IDs, which do not consider the cost of computation. The original technique converts an ID to a symmetric decision tree [4]. An algorithm which operates on the graphical structure is given in [13].

Recent advances in efficient computation in Bayesian networks [9, 7, 6] provides a framework for efficient computation of expected value and optimal policies [12, 5]. Heuristic search has also been applied to finding policies for IDs [10] using these advances. We use Bayesian networks (BNs) as the underlying computational engine for our technique to compute posterior probabilities and expected values [12].

A number of researchers have described iterative approaches to solving influence diagrams. Heckerman *et al.*, [2] and Lehner and Sadigh [8] use tree structures to represent policies, and use a greedy approach to incremental improvement of the tree structure. Both approaches use a single tree to represent the policy. Lehner and Sadigh define optimality of a decision tree with respect to the number of nodes in the tree, and give a general property which guarantees that an optimal decision tree of a certain size can be found by greedy search. Heckerman *et al.*weigh the value of the tree against the cost of computing it, but the tree itself is intended as an alternative to on–line decision making.

Our work extends the current work by building a tree structure for each decision node, taking advantage of efficient probabilistic inference techniques. This combination creates a basis for on–line, resource bounded computation. Our empirical results also suggest that the simple greedy approach of one step look ahead, can be improved by a greedy approach using less than a full step look ahead.

3 Single stage computations

We use decision trees to represent decision functions. In this section, we define decision trees, and show how they are built. In Section 3.3, we consider the case where the decision problem has a single decision node, and extend the idea to IDs with multiple decision nodes in Section 3.4.

3.1 Decision trees

Let d be a decision node in an ID. A decision tree t for d is either a leaf labelled by an action $d_j \in \Omega_d$ or a non-leaf node labelled with some observable variable $x \in \Pi_d$. Each non-leaf has a child decision tree for every value $x_k \in \Omega_x$. An information predecessor $x \in \Pi_d$ appears at most once in any path from the root to a leaf. Each vertex v has a *context*, γ_v, defined to be the conjunction of variable assignments on the path from the root of the tree to v. The action at the leaf represents the action to be taken in the context of the leaf. Given an information state $w \in \Omega_{\Pi_d}$, there is a corresponding path through a decision tree for d, starting at the root leading to a leaf, which is labelled with the prescribed action for w. Note that the context of an action need not contain every variable in Π_d.

A decision tree represents a decision function. We will refer to the action prescribed by a decision function by $\delta(w)$ for information state w, or by d_l if l is a leaf on a given decision tree.

For a given leaf l, its context γ_l is extendible if it does not contain all the observable variables. We refer to the variables which are not in the context as *possible extensions*, writing ξ_l. We will use γ without subscript or argument when we need to refer to an arbitrary context. The symbol γ_ϕ represents the empty context, equivalent to the context at the root of a decision tree.

A decision tree is shown in Figure 1(b). The tree can be interpreted as a policy for the ID in Figure 1(a) as follows. The decision maker first considers the view from the window. If the view is cloudy, then the decision maker will determine what to do by consulting the radio weather report. However, if the view from the window is sunny or rainy, then the radio report need not be consulted at all, even though the report is available as information. Note that in this example, the decision tree represents a policy. In general, we will construct a decision tree for each decision node in an influence diagram.

3.1.1 Expected value

We define expected value of a decision tree, so that we can compare decision trees.

The expected value of the decision tree t is defined as follows:

$$E_t = \sum_{l \in t} u(d_l|\gamma_l)\mathrm{P}(\gamma_l)$$

where $u(d|\gamma)$ is the expected value of an action d in a context γ, and the summation is over all leaves in t.

The optimal decision tree is defined as the one whose expected value is greater or equal to the expected value of any other decision tree: t is optimal if for all t', $E_t \geq E_{t'}$. This definition of optimal does not take into account the cost of computation.

3.2 Extending a decision tree

A decision tree t can be extended by removing some leaf l with context γ_l, and replacing it with new vertex $x \in \xi_l$. The new vertex x must have a leaf for every value $x_j \in \Omega_x$, and the leaf will be labelled with an action $d_j \in \Omega_d$ which maximizes the expected utility in the new context $\gamma' = x_j\gamma_l$.

