Flexible Policy Construction by Information Refinement

by

Michael C. Horsch

B.Sc., University of Toronto, 1988

M.Sc., University of British Columbia, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

October 1998

Abstract

Decision making under uncertainty addresses the problem of deciding which actions to take in the world, when there is uncertainty about the state of the world, and uncertainty as to the outcome of these actions. A rational approach to making good choices is the principle of *maximum expected utility*: the decision maker should act so as to maximize the expected benefits of the possible outcomes.

The "textbook" approaches to decision analysis typically make the assumption that the computational costs involved are negligible. This assumption is not always appropriate. When computational costs cannot be ignored, a decision maker must be able to choose a trade-off between computational costs and object value.

This thesis proposes an approach to decision making called information refinement. It is an iterative, heuristic process which a decision maker can use to build a policy. We present three algorithms which use information refinement to construct policies for decision problems expressed as influence diagrams. The algorithms are intended for situations in which computational costs are not negligible, and are designed to give the decision maker control of the trade-off involved in the decision making process.

The first algorithm is an anytime algorithm for single stage decision problems. It constructs a policy by increasing the use of information available to the decision maker. The second algorithm applies the single stage algorithm to multi-stage decision problems using a fixed allocation of computational resources. The third algorithm is an anytime algorithm for multi-stage decision problems.

We show empirically that these algorithms are able to make decisions with high expected value with small computational costs. We provide empirically evidence for our claims, by applying our algorithms to a large number of large decision problems.

Contents

List of Tables

List of Figures

Acknowledgements

I am first and foremost grateful to David Poole for his academic guidance, his generosity with time and financial support, and his enthusiasm as a supervisor. Thanks go to my supervisory committee, for their patience and insight: Alan Mackworth, Craig Boutilier, David Kirkpatrick and Jim Little.

I am grateful to the Department of Computer Science for additional financial support during my studies, as well as for giving me the opportunity to teach. I'd especially like to thank all the staff at the department for their help over the years, especially Jean, Joyce, Monika, and Valerie. I also gratefully acknowledge financial support from NSERC.

I am especially grateful for the friendship of the community which has surrounded me here at UBC: my colleagues, especially Andrew Csinger, Carl Alphonce, and Chris Roehrig; my room-mates past and present, and my friends at Lutheran Campus Ministry (my friends, there are far too many of you to list).

My family, of course, have been a constant source of support.

MCH

Vancouver, 1998

Chapter 1

Introduction

Agents that act in uncertain environments must be able to make make good decisions with limited computational resources.

Bayesian decision theory addresses the concerns of making good decisions [43, 48]. In particular, decision making under uncertainty addresses the problem of deciding what actions to take in the world, when there is uncertainty about the state of the world, and uncertainty as to the outcome of these actions. Uncertainty is modelled with probability, and the preference among outcomes is modelled with utility.

A common approach to making good choices is the principle of *maximum expected utility*: the decision maker should act so as to maximize the expected benefits of the possible outcomes. Expected utility is the sum of the utilities of the possible outcomes weighted by the probability of those outcomes. Maximizing expected utility means that choices are made so that, under the explicit uncertainty of the situation, the outcome will be as good as can be expected.

This principle is simply stated, although there is much to be said about the application of this principle. A decision problem can be posed as a decision tree [40], a Markov decision process [35], an influence diagram [21], or as a variant of these representations. The "textbook" approaches which apply the principle of maximum expected utility to problems in these representations typically make the assumption of negligible computational costs [35, 21, 45].

There is considerable interest in how the principle of *maximum expected utility* can be realized in practice [11, 41, 42, 18, 47], since the assumption of negligible computational costs is not always appropriate. In domains such as medical informatics, decision problems may be so large that computing an optimal policy is not possible in practice. In these cases, sub-optimal policies might be computed off-line. In the domain of intelligent agent architectures, on-line decision making may require that the agent respond to a dynamic environment without allowing for time to compute an optimal policy. In this case, a sub-optimal policy may be able to serve the agent well.

A key insight to the problem of accounting for computational costs is that any computation performed in the service of choosing an action can also be considered an action [18, 42]. Computational actions have costs associated with time and memory, and outcomes in terms of the results of the computation. Given the tools of decision theory, a decision maker might try to reason explicitly about the costs and benefits of particular methods applied to a particular problem. Seen in this way, any choice between alternate computations would be as much a candidate for the principle of maximum expected utility as the choice of action.

It might seem that the problem of infinite regress would prevent the principle of maximum expected utility from being used at the meta-level, since any computation performed in the service of determining which computational method to use to solve a problem is itself a decision problem.

To unravel this situation, it is helpful to distinguish between these two types of action; an action in the world will be called an *object level* action; a computational action is at a different level, the *meta-level*. It is also helpful to distinguish between two types of value: the value of an action without considering the computational costs is the *object value*; the value of the action accounting for computational costs is the *comprehensive value*.

Under the assumption of negligible computational costs, the expected comprehensive value is maximized when the expected object value is maximized. When this assumption is not appropriate, the cost of maximizing expected object value could be prohibitive. In this case, a better comprehensive value may be obtained from an algorithm which finds an action which does not maximize expected object value, but is available with a smaller computational cost.

According to the principle of maximum expected utility, a decision maker would always compute so as to maximize comprehensive value. In theory, this means that the decision maker finds an optimal trade-off between computational costs and object value. In practice, this means that a decision maker must be able to choose a trade-off between computational costs and object value. The choice of a suitable trade-off is restricted by the repertoire of methods available to the decision maker. Horvitz [18] uses the term *flexible* to describe an algorithm which is able to make a trade-off between computational costs and object value.

One way to solve the decision maker's meta-level problem is to assess the expected comprehensive value of applying each of the available methods, and proceed to apply the method which has the best expectation. If the assessment process is itself computationally negligible, we need not consider the assessment itself in the comprehensive value. However, if the assessment process involves significant computation, the assessment process could be modelled as a meta-decision problem. As long as this meta-decision problem is simpler than the original decision problem, the infinite regress problem, mentioned above, can be avoided. The regress will terminate at the level at which the assessment process is so small as to be a trivial computation. While this is an interesting and important aspect of meta-reasoning, this thesis does not solve the problem of assessing the comprehensive value of computation. Our focus is on providing methods for solving decision problems.

This thesis proposes and studies three algorithms designed to give the decision maker control of the trade-off involved in the decision making process. We present an approach to decision making called information refinement. It is an iterative, heuristic process which a decision maker can use to build a policy. We propose three algorithms which use information refinement for constructing policies for decision problems expressed as influence diagrams. The algorithms are intended for situations in which computational costs are not negligible.

Two of our algorithms are *anytime* algorithms [7], which construct policies iteratively, improving them as the algorithms proceed; at any time, the decision maker can interrupt the deliberation process and take action according to the current policy. In this way, the decision maker has control over the deliberation process. The third algorithm gives the decision maker control by requiring an *a priori* allocation of resources; a policy will be constructed according to the allocation provided by the decision maker.

We show that these algorithms are able to make decisions with reasonably high expected value with reasonably small computational costs. The algorithms have the property that the object value of the result increases as computational resources are invested in the method. We provide empirical evidence for our claims, by applying our algorithms to a number of large decision problems. These problems are large enough that traditional methods are infeasible. In this way, we have increased the repertoire of methods available to a decision maker.

1.1 Overview

In this section, we demonstrate our approach with a simple decision problem. In this problem, a decision maker is trying to determine whether to bring an umbrella on an outing. The decision maker has heard the radio weather report, and has seen the current local weather state by looking out the window. The problem is that neither source of information is perfect; the radio report has limited predictive ability, and the local weather may not predict how the weather will turn over the course of the outing. If it rains, the decision maker would prefer to have the umbrella, and prevent getting wet. On the other hand, the decision maker may have a distaste for being seen with an umbrella on a sunny day. The problem facing our hypothetical decision maker is this: given the current state of information, should she bring the umbrella?

Bayesian decision theory's answer is that a rational decision maker should act so as to maximize expected utility. In order to see how this is done, we introduce the following notation. Let S be the set of possible states of "the world," and O be the set of possible observations which can be made about the world. Let D be the set of choices available to the decision maker. Uncertainty about the states of the world are modelled with the probability distribution $\mathrm{P}(S)$. The uncertainty in the observations about the world can be modelled with the conditional probability distribution $\mathrm{P}(O|S)$, which says how likely it is to make a particular observation given that the world is in a certain state. The decision maker's preferences depend on the state of the world, and the action chosen; these preferences can be modelled with a utility function $u(D, S)$, which maps all pairs of state and action to a real number.

The problem is to choose a policy, which tells the decision maker how to act for every possible observation. Each policy $\delta : O \to D$ has an expected value E_δ, which is defined as follows:

$$E_\delta = \sum_{s \in s, o \in O} u(\delta(o), s)\mathrm{P}(O = o|S = s)\mathrm{P}(S = s)$$

The principle of maximum expected utility, applied without consideration of the cost of computation, states that the decision maker should choose the policy δ^* which maximizes E_δ.

In our example, the world states might be modelled with the two possibilities, rain and sunshine. The possible observations come from the possible weather reports, and the possible ways the local weather could be described. Let us represent the possible radio reports as one of sunny, cloudy or rainy, and for simplicity, we use the same three descriptions for the decision maker's observation of the local weather. The possible observations are the set of combinations of these two sources. The decision maker must decide to take the umbrella or leave it at home. Finally, we suppose that the decision maker prefers to use an umbrella if necessary, but prefers not to carry the umbrella if it is sunny; if the decision maker gets wet because the umbrella is at home, it's very bad, and it's almost as bad if someone sees him carrying an umbrella on a sunny day. For brevity, we leave the probability distributions and the utility function unspecified.[1]

Our decision maker can solve his problem by finding a policy, which tells him what to do for any observation. One way to find a policy is to enumerate the possible observations, and choose the best action for each. For our simple problem there are nine different observations which could be made. For each observation, and for each state of the world, and for each possible action, the decision maker could ask how likely the world would be in state $S = s$ given the observation that $O = o$, and determine the value of taking action $D = d$ in state $S = s$. For a human decision maker, working with pencil and paper, this is not a trivial calculation, though for a properly programmed computer it is quite simple.

The last wrinkle we throw into our problem is to suppose that the decision maker is under some time pressure: the decision maker intends to catch a bus which should arrive at the bus stop "shortly." Suppose that there will not be enough time find the optimal policy, and still catch the bus.

[1]We return to this example in Chapter 2, with slightly different notation. The probabilities and utilities are fully specified there.

Our approach to solving this problem is to construct a policy iteratively. We first consider the best course of action without using any of the observations. In our example, the decision maker may decide that, based on the preferences and probabilities, that rain is likely enough to warrant bringing an umbrella. If there is still time, this policy could be refined by choosing one of the possible observations, and using this piece of information to decide what to do. In our example, the decision maker may have time to consider how his action would change based on what was heard on the radio weather report. For example, if the radio says sunny, leave the umbrella at home; if the radio says cloudy or rainy, take the umbrella, as previously decided. At this point, the decision maker can be interrupted (by the need to catch the bus), and can take the action appropriate to his state of information.

1.2 Outline

We begin in Chapter 2 by presenting the background work upon which our work is based: influence diagrams, which are used as a representation for general class of decision problems; Bayesian networks, which are used in our implementation for computation of expected value and posterior probabilities; and decision trees, which are used to represent decision functions, mapping observation to action.

Chapter 3 introduces information refinement in single stage decision problems. The core algorithm and representation are introduced, and several examples are given. We show that the algorithm converges to the optimal policy, and that convergence in single stage problems is asymptotically equivalent to current algorithms which determine the optimal policy. The need for heuristics in the algorithm is presented, and several heuristics are presented and discussed. We apply the algorithm to a large number of single stage decision problems, to demonstrate the general behaviour of the heuristics. Our claim is that information refinement can construct non-optimal policies which are valuable to the decision maker, when computational costs are not negligible. To support this claim, we show empirically that for a range of computational resource expenditures, the information refinement approach provides more valuable policies than methods which construct optimal policies by exhaustive enumeration of the information space.

Chapter 4 extends the information refinement approach to multi-stage problems. We present two algorithms, both of which are based on the single stage algorithm presented in Chapter 3. The first algorithm applies the single stage algorithm to the stages of the multi-stage decision problem in the traditional dynamic programming sequence. It is a "contract" algorithm, which computes a policy for a given allocation of computational resources. The issue of determining a suitable allocation of resources is discussed.

The second algorithm has the "anytime" property, namely that it maintains a current best policy throughout its execution, and expends computational resources to improve the policy. It applies the core of the the single stage algorithm to the stages in the multi-stage decision problem in any order; the actual ordering is according to a heuristic measure of increase in expected value. The anytime algorithm makes use of an additional level of heuristic guidance. The behaviour of the algorithms under various heuristics is demonstrated.

These two algorithms are shown to converge to the optimal policy in a finite number of steps. The time complexity of the contract algorithm is asymptotically equivalent to traditional dynamic programming algorithms; the anytime algorithm is asymptotically worse (by an exponential factor) than traditional dynamic programming algorithms. The claim for both algorithms is that they provide valuable policies for multi-stage decision problems using a fraction of the resources required to determine the optimal policy. To support this claim, we apply the algorithms to a number of large decision problems (the information space of these problems is on the order of 2^{60} states), and show that for a wide range of computational resource expenditures, the sub-optimal policies constructed using the information refinement approach are more valuable than any policy (or fraction thereof) constructed using exhaustive enumeration of the information space and dynamic programming.

We summarize our contribution in Chapter 5 and outline the future directions of this research.

Chapter 2

Background and Related Work

We present Bayesian networks [32] as a representation of probabilistic knowledge, and as a basis for computation of posterior probabilities. Thereafter, we will discuss influence diagrams [21], which augment Bayesian networks with actions and preferences. As well, much of the numerical computation that we require in decision making is performed using Bayesian network techniques. We follow this discussion by a section covering the basics of decision tree learning, on which we base our iterative solution of decision problems.

2.1 Bayesian networks

A Bayesian network [32] represents a joint probability distribution as a directed acyclic graph (DAG). The nodes in the DAG represent random variables, which we assume have a finite number of discrete values. We denote random variables using upper case letters (*e.g.*, X, Y, Z) or capitalized words (*e.g.*, *Alarm*). For any

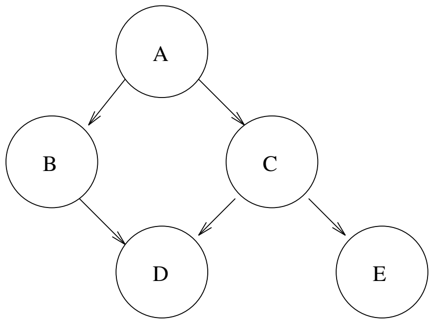


Figure 2.1: A simple Bayesian network.

random variable X, we denote the set of its values using Ω_X; a particular value from this set is denoted using lower case, *e.g.*, $\Omega_X = \{x_0, x_1, x_2, \ldots, x_k\}$; *e.g.*, $\Omega_{Alarm} = \{on, \mathit{off}\}$.

An arc from X to Y indicates that, in the model, X is considered to be a direct influence on Y. The *parents* for a node X is the set of direct predecessors of X in the DAG, denoted Π_X. A node X and all of its parents Π_X is called a *family*; each family has an associated conditional probability distribution which quantifies the effects of the parents on the child node: $\mathrm{P}(X|\Pi_X)$. If X has no parents in the Bayesian network, $P(X|\Pi_X) = P(X)$.

The joint probability distribution of all the random variables assessed in the domain can be represented by a Bayesian network, and is factored as follows:

$$\mathrm{P}(X_1, \ldots, X_n) = \prod_{i=1}^{n} \mathrm{P}(X_i|\Pi_{X_i})$$

Each factor in the product corresponds to a family in the network: the conditional probability of a node given all its direct predecessors.

Figure 2.1 shows a simple example of a Bayesian network. The factorization for this example is as follows:

$$\mathrm{P}(A, B, C, D, E) = \mathrm{P}(A)\mathrm{P}(B|A)\mathrm{P}(C|A)\mathrm{P}(D|BC)\mathrm{P}(E|C)$$

This representation is valuable for two reasons. First, domain knowledge can be described in a highly modular fashion. Second, the resulting graph structure can be used as a computational engine for computing posterior probabilities far more efficiently than the naive use of the joint probability distribution.

A Bayesian network encodes conditional independence: a node is independent of its non-descendants given an assignment of values to its parents. A Bayesian network also allows computation of conditional independence between any two sets of nodes, by graphical analysis called *d-separation*, rather than by numeric computation [32].

2.1.1 Inference using Bayesian networks

Bayesian networks are used to compute posterior probabilities, *i.e.*, beliefs after all available evidence has been taken into account. This section outlines some inference procedures for computing posterior probabilities. Our discussion focusses on the use of so-called "clique–tree propagation" methods, because we use them to compute posterior probabilities and expected utilities during decision making. In principle, any of the methods described below could have been used in our algorithms.

The problem of computing posterior probabilities, exactly or approximately, in Bayesian networks is NP–hard in general [5, 6], but in many cases the domain knowledge can be structured or simplified so that computation is feasible.

Kim and Pearl [24, 32] developed a polynomial algorithm for computing posterior probabilities for a special class of Bayesian network, known as *polytrees*. A polytree is a Bayesian network in which there is at most one undirected path between any two nodes. The algorithm works by message passing, each node passing probabilistic information it knows about evidence to any of its children and parents who have not already been informed. The polytree structure allows the evidence to be separated, and guarantees that a node does not receive multiple messages about a single piece of evidence. The complexity of the polytree algorithm is polynomial in the number of nodes in the network.

For networks which are not polytrees, several exact techniques exist, although in the worst case the problem is intractable. One method, called "cutset conditioning," decouples the general graph by finding a subset of nodes which, when their values are fixed, effectively creates a polytree [31]. The polytree algorithm of Kim and Pearl can proceed on the decoupled network, once for each combination of values in the cutset; the results are combined, weighted by the joint probability of the cutset. The drawback to this method is that finding a small cutset is crucial; finding the smallest cutset is NP-hard, and even with a small cutset, the technique is prone to combinatorial explosion.

Another technique for general DAGs precompiles a network into a secondary polytree-like structure called a join tree. Several variations on this technique exist, and are based on the idea of clustering the nodes of the original DAG. A node in the DAG must appear in at least one cluster, and furthermore, it must appear in a cluster with all of its parents; the conditional probability distribution of a node given its parents is placed in one of the clusters in which the node and its parents appear. The clusters are connected with undirected arcs such that there is only one path between any two clusters, and if a node from the DAG appears in two clusters, it must also appear in all the clusters on the path between the two.

There are several techniques for computing a join tree from a Bayesian network, the details of which can be found in [31, 25, 23, 46]. Computing an optimal join tree is an NP-complete problem, so heuristics are used to find good join trees. The cost of computing the join tree is amortized over the use of the network. For example, a Bayesian network representing diagnostic medical knowledge (*e.g.*[13]) can be compiled once and used thereafter for arbitrary consultations.

The resulting join tree can be used to compute posterior probabilities using a variant of the polytree algorithm called "clique–tree propagation" [25, 23]. Each cluster in the join tree has a "potential function," which is a function of the variables in the cluster. The conditional probability table for a random variable in the Bayesian network is assigned to exactly one of the clusters in which the variable appears. The potential function is initialized to the product of the conditional probability tables which appear in the cluster.

Evidence is expressed as "likelihood functions" which describe the likelihood of the evidence. When evidence is observed, a likelihood function is assigned to a cluster, and the potential of this cluster is revised by multiplying the likelihood function with the previous potential function. Typically, evidence is certain, and the likelihood function assigned to a cluster reduces the dimensionality of the potential by asserting that a value is known with certainty.

Evidence may affect variables outside the cluster; the propagation of evidence to the rest of the join tree is performed by message passing. A message is passed from one cluster to an adjacent cluster; the message has the dimensions of the variables which are common to the two clusters. These messages represent the effect of evidence on the common variables, and are computed by marginalizing or "summing out" all the variables which are not dimensions of the message. When a cluster receives a message from its neighbour, it builds a new potential by taking the product of the message and its previous potential. The resulting potential represents the joint probability distribution of the variables in the cluster, given the evidence seen from all messages. The tree structure representation allows a cluster to send a message along any edge from which it did not receive a message.

There are two ways to propagate the effects of evidence throughout the join tree. The first is called "DistributeEvidence," which sends the effects of observed evidence from a particular cluster to the remaining clusters. The process is started by a cluster containing evidence, which sends a message to each of its neighbouring clusters. The process continues as clusters that have received a message send messages along edges from which it did not receive a message. The process ends when messages reach the leaves of the join tree.

The second method is called "CollectEvidence," which gathers messages from all clusters. Clusters with only one edge start the process by sending messages along that edge; the remaining clusters propagate the messages to its neighbours from which it did not receive a message. The collecting cluster does not send out any messages; its potential is the product of its original potential, and the effects of all evidence observed, *i.e.*, it represents the joint distribution of the variables in the cluster and the evidence observed throughout the network.

The complexity of computing posterior probabilities in a join tree depends on the size of the largest cluster *i.e.*, the product of the number of values of each node in the cluster; for sparse networks, the number of nodes in a cluster will be small enough to make computations in join trees very effective.

Clique–tree propagation has a very useful property which we exploit. For a fixed collection of evidence, the posterior probability for all nodes in the network can be computed once the evidence has been distributed to all the clusters in the cluster tree.

A Bayesian network can be used to structure query-based computations of posterior probability [27, 52, 53]. For any given query, the posterior probability can be computed by marginalization, *i.e.*, summing out the random variables not mentioned in the query. The remaining nodes are structured by the Bayesian network's factorization of the joint probability distribution, and the query can be computed by taking their product. This general technique depends highly on the *elimination ordering*; an effective ordering minimizes the number of summations and multiplications in the process.

Because the problem of computing posterior probabilities in Bayesian networks is NP-hard in general, approximation techniques have been developed. Approximating posterior probabilities to within a given error bound has also been shown to be an NP-hard problem [6]. Approximation schemes include those based on stochastic simulation [14, 4] and those based on search [34].

2.2 Influence Diagrams

An influence diagram (ID) is a directed acyclic graph representing a sequential decision problem under uncertainty [21]. An ID models the subjective beliefs, preferences, and available actions from the perspective of a single decision maker. An influence diagram augments the Bayesian network representation with the ability to represent events which the decision maker can control (decision nodes) and nodes which represent the decision maker's preferences (value nodes).

Nodes in an ID are of three types. Circle shaped *chance* nodes represent random variables, *i.e.*, events 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. See Figure 2.2.

Arcs represent dependencies. Arcs into chance nodes represent probabilistic dependency. As in a Bayesian network, a chance node is conditionally independent of its non-descendants given its parents. There is a conditional probability distribution associated with every chance node, quantifying the probabilistic dependency of the child on the parent nodes. If a chance node has no parents, the probability distribution is unconditional.

An arc into a decision node represents an information dependency. A decision maker will observe a value for each of a decision node's direct predecessors before an action must be taken. Therefore, the parents of a decision node are called *information predecessors*; they can be either chance nodes or decision nodes.

The decision maker's preferences are expressed as a function of the value

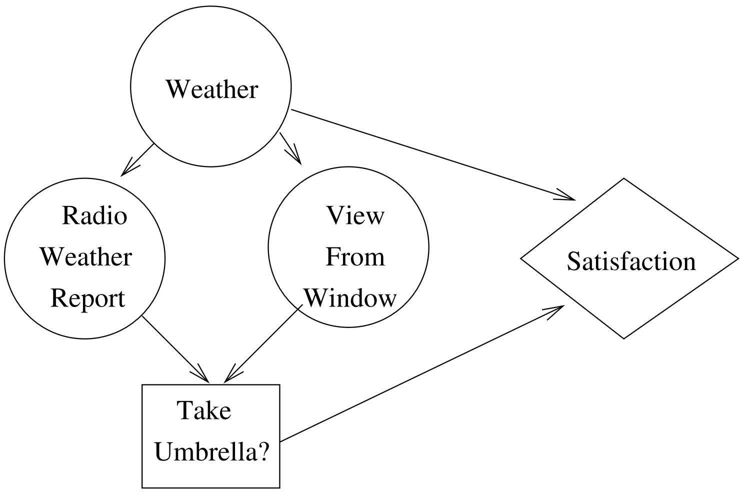


Figure 2.2: The extended weather decision problem.
This simple influence diagram describes the problem in which a decision maker must choose whether to bring an umbrella, based on information gained from listening to the radio and looking out the window. Only the dependency information is shown. The state space for the events, the conditional probabilities, and the value function are given in Table 2.1.

node's parents. Arcs into the value node represent functional dependency.

The notation for random variables presented in the previous section will be used for influence diagrams as well. The set of parents of a decision node D is denoted Π_D, and these will be called information predecessors. The set Ω_D is the set of available actions represented by the decision node D. 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 for D*. Value nodes take real ($\Re$) values, and often we will normalize these values to $[0, 1]$ or $[0, 100]$.

The conditional probability tables and the value function contain numerical information which represents the knowledge of the decision maker. In general, when posing a decision problem, this knowledge must be elicited from the decision maker, and it can be a painstaking process. For our examples, we will present the numerical information as if the elicitation process had already been completed beforehand.

For example, Figure 2.2 shows a simple influence diagram, (Table 2.1 contains the numerical information for this example). The ID represents the information relevant to a hypothetical decision maker, whose problem is to decide whether to take an umbrella on an outing. The goal is to maximize the decision maker's expected *Satisfaction*, which depends on the *Weather* and decision maker's decision to *Take Umbrella?* The decision maker can choose to *Bring Umbrella*, or *Leave Umbrella*.

The decision maker has two sources of information: a *Radio Weather Report*, and the *View From Window*. These events are explicitly assumed to be independent given the weather, and both have three possible outcomes: *sunny*, *cloudy*, and *rainy*. The *Weather* is also an event, but it is not directly observable at the time an action must be taken; it has two states: *sunshine* and *rain*.

The three chance nodes, and the arcs between them form a small Bayesian network. To complete the specification of the problem, conditional probability tables of the form $\mathrm{P}(\textit{Weather})$, $\mathrm{P}(\textit{Radio Weather Report}|\textit{Weather})$, and $\mathrm{P}(\textit{View From Window}|\textit{Weather})$ are needed. These probabilities represent the decision maker's subjective assessment of the child's probabilistic dependence on its parents. For example, suppose the decision maker believes that sunshine is quite likely (probability 0.7), whereas rain is unlikely (probability 0.3). Also, suppose the decision maker believes that the radio weather report will likely predict sunny weather on a day which will turn out to be sunny, but may also predict cloudy or rainy weather. See Table 2.1.

The value function, $\textit{Satisfaction}(\textit{Weather}, \textit{Take Umbrella})$, is an assessment of the decision maker's preference. This numerical data is listed in Table 2.1. This table supposes that the decision maker prefers to be without the umbrella on a sunny day, but prefers to have the umbrella if it is raining. In this example the value function is expressed so that high values are preferred to low values. All the examples in this thesis will follow this convention.

A *policy* prescribes an action (or sequence of actions, if there are several decision nodes) for each possible combination of outcomes of the information predecessors. One of the possible policies for the above example directs the decision maker to take an umbrella, regardless of what the radio or the view from the window

$\mathrm{P(Weather)}$

sunshine	rain
0.700	0.300

$\mathrm{P(RadioWeatherReport|Weather)}$

sunny	cloudy	rainy	Weather
0.700	0.200	0.100	Sunshine
0.150	0.250	0.600	Rain

$\mathrm{P(ViewFromWindow|Weather)}$

sunny	cloudy	rainy	Weather
0.800	0.150	0.050	Sunshine
0.100	0.150	0.750	Rain

Satisfaction

	Weather	Take Umbrella
20.000	sunshine	take it
100.000	sunshine	leave at home
70.000	rain	take it
0.000	rain	leave at home

Table 2.1: Numerical data for the extended weather problem.
This data completes the specification of the influence diagram in Figure 2.2. The conditional probability tables for the chance nodes in the influence diagram are listed, as well as the value function for the decision maker's satisfaction.

may indicate. An *optimal policy* is a policy which maximizes the decision maker's expected value, 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 may be better off with a policy which is not optimal in the above sense.

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

An influence diagram is *oriented* if it has a value node; an oriented influence diagram indicates how the outcomes and choices are relevant to the decision maker. An influence diagram is *regular* if the value node has no successors and if there is a directed path containing all of the decision nodes. The directed path in a regular influence diagram indicates a temporal ordering. A regular influence diagram with k decision nodes is said to have k stages. An action taken at the ith stage, *i.e.*, the ith decision node, occurs after all the actions at decision nodes which precede it, and before all actions at stages after it. An influence diagram is called "no-forgetting" if a decision node D_i and all its parents Π_{D_i} are also parents of all decision nodes later in the sequence. The "no-forgetting" property implies that a decision maker can always remember the actions and information of the preceding stages.

2.2.1 Making decisions with influence diagrams

An influence diagram represents a decision problem. Decision making is a deliberative process, the result of which is a policy, mapping information states to actions. The decision maker executes the policy by observing the actual state of the information predecessors, and taking the action dictated by the policy for this information state. As the decision maker moves forward during execution, the decision maker's history of observations and actions increases linearly with the number of actions taken. Thus, the assumption that a decision maker will not forget its own history is reasonable for small finite stage decision problems.

Given an influence diagram, a policy can in principle be constructed by choosing a sequence of actions for each information state. For a decision node with k binary information predecessors, the number of information states for the decision node is $O(2^k)$. The number of decision functions for a binary decision node with k predecessors is therefore $O(2^{2^k})$.

Enumerating the possible policies is only feasible for very small problems. More efficient techniques are used to construct policies by enumerating the information space, *i.e.*, enumerating the possible histories of the agent. The information space is smaller than the space of possible policies, but it can still be quite large; if the k predecessors of the last decision node have b values each, then information space has b^k states.

A number of algorithms have been developed which are based on the idea of dynamic programming. The basic approach is presented first, followed by a brief account of some specific algorithms. Our presentation is based on [36], Chapter 7, This review has several purposes. It emphasizes the inductive nature of the approach, which constructs an optimal policy starting at the end of the decision sequence. As well, it demonstrates the importance of expected value during the computation of an optimal policy. In particular, we point out that expected value is defined in the dynamic programming approach because the information states are enumerated exhaustively. In the information refinement approach to policy construction, we take special steps to define expected value for incomplete contexts (Chapter 4).

The dynamic programming approach computes an optimal policy in stages. We define inductively the function v_k, which measures the expected value of acting according to the optimal policy starting from stage k.

We start with the base case of the induction. The value function for value node V is $v : \Omega_{\Pi_V} \to \Re$. We define $v_{n+1} = v$ to be a synonym for the value function.

For the kth stage of the decision problem, the dynamic programming approach has constructed an optimal policy, denoted $\Delta_{k+1}^n = \{\delta_{k+1}^*, \ldots, \delta_n^*\}$, for decision nodes $D_{k+1}, \ldots, D_n$.

Let $E[v_k|d, w]$ represent the expected value of taking an action $d \in \Omega_{D_k}$ at stage k in context $w \in \Omega_{\Pi_{D_k}}$, followed by acting according to the optimal policy Δ_{k+1}^n for the remaining execution steps:

$$E[v_k|d, w] = \sum_{x \in \Omega_{\Pi_{D_{k+1}}}} v_{k+1}(x)\mathrm{P}(x|d, w)$$

The inductive step at this stage is to use this policy to determine a decision function δ_k for decision node D_k. A case analysis of the information space for D_k

is performed such that, for every $w \in \Omega_{\Pi_D}$ the decision maker determines

$$\delta_k^*(w) = \arg\max_{d \in \Pi_D} E[v_k | d, w]$$

The case analysis is performed to determine δ_k because the optimal policy is not yet known for the decision nodes $D_1, \ldots, D_{k-1}$; the case analysis examines all the possible sequences of action before stage k.

The value of following this policy starting in an information state $w \in \Omega_{\Pi_{D_k}}$ at stage k is given by the value function $v_k : \Omega_{\Pi_{D_k}} \to \Re$.

$$v_k(w) = E[v_k | w, \delta_k^*(w)]$$

Example: The Car Buyer

The Car Buyer problem is a well known example from the literature [20, 36, 37] which we use to illustrate the techniques of this chapter.

The influence diagram in Figure 2.3 represents the knowledge relevant to a decision maker deciding whether or not to buy a particular used 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. The car could be a “lemon” (a bad purchase) or a “peach” (a good purchase). A “lemon” is defined as a car with 6 defective subsystems, and a “peach” has only one defective subsystem. The decision maker’s model of the car counts 10 subsystems in total. The decision maker’s prior beliefs about the car indicate that the car is probably a peach (with probability 0.8), but could be a lemon (with probability 0.2). It will cost the decision maker \$40 to make a “peach” road-worthy, but \$200 to repair a “lemon”.

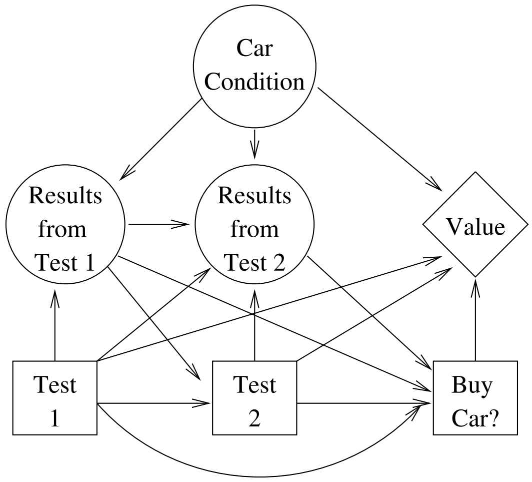


Figure 2.3: The Car Buyer influence diagram.
The decision maker must decide to buy a particular car or not, based on the information gathered in optional tests.

The car's condition is modelled by a chance node, labelled *Car Condition* in the influence diagram. Its two values are *peach* and *lemon*.

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. There are three test options available: check the steering subsystem alone, at a cost of $9; test both the fuel and the electrical subsystems, at a cost of $13; perform a two test sequence, starting with the transmission, at a cost of $10, then optionally, testing the differential, at an additional cost of $4.

These tests are modelled in the influence diagram by a two decision node sequence. The first decision node is labelled *Test 1*, and its values are *no test, steering, fuel and electrical system*, and *transmission*. The second decision node completes the optional two test sequence of checking the transmission and differential; it is labelled *Test 2*, and has values *no test* and *differential*.

The test results are modelled using chance nodes which count the number of defects found in the test. The node *Results from First Test* has values *no results, no defects, one defect, two defects*. The tests are guaranteed to find a defect in a tested subsystem if the subsystem is actually defective. If no test were performed, no results are available; otherwise, the test could find no defects, or up to 2 defects. The *Results from Test 2* are only available if the decision maker performed the first test on the transmission, and decided to follow through with the test on the differential.

The third decision node, labelled *Buy Car* represents the options available to the decision maker with respect to purchasing the car. The decision maker can decide not to buy it, to purchase the car at the price of \$1000, or to purchase the car at the higher price of \$1060 which includes a guarantee that the dealer will pay the full repair cost if the car is a "lemon." Before making the decision to buy, the decision maker has all the test and result information available.

The value function for this problem is an account of the dollar costs involved. The dealer's price for the car is \$1000, and the blue-book price of the car is \$1100; so the car is a good value if it is not a lemon. The tests are available at the indicated costs, so the decision nodes representing the tests are connected to the value node.

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. This problem is well known for its asymmetry; some combinations of tests and results are logical impossibilities. The conditional probability tables for this problem can be found in Table 2.2.

The exhaustive enumeration method reviewed in this section performs the following steps. We start with the last decision node, *Buy Car*. It has 3 actions, and 4 informational predecessors, which combine to form an information space with 96 states. For each information state, the process finds the optimal action by computing the expected value for all actions in the given state. Since the car's condition is not observable directly, the best action is based on the expectation of the car's condition. For each tuple $(t_1, r_1, t_2, r_2) \in \Omega_{\Pi_B}$

$$\delta_3(t_1, r_1, t_2, r_2) = \arg\max_{b \in \Omega_B} \sum_{c \in \Omega_C} v(t_1, t_2, b, c)\mathrm{P}(c|r_1, r_2)$$

where v represents the value function at the value node. In this notation, we have abbreviated the node labels using the initial letter, *e.g.*, the node labelled *Buy Car* is denoted B, and b is a variable which is constrained to the set Ω_B; *e.g.*, the first test is denoted T_1, and t_1 is a variable which is constrained to the set Ω_{T_2} *etc.*. The expected value of starting in a given state (t_1, r_1, t_2, r_2) and following the δ_3 is given by:

$$v_3(t_1, r_1, t_2, r_2) = \sum_{c \in \Omega_C} v(t_1, t_2, \delta_3(t_1, r_1, t_2, r_2), c)\mathrm{P}(c|r_1, r_2)$$

The decision node *Test 2* has two actions, and 2 information predecessors; the information space has a total of 16 states. For each information state, the decision maker computes the expected value of all choices for this test, given the state. The expected value is computed assuming that the decision maker will follow the decision function δ_3. For each tuple $(t_1, r_1) \in \Omega_{\Pi_{T_2}}$

$$\delta_2(t_1, r_1) = \arg\max_{t_2 \in \Omega_{T_2}} \sum_{r_2 \in \Omega_{R_2}} v_3(t_1, r_1, t_2, r_2)\mathrm{P}(r_2|t_1, r_1, t_2)$$

$\mathrm{P}(\mathrm{CarCondition})$

peach	lemon
0.8	0.2

$\mathrm{P}(\mathrm{Result1}|\mathrm{Test1}, \mathrm{CarCondition})$

no result	zero	one	two	Test 1	Car Condition
1.0	0	0	0	no test	all
0	0.9	0.1	0	steering	peach
0	0.4	0.6	0	steering	lemon
0	0.8	0.2	0	fuel and elect.	peach
0	0.13	0.53	0.33	fuel and elect.	lemon

$\mathrm{P}(\mathrm{R2}|\mathrm{Test1}, \mathrm{Result1}, \mathrm{Test2}, \mathrm{CarCondition})$

no result	zero	one	Test 1	Result 1	Test 2	Car Condition
1.0	0	0	no test	all	all	all
1.0	0	0	steering	all	all	all
1.0	0	0	fuel and elect.	all	all	all
1.0	0	0	transmission	no result	all	all
0	0.89	0.11	transmission	zero	differential	peach
0	0.67	0.33	transmission	zero	differential	lemon
0	1.0	0	transmission	one	differential	peach
0	0.44	0.56	transmission	one	differential	lemon
1.0	0	0	transmission	two	all	all

Table 2.2: Conditional probability tables for the Car Buyer problem.
Because many of the entries in the tables are zero, not all of them are explicitly listed. The label "all" indicates that the probability distribution applies for all values of a random variable.

The expected value of starting in a given state (t_1, r_1) and following δ_2 and δ_3 is given by

$$v_2(t_1, r_1) = \sum_{r_2 \in \Omega_{R_2}} v_3(t_1, r_1, \delta_2(t_1, r_1), r_2) \mathrm{P}(r_2 | t_1, r_1, \delta_2(t_1, r_1))$$

Finally, the last decision node has no predecessors; the optimal action at this stage is given by

$$\delta_1 = \arg \max_{t_1 \in \Omega_{T_1}} \sum_{r_1 \in \Omega_{R_1}} v_2(t_1, r_1) \mathrm{P}(r_1 | t_1)$$

The expected value of following the policy $\{\delta_1, \delta_2, \delta_3\}$ is given by

$$v_1 = \sum_{r_1 \in \Omega_{R_1}} v_2(\delta_1, r_1) \mathrm{P}(r_1 | \delta_1)$$

Algorithms for computing optimal policies

The basic dynamic programming approach is employed in several algorithms. The original technique converts an ID to a symmetric decision tree [21]. A decision tree is a structure which explicitly represents all the combinations of action and outcome. The root node of a decision tree is the first decision, and a path from the root to a leaf in the tree identifies a sequence of actions to be taken, and a possible state that the decision maker might face. Each action is the root of a subtree which identifies the possible outcomes for that action, and the leaves of the tree identify the decision-maker's preference function for that state. The decision tree can be used to compute the optimal policy by folding back the value of the leaf state, weighted by the probability of that state occurring. A policy is created by selecting the action which leads to the highest expected value of the possible outcomes.

Shachter [45] describes an algorithm which applies the dynamic programming approach directly to the influence diagram. This algorithm works backwards from the last decision, computing the optimal choice for each informational state by selecting the action which maximizes the expected value for the information state. This is accomplished by a sequence of value preserving reductions on the graph.

An influence diagram can be converted into a Bayesian network [44, 22], and an optimal policy can be constructed by computing posterior probabilities in the network. One of the advantages to this technique is that the computation of posterior probabilities can be done using efficient algorithms developed for Bayesian networks.

In the transformation proposed by Shachter and Peot [44], the value node V is converted to a binary chance node with values $\{true, false\}$. To avoid ambiguity, the chance node derived from the value node in the influence diagram will be called the value/chance node in the Bayesian network. The parents of the value node are parents of the value/chance node. The value function $v : \Omega_{\Pi_V} \rightarrow \Re$ is normalized to $[0, 1]$ and these normalized values are used as the probability distribution for the value/chance node, *i.e.*, for each $w \in \Omega_{\Pi_V}$, $\mathrm{P}(V = true|w) \propto v(w)$. A decision node in the influence diagram is converted to a chance node with a uniform probability distribution over its possible values for all its direct predecessors. Chance nodes in the influence diagram are not changed. This transformation results in a Bayesian network whose probability model represents the expected object value of the decision maker's initial policy of acting randomly with uniform probability over its actions.

Figure 2.4 shows a graphical example of the conversion. The conditional probabilities for the chance nodes *Take Umbrella* and *Satisfaction* are shown in

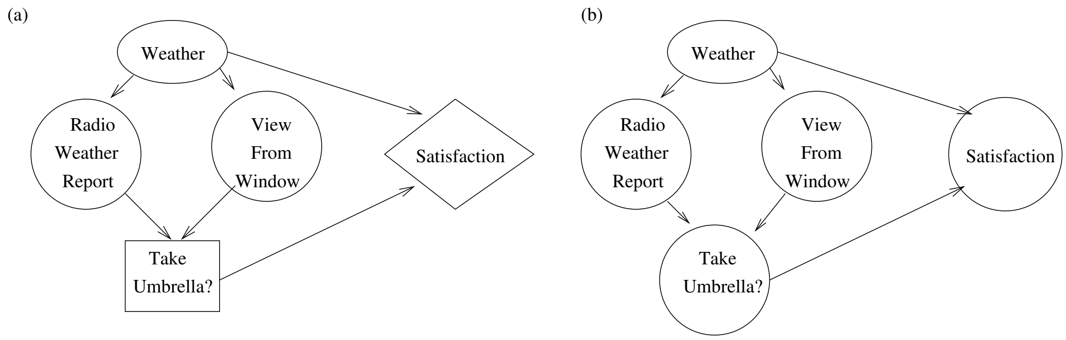


Figure 2.4: The transformation of an influence diagram.
(a) The Extended Weather Problem influence diagram, and (b) a Bayesian network used to compute expected value. The decision node and the value node are converted to chance nodes.