Recall that each leaf in the tree is labelled with an action. The action is chosen on the basis of being the best action for the context γ_l of the leaf l:

$$d_l = \arg\max_{d_i \in \Omega_d} u(d_i|\gamma_l)$$

We observe that this can be computed with one query to a Bayesian network [12] (see Section 3.5). If a vertex has b values, b queries to the BN are required to compute its leaves. Another b queries are needed to compute expected value for each leaf.

For a decision tree t, and leaf node l, we define the expected value of improvement, $EVOI_t(l, x)$, to be the increase in expected value when t is extended at l with some $x \in \xi_l$, resulting in a new tree t':

$$EVOI_t(l, x) = E_{t'} - E_t$$

The *best extension of* t *at* l is given by:

$$\begin{aligned} x_l^* &= \arg\max_{x \in \xi_l} EVOI_t(l, x) \\ &= \arg\max_{x \in \xi_l} \sum_{x_j \in \Omega_x} u(d_j|x_j\gamma_l)\mathrm{P}(x_j|\gamma_l) \end{aligned}$$

This x_l^* represents a local improvement. Note that $EVOI_t$ only depends on l, its context, and the possible extensions at l. Therefore we can look at improvements to each leaf independently of possible improvements to other leaves, which we exploit for efficiency. Finally, the posterior probability $\mathrm{P}(x_j|\gamma_l)$ requires one additional BN computation. Therefore, to find a best extension for a particular leaf, we require $2(n-k)+1$ BN calculations, where n is the number of predecessor for the decision node, and k is the size of the leaf's context.

3.3 The single stage algorithm

The intuition for the single stage algorithm can be stated as follows. Given a decision tree t, the algorithm extends the tree by choosing a leaf, and finding its best extension. The process continues until some stopping criteria is met. We have assumed that the ID has a single decision node. In the next section, we show how to find policies for IDs with several decision nodes.

The basic algorithm can be given as follows:

```
procedure DT1
  Input:
    influence diagram ID
    decision node d
  Output:
    a decision tree for d

  Start with the tree as a single leaf
  Do {
    Choose a leaf in the tree to extend
    Replace the leaf with its best extension
  } Until   (stopping criteria are met
             or no more leaf nodes to extend)
  Return the tree
```

The sequence of trees created by `DT1` is such that the expected utility of the next tree is never less than that of the previous tree. However, there is no guarantee that the expected utility will always increase with every extension of the tree. Furthermore, an ID could be constructed in which the expected utility of a tree is arbitrarily far from the expected utility of the optimal tree as long as every leaf node is still extendible.

The following results hold, regardless of the manner in which a leaf is chosen for extension.

Theorem 1 *Let n be the number of information predecessors for a decision node d in an ID. Furthermore, assume that the number of states for any node in the ID is bounded by a constant b. The total number of extensions computed when* `DT1` *computes the optimal decision tree for d is less than* $\frac{b^{n+1}}{(b-1)^2}$.

This follows from the observation that the number of extensions considered at a leaf depends on the size of its context: if decision node d has n predecessors, and a leaf has k of these predecessors in its context, then the number of extensions which must be examined to choose the maximum is $n-k$. If the number of states a predecessor can take is bounded by constant b, then during the course of constructing the optimal decision tree, `DT1` extends b^k leaf nodes whose context is of size k. Therefore the total number of extensions considered by `DT1` is

$$\begin{aligned}\sum_{k=0}^{n-1}(n-k)b^k &= b^{n-1}\sum_{k=0}^{n-1}(n-k)b^{k-n+1} = b^{n-1}\sum_{i=1}^{n-1} ib^{1-i} \\ &\leq b^{n-1}\sum_{i=1}^{\infty} ib^{1-i} = b^{n-1}\left(1-\frac{1}{b}\right)^{-2}\end{aligned}$$

We observe that this result implies that IDs with binary nodes form the worst case for our technique. Note also that this result counts the number of different possible improvements to the tree throughout the process until the optimal decision tree is found. This number is less than the number of leaf nodes which need to be computed, by a factor of b.