Table 2.3 (the distributions for the remaining chance nodes are unchanged from Table 2.1).

Shachter and Peot [44] show that, in general, the best action $d^* \in \Omega_D$ for a given information state $w \in \Omega_{\Pi_D}$ can be found by choosing the action $d \in \Omega_D$ which maximizes $\mathrm{P}(d, w|V = \mathit{true})$. The normalized expected value of taking action $d \in \Omega_D$ in information state $w \in \Omega_{\Pi_D}$ is given by $\mathrm{P}(V = \mathit{true}|d, w)$:

$$\begin{aligned} E[v|d,w] &= \sum_{u \in \Omega_{\Pi_V}} v(d,u)\mathrm{P}(u|d,w) \\ &\propto \sum_{u \in \Omega_{\Pi_V}} \mathrm{P}(V = \mathit{true}|d,u)\mathrm{P}(u|d,w) \\ &= \mathrm{P}(V = \mathit{true}|d,w) \end{aligned}$$

where $v(d, u)$ is the value function at the value node. The optimal action in a given context maximizes $\mathrm{P}(d|V = \mathit{true}, w)$. By Bayes' rule, we have:

$$\arg\max_{d \in \Omega_D} E[v|d,w] \quad = \quad \arg\max_{d \in \Omega_D} \mathrm{P}(V = \mathit{true}|d,w)$$

$\mathrm{P(TakeUmbrella|RadioWeatherReport, ViewfromWindow)}$

Take It	Leave At Home	Radio Weather Report	View from Window
0.5	0.5	sunny	sunny
0.5	0.5	sunny	cloudy
0.5	0.5	sunny	rainy
0.5	0.5	cloudy	sunny
0.5	0.5	cloudy	cloudy
0.5	0.5	cloudy	rainy
0.5	0.5	rainy	sunny
0.5	0.5	rainy	cloudy
0.5	0.5	rainy	rainy

$\mathrm{P(Satisfaction|Weather, TakeUmbrella)}$

True	False	Weather	Take Umbrella
0.2	0.8	Sunshine	Take It
1.0	0.0	Sunshine	Leave At Home
0.7	0.3	Rain	Take It
0.0	1.0	Rain	Leave At Home

Table 2.3: Data for the Bayesian network in Figure 2.4b.
In the transformation of the influence diagram, the value node Satisfaction *is converted to a binary chance node; its conditional probabilities are the normalized values. The decision node,* Take Umbrella *is converted to a chance node, and given a uniform conditional probability table. The conditional probability distributions for the remaining chance nodes are found in Table 2.1.*

$$
\begin{aligned}
&= \arg\max_{d \in \Omega_D} \frac{\mathrm{P}(d|V = \mathit{true}, w)\mathrm{P}(V = \mathit{true}|w)}{\mathrm{P}(d|w)} \\
&= \arg\max_{d \in \Omega_D} \mathrm{P}(d|V = \mathit{true}, w)
\end{aligned}
$$

The equality holds since $\mathrm{P}(d|w)$ is a uniform distribution, and does not affect the maximum; as well, $\mathrm{P}(V = \mathit{true}|w)$ is a constant. This result justifies the use of a Bayesian network, transformed from an influence diagram as described above, to compute expected value.

Both of these quantities can be computed in a Bayesian network transformed from an influence diagram as described above. For example, the Bayesian network can be compiled into a join tree, which can compute posterior probabilities efficiently, using DistributeEvidence and CollectEvidence operations [23]. As described in Section 2.1, once evidence has been distributed throughout the network, the posterior probability of any chance node (given the evidence) in the network can be ascertained without additional cost.

Each query for posterior probability is potentially expensive, since computation in Bayesian networks is NP-hard [5, 6]. However, any method for computing expected value in an influence diagram is going to incur similar costs.

Using this transformation, a decision problem can be solved by a series of computations in the Bayesian network. Starting with the last decision node, D_k, the process establishes an optimal decision function by making the query $\mathrm{P}(D_k|V = \mathit{true}, w)$ for each $w \in \Omega_{\Pi_{D_k}}$; this query returns a probability distribution over the possible actions in Ω_{D_k}. The best action for each w maximizes this distribution, as shown above. The optimal decision function for the decision node is then *in-*

stalled into the Bayesian network, by setting the conditional probability table of the decision node to:

$$\mathrm{P}(D = d|w) = \begin{cases} 1.0 & \mathrm{if} d = \arg\max_{d \in \Omega_D} \mathrm{P}(d|V = \mathit{true}, w) \\ 0.0 & \mathrm{otherwise} \end{cases}$$

This process is iterated over all the decision nodes in reverse time order: $D_k, \ldots, D_1$.

The number of queries required depends on the number of direct predecessors of the decision nodes. If the decision node has k direct predecessors, and each one has b values, there are b^k information states. Therefore, the number of queries needed to find an optimal decision function for a decision node is b^k. Under the assumption of no-forgetting, k increases at least linearly with the number of decision nodes.

The clique–tree propagation techniques, used to compute posterior probabilities in Bayesian networks, can be augmented by a maximization operation [44, 22]. The influence diagram is converted to a join tree; the potential functions for a cluster are as for Bayesian networks: the products of the conditional probability tables assigned to a cluster. The decision nodes do not have probability distributions, so their appearance in a cluster does not affect the initial potential for the cluster. When a message needs to be sent to an adjacent cluster, the maximization operation can be used to remove a decision node from the message, just as the "summing out" operation removes a chance node from the message. With this operation, a single "CollectEvidence" phase can compute an optimal policy with a single pass through the network [44, 22]. This pass examines the entire information space, as determined by the decision nodes' direct predecessors. However, the computations of expected value and posterior probability are computed more efficiently than by issuing queries.

2.3 Decision Trees

A decision tree is a representation for discrete functions. Decision trees are used in many ways: as the representation of a classification function in machine learning; as a representation of a decision problem in decision analysis.

Decision trees can be used to represent functions extensionally. Specific classes of discrete functions can be represented extremely succinctly using decision trees, and many functions can be approximated using decision trees.

In Chapter 3, an algorithm will be presented which uses decision trees to represent decision functions, *i.e.*, the mapping from the observations which a decision maker will make, to the action to be taken. This algorithm is closely related to the machine learning technique of *top-down induction of decision trees* (TDIDT). This section lays the groundwork for Chapter 3 by introducing terminology and notation, and emphasizes key issues for the comparison of TDIDT with the approach presented in Chapter 3.

2.3.1 Terminology and notation

A decision tree can be used to represent a discrete function. If $A_1, \ldots, A_k$ are finite sets (called attributes; elements of an attribute set are called values), and G is a finite set called the goal attribute, we can use a decision tree to represent the function:

$$f : A_1 \times \ldots A_k \to G$$

If the attributes are b-ary, there are b^k elements in the attribute space.

The decision tree representation uses the attributes as *internal vertices,* with a branch for each value of the attribute. The *leaf vertices* are elements of the goal predicate. The function is evaluated for a given input by descending through the tree; at each internal vertex the branch corresponding to the input's value for that attribute is descended. When a leaf is reached, the leaf value is the output of the function.

The path from the root of the decision tree to a given vertex is called a *context*. An attribute can appear at most once in any context.

2.3.2 Induction of decision trees

Decision tree induction is a machine learning technique which learns by example [38]. The object of decision tree induction is to learn a classification rule, represented by a decision tree, for a domain of interest. A set of examples of the domain are obtained, and are classified by an external agent. The examples consist of attributes and values for the domain. Given this set, called the *training set*, a decision tree can be constructed or "learned" such that the decision tree will classify the examples in the training set by examining the attributes of the data. The purpose of learning decision trees is to be able to classify examples which are not part of the training set.

A naive method for constructing decision trees is to generate all the possible trees which classify the training data and choose the smallest (the choice of the smallest tree which classifies the training data is a form of bias, and is usually justified by appeals to Ockham's Razor). This method is very expensive computationally. Decision trees are usually constructed inductively, as follows.

One of the attributes is chosen to be the root of the tree, the training set is split into k subsets, one subset for each value of the chosen attribute. The kth subset contains only those examples from the original set which are consistent with the kth value of the chosen attribute. A subtree is computed for each of the k subset using the remaining attributes

A key research issue is the choice of attribute at any level of the inductive recursion. The intuition is that a good choice of an attribute will lead to a better tree. This issue is clouded by a number of factors. First, it is desirable that the tree be consistent with the training set. There are many possible trees which fit this description, and smaller trees may be preferred to larger trees. The choice of attributes may have a significant impact on the size of the tree. Second, smaller trees are often preferred as a bias to prevent *over-fitting*, *i.e.*, the mistake of assuming that chance correlations in the training set are correlations in the population. Third, it seems that many of the learning domains to which TDIDT has been applied are favourably predisposed to classification [15]; the accuracy of decision trees constructed for many domains are not much more accurate than single level decision trees, *i.e.*, trees which use a single attribute to classify the domain.

Many heuristics have been studied, including information gain [38], information gain ratio [39], the GINI index [2], among many others.

Decision tree inducing algorithms are typically presented recursively (as above), and the result of the algorithm is a tree which classifies the training set.

The algorithm can be expressed iteratively as well. The motivation for an iterative version comes from the idea of having partial solutions available.

In the iterative approach, each leaf in the decision tree is associated with two sets: the set of training examples consistent with the path from root to leaf; and the set of attributes not used on the path from root to leaf. A leaf node is *extensible* if the list of unused attributes is not empty. An *extension* of the leaf consists of an internal node, representing one of the unused attributes, and a leaf node for each of the attributes values. The classification at each leaf is made by some criterion applied to the set of examples at the leaf; for example, simple majority can be used. For each new leaf, an attribute is chosen from the list of unused attributes, and each new leaf's training set is obtained by splitting the extended leaf's training set on the chosen attribute, *i.e.*, the training set is split into k subsets, the kth subset containing all the elements of the training set which are consistent with the kth value of the attribute.

The iterative algorithm shown in Figure 2.5 creates a sequence of trees; each is able to classify all the examples, although not necessarily correctly. The value of this algorithm is that it can be interrupted, and a useful tree is available. The iterative algorithm creates leaf nodes where the recursive algorithm would build a subtree. The iterative version also deletes extensible leaf nodes, refining the classification at that leaf. Thus the iterative version does more work than the recursive version, but has partial results available for use as a classifier. This approach may be of value to a learning agent which must build learn a classification when there is no time to learn a larger tree.

```
procedure Decision-Tree-Learn
  Input:
    training set T
    attribute set A
    default classification c
  Output:
    a decision tree

  Start with the tree as a single leaf, c
  Do {
    Choose an extensible leaf
    Replace the leaf with an extension:
      Choose an unused attribute
      Split the leaf's training set
      Create new leaf nodes
  } While there are extensible leaf nodes
  Return the tree
```

Figure 2.5: An algorithm for learning decision trees from data.

Our approach to constructing policies for influence diagrams is based on the induction of decision trees. Instead of training examples, we have an information space. Instead of classification at the leaf vertices, we will have actions. We also develop a number of heuristics which choose which internal vertices to create.

2.4 Summary

This chapter presented the problem of decision making under uncertainty. We presented Bayesian networks, which structure the probabilistic knowledge for efficient computation of posterior probabilities. Influence diagrams were presented as representations of a class of sequential decision problems under uncertainty, and we outlined some methods which can compute optimal policies. We discussed the machine learning technique of decision tree learning, which is the basis for the algorithms to be proposed in the following chapters. Chapter 3 presents an algorithm for single-stage influence diagrams; Chapter 4 develops similar algorithms for multi-stage problems.

Chapter 3

Information Refinement

This chapter presents *information refinement*, an incremental approach to the construction of decision functions; each increment includes more of the information available to the decision maker at the time an action must be taken. Decision functions are represented as decision trees, where internal vertices are information predecessors, and leaf vertices are actions. This approach is similar to learning classification trees in machine learning as discussed in Section 2.3, and other work the compilation of decision models into simple rules which can be executed by human decision makers [12, 26].

The advantages of this approach are that information refinement is incremental, and small decision trees which use only a small subset of the available information can be computed with relatively little cost. The optimal policy may not be representable by small decision trees; however, if the decision maker cannot assume negligible computational costs, small decision functions may have positive comprehensive value, whereas large decision trees may not.

This chapter presents information refinement for single stage decision problems. Chapter 4 extends the ideas to multi-stage decision problems.

3.1 Decision trees as decision functions

A decision function for a decision node maps information states to actions. In this section we formalize the representation of decision functions as decision trees. We emphasize that in this context, decision trees are used to represent the "solution" to decision problems; this is in contrast to the use of decision trees to represent decision problems (as in [40]).

Let D be the decision node in a single-stage influence diagram. A *decision tree t for D* is either a leaf vertex labelled by one of the actions $d_j \in \Omega_D$, or a tree whose root is a non-leaf vertex labelled with some information predecessor $X \in \Pi_D$, and whose subtrees are also decision trees. Each non-leaf labelled with X has an edge for every value $x_k \in \Omega_X$; the edge for $x_j \in \Omega_X$ corresponds to the *variable assignment* $X = x_j$. 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.

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 (this is guaranteed by the fact that each internal vertex has an edge for each of its values). Such a path from root to leaf l is called a *leaf context*, and is written γ_l.

Note that a leaf context need not contain an assignment for every node in

Π_D. If the context contains all elements of Π_D, it is called *complete*; if every leaf on a decision tree has a complete context, the tree is called complete.

The action at leaf l is the action to be taken for any information state which is consistent (in its variable assignments) with γ_l. If a node is not included in a given leaf context, the action at the leaf is taken for all values of that node. Thus, a single leaf context can summarize many information states, and a single action is specified for all information states summarized by a leaf context.

The probability of a context $\mathrm{P}(\gamma)$ is the probability of the variable assignments in the context according to the probability distribution represented in the influence diagram. The probability is marginalized over all the chance nodes not in the context.

Example Recall the simple decision problem about umbrellas described in Chapter 2 (see Figure 2.2 on page 19). A decision function in the form of a decision tree for this problem is shown in Figure 3.1. The root of the tree, namely *View from Window*, is an information predecessor for the decision. The three edges out of the root correspond to the three values for the node *View from Window*. Two of these three edges lead directly to leaf vertices, which are labelled with actions. The middle edge leads to the second information predecessor, *Radio Weather Report*. The three edges from this vertex correspond to the node's three values. Each edge from *Radio Weather Report* leads to a leaf.

The tree can be interpreted as a policy for the influence diagram in Figure 2.2 as follows. Suppose that the decision maker has already observed the world, and observes that the view from the window is cloudy, and the weather report from the

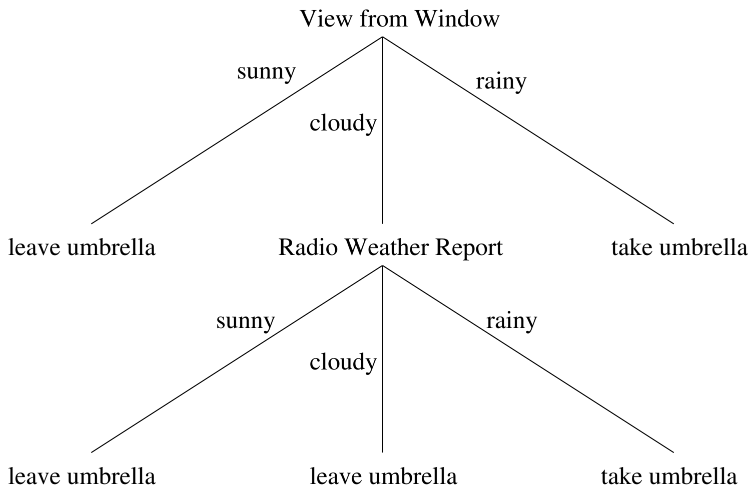


Figure 3.1: A decision tree representation of a policy.
A decision tree representation of a policy for the Weather problem in Figure 2.2.

radio indicated sunny weather. Starting at the root of the tree, the decision maker takes the middle branch, corresponding to the observation from the window, and then takes the left branch, corresponding to the radio report. The leaf indicates that the decision maker should leave the umbrella at home. If the view from the window had been either sunny or rainy, the information from the radio would not have been used, as both the left and right branches lead directly to actions which do not depend on the radio report.

The value node V in an influence diagram represents a value function $v : \Omega_{\Pi_V} \to \Re$. The expected value of an action $d \in \Omega_D$ in context γ is given by:

$$E[v|d, \gamma] = \sum_{w \in \Omega_{\Pi_V}} v(w)\mathrm{P}(w|d, \gamma)$$

The *expected object value* of a decision tree t is defined as follows:

$$E_t = \sum_{l \in t} E[v|d_l, \gamma_l]\mathrm{P}(\gamma_l)$$

where $\mathrm{P}(\gamma_l)$ is the probability of the context γ_l according to the probability distribution represented in the influence diagram. The summation is over all leaves in t. This definition corresponds to the classical definition of expected value of a policy.

Decision functions could also be represented as tables: each row in the table specifies a unique information state and the action prescribed for the state. This table grows exponentially in the number of information predecessors for the decision node. A complete tree is equivalent to a complete table. Exhaustive enumeration of the information space (using any of the algorithms discussed in Section 2.2) is equivalent to building a complete tree, and specifying an optimal action for each leaf context.

The key idea behind using decision trees to represent decision functions is that leaf contexts have the potential to summarize many information states. There are two situations in which decision trees have representational advantage over the tabular representation:

- Small decision trees may be able to represent the optimal policy compactly.
- Small decision trees may be able to approximate an optimal policy, or represent a policy which is approximately optimal.

In the next section, we present an incremental algorithm for building decision trees for single stage decision problems. The algorithm is intended to build decision trees which are not complete, but are valuable to the decision maker. An incomplete decision tree will be valuable to the decision maker if it is the result of a good trade-off between object value and resource costs.

3.2 Information refinement

Information refinement constructs decision functions in a manner similar to the way decision trees are learned in machine learning, [38, 39, 2]. In decision tree learning, a finite set of training examples is used to induce a decision tree which classifies the training set. The information refinement algorithm "classifies" the finite set of information states, where the "class" is an action to be taken in a context. Decision tree learning algorithms split the set of training instances using attributes; information refinement splits incomplete leaf contexts using information predecessors. In both cases, heuristics are used to determine how to split the branch.

A difference in the two tasks is the motivation for preferring "small" trees. In decision tree learning, small trees are often preferred to large trees, all else being equal: this preference is a bias used to avoid over-fitting. In contrast, information refinement prefers small decision trees if they represent a good trade-off between object value and computational costs.

The main idea in information refinement is that a decision function is constructed incrementally, starting from a very simple tree. It is intended that each incremental step result in an improved policy. Furthermore, the tree always represents a policy, in the sense that it always has an action for every information state.

Let D be the decision node in the influence diagram, and let t be a decision tree for D. The information predecessors which may be used in the decision tree are the parents Π_D of the decision node D. For a given leaf l in t, its context γ_l is *extensible* if it is not complete and if $\mathrm{P}(\gamma_l) > 0$. Let ξ_l be the set of information predecessors which are not in the context; this set is called the *possible extensions* *for* l.

Let l be an extensible leaf with context γ_l and possible extensions ξ_l. The context γ_l is refined by choosing a single information predecessor $X \in \xi_l$, creating a new context $x_j\gamma_l$ for each of the values $x_j \in \Omega_X$. For each new context, an action is chosen which maximizes the expected object value for the context $x_j\gamma_l$:

$$d_j = \arg\max_{d_i \in \Omega_D} E[v|d_i, x_j, \gamma_l] \tag{3.1}$$

This action will be called the *MEV* action. Section 3.3 shows how to compute the *MEV* action.

The remainder of this chapter assumes that the action at the leaf of a decision tree is the *MEV* action for the given context.

In terms of decision trees, refining an extensible leaf's context corresponds to constructing a subtree with root X, and a leaf labelled d_j for each element of Ω_X, as above. This subtree is called an *extension subtree for the context* γ_l. A new decision tree t' is derived from t by replacing the leaf l in t with the extension subtree for γ_l.

The basic information refinement algorithm is given in Figure 3.2. The algorithm starts by determining the *MEV* action for the empty context. It proceeds thereafter by iteratively choosing an extensible leaf context, then choosing an information predecessor from the possible extensions with which to replace the leaf.

The algorithm produces a sequence of decision trees, each of which can be used by the decision maker at the time a decision must be made. The quality of the trees in this sequence depends in part on the choices made by the algorithm, and also on the properties of the decision problem. Because the *MEV* action is chosen at

```
procedure DT1
    Input:
      single stage influence diagram with decision node D
    Output:
      a decision tree for D

1.      Start with the tree as a single leaf, labelled with
          action d_j = arg max_{d_i ∈ Ω_D} E[v|d_i]
2.      Do {
2a.       Choose a leaf l whose context is extensible
2b.       Choose a node X ∈ ξ_l
2c.       Construct an extension subtree for l using X
2d.       Replace l with the extension subtree
2e.     } Until stopping criteria are met or tree is complete
3.      Return the tree
```

Figure 3.2: The information refinement algorithm for single stage influence diagrams.

each leaf, the expected object value is monotonically non-decreasing.[1] The issues of choosing a leaf context to extend, and choosing a new node X are discussed in Sections 3.4.2 and 3.4.1, respectively. We discuss the effects of these choices in their respective sections.

We can call `DT1` an anytime algorithm [7], since a best policy exists throughout its execution, and the expected object value of the policy is non-decreasing as resources are allocated.

There are three situations in which the algorithm could stop. First, the decision maker may have computational resources to complete the decision tree. We will see that a complete tree represents the optimal policy. A second way to end the refinement process would be for the decision maker to interrupt the information refinement process, and take as its policy the best decision tree constructed so far. Finally, the information refinement algorithm could stop after it has consumed an *a priori* allocation of resources, which the decision maker has deemed sufficient. This will depend on the resource constraints on the decision maker's deliberation.

3.2.1 Example: The Extended Weather Problem

One possible trace of the `DT1` algorithm for the construction of the decision tree in Figure 2.2 follows. Let D, V, R, S stand for *Take Umbrella*, *View from Window*, *Radio Weather Report* and *Satisfaction*, respectively.[2] This trace represents an ideal computation, making choices for steps 2a and 2b which are, from the point of view from an omniscient viewer, optimal; however, for purposes of the discussion, these

[1] This is proved in Section 3.4.1.
[2] D is for "decision."

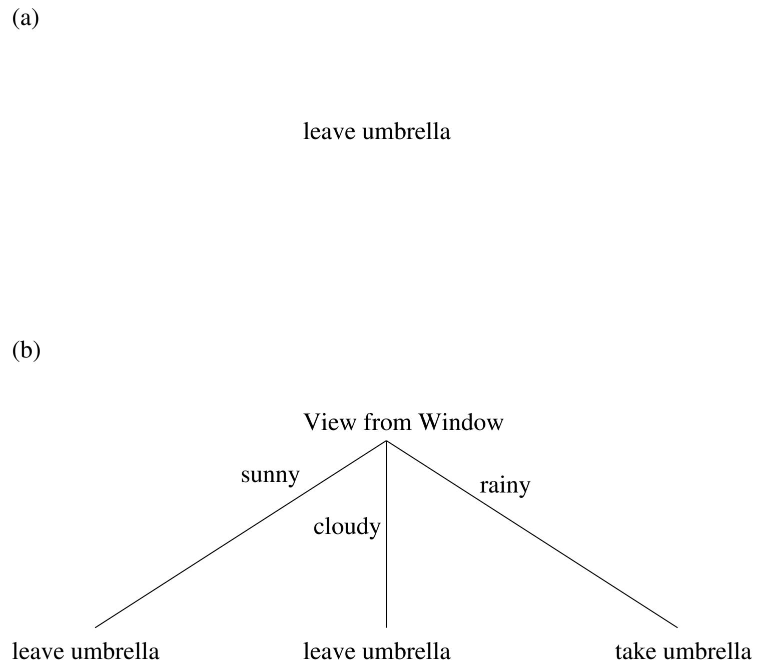


Figure 3.3: Two steps in the refinement process.
Two decision trees in the refinement of the Weather problem in Figure 2.2 are shown. (a) The initial decision tree; a single action, leave umbrella*, is prescribed for all information states. (b) The first refinement uses the information predecessor* View from Window*. The tree in Figure 3.1 is the third in this sequence of refinements.*

choices should be seen as arbitrary. The ways in which one might make these choices are discussed in later sections.

Before the algorithm begins, there is no decision tree (in the previous section, we used the description "the decision tree is empty"). If the decision maker is forced to act before any computation can be performed, the decision maker could choose some action from the possible actions at random.

Under the assumption that each choice is equally probable, the expected value of a random action is

$$\begin{aligned} E[\mathrm{S}] &= \frac{1}{2}\left(\sum_{w\in\Omega_{\Pi_D}} E[\mathrm{S}|D=\mathrm{take}, W=w]\mathrm{P}(W=w)\right. \\ &\quad \left. + \sum_{w\in\Omega_{\Pi_D}} E[\mathrm{S}|D=\mathrm{leave}, W=w]\mathrm{P}(W=w)\right) \\ &= 52.5 \end{aligned}$$

The algorithm initializes the decision tree by choosing an action which maximizes the expected object utility for the empty context. For $d \in \Omega_D$,

$$E[\mathrm{S}|D=d] = \sum_{w\in\Omega_{\Pi_D}} E[\mathrm{S}|D=d, W=w]\mathrm{P}(W=w)$$

In this example, d is one of *take umbrella* or *leave umbrella*. Using the numeric data from Section 2.2 (page 18), we have $E[\mathrm{S}|$*D=take umbrella*$] = 35.0$ and $E[\mathrm{S}|$*D=leave umbrella*$] = 70.0$. Thus the best choice, when none of the available information is in the decision function, is to leave the umbrella at home. The initial decision tree t_1 is shown in Figure 3.3(a), and the expected object value of this tree is 70.0.

The iterative process begins by choosing a leaf, and there is currently only one leaf in the tree, which has the empty context. The empty context is extended by choosing one of the information predecessor, say *View from Window*. An action is chosen for each of the values for this node; for each $d \in \Omega_D$ and $v \in \Omega_V$,

$$E[\mathrm{S}|d, v] = \sum_{r\in\Omega_R} E[\mathrm{S}|d, v, r]\mathrm{P}(r|v)$$

The respective numerical values are as follows:

$$
\begin{aligned}
E[\textit{S}|\textit{D=take umbrella, V=sunny}] &= 22.5424 \\
E[\textit{S}|\textit{D=leave umbrella, V=sunny}] &= 94.9153 \\
E[\textit{S}|\textit{D=take umbrella, V=cloudy}] &= 35.0 \\
E[\textit{S}|\textit{D=leave umbrella, V=cloudy}] &= 70.0 \\
E[\textit{S}|\textit{D=take umbrella, V=rainy}] &= 63.2692 \\
E[\textit{S}|\textit{D=leave umbrella, V=rainy}] &= 13.4615
\end{aligned}
$$

Choosing the *MEV* action for each observation, and replacing the single leaf with the extension subtree, results in the decision tree t_2 shown in Figure 3.3(b). The expected object value for this tree is 82.95.

The algorithm now chooses one of the three leaf vertices, say *View from Window* $=$ *cloudy*. There is only one possible extension for this context, namely *Radio Weather Report*. An action is chosen for each of the possible radio reports; for each $d \in \Omega_D$ and $r \in \Omega_R$, we compute $E[\mathrm{S}|drv]$, where v is the context of the leaf: the view from the window is cloudy. The respective numerical values are as follows:

$$
\begin{aligned}
E[\textit{S}|\textit{D=take umbrella, R=sunny, V=cloudy}] &= 24.2056 \\
E[\textit{S}|\textit{D=leave umbrella, R=sunny, V=cloudy}] &= 91.5888 \\
E[\textit{S}|\textit{D=take umbrella, R=cloudy, =cloudy}] &= 37.4419 \\
E[\textit{S}|\textit{D=leave umbrella, R=cloudy, =cloudy}] &= 65.1163 \\
E[\textit{S}|\textit{D=take umbrella, R=rainy, V=cloudy}] &= 56.0 \\
E[\textit{S}|\textit{D=leave umbrella, R=rainy, V=cloudy}] &= 28.0
\end{aligned}
$$

Choosing the *MEV* action for each new leaf context and replacing the leaf with the extension subtree results in decision tree t_3, as illustrated in Figure 3.1. The expected object value of the tree is 84.0.

The algorithm might proceed to extend the remaining two extensible leaf contexts. However, there is no value to the effort, since neither refinement will change the policy of the decision maker. For example, the algorithm could extend the leaf with context *V=sunny*. The split would consider the various values of the information predecessor *R*:

$$\begin{aligned}
E[S|D{=}\textit{take umbrella } R{=}\textit{sunny } V{=}\textit{sunny}] &= 20.5675 \\
E[S|D{=}\textit{leave umbrella } R{=}\textit{sunny } V{=}\textit{sunny}] &= 98.8651 \\
E[S|D{=}\textit{take umbrella } R{=}\textit{cloudy } V{=}\textit{sunny}] &= 23.1381 \\
E[S|D{=}\textit{leave umbrella } R{=}\textit{cloudy } V{=}\textit{sunny}] &= 93.7238 \\
E[S|D{=}\textit{take umbrella } R{=}\textit{rainy } V{=}\textit{sunny}] &= 32.1622 \\
E[S|D{=}\textit{leave umbrella } R{=}\textit{rainy } V{=}\textit{sunny}] &= 75.6757
\end{aligned}$$

In all of the information states consistent with *V=sunny*, the optimal action is to leave the umbrella, which is exactly what the leaf had indicated. The policy would not change with this refinement, even though the tree would increase in size. This is due to the fact that the observation *V=sunny* is a strong indication that the weather will be sunny, even if the radio report gives different information.

Refining the other leaf node would also result in a split which brought no value to the tree, and we do not show this detail. We leave the tree, for the purposes of our example, as in Figure 3.1, and end the trace.

The policy as given in Figure 3.1 is the smallest decision tree representing the optimal policy for this problem. A table expressing this policy would require 9 distinct entries (one for each combination of the two information predecessors. The decision tree representation requires only 5 distinct contexts.

The fact that the decision tree represents the optimal policy as well as it does is a property of the decision problem itself. The information predecessors are highly predictive of the actual weather in this model. One could modify the model (*i.e.*, the conditional probabilities and the value function) so that the optimal policy can only be expressed as a complete tree.

3.2.2 Properties of information refinement

For a decision tree t, and extensible leaf l, the *expected value of improvement*, $EVI_t(l, X)$, is the increase in expected object value of the decision function when γ_l is refined using $X \in \xi_l$, resulting in a new tree t':

$$EVI_t(l, X) = E_{t'} - E_t$$

This measure would be equivalent to the expected value of perfect information [29] if X were not an information predecessor. Since X is by definition part of the available information, the increase in object value is a rough measure of how much of the available computational resources a decision maker should invest in making the refinement. In contrast, the expected value of perfect information measures the most a decision maker should pay to have the information made available. In this case, the decision maker might also have to pay an additional amount for any computation making use of the new information.

We note that EVI_t is a myopic measure of value. It is possible that a single information predecessor X has $EVI_t(l, X) = 0$, at a particular leaf l, but when used in combination with some other possible extension, the expected object value of the tree would increase. A refinement which is known to result in no increase in expected object value is treated in two separate cases. If there are several possible extensions, and they all fail to increase the value of the tree, one of them is chosen to refine the tree. It is possible that a combination of extensions will improve a tree, when any single refinement will not. However, if there is only one information predecessor left in the possible extensions for a leaf, and it does not increase the value of the tree, it can be discarded. This can result in a small savings in space when representing a complete information state in the decision tree.[3]

The following proposition relates the EVI_t to the refinement of a context.

Proposition 1 Let t be a decision tree, with leaf l with context γ_l and $X \in \xi_l$.

$$EVI_t(l, X) = \mathrm{P}(\gamma_l) \sum_{x_j \in \Omega_X} (E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l) - E[v|d_l, \gamma_l])$$

where d_j is the action labelling the leaf vertex corresponding to $x_j \in \Omega_X$, and d_l is the action labelling leaf vertex l.

Proof: By definition:

$$E_t = \sum_{k \in t} E[v|d_k, \gamma_k]\mathrm{P}(\gamma_k)$$

where the sum is over the leaf vertices in the tree t. By definition,

$$EVI_t(l, X) = E_{t'} - E_t$$

Thus

$$EVI_t(l, X) = \sum_{k \in t'} E[v|d_k, \gamma_k]\mathrm{P}(\gamma_k) - \sum_{k \in t} E[v|d_k, \gamma_k]\mathrm{P}(\gamma_k)$$

[3]Note that a larger reduction may be attained by stopping the information refinement process before the contexts in the tree near completion. The value of such a tree is investigated in Section 3.5 (p. 83).

Now t' differs from t only at leaf l.

$$\begin{aligned} EVI_t(l, X) &= \sum_{k \neq l} (E[v|d_k, \gamma_k]\mathrm{P}(\gamma_k) - E[v|d_k, \gamma_k]\mathrm{P}(\gamma_k)) \\ &\quad + \sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j, \gamma_l) - E[v|d_l, \gamma_l]\mathrm{P}(\gamma_l) \end{aligned}$$

The first term in the above expression sums to zero, since the trees are identical whenever $k \neq l$. The second term represents the value of the extension to t at l, and the third term is the previous value of the tree t at l. By the multiplication rule of probabilities:

$$\mathrm{P}(x_j, \gamma_l) = \mathrm{P}(x_j|\gamma_l)\mathrm{P}(\gamma_l)$$

Therefore:

$$EVI_t(l, X) = \mathrm{P}(\gamma_l) \sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l) - E[v|d_l\gamma_l]\mathrm{P}(\gamma_l)$$

from which the result follows. □

The proposition justifies the incremental nature of the algorithm: the increase in value due to extending each leaf can be computed independently. The value of an extension does not depend on the state of the rest of the tree. That is, the only context which can affect EVI_t is the context being extended. Finally, the proposition does not depend on the choice of *MEV* action which labels every leaf vertex; it only depends on the fact that only one context is changing. Proposition 3 will show that because the *MEV* action is used, EVI_t is non-negative.

The next property establishes that we can compare two different refinements to tree t at a given context by comparing the value of actions of the refined contexts.

Proposition 2 Let t be a decision tree with extensible leaf l, and $X, Y \in \xi_l$. Then $EVI_t(l, Y) > EVI_t(l, X)$ if and only if

$$\sum_{y_i \in \Omega_Y} E[v|d_i, y_i, \gamma_l]\mathrm{P}(y_i|\gamma_l) > \sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l)$$

where d_i is the action labelling the leaf vertex corresponding to $y_i \in \Omega_Y$ and d_j is the action labelling the leaf vertex corresponding to $x_j \in \Omega_X$.

Proof: By Proposition 1,

$$\begin{aligned}
& EVI_t(l, X) - EVI_t(l, Y) \\
&= \mathrm{P}(\gamma_l)\left(\sum_{x_j \in \Omega_X} E[v|d_j, x_j\gamma_l]\mathrm{P}(x_j|\gamma_l) - E[v|d_l, \gamma_l]\right) \\
&\quad -\mathrm{P}(\gamma_l)\left(\sum_{y_j \in \Omega_Y} E[v|d_j, y_j, \gamma_l]\mathrm{P}(y_j|\gamma_l) - E[v|d_l, \gamma_l]\right) \\
&= \mathrm{P}(\gamma_l)\left(\sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l) - \sum_{y_j \in \Omega_Y} E[v|d_j, y_j, \gamma_l]\mathrm{P}(y_j|\gamma_l)\right)
\end{aligned}$$

This difference is positive if and only if

$$\mathrm{P}(\gamma_l)\left(\sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l) - \sum_{y_j \in \Omega_Y} E[v|d_j, y_j, \gamma_l]\mathrm{P}(y_j|\gamma_l)\right) > 0$$

The result follows, since an extensible context has non-zero probability by definition. □

Again, this proposition does not depend on the use of the *MEV* action at the leaf vertices, but only on the fact that the tree remains unchanged except at the leaf which is being refined.

The next property establishes that EVI_t is non-negative, provided that *MEV* actions are chosen to label the leaf vertices according to Equation 3.1.

Proposition 3 Let t be a decision tree for decision node D with extensible leaf l, and $X \in \xi_l$. If t' is constructed from t by replacing the leaf l with an extension subtree rooted at X, having leaf vertices for each $x_j \in \Omega_X$ labelled with d_j such that

$$d_j = \arg\max_{d_i \in \Omega_D} E[v|d_i, x_j, \gamma_l]$$

then

$$EVI_t(l, X) \geq 0$$

Proof: Let $d_l \in \Omega_D$ be the action labelling the leaf l. By Proposition 1,

$$EVI_t(l, X) = \mathrm{P}(\gamma_l)\left(\sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l) - E[v|d_l, \gamma_l]\right)$$

Rewriting

$$E[v|d_l, \gamma_l] = \sum_{x_j \in \Omega_X} E[v|d_l, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l)$$

and rearranging the terms, we have:

$$\begin{aligned} EVI_t(l, X) &= \mathrm{P}(\gamma_l)\left(\sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l) - E[v|d_l, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l)\right) \\ &= \mathrm{P}(\gamma_l)\sum_{x_j \in \Omega_X}\left(E[v|d_j, x_j, \gamma_l] - E[v|d_l, x_j, \gamma_l]\right)\mathrm{P}(x_j|\gamma_l) \end{aligned}$$

Since

$$d_j = \arg\max_{d_i \in \Omega_D} E[v|d_i x_j \gamma_l]$$

$E[v|d_j, x_j, \gamma_l] \geq E[v|d_l, x_j, \gamma_l]$, with equality whenever $d_j = d_l$ for all j. Because each term in the summation is non-negative, the sum is also non-negative. $\square$

This implies that the sequence of trees created by `DT1` is such that the expected object value of the next tree is never less than that of the previous tree.

However, there is no guarantee that the object value will always increase with every extension of the tree. If EVI_l is zero, the *MEV* actions at each new leaf can be the same as the action at the leaf which the extension replaced.

Proposition 4 A complete decision tree, constructed by `DT1` maximizes the decision maker's expected object value.

Proof: The complete tree contains all complete contexts. Each leaf is labelled with an action which maximizes the expected object utility for the context. Therefore expected object value is maximized. □

If `DT1` is run until a policy is complete, the resulting decision function is optimal (in the sense of maximizing expected object value). Note that this convergence depends only on the choice of action at each leaf. Before this claim is stated formally, we will examine the possibilities for choices made in steps 2a and 2b. The formal statement of convergence is made as Corollary 7 at the end of Section 3.4.

3.3 Computing expected value

In order to compute extensions for decision trees, we need to be able to compute expected value of action $d \in \Omega_D$ in a given context, γ. The transformation of influence diagrams to Bayesian networks due to Shachter and Peot [44] can be used to compute expected value, (see 2.2).

Our transformation departs from that of Shachter and Peot in the conversion of the decision node. In their transformation, the decision node's direct predecessors are also direct predecessors of the decision/chance node in the Bayesian

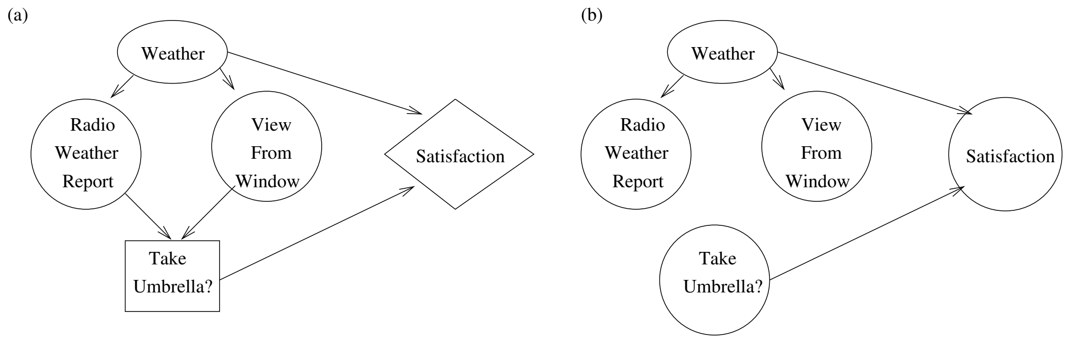


Figure 3.4: The transformation of an influence diagram.
(a) The Extended Weather Problem influence diagram, and (b) the Bayesian network used to compute expected utility. The decision node is converted to a root chance node, and the value node is converted to a chance node whose dependencies are that of the value function.

network. Our conversion of the decision node creates a chance node with no predecessors. Our transformation saves the trouble of building larger uniform CPTs, and the resulting clique tree is much simpler.

This transformation is justified, since, at the start of our refinement process, none of the information predecessors are actually used in the policy. Furthermore, a uniform conditional probability distribution renders the chance node independent of its predecessors, making all arcs into the decision/chance node trivial. For example, if A and B are binary chance nodes, and if $\mathrm{P}(B|A)$ is a uniform distribution (*i.e.*, $\mathrm{P}(B|A) = 0.5$ for all values of A and B), $\mathrm{P}(B) = \mathrm{P}(B|A)$ for any values of A and B.

Figure 3.4 shows a graphical example of the conversion. The conditional probabilities for the chance nodes *Take Umbrella* and *Satisfaction* are shown in Table 3.3 (the distributions for the remaining chance nodes are unchanged from

$\mathrm{P}(\mathrm{TakeUmbrella})$

Take It	Leave At Home
0.5	0.5

$\mathrm{P}(\mathrm{Satisfaction}|\mathrm{Weather}, \mathrm{TakeUmbrella})$

True	False	Weather	Take Umbrella
0.2	0.8	Sunshine	Take It
1.0	0.0	Sunshine	Leave At Home
0.7	0.3	Rain	Take It
0.0	1.0	Rain	Leave At Home

Table 3.1: Data for the Bayesian network in Figure 3.4b.
In the transformation of the influence diagram, the utility node Satisfaction *is converted to a binary chance node; its conditional probabilities are the normalized values. The decision node,* Take Umbrella *is converted to a root chance node, and given a uniform conditional probability table. The conditional probability distributions for the remaining chance nodes are found in Table 2.1 (page 22).*

Table 2.1). Note that the information arcs into the decision node of the influence diagram are dropped in the Bayesian network.