Corollary 2 *To compute the optimal decision tree for a decision node with n informational predecessors,* `DT1` *requires $O(b^n)$ queries to a Bayesian network.*

Recall that each extension requires $2b$ queries to the BN (Section 3.2). This result implies that `DT1` requires a only constant factor more BN queries than traditional dynamic programming algorithms [12].

3.3.1 A greedy sequence of extensions

Recall the definition of the best extension of a decision tree t at a leaf l. Here we define the notion of the best extension for the whole tree: the tree is extended by picking the leaf whose best extension yields a higher expected value than the best extension of any other leaf in the tree.

More formally, the best extension for a decision tree t is defined to be the extension x_t^* at some leaf l such that for all leaf nodes $l' \neq l$:

$$EVOI_t(l, x_l^*) \geq EVOI_t(l', x_{l'}^*)$$

We can reduce this to the following: given a decision tree t, with leaves $L = \{l_0, l_1, \ldots, l_k\}$, the best extension to t is given by:

$$x_t^* = \arg\max_{l \in L} \sum_{x_j \in \Omega_{x_l^*}} u(d_j | x_j \gamma_l) \mathrm{P}(x_j | \gamma_l) - u(d_l | \gamma_l)$$

where x_l^* is the best extension for a given leaf l, and γ_l is the context of leaf l. The action d_j maximizes the decision maker's expected value in each new context.

This expression emphasizes that extensions to all leaves can be computed independently, and the best one chosen from them. Furthermore, any extension which is not chosen as x_t^* can be cached, and possibly used later without recomputation.

Finding the best extension to a tree may seem to require an additional non-trivial amount of computation. However, so long as we choose to extend the chosen leaf by its best extension, the greedy extension changes only the ordering in which the tree is extended. A simple implementation would use a priority queue to order the extension sequence; extensions stored in the queue represent a computational investment, and if the computation continues until the optimal tree (it could be asked to stop earlier), none of the work put into the queue is wasted. However, as we show in Section 4, the one step look head turns out not to be the best investment of computational resources.

3.4 Multiple decision nodes

We have seen that a decision function can be constructed for an influence diagram containing one decision node. For IDs which have several decision nodes, we work in the usual order (last decision to first). We allow the algorithm to "split" on a decision node's values, assigning a uniform distribution

over the decision variable, even though the decision maker has control. We define the expected value of a non-leaf vertex labelled with a decision node to be the maximum expected value of its children. The decision vertex retains all its leaves, as some future extension may shift the maximum expected value to another branch.

3.5 Implementational details

We have implemented the algorithm using the greedy approach of the previous section. We use a BN to compute expected value, and make some effort to make the implementation reasonably efficient.

3.5.1 A BN as computational engine

Computation of expected value is based on the conversion of IDs to BNs [12].

The utility node is converted to a binary chance node whose conditional probability distribution is the normalized utility function. Decision nodes in the ID are converted to root chance nodes with uniform probability distributions.[1] The BN derived in this way from an ID is subsequently compiled into a join tree [6], which can compute posterior probabilities efficiently.

The best action $d^* \in \Omega_d$ to be performed in a state $w \in \Omega_{\Pi_d}$ can be found by choosing the action which maximizes the query $\mathrm{P}(d|w,v)$; the expected value of the best action is computed by querying the utility node $\mathrm{P}(v|w,d^*)$.

We generalize this result in terms of choosing an action which maximizes expected utility in a given context (recall that a context may not include a value for every observable variable in Π_d). The corresponding queries are: $\mathrm{P}(d|v,\gamma)$, and $\mathrm{P}(v|d^*,\gamma)$.

3.5.2 Other efficiency issues

We use a priority queue to order the sequence of extensions to a given tree. The queue contains elements having a context, and a subtree labelled with the implied leaf's best extension, with leaf nodes attached to it, as in Section 3.2. In effect, when the implementation pulls an item from the queue, it is almost trivial to replace the leaf with a subtree, because it was pre-computed. Most

[1] This does not affect the probability distribution, and makes the size of the largest cliques a function of the probabilistic information in the ID. The informational predecessors remain part of the ID, but not the underlying BN.

of the work happens after the leaf is replaced. For each of the new leaf nodes, we compute the new best extension, its best subtree, and enter it into the queue.