Shachter and Peot show that the best action $d^* \in \Omega_D$ for a given information state $w \in \Omega_{\Pi_D}$ can be found by choosing the action which maximizes $\mathrm{P}(d, w|V = \mathit{true})$; see Section 2.2. We specialize this result in terms of choosing an action which maximizes expected utility in a given leaf context γ (recall that a leaf context may not include an assignment for every information predecessor in Π_d). The normalized expected utility of an action d in context γ is given by $\mathrm{P}(V = true|d, \gamma)$:

$$\begin{aligned} E[v|d,\gamma] &= \sum_{w \in \Omega_{\Pi_V}} E[v|d,w]\mathrm{P}(w|\gamma) \\ &\propto \sum_{w \in \Omega_{\Pi_V}} \mathrm{P}(V = true|d,w)\mathrm{P}(w|\gamma) \end{aligned}$$

$$= \mathrm{P}(V = true|d, \gamma)$$

The optimal action in a given context maximizes $\mathrm{P}(d|V = true, \gamma)$:

$$\begin{aligned}
\arg\max_{d \in \Omega_D} E[v|d\gamma] &= \arg\max_{d \in \Omega_D} \mathrm{P}(V = true|d, \gamma) \\
&= \arg\max_{d \in \Omega_D} \frac{\mathrm{P}(d|V = true, \gamma)\mathrm{P}(V = true|\gamma)}{\mathrm{P}(d|\gamma)} \\
&= \arg\max_{d \in \Omega_D} \mathrm{P}(d|V = true, \gamma)\mathrm{P}(V = true|\gamma) \\
&= \arg\max_{d \in \Omega_D} \mathrm{P}(d|V = true, \gamma)
\end{aligned}$$

The equality holds since $\mathrm{P}(d|V = true, \gamma)$ is a uniform distribution, and does not affect the maximum; as well, $\mathrm{P}(d|V = true, \gamma)$ is constant for all values of $d \in \Omega_D$.

This Bayesian network is used to compute expected utility, and posterior probabilities which are needed during the process of information refinement. A single query to the Bayesian network can establish the best action in a context, and a second query can determine the expected value of the best action.

Both of these quantities can be computed in a Bayesian network transformed from an influence diagram as described above. The Bayesian network can be compiled into a join tree, which can compute posterior probabilities efficiently, using DistributeEvidence and CollectEvidence operations [23]. As described in Chapter 2, once evidence has been entered, the posterior probability of any chance node (given the evidence) in the network can be ascertained without additional cost.

Each query for posterior probability is potentially expensive, since computation in Bayesian networks is NP-hard [5, 6]. However, any method for computing expected value in an influence diagram is going to incur similar costs.

We are treating the Bayesian network as a “black box” which can return answers to our queries for the posterior probability of any single chance node given some evidence. We use a single query of posterior probability to the Bayesian network as an atomic step in our analyses of computational cost. We can compare our queries directly to the queries proposed by Shachter and Peot [44], in their query based algorithm, with one exception. In order to determine the optimal action in a context, we must first ensure that $\mathrm{P}(V = \mathit{true}|\gamma) \neq 0$; in other words, we must ensure that there is a possibility that some action will result in an outcome which is not the worst. This query could be avoided if we could query for $\mathrm{P}(D, V|\gamma)$, from which we could determine if $\mathrm{P}(V = \mathit{true}|\gamma) = 0$ (by summing over Ω_D) as well as $\mathrm{P}(D|V = \mathit{true}, \gamma)$ (by dividing by $\mathrm{P}(V = \mathit{true}|\gamma)$ if this quantity is not zero) without further queries.

3.3.1 Computing extensions

Recall from Section 3.2 that an extension subtree is rooted by an information predecessor which has an edge for each of the its values. Each child of this root is a leaf, labelled with an action which maximizes expected object utility in a given context. An algorithm for computing an extension subtree is given in Figure 3.5.

Suppose the leaf we wish to replace has context γ_l, and suppose this leaf is refined using information predecessor $X \in \xi_l$. We compute the posterior probability distribution $\mathrm{P}(X|\gamma_l)$, and then, for each value $X = x_j$, we construct a leaf labelled with the best action to be done in the new context $x_j\gamma_l$. The *MEV* action

procedure build-extension
Input:
context γ
information predecessor $X \in \xi_l$
Output:
an extension subtree for X at γ

1. Determine $\mathrm{P}(X|\gamma)$
2. Build a vertex for X
3. for each $x_j \in \Omega_X$
4. if $\mathrm{P}(V = true|x_j, \gamma) \neq 0$
5. Add a leaf to vertex X labelled with $d^* = \arg\max_{\Omega_D} \mathrm{P}(D|V = true, x_j, \gamma)$
6. Label edge to leaf vertex x_j with $\mathrm{P}(x_j|\gamma)$
7. Return the extension subtree

Figure 3.5: An algorithm to compute an extension subtree for context γ using information predecessor X.

can be computed using a single query to the Bayesian network, $\mathrm{P}(D|v, x_j\gamma_l)$, which returns a posterior probability distribution for D; the best action is the one which maximizes this distribution. Finally the expected value of the best action d_j for leaf corresponding to x_j is computed, using a single query $\mathrm{P}(v|d_j, x_j\gamma_l)$.

In addition to the labels of the vertices in the decision tree, conditional probability tables $\mathrm{P}(X|\gamma_l)$ are stored at each internal vertex X. These values can be used to determine the probability of any context, using the chain rule for probabilities:

$$\mathrm{P}(x_j, \gamma_l) = \mathrm{P}(x_j|\gamma_l)\mathrm{P}(\gamma_l).$$

and performing the multiplication by following the path from the root to the vertex X. This allows us to cache computations of posterior probability, so that we do not need to recompute them.

We note that before we can determine *MEV* action in context $x_j\gamma_l$, our implementation performs a check to ensure that $\mathrm{P}(V = true|x_j\gamma_l) > 0$. There are two explanations for this probability being zero. First, the probability $\mathrm{P}(x_j, \gamma_l) = 0$, in which case the event $x_j\gamma_l$ cannot occur, and so no action needs to be chosen for it. Second, the expected value for all events consistent with the context $x_j\gamma_l$ could be zero. In this case, all actions lead to the same value, and the choice between actions is of no consequence.

Thus, in our implementation, computing an extension subtree for a given leaf requires three queries to the Bayesian network for every value $x_j \in \Omega_X$; an additional query for the posterior probability $\mathrm{P}(X|\gamma_l)$. Thus, for each possible extension, $3b + 1$ queries are needed to compute the extension subtree, where b is the number of values in Ω_X. The factor of three in this figure is the result of our implementation's effort to guard submitting impossible evidence to the Bayesian network. If it is possible to query $\mathrm{P}(D, V|\gamma)$, the factor of three could be reduced to two.

3.4 Heuristics and strategies for information refinement

The `DT1` algorithm does not indicate how the choices in steps 2a (the choice of a leaf in the tree) and 2b (the choice of an information predecessor with which the chosen leaf's context can be refined) are made. This section presents the *heuristics* which are used to choose a leaf to refine, and the *strategies* which are used to choose an extension for a given leaf. The order in which leaf vertices are refined and the extensions made to the leaf vertices is not important if the decision maker intends to build a complete decision tree. However, if the algorithm is to be used as an anytime or flexible algorithm, the choices of leaf and extension become important. We want to build valuable policies with few splits.

In Section 3.4.2 we explore heuristics which we use to determine where to refine a decision tree. These heuristics make use of information computed during the computation of extensions for a given leaf; relative to the cost of building extensions, these heuristics are negligible. We also explain the need for heuristics. Section 3.4.1 discusses strategies for choosing an extension for a given leaf.

3.4.1 Strategies for choosing an extension for a given leaf

A given leaf with an extensible context can be extended by choosing an information predecessor which is not already in the context. In this section we discuss three *strategies*: Maximal Extension, Greedy Extension, and Random Extension. Each strategy chooses a single information predecessor with which to refine the leaf's context.

These strategies will choose an information predecessor by constructing a number of extension subtrees for the given leaf (at least one, but possibly more). The number of subtrees constructed during the refinement of a single leaf depends on the strategy used to choose an extension, as well as the EVI_t. However, the number of subtrees constructed during the refinement of a given leaf is independent of the manner in which the leaf was chosen to be extended.

Maximal extensions

The Maximal Extension strategy chooses the best extension subtree to a given leaf l on the basis of EVI_t:

$$
\begin{aligned}
X_l^* &= \arg\max_{X \in \xi_l} EVI_t(l, X) \\
&= \arg\max_{X \in \xi_l} \sum_{x_j \in \Omega_X} E[v|d_j, x_j, \gamma_l]\mathrm{P}(x_j|\gamma_l)
\end{aligned}
$$

because of Proposition 2. Recall that d_j is chosen to maximize $E[v|d_j, x_j, \gamma_l]$, as per Equation 3.1. The X^* is determined by explicitly extending l with each possible extension in ξ_l.

The Maximal Extension strategy enumerates all possible extensions for a given leaf, but uses only one of these. Therefore, for each leaf in the sequence of trees constructed by `DT1`, many extension subtrees are constructed which will never be placed in the decision tree. However, the total number of extension subtrees computed by this strategy to complete the decision tree is only a constant factor greater than the number of information states, as established by the following theorem.

Theorem 5 Let n be the number of parents for decision node D in an influence diagram, and let b be the number of values for each of D's parents, $b > 1$. The total number of extension subtrees computed by the Maximal Extension strategy in constructing a complete decision tree for D is $O(b^n)$, regardless of the manner in which the leaf vertices are chosen.

Proof: This follows from the observation that the number of extensions considered by Maximal Extension strategy at a leaf depends only on the size of its context: if decision node D has n parents, and a leaf in the decision tree has k of these in its context, then the number of extensions which are examined to choose the maximum is $n - k$. Recall that the algorithm replaces a leaf with an extension subtree; therefore, every internal vertex in the complete tree represents one out of $n-k$ possible vertices. In the complete decision tree, there are b^k internal vertices at depth k, for any $k \leq n$; therefore, the algorithm constructs $n-k$ extension subtrees for each of the b^k vertices at depth k. Extension subtrees are constructed for leaf vertices at depth $0 \leq k \leq n-1$, because the first decision tree in the sequence is a leaf with an empty context ($k = 0$); also, the leaf vertices of the complete tree have complete contexts ($k = n$), and are never extended. Summing over k, the total number of extension subtrees constructed by the Maximal Extension strategy is

$$\sum_{k=0}^{n-1}(n-k)b^k = \frac{b^{n+1} - b(n+1) + n}{(b-1)^2}$$

□

Each extension subtree requires $3b+1$ queries for a given context (see Section 3.3.1). Therefore, using the Maximal Extension strategy for choosing an extension, the complete decision tree requires $O(\frac{(3b+1)b^{n+1}}{(b-1)^2}) = O(b^n)$ queries to the Bayesian network. Finally, the initialization step (step 1) requires 3 queries. We have assumed that no extra queries are needed to choose leaf vertices to extend; Section 3.4.2 proposes a number of methods for choosing a leaf to extend which do not require any extra queries.

This implies that constructing a decision tree iteratively to completion requires only a constant factor more computational effort than building a policy by enumerating the information states. In particular, when all the nodes in an influence diagram are binary, *i.e.*, $b = 2$, there are $2^{n+1} - (n+2)$ extension subtrees constructed during the building of a complete decision tree, using $7(2^{n+1} - (n+2))$ queries. Including the step which initializes the decision tree, $7(2^{n+1} - (n+2)) + 3$ queries are made in total. Exhaustive enumeration requires 2^{n+1} queries. In Section 3.5, we will apply the information refinement approach to single stage problems in which $n = 8$. The Maximal Extension strategy will require 3517 queries to complete a decision tree for a influence diagram with 8 binary chance nodes. In contrast, the exhaustive enumeration technique will require 512 queries.

Random myopic extensions

Another strategy for choosing an extension is to pick an information predecessor from the possible extensions at random. This strategy does not make any comparison of any extension subtrees, and thus will minimize the number of extension subtrees computed by `DT1` when a complete decision tree is constructed.

Theorem 6 Let n be the number of parents for decision node D in an influence diagram. Furthermore, suppose that each of D's parents has at most b values, $b > 1$. The total number of extension subtrees computed by the Random Extension strategy in constructing a complete decision tree for d is $O(b^n)$.

Proof: This follows from the observation that only one extension is computed for every extensible leaf. Thus the number of extensions equals the number of internal nodes in the complete tree, which is:

$$\sum_{k=0}^{n-1} b^k = \frac{b^n - 1}{b - 1}$$

Therefore the number of extensions computed by the Random Extension strategy is $O(b^n)$. □

Each extension requires $3b + 1$ queries to the Bayesian network; the initialization of the tree requires 3 queries. Therefore, a complete tree constructed using the Random Extension strategy requires $O((3b+1)b^n + 3) = O(b^n)$ queries.

In particular, if all the chance nodes in the influence diagram are binary (*i.e.*, $b = 2$), the Random Extension strategy requires $7 * 2^n + 3$ queries to construct a complete tree. Exhaustive enumeration would require 2^{n+1} queries.

Greedy myopic extensions

For a given extensible leaf l in decision tree t, the Greedy Extension strategy examines the possible extensions ξ_l in a fixed order, constructing extension subtrees until one, say X, is found for which $EVI_t(l, X) > 0$.

In the case that no extension increases the expected value of the tree, one is chosen at random. An extension is made even when $EVI_t(l, X) = 0$ because of the myopic nature of the algorithm. The decision problem might be such that all single observations do not increase the expected utility, but a combination of observations might increase the expected utility. Obviously, if there is only one possible extension, and the $EVI_t(l, X) = 0$ for that $X \in \xi_l$, the leaf l is not replaced (See Section 3.2.1 for an example of this).

The performance of this strategy depends on the ordering of the possible extensions, and many orderings are possible.

The ordering used in our implementation of this strategy is consistent with the intuition that "recent" observations are likely to be more important than less recent observations. The DAG structure of an influence diagram implies a partial ordering on the nodes, in which a node occurs before any of its children. The DAG's partial ordering can be determined and expressed as a total order before refinement begins [45]. The reverse of this total order puts recent observations before observations which are earlier temporally. Note that this ordering is applicable to any influence diagram, but is of particular interest to multistage problems (in which the decision nodes imply a time sequence).

Clearly, other orderings could also be used. One of the criteria for choosing an ordering is computational cost. The ordering we use involves no additional overhead during the refinement, as it can be computed once before the information refinement begins.

Regardless of the method for ordering the extensions, the number of extension subtrees computed by the Greedy Extension strategy to find the complete tree is bounded above by the number required for the Maximal Extension strategy. At the other extreme, the Greedy Extension strategy requires only as much effort as the Random Extension strategy.

Corollary 7 The series of decision trees constructed by `DT1` converges to a policy which maximizes expected object utility after $O(b^n)$ Bayesian network computations.

Summary: Strategies

This section has outlined three strategies which can be used to refine a given leaf's context. These strategies compare extension subtrees by computing the actual object value of the subtree. The cost of using these strategies, in relation to the computation of a complete decision tree, has been shown to be only a constant factor more effort than the cost of finding a policy by enumerating the information space; this assumes that the method by which the leaf vertices are chosen has negligible cost. The empirical behaviour of these strategies in single-stage influence diagrams is explored in Section 3.5. Chapter 4 uses these strategies in multi-stage influence diagrams.

3.4.2 Heuristics for choosing a leaf to extend

In this section, four heuristics are presented which are used to choose which leaf vertex to extend. The heuristics discussed in this section are domain independent, and do not require queries to the Bayesian network over and above those needed to construct the extension subtree. In other words, the information used by the heuristics described in this section make use of expected values and probabilities computed by the strategies of the previous section. In this way, the costs of the heuristics are negligible.

The following definition will be helpful in discussing the issues involved; it presents a "gold standard" for making myopic choices.

Definition 1 Let t be a decision tree for decision node D, with extensible leaf vertex l. The *maximum expected object value of improvement*, $MEVI_t(l)$, is defined as follows:

$$MEVI_t(l) = \max_{X \in \xi_l} EVI_t(l, X)$$

□

In other words, $MEVI(l)$ is the most a decision tree could increase in value by a single refinement at leaf l. This is just the value of the maximal extension for a decision tree at a given leaf.

Step 2a of the information refinement algorithm could choose to refine the leaf whose $MEVI(l)$ was highest: $l^* = \arg\max_{l \in t} MEVI_t(l)$. In this case, $MEVI(l^*)$ is the most a decision tree could increase by a single refinement of any leaf in the tree.

The $MEVI_t(l)$ can be computed by constructing all possible extension subtrees for a given leaf l, and keeping the one whose EVI_t is maximal; Section 3.4.1 discusses this approach, and presents two alternatives. The leaf l^* can be determined by finding $MEVI_t(l)$ for all leaf vertices. In other words, to find the best leaf to extend, we have proposed to extend all the leaf vertices in the tree. This is a reasonable approach for building decision trees in a breadth first manner. However, it does not solve the problem of how to choose a single leaf to refine.

It would also be reasonable to determine $MEVI_t(l)$ for all leaf vertices in the tree, but only change the tree at l^*. A reasonable implementation of this strategy would save the results for the other leaf vertices in a cache, and only compute $MEVI_t(l)$ for the new leaf vertices. This is possible, since EVI_t values at a given leaf will not change when a different leaf is extended, as implied by Proposition 1. The best $MEVI_t(l)$ can be chosen from the cache, and the new results.

We observe that the extensions in the cache could add significant value to the tree. For each leaf in the tree, this approach has already computed an extension which maximizes EVI_t. If the nodes in the influence diagram are binary, the number of leaf vertices in the cache is double the number of leaf vertices in the tree. The expected value of the tree does not include the value of the extensions stored in the cache.

This approach is reasonable if the goal is to keep the tree small, as in Lehner and Sadigh [26]; a more detailed discussion is found in Section 3.6. However, for an anytime algorithm, it makes sense to make use of the results of computation immediately. In other words, we want to avoid using a cache of stored results, favoring instead to put the results of extending a leaf directly in the tree, rather than in a cache. In this way, the value of the tree includes the value all the extensions computed.

This choice implies that we cannot choose to extend the tree by maximizing $MEVI_t(l)$; if we compute $MEVI_t(l)$ for a given leaf l, the best myopic extension is placed in the tree immediately. This approach will result in a depth-first or breadth-first construction of the decision tree. We have not solved our problem, which is to find the best single extension to the tree.

The heuristics described in this section are used to choose a leaf vertex to extend. The general approach is as follows. When a leaf vertex is created during the refinement of a context, it is also assigned a heuristic value. A priority queue is used to contain the set of extensible leaf vertices in the tree, ordered by decreasing heuristic value. If a particular leaf's context has probability zero (*i.e.*, $\mathrm{P}(x_j|\gamma_l) =$

0), the leaf is not put into the queue.

Example Recall the extended weather decision problem from Section 3.2.1, in which a sequence of three decision trees were constructed. The second tree t_2 in this sequence is pictured in Figure 3.3b. The $MEVI_{t_2}$ values for the leaves are as follows:

$$MEVI_{t_2}(\mathrm{V} = \mathrm{sunny}, \mathrm{R}) = 0.0$$
$$MEVI_{t_2}(\mathrm{V} = \mathrm{cloudy}, \mathrm{R}) = 1.05$$
$$MEVI_{t_2}(\mathrm{V} = \mathrm{rainy}, \mathrm{R}) = 0.0$$

where *R* stands for *Radio Weather Report*, and *V* stands for the *View from Window*. The discussion of each heuristic below will include the evaluation of the heuristic on the leaf vertices of t_2 for this problem.

The Most Likely Context heuristic

This heuristic uses the probability of a leaf's context, $\mathrm{P}(\gamma_l)$; the leaf with the most likely context is always refined first. The most likely context is a good context to extend, since it invests computational effort in those contexts which, by the decision maker's model of the problem, are most likely to occur.

To compute $\mathrm{P}(\gamma_l)$, we make use of the posterior probabilities of contexts which are stored in the internal vertices of the decision tree, as outlined in Section 3.3.1. Thus this heuristic incurs a memory cost which is linear in the number of internal vertices in the tree (because the probabilities $\mathrm{P}(X|\gamma_X)$ are stored at each internal vertex labelled with information predecessor X), and a computational cost, which is linear in the depth of the given leaf. This can be much simpler than using the Bayesian network to compute $\mathrm{P}(\gamma_l)$.

Example Recall the extended weather decision problem from Section 3.2.1, in which a sequence of three decision trees were constructed. The second tree t_2 in this sequence is pictured in Figure 3.3b. Under the Most Likely Context heuristic, each leaf l of this tree is given the heuristic value $\mathrm{P}(\gamma_l)$; the three contexts have the following probabilities:

$$\begin{aligned}
\mathrm{P}(\mathrm{V} = \mathrm{sunny}) &= 0.59 \\
\mathrm{P}(\mathrm{V} = \mathrm{cloudy}) &= 0.15 \\
\mathrm{P}(\mathrm{V} = \mathrm{rainy}) &= 0.26
\end{aligned}$$

The Most Likely Context heuristic suggests that the context *V= sunny* should be refined first. This heuristic is misleading in this particular instance, as only the middle branch has positive $MEVI_{t_2}$, and this branch is ranked last according to likelihood. Empirically (as shown in Section 3.5) this heuristic is often effective.

The Post Hoc heuristic

The intuition for this heuristic is that it may be valuable to invest computational resources by refining contexts where previous refinement has provided the best previous results. This heuristic is also of interest as it is used implicitly by the algorithms described in [12, 26].

When a context of decision tree t is refined, an extension subtree, rooted at some $X \in \xi_l$. This extension subtree replaces a leaf l resulting in the new decision tree t'. The value of this subtree is $EVI_t(l, X)\mathrm{P}(\gamma_l)$. The Post Hoc heuristic uses this value as a means for estimating the value of refining each leaf vertex of the extension subtree. That is, for each leaf l_j in the subtree, the Post Hoc heuristic uses the value $EVI_t(l, X)\mathrm{P}(\gamma_l)$ to estimate $MEVI_{t'}(l_j)$.

The first factor, $EVI_t(l, X)$, is computed when the extension to γ_l is computed. The second factor, $\mathrm{P}(\gamma_l)$, can be computed using cached posterior probabilities stored at the internal vertices, as described in the Most Likely Context heuristic.

The Post Hoc heuristic is not obviously a good one, but it arises naturally from the attempt to choose the leaf which maximizes $MEVI_t$. The following four paragraphs describe the process of maximizing $MEVI_t$; we show that this is in fact, an implicit implementation of the Post Hocheuristic. Our implementation of the heuristic is explicit, and is described below.

One way to maximize $MEVI_t$ is to search through every possible refinement of every leaf in t; only the best refinement is used, and all other possible refinements are discarded. This is obviously too inefficient for practical use, since a large number of refinements are recomputed.

A more efficient method for maximizing $MEVI_t$ is to find a possible extension which maximizes $EVI_t(\gamma, X)$ for each leaf context γ. Proposition 1 implies that the best extension subtree for each leaf can be computed once (*e.g.*, using the Maximal Extension strategy), and stored in a cache.

With a cache of subtrees, the algorithm can choose the best extension subtree from the cache (on the basis of $MEVI_t$), and apply it to the tree; this is the approach of [26]. Such an algorithm would replenish the cache by extending, if necessary, each of the new leaf vertices which were just previously added to the tree.

In effect, the cache places a delay between the construction of a subtree, and its placement in the tree. This delay can be removed, since the cache contains subtrees which could be added to the tree at the time they are added to the cache. In this way, the work of constructing an extension subtree brings immediate benefit to the value of the decision tree. The cache is therefore no longer a repository for subtrees, but a collection of pointers to the most recent subtrees added to the tree. The best subtree in the cache is the best subtree of those recently added to the tree; and when it is removed from the cache, each of its leaf vertices are refined. The best subtree has value $EVI_t(l, X)\mathrm{P}(\gamma_l)$, and each of the leaf vertices is refined on account of this value. This is the Post Hoc heuristic, used implicitly.[4]

In our algorithm, the Post Hoc heuristic is used explicitly, rather than implicitly, as described above. A heuristic value is assigned to each leaf when the extension subtree is created, namely $EVI_t(l, X)\mathrm{P}(\gamma_l)$, where l is the leaf vertex which is being refined, and X is the root of the extension subtree. The Post Hoc heuristic uses the value of an extension subtree to estimate the value of refining the leaf vertices in the extension subtree.

This heuristic uses value information which is already known, as opposed to making extra effort to estimate value information. The disadvantage of this heuristic is that the value of past effort doesn't always correspond to a good context for future

[4]Both [12, 26] mention the use of a cache to make their algorithms more efficient. Thus, it seems their algorithms use the Post Hoc heuristic implicitly. However, the concerns are somewhat different; see Section 3.6 for a discussion.

effort. Another disadvantage is that all the leaves of the new extension have the same value.

Example Recall the extended weather decision problem from Section 3.2.1, in which a sequence of three decision trees were constructed. Let t_1 and t_2 be the first and second trees in this sequence, respectively; these are pictured in Figure 3.3a and 3.3b (page 52). Now $EVI_{t_1}(\gamma_0, V) = E_{t_2} - E_{t_1} = 82.95 - 70.0 = 12.95$, and $\mathrm{P}(\gamma_0) = 1.0$ since t_1 has only a single context accounting for the entire information space. Thus, under the Post Hoc heuristic, each leaf of this tree is given the heuristic value 12.95. This is a heuristic suggestion that, in refining t_2, all possible leaf vertices have equally likely $MEVI_{t_2}$ extensions. In this case, the suggestion is misleading, as only the middle branch has positive $MEVI_{t_2}$.

The second best action heuristic

Recall that a leaf is labelled with an action which maximizes the expected object value for the context of the leaf (step 2c of `DT1` and Equation 3.1). The Second Best Action heuristic uses the expected value of the second best action (the runner up to the action labelling the leaf) to order the leaf nodes.

The Second Best Action heuristic uses the probability distribution $\mathrm{P}(D|v^*, x_j\gamma_l)$, which was computed to find the best action (for labelling the leaf); recall that this distribution contains values which are proportional to $E[v|d_i, \gamma_l]$. The distribution is scanned for the second best value. For single stage decision problems, this value is used. For multi-stage decision problems, a minor modification is needed; this will be discussed in Chapter 4.

The intuition behind this heuristic is that a single action may be best on average for all information states covered by the context of a leaf. There are two reasons that an action can be best on average. First, this action might be the best action in all information states covered by the context. Second, this action might be the best compromise action for all the information states covered by the context. The value of the runner up action provides some insight to distinguishing these possibilities. If the second best action is relatively low in expected value, it seems likely that the best action is best in most of the information states covered by the context. If the second best action is relatively high, it seems likely that it might be preferable to the best action in a significant number of information states covered by the context.

The expected value of the second best action may indicate contexts whose refinement will lead to different actions at the new leaf vertices (and therefore result in positive EVI_t). It is also possible that the decision maker is indifferent to the two different actions in all information states consistent with the given context.

Because this heuristic uses the values $\mathrm{P}(D|v^*, x_j\gamma_l)$, which are computed during the construction of an extension, this heuristic has the same cost as the Post Hoc heuristic. Unlike the Post Hoc heuristic, each leaf may have a unique second heuristic value.

Example Recall the extended weather decision problem from Section 3.2.1, in which a sequence of three decision trees were constructed. The second tree t_2 in this sequence is pictured in Figure 3.3b. Under the Second Best Action heuristic, each leaf of this tree is given a heuristic value corresponding to the value of the alternate action, as follows:

$$\begin{aligned}
E[\text{S}|\text{D=take it, V=sunny}] &= 22.5424 \\
E[\text{S}|\text{D=take it, V=cloudy}] &= 35.0 \\
E[\text{S}|\text{D=leave at home, V=rainy}] &= 13.4615
\end{aligned}$$

Under this heuristic, the context *V=cloudy* is ranked the highest. This context happens to be the only context with positive $MEVI_{t_2}$.

For this example, the various heuristics make different suggestions, and only the second best heuristic guesses correctly. This behaviour should not be seen as general behaviour; an empirical demonstration of the heuristics is found in Section 3.5.

The random leaf heuristic

This heuristic chooses a leaf at random, with uniform probability. This is very cheap computationally. If any of the above heuristics are effective, they must perform better on average than random choices, considering the relative computational costs.

3.5 Information Refinement Applied

In Section 3.4.1, the information refinement algorithm was shown to converge to the optimal policy after $O(b^n)$ steps; this is the same order as traditional methods using exhaustive enumeration of the information space. Because the information refinement algorithm summarizes information states using contexts, this approach may construct policies which are valuable to the decision maker without exploring the whole information space.

The degree to which a policy constructed by information refinement approximates the optimal policy is an important way of evaluating the algorithm. The optimal policy could have expected object value as high as $\max_{w \in \Omega_{\Pi_V}} v(w)$, where v is the value function for the value node in the influence diagram. Therefore, a loose upper bound on the error in the expected value of an anytime policy is given by:

$$E_{\delta^*} - E_\delta \leq \max_{w \in \Omega_{\Pi_V}} v(w) - E_\delta$$

This estimate is exact for influence diagrams that have an optimal policy with expected object value equal to $\max_{w \in \Omega_{\Pi_V}} v(w)$. In general, the decision maker does not know E_{δ^*} without computing a complete policy.

In this section, we explore the space of single-decision influence diagrams, and show the performance of information refinement. We show that there are influence diagrams for which information refinement is able to produce good approximations for the optimal policy at lower cost than exploring the information space exhaustively. We argue that these problems have properties in common with influence diagrams which are instances of real decision problems for real decision makers. We will also see how well the various heuristics and strategies work.

3.5.1 The sample space of influence diagrams

The space of all influence diagrams with a single decision node is vast. The degrees of freedom include:

- the number of chance nodes
  - in total

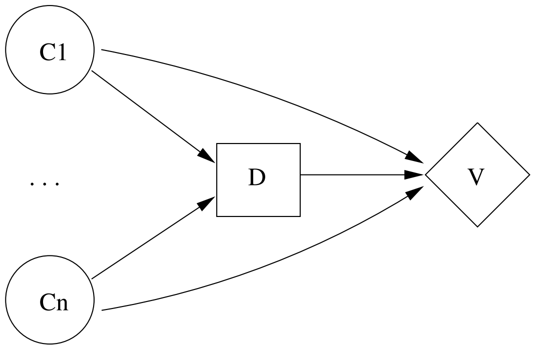


Figure 3.6: A template for a simple influence diagram.
This template influence diagram has one decision node and n informational predecessors.

  - which are parents of the decision node D: n_D
  - which are parents of the value node V: n_V

- the number of values taken by a chance node
- the interdependence of the chance nodes
- the probability distribution over the chance nodes
- the number of values taken by the decision node
- the value function

Because it is not feasible to sample this entire space, we investigate the empirical performance of information refinement on randomly generated influence diagrams, sampled from a specific class of influence diagram. This class is described below. The remainder of this subsection describes this class of problems, and shows how variations within this class result in problems with interesting properties.

This class of influence diagram is pictured in Figure 3.6. For brevity, this class is called the "1-ID(n)" class, where n is the number of chance nodes. For simplicity, we consider only binary-valued chance nodes, and a binary-valued decision node.

The 1-ID(n) class has the property that all the chance nodes are parents of the decision node and the value node. Any influence diagram with a single decision node can be reduced to an influence diagram which has this property by summing out all the chance nodes which are not information predecessors (using Shachter's algorithm [45], or variable elimination [53]). This does not change the probability distribution represented by the influence diagram, but rather, it represents the probability distribution in terms of only the directly observable events. As well, the size of the information space does not change. This reduction does not limit the probability distributions which can be represented; rather, this reduction represents the same distribution differently, perhaps less compactly.

In the 1-ID(n) class, the chance nodes are conditionally independent, *i.e.*, there is no interdependence between the chance nodes. The size of the information space depends on the number of information predecessors, and not on their interdependence. Therefore, the total number of steps required to enumerate the information space exhaustively is not affected by this constraint. The conditional independence of the chance nodes does constrain the probability distributions which can be expressed.

The above description of the class 1-ID(n) leaves n unspecified. Informal experiments suggested that using $n = 8$ leads to sample problems which are large enough to demonstrate the behaviour of information refinement, and small enough to let information refinement be applied to a large number of sample problems.

The class also allows for some variation within the class in terms of the probability distributions for the chance nodes, and the value function. These dimensions affect the performance of information refinement, as follows. If the probability distributions are such that a small number of contexts contain a majority of the probability mass in the problem, an anytime policy which summarizes the least likely outcomes may be valuable. If the value function does not depend on all of its inputs, an anytime policy may be valuable, if it distinguishes the relevant outcomes. These two factors interact, since an unlikely outcome might be extremely good (or bad) in value.

In order to demonstrate the dependence of information refinement on the probability distribution and the value function, we need to assess these aspects quantitatively. Below, we define two measures, which we call $vsize$ and $pr95$, which are prior assessments of the value function and probability distribution, respectively. For the 1-ID(8) influence diagrams, these measures can be computed fairly simply; for more general influence diagrams, it may be too expensive to compute these measures.

The dependency of a value function on the observable outcomes is assessed by determining the size of the smallest decision tree representation for the function. This measure, called $vsize$, is a property of the value function. We are interested in large problems, so we approximate the smallest decision tree by building a decision tree for the value function using standard heuristic techniques. In particular, we count the number of internal vertices in the decision tree representation computed using the *information gain* heuristic [38]. This measure provides an *a priori* assessment of one aspect of the difficulty a given decision problem with respect to information refinement. Note that for 1-ID(n), $\Pi_V = \Pi_D \cup \{D\}$.

The skewness of the probability distribution over the information states is assessed using the following measure: the smallest number of information states which are required to describe 95% of the probability mass for all the information states. We refer to this measure as $pr95$.

The $vsize$ and $pr95$ measures indicate some of the properties of a given influence diagram. For our experiments with 1-ID(n), a large number of influence diagrams are constructed randomly, so as to vary in terms of these measures. We describe the construction process below.

In order to create skewed probability distributions for the influence diagrams, the prior probability distribution for each chance node was selected at random according to the parameterized probability density function (PDF): $f_p(x) = kx^{k-1}$ for $x \in (0, 1]$, where $k = p/(1 - p)$; the expected value of this distribution is p. By varying p, random numbers can be generated with any arbitrary mean. By carefully selecting p, probability distributions can be created which are very skewed. Since the chance nodes are independent, for each chance node C_i in the influence diagram, one parameter x_i was drawn from f_p for a fixed p; the conditional probability table for the chance node given to the chance node is $(x_i, 1 - x_i)$. For 1-ID(n), there are n independent quantities which determine the probability distribution for the influence diagram. Our experiments use several values for p.

In order to construct value functions with varying dependencies on its inputs, the following procedure was used. The parents of the value node were represented in a list. With probability b, the first of these nodes would be used to split the value tree at the current position; with probability $1-b$, the first node was discarded, and the procedure repeated for the next node in the list. This procedure is applied recursively until there are no more nodes to split on or to discard. The decision node was always used as the last split (*i.e.*, with probability 1). The leaves of the value tree were selected from $[0, 1000]$ from a uniform probability distribution. By varying the parameter b, value functions with more or less structure could be constructed.

For value functions with n chance node predecessors, the expected number of splits in a value tree for a fixed b is given by the equation:[5]

$$E(b, n) = 2(b+1)^n - 1 \tag{3.2}$$

When $b = 1$, every parent node is used in every branch of the tree, for a total of $2^n - 1$ internal vertices. When $b = 0$ only the decision node is used, for a single internal vertex.

[5]With probability b, the first node in the list is used to split the tree. Both of these trees have an expected number of splits equal to $E(b, n-1)$. With probability $1-b$, the first node in the list will be discarded and a tree will be constructed using $n-1$ nodes. Thus we have the recurrence relation:

$$\begin{aligned} E(b,n) &= b(1 + 2 * E(b, n-1)) + (1-b)E(b, n-1) \\ &= E(b, n-1)(b+1) + b \end{aligned}$$

The decision node is always used in the tree as the last split; therefore $E(b, 0) = 1$. It can be verified that the solution to this recurrence relation is

$$E(b, n) = 2(b+1)^n - 1$$

Note that this procedure creates a decision tree structure with values at the leaves which are chosen from a uniform distribution. The assignment of random values at the leaf nodes will also have an effect on the over-all structure of the value function. By chance, some branches could have similar values (similar enough that the maximization of expected value does not distinguish between them). The point here is that the structure created by the above procedure does not determine the structure of the value function absolutely. The $vsize$ measure could be smaller than the number of nodes used to create the value function.

The two parameters p and b parameterize the creation of influence diagrams, but some variation is still expected. The two measures, $vsize$ and $pr95$, were presented to characterize the influence diagrams which were created. For 1-ID(8), $vsize \in \{1, \ldots, 511\}$ (since 2^8 - 1 = 511) and $pr95 \in \{1, \ldots, 486\}$ (since 95% of 2^8 is about 486). If $vsize$ is small, then the value function has a small tree representation. If $pr95$ is small, the joint probability distribution of the information states is skewed.

3.5.2 Experiment

In order to judge the effectiveness of our algorithm, and the heuristic components, the information refinement algorithm was applied to several hundred influence diagrams with the topology of 1-ID(8), described above.

The 1-ID(n) class of influence diagram allows variation in three important dimensions. The choice of $n = 8$ makes the problem non-trivial in size. The class allows variation in the probability distribution and value function, which we explore in this experiment.

The conditional independence of the chance nodes in the 1-ID(n) simplifies the sampling of probability distributions for our experiment. It also excludes more complicated probability distributions. However, our experiments are intended to explore the $pr95 - vsize$ space, because these measures reflect general properties of influence diagrams that affect our algorithms.

Four points in the $pr95 - vsize$ space were chosen so that the influence diagrams created would exhibit differences in the skewness of probability distributions, and the structure of the value function. The point $(pr95, vsize) = (10, 10)$ was chosen as an extreme in the space. The point $(pr95, vsize) = (200, 200)$ was chosen as a point roughly in the center of the space. The points $(pr95, vsize) = (10, 200)$ and $(pr95, vsize) = (200, 10)$ were chosen to illustrate how the algorithm is affected by changes in one or other dimension.

Influence diagrams were created to have these $(pr95, vsize)$ properties on average. The parameters p and b were selected to create these four points as follows: $p = 0.9757$ for $pr95 = 10$; $p = 0.55$ for $pr95 = 200$; $b = 0.2375$ for $vsize = 10$; $b = 0.7794$ for $vsize = 200$.

It happens that $vsize$ is very strongly correlated with the number of splits used to build the value function: in most cases, the $vsize$ of a value function is exactly the number of splits used to create the value function using b. Note that this is not necessarily true, since the method we used to create the value trees could result in a $vsize$ which is smaller than the number of splits used to build the value tree. Equation 3.2 was used to build value functions of clustered around a specified

	(10,200)			
	Value		Avg	STD
p	0.9757	$pr95$	8.53	2.2
b	0.7794	$vsize$	201.0	72.4

	(200,200)			
	Value		Avg	STD
p	0.55	$pr95$	192.5	61.9
b	0.7794	$vsize$	191.5	73.3

	(10,10)			
	Value		Avg	STD
p	0.9757	$pr95$	8.72	2.9
b	0.2375	$vsize$	10.4	9.3

	(200,10)			
	Value		Avg	STD
p	0.55	$pr95$	173.5	70.1
b	0.2375	$vsize$	10.4	9.0

Table 3.2: Four points in $pr95$-$vsize$ space.
One hundred influence diagrams were created at each point in this space. The given values for p *and* b *resulted in influence diagrams clustered around these points in the space. The average* $pr95$ *and* $vsize$ *of each set of 100 influence diagrams is shown, with standard deviations.*

$vsize$.

The correspondence between p and $pr95$ was estimated by creating 10 influence diagrams at 10 equally spaced values for p, and computing $pr95$ for each. With this estimate of the relationship between p and $pr95$, p values were interpolated to produce influence diagrams with the desired $pr95$ measures.

One hundred influence diagrams were created for each of the four points in the space. The average $(pr95, vsize)$ values are listed in Table 3.2. Note that the standard deviations are sometimes quite large. The influence diagrams are centered around the four points, but some of them are relatively far from the intended point.

The information refinement algorithm was applied to each problem using each of the 12 heuristic/strategy combinations. Each application counted the number of refinements made to the tree, the number of queries needed for each refinement, and the value of the tree at each refinement. The algorithms were run to completion; *i.e.*, the complete decision tree was constructed for each influence diagram.

We describe the behaviour of an implementation of the single stage algorithm, running the procedure until the complete decision tree is achieved. The data points we collect represent decision trees in terms of the tree's expected object value, the number of leaf nodes in the tree, and the number of queries made to the Bayesian network during the construction of the tree.

The point at which information refinement has constructed a policy worth 100% of the optimal policy's expected value was recorded as the "100% point." The point at which the current policy is worth 95% of optimal was recorded as the "95% point."

Averages for the number of queries used to reach the 95% and 100% points were computed for each set of influence diagrams and for each heuristic/strategy combination. As well, the average number of leaf vertices in the decision trees at the 95% and 100% points were computed.

Note that the information refinement algorithm can not tell if the policy is optimal by looking at its expected value. However, if there are no more extensible leaf vertices, the policy is guaranteed to maximize expected object utility (Corollary 7, page 73). The data points were recorded when the refinement process was complete.

The data are summarized in tables: Table 3.3 and Table 3.4 report the average number of queries used by information refinement for all the problems.[6] These

[6]A single table would have been ideal; however, such a table is too large to fit on a single page. The placement of these tables was chosen so that the variations in $pr95$ are horizontal, and variations in $vsize$ are vertical. The results from the $(10, 10)$ influence diagrams are located in the lower left

two tables have two subtables each: Table 3.3 summarizes the data collected for the points $(10, 10)$ (lower subtable (b)), and $(10, 200)$ (upper subtable (a)); Table 3.4 summarizes the data collected for the points $(200, 10)$ (lower subtable (b)) and $(200, 200)$ (upper subtable (a)). Each subtable lists means and standard deviations for a single set of influence diagrams, showing the number of queries used to construct policies reaching 95% and 100% points.