We avoid recomputing expected value by storing the expected value of a leaf, and the expected value of each extension in the queue. As well, we store the posterior probability each of non-leaf node given its context. Thus, we need only to compute expected value for a new context; all the remaining information is obtained by look up.

In order to compute $EVOI_t(l, x)$, we need to compute the posterior probability $\mathrm{P}(x_j|\gamma_t(l))$. We order the exploration of extensions so that we can enter $\gamma_t(l)$ once, and query the BN for the posterior probabilities of $x \in \xi_l$ on the basis of one DistributeEvidence computation. We still require this computation once for every leaf, but this is insignificant compared to the $2k$ CollectEvidence computations required to compute the expected value for all k of the leaf's possible extensions.

In our implementation, any leaf node whose context contains logical impossibilities are not extended nor are they considered further. Empirically, this could have a great effect on the cost of computation and the size of the resulting tree, as shown in the first example in the next section.

4 Performance

In this section, we explore the behaviour of a greedy `DT1` empirically. We demonstrate the algorithm on a small well known problem, and a set of larger random problems.

4.1 Terms of comparison

We describe the behaviour of an implementation of `DT1`, running the procedure until the optimal decision tree is achieved. The data points we will be collecting represent decision trees in terms of the tree's expected value, the number of BN computations required to achieve the tree, and the number of interior nodes in the tree. Note that each data point could be used as an anytime solution.

The expected value of a decision tree has been defined earlier. In our examples, expected value is normalized to $[0, 1]$. We measure the computation cost in terms of the number of BN computations required. Recall that each

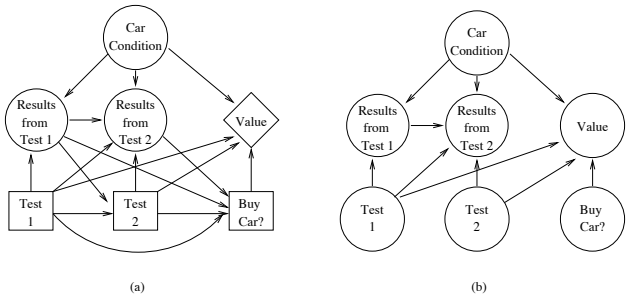

Figure 2: *The Car Buyer Problem. Part (a) shows the influence diagram; part (b) shows the BN used for computation of expected value for the last decision node,* Buy Car?*.*

extendible leaf requires $2(n-k)+1$ BN computations, where n is the number of predecessors of the decision node, and k is the number of observations in the context of the leaf.

The size of our decision trees is measured in terms of the number of non-leaf vertices in the tree. Assuming binary valued predecessors of a decision node, the number of actions in a tree of size s is $s+1$, as shown in Section 3.

We compare our implementation to an idealized dynamic programming algorithm, in which a BN computation is required for every possible observable state [12]. If a decision node in an ID has n binary predecessors, then the dynamic programming algorithm requires exactly 2^n BN computations to find the optimal decision function containing 2^n information states. If required, the expected value of the decision function could be computed with a single BN computation after the decision function has been established. Finally, since the dynamic programming algorithm only produces a single solution, it falls under the category of *inflexible*, and for emphasis and brevity, we refer to it as such.

4.2 A classical example

To illustrate the behaviour of `DT1`, we will show its behaviour on the well-known Car Buyer problem, Figure 2 [14]. There are three decision nodes in this problem; in this section, we use `DT1` to find a decision function for the last decision node.

Briefly, the ID represents the knowledge relevant to a decision maker

deciding whether or not to buy a particular car. The decision maker has the option of performing a number of tests to various components of the car, and the results of these tests will provide information to the decision to buy the car. The actual condition of the car is not observable directly at the time the decision maker must act, but will influence the final value of the possible transaction. A policy for this problem would indicate which tests to do under which circumstances, as well as a prescription to buy the car (or not) given the results of the tests. Due to space constraints, none of the numerical data required to complete the specification of this problem is shown; this information can be found in [10, 14]. This problem is well known for its asymmetry; some combinations of tests and results are logical impossibilities.

Figures 3 and 4 show the performance of `DT1` on the last decision stage in the problem.