Table 3.5 and Table 3.6 report the average number of leaf vertices used to represent the anytime policies. The means and standard deviations of the size of the decision trees in terms of the number of leaf vertices are shown for the 95% and 100% points. This data indicates the size of the decision trees computed by information refinement.

Note that for all the influence diagrams summarized by these tables, exhaustive enumeration would require 512 queries to complete the optimal policy, using 256 leaf vertices.

We discuss the results of each of these points below.

The point (10,10)

Table 3.3b summarizes those influence diagrams with very skewed probability distributions, and very small value functions. Of the 100 problems in this set, 27 had optimal policies which could be expressed as a single action; *i.e.*, after initialization, no refinement was needed to improve the policy. Of the remaining 73 problems, 43 had policies for which the initial policy achieved 95% of the value of the optimal policy. The table shows that for these problems, the average number of queries

hand corner, “the origin.”

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	42.4	43.1	895.0	193.9
	Maximal	76.0	71.0	1435.0	254.2
	Random	33.2	24.9	543.1	113.3
Post Hoc	Greedy	117.7	175.0	1171.9	364.2
	Maximal	76.1	67.7	1838.8	543.2
	Random	191.8	252.7	874.5	159.7
Most Likely	Greedy	40.9	37.0	915.5	203.4
	Maximal	74.8	66.4	1434.2	258.3
	Random	29.6	20.9	543.1	117.3
Random Leaf	Greedy	332.8	465.2	1446.5	267.0
	Maximal	134.7	258.6	2110.1	427.9
	Random	263.1	304.5	935.6	156.7

(a) (10,200)

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	18.2	32.3	494.7	479.2
	Maximal	25.5	43.5	514.1	544.1
	Random	16.1	23.2	334.0	233.1
Post Hoc	Greedy	31.1	81.0	367.8	506.3
	Maximal	28.1	58.2	354.5	526.0
	Random	99.0	224.2	579.7	408.7
Most Likely	Greedy	18.4	33.3	552.8	494.9
	Maximal	25.7	44.9	550.3	565.2
	Random	16.3	23.7	371.7	253.3
Random Leaf	Greedy	48.8	175.8	670.7	711.8
	Maximal	31.4	76.7	691.5	754.7
	Random	124.9	261.6	631.8	456.0

(b) (10,10)

Table 3.3: Summary of results for 1-ID(8): queries (first half).

These two tables show the results of information refinement applied to two sets of 100 influence diagrams, in terms of the number of queries used. The columns represent the mean (and standard deviation) for the number of queries needed to find policies whose value is at least 95% or 100% of the value of the optimal policy. The summary continues in Table 3.4.

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	840.1	299.2	2404.8	250.5
	Maximal	1276.4	431.2	3335.9	290.4
	Random	585.9	194.1	1724.0	79.9
Post Hoc	Greedy	1661.4	518.8	2416.3	297.6
	Maximal	2288.8	833.5	3392.9	242.3
	Random	1314.4	325.1	1768.5	22.0
Most Likely	Greedy	907.8	312.5	2439.1	258.1
	Maximal	1382.5	456.0	3390.9	252.9
	Random	628.5	214.8	1750.0	66.8
Random Leaf	Greedy	1916.7	403.1	2502.6	135.7
	Maximal	2516.8	632.2	3423.6	160.7
	Random	1402.9	280.4	1766.6	31.4

(a) (200,200)

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	207.8	299.9	1003.2	1059.0
	Maximal	197.5	308.8	960.0	1130.3
	Random	208.9	221.4	1041.7	684.5
Post Hoc	Greedy	355.1	649.2	906.4	1116.4
	Maximal	333.2	712.0	895.7	1202.2
	Random	647.6	645.9	1207.3	766.7
Most Likely	Greedy	247.7	352.6	1100.0	1139.9
	Maximal	223.7	370.6	1044.1	1239.1
	Random	249.9	256.2	1202.1	739.9
Random Leaf	Greedy	648.0	871.4	1335.2	1233.4
	Maximal	407.8	766.2	1151.9	1348.5
	Random	690.7	659.3	1194.4	764.4

(b) (200,10)

Table 3.4: Summary of results for 1-ID(8): queries (second half).
This table summarizes the results from applying information refinement to two sets of 100 influence diagrams. This table is organized into two subtables, each corresponding to one of the sets. This table, with Table 3.3, presents a summary of the data collected for all four sets of influence diagram.

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	3.7	2.9	73.5	16.7
	Maximal	2.6	2.2	72.4	16.4
	Random	5.3	3.6	78.2	16.2
Post Hoc	Greedy	9.2	13.2	96.3	34.9
	Maximal	2.4	1.5	91.1	40.6
	Random	28.0	36.1	125.5	22.8
Most Likely	Greedy	3.6	2.5	75.5	16.8
	Maximal	2.5	1.9	72.6	16.7
	Random	4.8	3.0	78.2	16.8
Random Leaf	Greedy	28.3	40.0	127.2	26.8
	Maximal	5.5	13.2	121.8	32.3
	Random	38.2	43.5	134.2	22.4

(a) (10,200)

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	1.8	1.9	29.4	29.3
	Maximal	1.4	1.1	24.2	28.0
	Random	2.9	3.3	48.3	33.3
Post Hoc	Greedy	2.5	4.8	19.5	31.6
	Maximal	1.5	1.3	13.1	27.0
	Random	14.7	32.0	83.4	58.4
Most Likely	Greedy	1.8	2.0	33.0	30.8
	Maximal	1.5	1.2	26.3	29.5
	Random	2.9	3.4	53.7	36.2
Random Leaf	Greedy	3.8	11.5	43.3	49.2
	Maximal	1.8	3.3	36.9	44.4
	Random	18.4	37.4	90.8	65.1

(b) (10,10)

Table 3.5: Summary of results for 1-ID(8): size (first half).

These two tables show the results of information refinement applied to two sets of influence diagrams, in terms of the size of the decision trees. The data represent the mean (and standard deviation) for the number of leaf vertices used in policies whose value is at least 95% or 100% of the value of the optimal policy. The summary continues in Table 3.6).

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	68.9	27.5	236.0	28.1
	Maximal	58.0	25.3	234.8	28.9
	Random	84.3	27.7	246.9	11.4
Post Hoc	Greedy	154.4	56.2	237.5	35.7
	Maximal	135.9	68.1	240.3	27.6
	Random	188.3	46.4	253.2	3.1
Most Likely	Greedy	75.3	29.1	240.4	27.4
	Maximal	66.5	28.0	241.0	25.5
	Random	90.4	30.7	250.6	9.5
Random Leaf	Greedy	186.5	42.7	248.6	12.1
	Maximal	169.1	50.6	246.2	15.0
	Random	201.0	40.1	252.9	4.5

(a) (200,200)

Heuristic	Strategy	95%		100%	
		Mean	S.D.	Mean	S.D.
Second Best	Greedy	11.8	17.6	72.8	83.5
	Maximal	7.4	12.8	56.0	72.7
	Random	30.4	31.6	149.4	97.8
Post Hoc	Greedy	22.4	46.4	63.2	88.1
	Maximal	15.6	38.9	49.4	78.0
	Random	93.1	92.3	173.1	109.5
Most Likely	Greedy	14.3	22.3	81.2	91.3
	Maximal	9.0	17.2	63.4	82.3
	Random	36.3	36.6	172.3	105.7
Random Leaf	Greedy	49.7	70.7	105.6	101.4
	Maximal	23.8	51.2	77.3	95.0
	Random	99.2	94.2	171.2	109.2

(b) (200,10)

Table 3.6: Summary of results for 1-ID(8): size (second half).
This table, along with Table 3.5, summarizes the results of applying information refinement to the four sets of influence diagrams.

needed to reach the 95% point is very much less than 512. The data also shows that on average, to compute the optimal policy, five of the variations of the information refinement algorithm required fewer than 512 queries; seven of the variations required more than 512.

The least average number of queries to reach the 95% point on these problems was achieved by the Most Likely Context/Random Extension combination, which required only 16.1 queries on average. Recall that for 70 of these, the 95% point was reached during the initialization step of the algorithm. The remaining problems averaged 46.6 queries to reach the 95% point. The Most Likely Context/Random Extension combination had very similar results.

The Random Leaf/Random Extension Extension combination is shown to have the worst average performance for reaching the 95% point of all the combinations. The combination required 124.9 queries on average, which is still much less than would be required by exhaustive enumeration. Of the 30 problems whose initial policy has less than 95% of the optimal expected utility, the average number of queries needed was 409.5.

The averages for the Most Likely Context heuristic (all strategies) and the Second Best Action heuristic (all strategies) showed nearly identical averages for reaching the 95% point. When combined with the Greedy Extension strategy, the two heuristics differed on only one of the 100 problems. When combined with the Maximal Extension strategy, the two heuristics differed only on the very same problem. This nearly identical behaviour was not observed for these two heuristics using the Random Extension strategy on a problem by problem basis, but the average performance was quite similar.

Another similarity is apparent between the Post Hoc heuristic and the Random Leaf heuristic. For these two heuristics, the Random Extension strategy requires more queries on average to reach the 95% point. The Maximal Extension strategy required marginally fewer queries than the Greedy Extension strategy. Due to the high variance of the data, this difference is not conclusive. However, a closer look at the raw data was more conclusive. On 5 problems, the Post Hoc/Greedy Extension combination found policies for the 95% point with fewer queries than the Random Leaf/Greedy Extension combination; on 3 problems, the Post Hoc/Greedy Extension combination found policies for the 95% point with more queries than the Random Leaf/Greedy Extension combination; on 92 of the problems, the two combinations required an identical number of queries to reach the 95% point. Recall that 70 of the problems have initial policies worth at least 95% of optimal; so for 22 of the 30 problems which did not initialize to the 95% point, the Post Hoc/Greedy Extension and Random Leaf/Greedy Extension combinations reach the 95% point using identical numbers of queries. Another similarity between the Post Hoc heuristic and the Random Leaf heuristic is the performance of the Random Leaf strategy: it had the poorest performance of the three strategies. Compare this to the fact that the Random Leaf strategy had the best performance for the Most Likely Context and Second Best Action heuristics.

The standard deviations in the table reveals that there is a lot of variation in the raw data. This variation is due in part to the sampling of the influence diagrams. The method used to create "similar" influence diagrams created problems with $pr95$

between 3 and 21, and v between 1 and 63. Another factor in the large variance of the data is the degree to which the optimal policy can be approximated by a small decision tree. This factor is of interest, but it is not completely determined by the properties $pr95$ and v.

The Second Best Action/Random Extension combination used the fewest queries on average to reach the 100% point, needing 334.0. The Most Likely Context/Random Extension combination used 371.7 queries on average, which, given the large variation in the data, is comparable. Similarly, the Post Hoc/Maximal Extension combination required 354.5 queries on average, and the Post Hoc/Greedy Extension required 367.8 queries. These combinations used fewer queries on average than would be required by exhaustive enumeration.

The data in Table 3.5b show the average size of the decision trees computed for these problems, in terms of the number of leaf vertices in the tree at the 95% and 100% points. The Maximal Extension strategy computed the smallest decision trees (*e.g.*, 1.4 leaf vertices on average when combined with the Second Best Action heuristic), while the Random Extension strategy built trees which were on average, much larger. Complete trees have a maximum of 256 leaf vertices, but may be smaller if some information states have probability zero. The trees which achieved the 95% point were much smaller than the complete trees for these problems.

The point (200,200)

Table 3.4a summarizes the data collected from the influence diagrams whose probability distributions are much less skewed, and the value functions are much larger.

The table shows that information refinement requires more than 512 queries to reach the 95% point, and much more than 512 queries to find an optimal policy.

The data in Table 3.4a has several qualitative similarities to Table 3.3b. The Second Best Action/Random Extension combination reached the 95% point with the smallest average number of queries, 585.9. This was closely followed by Most Likely Context/Random Extension which requires 628.5 queries on average. The difference, again, is too small to be conclusive. However, the raw data shows that for 71 of the 100 problems, Second Best Action/Random Extension required fewer queries to reach the 95% point than does Most Likely Context/Random Extension; for three of the problems, the number of queries needed was the same; for the remaining 26 problems, Most Likely Context/Random Extension required fewer queries than Second Best Action/Random Extension. The two heuristics, Second Best Action and Most Likely Context had comparable results for each strategy. This observation was also made for Table 3.3b.

Table 3.4a also shows that the Post Hoc and Random Leaf heuristics required about double the number of queries to reach the 95% point, compared to the data for Second Best Action and Most Likely Context heuristics.

On average, an optimal policy (*i.e.*, the trees at the 100% point) was computed with least average cost by the Second Best Action/Random Extension combination. However, the table shows that, on average, the choice of heuristic was almost irrelevant when computing the optimal policy. The Random Extension strategy used the fewest queries, and the Maximal Extension strategy used the most. The analysis in Section 3.4.1 demonstrated that the strategies would show this behaviour when constructing complete trees. The complete trees for the influence diagrams of this set (200,200) have 255 non-leaf vertices; the trees at the 100% point averaged 234.8 internal vertices; very close to complete trees.

When a complete tree is constructed, the difference between the heuristics is mainly the order in which the information predecessors are added to the decision tree, although there will be some variation as to whether an information predecessor will be added to the tree as the last possible extension. Much more important, for the complete tree, is the strategy used to extend the context at a given leaf, as this affects the number of queries used to compute each extension.

Table 3.6a shows the average size of the policies for these problems. For most of the heuristics, the Maximal Extension strategy constructed smaller trees. For the 95% point, the Random Leaf heuristic builds much larger trees on average than the other heuristics. However, the average size of the 100% point are roughly equal, and are only slightly smaller than the largest decision tree for 1-ID(8) problems. This means that for these problems, an optimal policy needs to consider most of the information predecessors.

The points (10,200) and (200,10)

The information refinement algorithm was applied to two other sets of influence diagrams. The data in Table 3.3a summarize the behaviour of the various heuristic/ strategy combinations on 100 problems with highly skewed probability distributions, and large value functions, *i.e.*, $pr95 = 10$ and $vsize = 200$. The data in Table 3.4b summarize the behaviour of the various combinations on 100 problems with less skewed probability distributions, and small value functions, *i.e.*, $pr95 = 200$ and $vsize = 10$.

Table 3.3a shows the results for the (10,200) problems. It shows that on average, all the combinations were able to reach the 95% point with fewer than 512 queries. The Most Likely Context/Random Extension combination showed the best average performance with 29.6 queries, and the worst was Random Leaf/Greedy Extension, with 332.8 queries.

Comparing the parts (a) and (b) of Table 3.3, some notable similarities can be observed under the 95% column. The Second Best Action heuristic and the Most Likely Context heuristic are very closely matched in both tables. The Random Extension heuristic obtained the highest average number of queries for each strategy in both tables. The Maximal Extension strategy obtained very similar results for all heuristics except the Random Extension heuristic.

Many of these similarities are also true for the 100% column in parts (a) and (b) of Table 3.3. The Second Best Action and Most Likely Context heuristics had very similar average results, and the Random Extension heuristic obtained the highest average number of queries for each strategy in both tables.

The results in Table 3.4b for the (200,10) problems share many of these similarities. In general, the Second Best Action and Most Likely Context heuristics had very similar results, and were superior to the other two heuristics. However, for these problems, the Maximal Extension strategy obtained the best performance in reaching the 95% and the 100% points. This is different from all the other tables, in which the Random Extension strategy obtained the fewest number of queries.

Another interesting observation about Table 3.4b is that the Post Hoc/Maximal Extension combination reached the 100% point with the fewest average number of queries.

Convergence behaviour

The data in Tables 3.3 and 3.4 represent one view of the data for the experiment. The tables show the average number of queries required to construct a "valuable" policy. The tables hide such details as the average value of a policy after a number of queries, and the actual convergence pattern. The choice of the 95% point as a valuable policy is somewhat arbitrary, made for comparison between problems of varying properties.

In this section, we take another view of the behaviour of all the heuristic/ strategy combinations of information refinement on the influence diagrams of the (200,200) set. We are interested in these in particular, since the data in Table 3.4a show that information refinement reaches the 95% point with more than 512 queries. The data we present show the value of the policies constructed by information refinement as a function of the number of queries.

Table 3.4a is shown in Figure 3.7 plots the performance profile of the information refinement algorithm using the Random Extension strategy for each of the heuristics. The average expected value of the decision trees for the problems is plotted against the number of queries made to the Bayesian network. In this graph, the convergence of the various combinations can be compared. The vertical line in graph indicates the 512 query point, *i.e.*, the point at which traditional methods

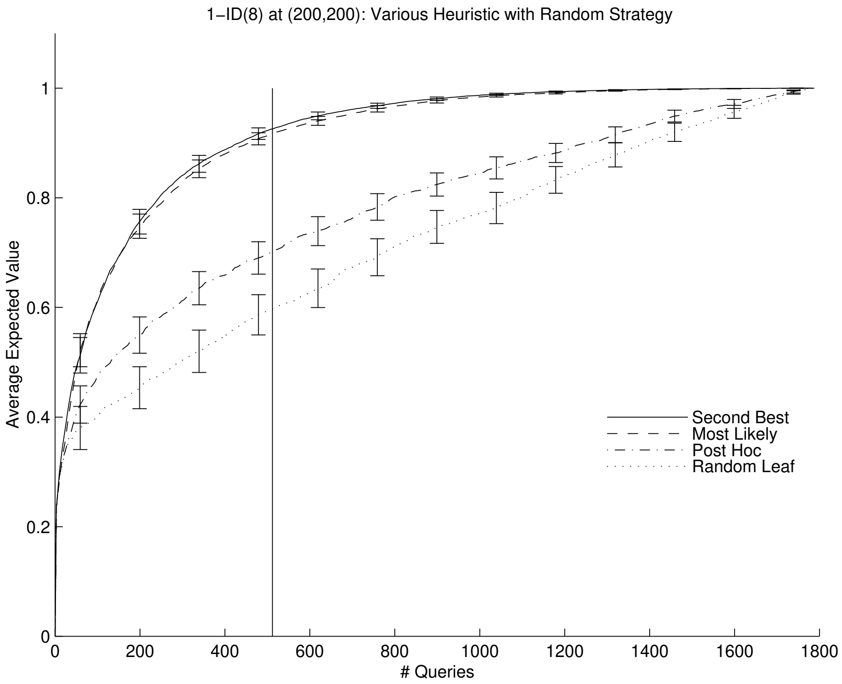


Figure 3.7: Performance profile for the Random Extension strategy.
A graph of the average expected value of policies constructed by information refinement as a function of the number of queries. This graph shows the results of all four heuristics paired with the Random Extension strategy, when applied to the problems in the (200,200) set. The error bars represent two standard deviations from the mean. Note that exhaustive enumeration would require 512 queries, marked in the graph by the vertical line to the right of the origin. This data is related to the data in Table 3.4a.

using exhaustive enumeration would complete an optimal policy. The error bars in the figure represent two standard deviations from the mean value.

Note that the underlying data for each problem is discrete; the value of a decision tree changes in discrete steps. For clarity, however, the average data is drawn using lines. As well, the expected value of the optimal policy is likely to be different for each problem. For this reason, the expected value of each policy was normalized in the following way: the value of acting randomly with uniform distribution over the possible actions (the first point recorded for each problem) was translated to 0.0, and the value of the optimal policy was translated to 1.0. The average performance was computed by averaging the scaled data points for each problem.

The graph demonstrates the similarities evident in the tables. The graph also shows that average expected value of the first few policies are the roughly equal for all the heuristics. After about 100 queries, distinct convergence patterns begin to become noticeable.

As noted above, none of the combinations reach the 95% point before 512 queries. However, the Second Best Action/Random Extension and Most Likely Context/Random Extension combinations produce trees which, on average, are worth 75.3% of the optimal policy after 206 queries (the average scaled value is 0.753, with standard deviation 0.11). If a decision maker does not have the resources to perform exhaustive enumeration, a policy worth 75% of the optimal policy may be sufficiently valuable.

Similar graphs for the Greedy Extension and Maximal Extension strategies

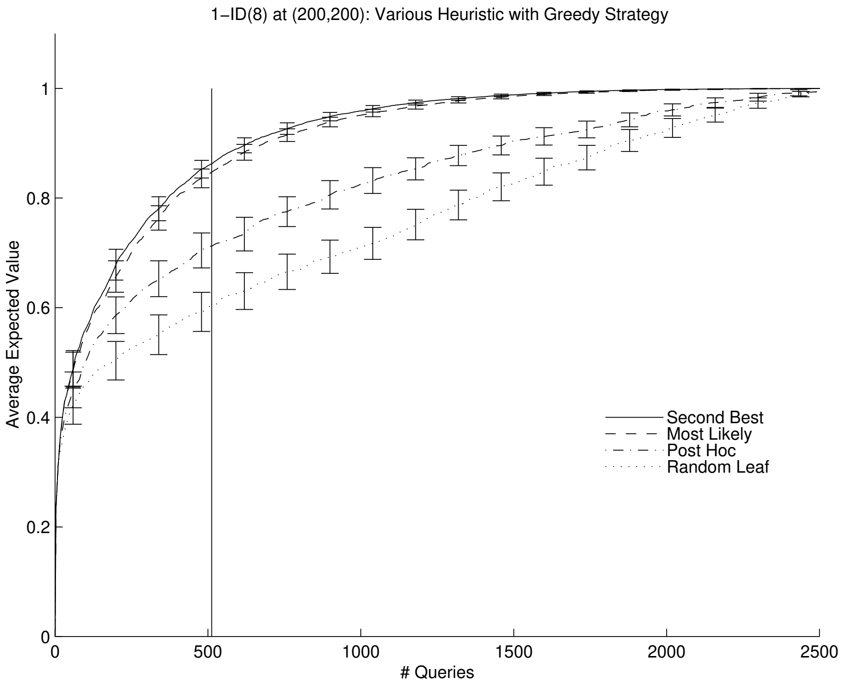


Figure 3.8: Performance profile for the Greedy Extension strategy.