Figure 3(b) compares the expected value of the decision tree and the size. Again, each point represents a decision tree improved by a single extension. Because of the asymmetries in the problem, the optimal policy can be represented with a decision tree with 7 internal vertices and 14 leaves. For this problem, the largest tree computed by `DT1` has 27 vertices and 17 leaves. The inflexible algorithm, unless designed to handle asymmetries, requires a table of 96 entries. The large difference between our trees and the inflexible solution is due to the asymmetries in the problem; our implementation does not extend contexts whose probability is zero.

Figure 3(b) illustrates the increase in expected value of each decision function, as a function of the number of BN computations. The very first decision function for this problem is available after just 2 BN calculations, and has an expected value which is less than 10% from optimal. Each subsequent point represents an improvement to the decision function by extending the tree by one node. The rightmost point represents the optimal policy.

For comparison, the inflexible algorithm requires 96 BN computations, to compute a table of 96 actions.

Figure 4 plots expected value versus the computational effort required; essentially, it shows where the work gets done. Much work goes into finding the first few trees, as each information state is small, and has many possible extensions. Half of the work is done to find a decision function which distinguishes 3 information states. As the decision tree gets larger, the number of possible extensions for each context gets smaller. Towards the end of the process, very few BN calculations are performed because there are no

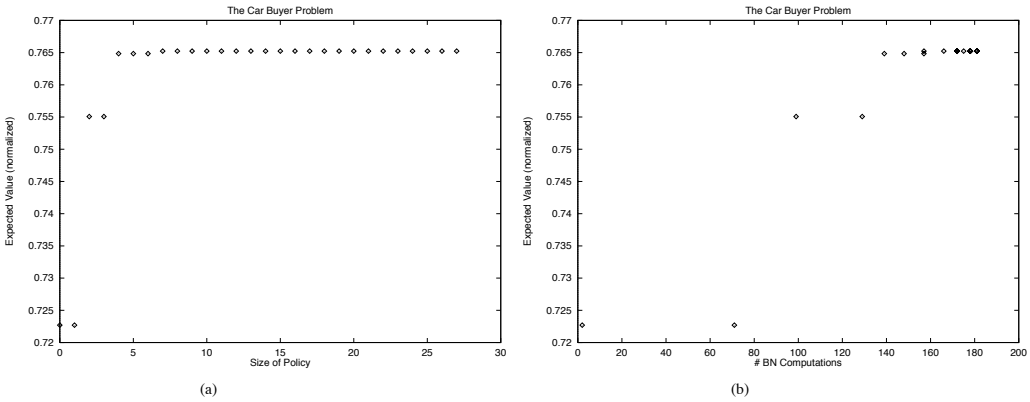

Figure 3: *The results of using* `DT1` *on last decision node of the Car Buyer Problem. (a) The expected value versus the size of the tree. (b) The expected value versus the number of BN computations. Each point represents a decision tree, and the rightmost point in both graphs represents the optimal decision function.*

extensions to the contexts pulled from the queue.

4.3 Performance on random single decision IDs

To show the performance of `DT1` on a single decision node, we have created a number of random influence diagrams with one decision node. Figure 5 illustrates the template ID which we have used to create random decision problems. The template problem has n chance nodes, each of which is an informational predecessor to the decision node. As well, each chance node is a predecessor of the value node. The template can be instantiated by choosing n; random binary probability distributions, *i.e.*, $\mathrm{P}(c_k)$, are chosen from a uniform distribution on $(0, 1)$. The utility function is also chosen from a uniform distribution $[0, 1]$.

Figure 6, shows the behaviour of `DT1` on seven instantiations with $n = 8$. Each point in these graphs represents a decision function; there are seven sets of data shown, one shape for each ID. Part (a) shows the increase in expected value as the trees increase in size. Part (b) shows the expected value of the decision function as a function of the work done by the algorithm in terms of BN calculations. The right most points represent the optimal decision tree

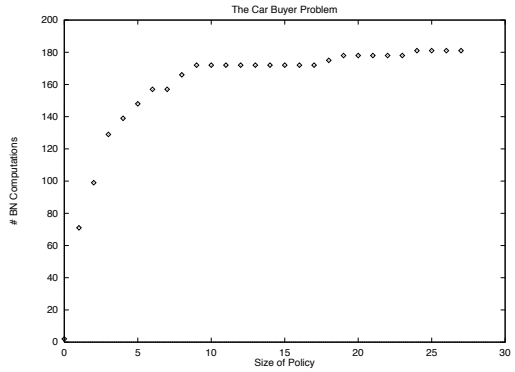

Figure 4: *Showing where the work gets done by* `DT1` *when computing decision functions for the last decision in the Car Buyer Problem.*

for each problem. The left hand endpoints represent the expected value of the decision tree continuing only a single leaf.

While the data tend to overlap the general trends are clear: expected value increases with the size of the tree, and the work done. However, the optimal policy can be computed by the inflexible algorithm in 256 BN computations, which is well before the trees produced by greedy `DT1` begin to level off in terms of expected value.

We observe that the greedy `DT1` does not invest computation time wisely. The graph of expected value versus size (Figure 6a) illustrates the concern mentioned in Section 3: that the greedy approach produces trees which are small but expensive to compute.

We have two final points to make about these graphs. First, even the first few trees can be more valuable to a decision maker than no decision function at all, if there are deadlines or other opportunity costs. Second, the IDs in these graphs are random, and symmetrical; there is reason to believe that decision problems faced by real decision makers contain more structure than our random problems do.

4.4 Performance on IDs with several decision nodes

`DT1` can be applied to each decision node in an ID, in the standard back to front dynamic programming order. We have applied this technique to the Car Buyer problem. The results are preliminary, but encouraging.

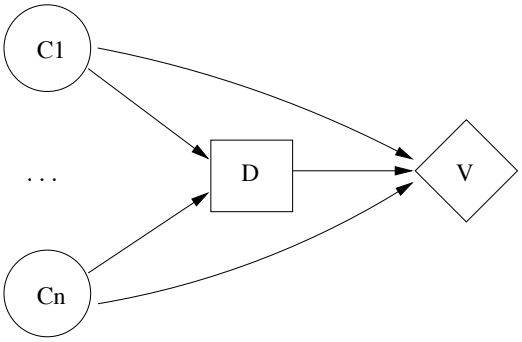

Figure 5: *An ID with one decision node and n informational predecessors.*

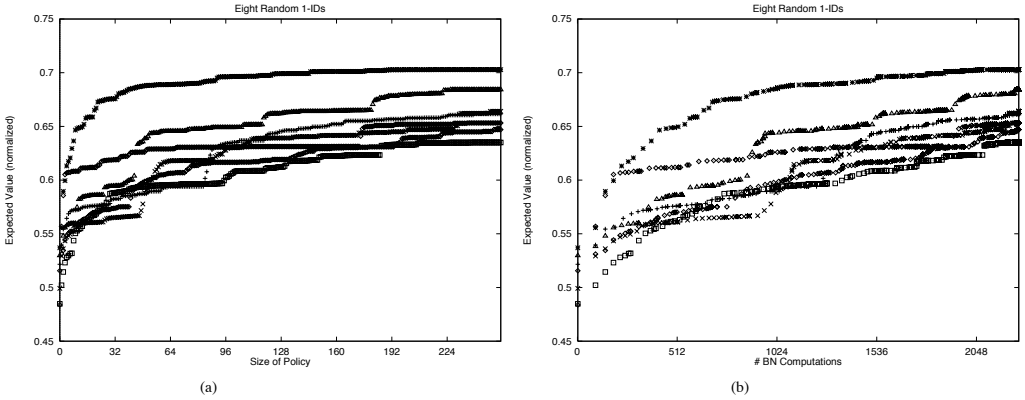

Figure 6: *Seven random IDs with 8 informational predecessors each were solved by an implementation of* `DT1`*. The improvement of the tree in terms of expected value: (a) as the tree increases in size, and (b) as the number of BN calculations performed increases.*

Because the optimal decision tree for the last decision node can be represented with a tree of 7 non-leaf vertices, we explored the space of all possible decision trees for the three decision nodes. Note that the first decision node has no informational predecessors, and the second has two. The maximum tree sizes for these two decision nodes are zero and 2 non-leaf vertices, respectively.

The optimal policy was found when the third decision tree had 2 non-leaf vertices, and the second decision tree no non-leaf vertices. In total, only 75 BN calculations were needed to compute the policy (71 of which were needed to find a decision tree for the last decision node), as opposed to the 113 BN calculations required by the inflexible method. A sub-optimal policy, in which the decision maker buys the car without any tests, is found with 6 BN calculations, and the expected value of this policy is about 3% less than the optimal policy's expected value.

5 Conclusions

We have shown how a decision function can be constructed iteratively. We have shown that the sequence converges necessarily to the optimal decision function. Asymptotically, the number of Bayesian network calculations required by the iterative technique is a constant factor larger than dynamic programming techniques. Furthermore, given a model of the cost of computation, the expected cost of the next iteration can be weighed against the increase in expected value of the decision function.

We have shown preliminary empirical results which are encouraging: small decision trees are of value to a decision maker, before inflexible techniques have produced the optimal decision tree. The greedy implementation was shown to spend much effort to find the best extension to the decision tree at a given leaf, resulting in trees which are small but expensive to compute. Another approach would be to choose a good extension, as opposed to the best extension. We expect that if less effort is spent to build a tree of a given size, the expected value of a decision tree will rise more quickly as measured by the number of BN calculations; correspondingly, the tree will also grow more quickly. We are currently exploring these and other ways to increase the value of early computational effort.

We have shown how to apply the greedy `DT1` to IDs with several decision nodes. While the algorithm can be applied in a straightforward manner to

such decision problems, we are exploring ways to balance the computational effort across the stages. The dilemma is that the decision maker may need to take action on the first decision node with some urgency, but all the computational effort could go into finding a decision function for the last decision node. We are looking at the effects of computing very small trees for decision nodes late in the sequence, and increasing the size for early decision nodes. Our preliminary experiments on simple IDs suggest that this technique may be successful.

We also are exploring the relationship between tree structured decision functions and tree representations of conditional probability tables, *e.g.*[14].

Acknowledgements

The authors would like to thank Brent Boerlage of Norsys Software Corp. for advice and support in the use of the Netica API as our Bayesian network engine, and for many discussions.

References

[1] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks (research note). *Artificial Intelligence*, 42(2–3):393–405, 1990.

[2] D. E. Heckerman, John S. Breese, and Eric. J. Horvitz. The compilation of decision models. In *Uncertainty in Artificial Intelligence 5*, pages 162–173, 1989.

[3] Eric J. Horvitz. Computation and action under bounded resources. Technical Report KSL–90–76, Departments of Computer Science and Medicine, Stanford University, 1990.

[4] R.A. Howard and J.E. Matheson. *The Principles and Applications of Decision Analysis*. Strategic Decisions Group, CA, 1981.

[5] Frank Jensen, Finn V. Jensen, and Soren L Dittmer. From influence diagrams to junction trees. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 367–373, 1994.

[6] F.V. Jensen, K.G. Olesen, and S.K. Andersen. An algebra of Bayesian belief universes for knowledge based systems. *Networks*, 20:637–660, 1990.

[7] S. L. Lauritzen and D. J. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *J. R. Statist Soc B*, 50(2):157–224, 1988.

[8] Paul E. Lehner and Azar Sadigh. Two procedures for compiling influence diagrams. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 335–341, 1993.

[9] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Reasoning*. Morgan Kaufmann Publishers, Los Altos, 1988.

[10] Runping Qi and David Poole. New method for influence diagram evaluation. *Computational Intelligence*, 11(3):498–528, 1995.

[11] Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, Cambridge, Mass., 1992.

[12] Ross Shachter and Mark Peot. Decision making using probabilistic inference methods. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 276–283, 1992.

[13] Ross D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871–882, 1986.

[14] J. E. Smith, S. Holtzman, and J. E. Matheson. Structuring conditional relationships in influence diagrams. *Operations Research*, 41:280–297, 1993.

[15] Nevin Lianwen Zhang. *A Computational Theory of Decision Networks*. PhD thesis, University of British Columbia, 1994.