A graph of the average expected value of policies constructed by information refinement as a function of the number of queries. This graph shows the results of all four heuristics paired with the Greedy Extension strategy, when applied to the problems in the (200,200) set. This data is related to the data in Table 3.4a.

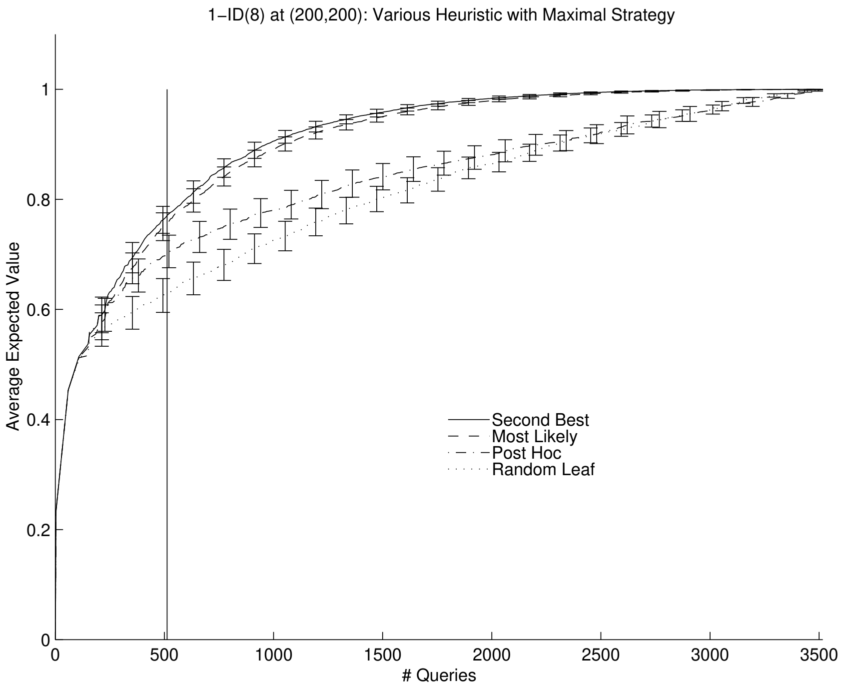


Figure 3.9: Performance profile for the Maximal Extension strategy.
A graph of the average expected value of policies constructed by information refinement as a function of the number of queries. This graph shows the results of all four heuristics paired with the Maximal Extension strategy, when applied to the problems in the (200,200) set. This data is related to the data in Table 3.4a.

are found in Figures 3.8 and 3.9, respectively. These graphs show that the Second Best Action and the Most Likely Context heuristics are very similar on average at all points. Note also that the Random Leaf heuristic shows the slowest average convergence in all cases.

3.5.3 Conclusions

The experiment was designed to compare the effects of the various heuristics and strategies on a restricted class of influence diagram, 1-ID(8). The expected object value of the policies was shown to increase on average with the computational resources queries used. The data show that the Second Best Action and Most Likely Context heuristics are superior to the Post Hoc and Random Leaf heuristics terms of the number of queries needed to construct policies which attained 95% of the optimal expected value.

The data also showed that the effectiveness of the combination of heuristics and strategies depends on the qualities of the decision problem. The Random Extension strategy combined with the Second Best Action heuristic was often able to build valuable policies with fewer queries than any other combination. However, the Random Extension strategy consistently built larger trees than the other strategies, implying that more of the information predecessors may be used in the decision tree.

The data shows that the $pr95$ and $vsize$ properties are correlated to the behaviour of the heuristics and strategies used by information refinement. For highly skewed probability distributions, or very small value functions, information refinement was able to find polices with 95% of the value of the optimal policy long before traditional methods could complete a policy. For problems with fairly flat probability distributions, and fairly large value functions, information refinement was unable, on average, to reach the 95% point using fewer than 512 queries. However, the Second Best Action and Most Likely Context heuristics were able to construct policies which have fairly high expected value even for these problems.

The next chapter applies the information refinement approach to multi-stage decision problems. Empirical data in that chapter demonstrate the value of the approach in finding policies for which exhaustive enumeration of the information space would be infeasible. For those experiments, we apply the Second Best Action heuristic and the Greedy Extension strategy, on the basis of the experiments presented here. The Random Extension strategy is not used in our multi-stage experiments; this strategy constructs large decision trees at fairly low cost, in terms of the number of queries used. For influence diagrams with a single decision node, this may be a reasonable trade-off. However, as we show in the next chapter, large decision trees increase the connectivity of the Bayesian network which we use to compute expected value and posterior probability. For influence diagrams with several decision nodes, the connectivity of the Bayesian network increases whenever a new information predecessor is included in a decision tree. This affects the efficiency of the computation within the Bayesian network, as the efficiency of exact computation depends on the connectivity of the network.

3.6 Related work

Lehner and Sadigh [26] discuss the issue of compiling a decision problem into a *situation-action* tree. Their concerns do not emphasize computational cost; their goal is to take a complex problem and create rules for use by human decision makers. The goal is to find the best decision tree of a certain size, regardless of the cost of computing them.

Their basic algorithm is equivalent to the Post Hoc/Maximal Extension variation of the information refinement algorithm presented in Section 3.2. It is interesting to note that the algorithm is presented as *greedy* but without a heuristic component. Their algorithm *precomputes* the best extension subtree for each leaf in the decision tree, and keeps the extension subtrees in reserve. The best extension in reserve is attached to the decision tree, and the reserve is augmented by refining the new contexts just added to the tree. In effect, the best reserved extension is used to invoke further extensions. This is equivalent to the Post Hoc heuristic, which orders refinements based on the value of previous refinement.

Lehner and Sadigh extend their basic algorithm, proposing one which searches for combinations of n observations to refine the tree on each refinement step (when $n = 1$, the algorithm is roughly equivalent to `DT1`). The argument for looking for multiple observations is that the probability of an extension with zero EVI_t is small when $n = 1$, but smaller when $n > 1$. The argument against this approach is based on the cost of examining all n step observation sequences.

Heckerman *et al.*[12] discuss an algorithm which is the equivalent of Post Hoc/Maximal Extension variation discussed here. Their interest is in representing a policy which can be used effectively by the decision maker on-line, in the following sense. The approach is to construct a decision tree which most effectively allows the decision maker to choose an action on-line. The costs of building the decision tree are not taken into account; the costs of using the decision tree is compared to the cost of other on–line approaches. In contrast, the information refinement approach is primarily concerned with the cost of constructing a decision tree.

Horvitz and Klein [19] describe a decision theoretic approach to categorization based on the of utility of states or actions comprising the category. By aggregating states with similar utility values, and actions with similar values, decision models can be simplified for increased efficiency. This approach is the opposite of the refinement approach, which splits aggregate states by determining differences in expected value.

Poh and Horvitz [33] present a greedy approach to exploring how random variables in a decision model might be refined, *i.e.*, be given a more fine grained set of values, to increase the utility of a decision. This work is intended to automate some of the effort that a decision analyst would put into reframing a decision problem, that is, extending the possible actions or outcomes or utilities for a decision model that has been already been constructed. This work deals with the refinement problem on a lower level than information refinement, which uses the random variables to split contexts.

Zhang & Boerlage [51] simplify decision problems by removing inconsistent information states and "insignificant details" before constructing a policy for the problem. The significance of the details in the information state is measured in terms of the effects of the information state on the posterior probabilities of (unobservable) state variables. This is similar to the approach taken by Horvitz and Klein [19], who measure significance by the effects of information state on the decision maker's expected utility. Both of these approaches construct their abstractions during a preprocessing stage to decision making. This (and similar) preprocessing is still available for use before policies are constructed using information refinement. However, the preprocessing overhead can be significant (as there are many information states to abstract). The information refinement approach requires very little preprocessing work.

For decision problems represented by Markov decision processes (MDPs) [1, 35], optimal policies have been approximated by aggregating states, *e.g.*[9, 8, 47]. These approaches exploit modifed representations of the the decision problem. White and Scherer [50] present an approximation algorithm for solving partially observable MDPs by using only a subset of the observations which are available over time. White and Scherer approximate the entire history using m most recent observations, whereas information refinement chooses which observations to include based on heuristic value.

Information refinement is closely related to "input generalization" which is used frequently to help deal with large state spaces. In reinforcement learning, the Q-learning algorithm [49] learns the expected value of acting in a given domain by compiling input, action and reward tuples. The so-called Q function which eventually determines the expected value function is simply implemented as a table. Chapman and Kaelbling [3] adapt the Q-learning algorithm for large input spaces by using a decision tree in place of the table to represent the Q-function. The decision tree is extended by "splitting" the function on significant input bits, as determined by a pair of significance tests (one for value significance, the other for perceptual significance). McCallum [30] describes a similar approach which makes distinctions in the Q-function based on selected past observations.

3.7 Summary

We have shown how a decision function for a decision node can be built incrementally. Asymptotically, the algorithm is only a constant factor worse than any algorithm which exhaustively enumerates the information set. The incremental algorithm can be used to construct a sequence of decision trees, each of which can be used by the decision maker as an anytime policy. A number of heuristics and strategies were presented which can be used to guide the refinement process.

The heuristics discussed in this chapter have minimal computational requirements. The strategies for choosing an extension dictate how much work goes into a single step of information refinement.

The combination of heuristic and strategy were demonstrated on a large number of decision problems. These trials demonstrate that information refinement creates decision functions with high value using only a small subset of the information available to the decision maker. We have shown that, for the problems we have created, the Most Likely Context and Second Best Action heuristics generate better anytime policies than the heuristic which makes refinements at random. For this reason, the Random Leaf and Post Hoc heuristics have been set aside from the investigation of multi-stage influence diagrams.

Chapter 4

Multi–stage decision problems

In this chapter, we apply information refinement to multi-stage decision problems. An algorithm is presented which applies information refinement sequentially to each decision node in an influence diagram. This algorithm constructs small decision trees for each decision node, approximating the optimal expected object utility at each stage.

A second algorithm is presented in which the refinements do not follow an established ordering. Domain independent heuristics guide the algorithm applying refinements to decision trees in the problem. In both cases we show the effect of the algorithm on several multi-stage decision problems.

4.1 Sequential refinement with resource allocation

The sequential refinement approach applies the single stage algorithm `DT1` to each decision node in the influence diagram. Refinement starts at stage n, building a

```
procedure SDT
  Input:
    Multi-stage influence diagram with decision nodes D_1, ..., D_n
    Resource allocation r_1, ..., r_n
  Output:
    Policy Δ = {δ_1, ..., δ_n}, a set of decision trees

  Δ ← empty set of decision trees
  For each decision node D_i, (i = 1, ..., n)
    δ_i ←the result of applying DT1 to D_i
        until allocation r_i is consumed
    Add δ_i to Δ
  Return the policy Δ
```

Figure 4.1: The sequential refinement algorithm.

decision tree for D_n, working backwards through the stages, until all the stages have decision trees. The size of the decision tree at a particular stage is determined by an *a priori* allocation of resources; refinement continues until the allocation for a particular stage is consumed.

We begin with a high level view of the algorithm (Figure 4.1), and then approach the details. The algorithm assumes that the decision maker has determined how much computation can be afforded in the construction of the policy, and has provided an allocation of computational resources. The algorithm applies the single stage algorithm `DT1` to each decision node in the problem, starting from the last and working backwards. The decision trees are not completed; rather, each tree is refined until its portion of the allocation has been consumed.

In the following sections, the details of the algorithm are presented. Because information refinement is intended to avoid exhaustive enumeration, there will be situations in which we require the expected value of an action in a particular context, without explicit resolution of all the decisions. In Section 4.1.1, we define expected value, accounting for these situations. Section 4.1.2 discusses the inductive nature of the refinement process. Section 4.1.3 discusses the issue of allocation of resources to each stage.

4.1.1 Expected value of a policy

Let us assume that that a policy Δ_{k+1}^n has been constructed for decision nodes $D_{k+1}, \ldots, D_n$. To construct a decision tree for decision node D_k, we need to be able to compute the expected value of an action in a context.

The expected value of an action is well defined when all choices represented by decision nodes have been resolved. The dynamic programming approach resolves these choices by exhaustive enumeration, and induction, as described in the previous section.

The information refinement approach is intended to avoid exhaustive enumeration, by constructing a decision tree which uses a few, relatively small contexts. In particular, a decision node D_i (where $i < k$) may or may not appear in a given context, indicating that the action at the leaf does not depend on the action at D_i. However, when computing expected value, it is not obvious what should be assumed about these actions.

In order to resolve the choices in $D_1, \ldots, D_{k-1}$, we provide a probability distribution over the actions of each these decisions. This allows us to define the expected value of an action in a context assuming that actions are chosen randomly through stages $D_1, \ldots, D_{k-1}$; thereafter, actions are chosen according to the policy already established for $D_{k+1}, \ldots, D_n$. In particular, we use uniform distributions for $D_1, \ldots, D_{k-1}$, giving no preference for one action over another in those stages in which deliberation has not yet occurred.

For the kth stage of the problem, sequential refinement has constructed a policy, denoted Δ^n_{k+1}, for decision nodes $D_{k+1}, \ldots, D_n$, as well as a corresponding value function $v_{k+1} : \Omega_{\Pi_{D_{k+1}}} \rightarrow \Re$, which is the value of following the policy Δ^n_{k+1} starting in information state w at stage $k+1$; the function v_{n+1} is the value function at the value node in the influence diagram.

The inductive step at this stage is to use this policy to determine a decision function δ_k for decision node D_k. Let l be a leaf in the decision tree for D_k, with context γ_l. The expected value of an action $d \in \Omega_{D_k}$ in the context γ_l is defined as:

$$E[v|d, \gamma_l] = \sum_{w \in \Omega_{\Pi_{D_k}}} \sum_{x \in \Omega_{\Pi_{D_{k+1}}}} v_{k+1}(x|d, w)\mathrm{P}(x|d, w)\mathrm{P}(w|\gamma_l)$$

The above discussion has focussed on the the definition of expected value in an influence diagram at an intermediate step in the sequential refinement algorithm. In the remainder of this section, the computational aspects of these definitions are discussed.

As in the single-stage algorithm, expected object value is computed by posing queries to a Bayesian network derived from an influence diagram. As before, the value node is converted to a chance node, whose probability distribution is the normalized value function.

Following [44], the decision functions $\delta_{k+1}, \ldots, \delta_n$ are installed by setting conditional probabilities in the Bayesian network derived from the multi-stage influence diagram. First, each information predecessor used in the decision tree δ_k is connected by a directed arc to the chance node D_k. Second, a conditional probability table is provided for the chance node representing D_k, such that for each leaf l in the decision tree and each $d \in \Omega_{D_k}$, we have

$$\mathrm{P}(d|\gamma_l) = \begin{cases} 1 & \text{if } \delta_k(\gamma_l) = d \\ 0 & \text{otherwise} \end{cases}$$

For the decision nodes $D_1, \ldots, D_{k-1}$, a uniform probability distribution is installed; these nodes are treated as chance nodes without predecessors. These can be constructed once before any refinement occurs. After a new decision tree δ_i has been computed, the new probability distribution for D_i can be installed, as described above.

4.1.2 Information refinement at stage k

With the policy Δ^n_{k+1} installed in the Bayesian network, the single stage algorithm `DT1` can be applied to D_k as if it were the only decision node in the influence diagram. The parents of D_k in the influence diagram include $D_1, \ldots, D_{k-1}$, since the influence diagram has no–forgetting arcs. These decision nodes have been converted to chance nodes with uniform probability distributions, as described in Section 3.3.

In Chapter 3, all refinements were made using chance nodes from the influence diagram. For multi-stage problems, some of the refinements will be made using decision nodes as internal vertices. The discussion in Chapter 3, regarding extensions constructed using chance nodes, applies in multi-stage problems as well.

The single stage algorithm `DT1` can choose which leaf to extend, and which information predecessor to split on according to the heuristics and strategies discussed in Chapter 3. In the remainder of this section, we discuss the issues which arise when decision nodes are used to construct extensions. In particular, we discuss how the extension is constructed, and the implications of the method.

Suppose that the decision tree for stage k is being constructed, and that the refinement algorithm has chosen to refine a context γ using one of the decision nodes, say $D_i \in \Pi_{D_k}$. During deliberation at stage k, the decision maker does not know which of the actions $d \in \Omega_{D_i}$ will be chosen during future deliberation at stage i. This uncertainty is modelled by assuming the decision maker could choose any of these actions, using a uniform probability distribution as described in the previous section. Since it is uncertain which value of D_i will have been taken, the internal vertex is labelled with D_i, and every edge (corresponding to each $d \in \Omega_{D_i}$) is given a uniform probability of being taken. Thus the decision tree for stage k has the uncertainty over previous actions built into its structure. Later, when the refinement process builds a decision tree for D_i, the uncertainty over action represented in the decision tree for D_k might be resolved: it is possible (but not necessary) that the deliberation at stage i will determine a decision function δ_i which will make some of the structure built into δ_k superfluous. In some cases, the uncertainty would be resolved entirely, such as when later deliberation chooses an action for D_i in context γ. In other cases, such as when later deliberation chooses an action for D_i in a context which is consistent with γ, the uncertainty may only be reduced.

4.1.3 Resource allocation

The remaining issue for this approach is to determine how much computation to invest in each decision tree. A simple technique would threshold the expected object value of the tree; however, we cannot predict the expected value of the complete tree, so it is impossible to tell if a smaller tree is sufficiently close to the maximal expected object value.

Instead, a fixed investment of computation for each stage is assigned *a priori*. The single stage algorithm is used as a procedure which stops when this allocation is consumed. The global resource allocation problem, that is, determining how to assign allocations to each stage is a meta-level problem. Our experiments, which are presented in Section 4.3.2 were performed to investigate this issue, suggest that considerable insight into the specific decision problem is necessary before an effective allocation can be made.

In our experiments, the allocation is given in terms of the number of refinements to be made in the tree. Alternatively, a more fine grained allocation could have been used: the number of queries to the Bayesian network.

4.1.4 Example: The Car Buyer problem

This particular decision problem is small enough that an exhaustive search through the space of possible resource allocations is feasible. Because of the asymmetries in the problem itself, very few of the observations are necessary to construct a policy which maximizes expected object value. As well, there are no information predecessors for the first decision node *Test 1*, so a single refinement (*i.e.*, the initialization of the tree) is all that is required for the last step.

The sequential refinement algorithm was applied to this problem with four resource allocations, which are discussed below. The expected value of the final policy were recorded, as well as the expected value of the policies after the algorithm stopped refining a stage. The number of queries required to construct the policy were also recorded. The initial policy, in which decision maker acts at random with uniform probability, is included in the data as well, and forms a common starting point for the allocations.

Figure 4.2 shows the smallest of the possible resource allocations. Each data set represents an allocation of resources, which was fixed *a priori*. Each data set is labelled using a triple, which indicates the resource allocation for the data set; the triple $< a, b, c >$ indicates that the first decision node has been allocated a refinement steps, the second decision has be allocated b refinement steps, and the third has been allocated c refinement steps. An allocation of zero for any decision node means that the decision tree is initialized, but not refined.

In the figure, the data for the four resource allocations are shown. Each allocation is represented by a line with four points. The first point for all four allocations is common, and represents the policy in which the decision maker chooses actions at random with uniform probability. The four allocations also have the second point in common; this point represents the initialization of the decision tree for *Buy Car?*, since this step is performed first by all allocations.

The smallest allocation assigned only initialization to each decision node; this data set is labelled $< 0, 0, 0 >$, indicating that each decision tree is initialized, but uses none of the available information. Its expected value is 0.816092. This policy is worth approximately 96% of the policy which maximizes expected object value, and requires only 9 queries to construct.

The allocation labelled $< 0, 1, 0 >$ requires more effort, but result in a policy with the same expected object value as the policy constructed using the $< 0, 0, 0 >$ allocation. The data set labelled $< 0, 0, 1 >$ shows that a refinement is made in the decision tree for the third decision node *Buy Car?* which allows the process to find the optimal policy. The data set labelled $< 0, 1, 1 >$ also results in a policy which maximizes expected object value; the extra refinement in the second stage does not result in higher value (in the graph, this refinement is seen as the horizontal line between 48 queries and 75 queries). Larger allocations are not shown, as they require more computational effort and do not construct superior policies.

4.1.5 Summary

A simple algorithm was presented which applies the single-stage information refinement procedure to each decision node, using a fixed sequence (*i.e.*, the reverse temporal order of the decision nodes) and *a priori* resource allocation.

A simple example was given in which it was observed that a fixed sequence of refinement steps could cause the decision maker to fail to find crucial observations. The refinement process is guided by heuristics (*i.e.*, the heuristics and strategies discussed in Chapter 3), and not all refinements result in increases to expected value. It may be better to spend more computation at some decision functions than others.

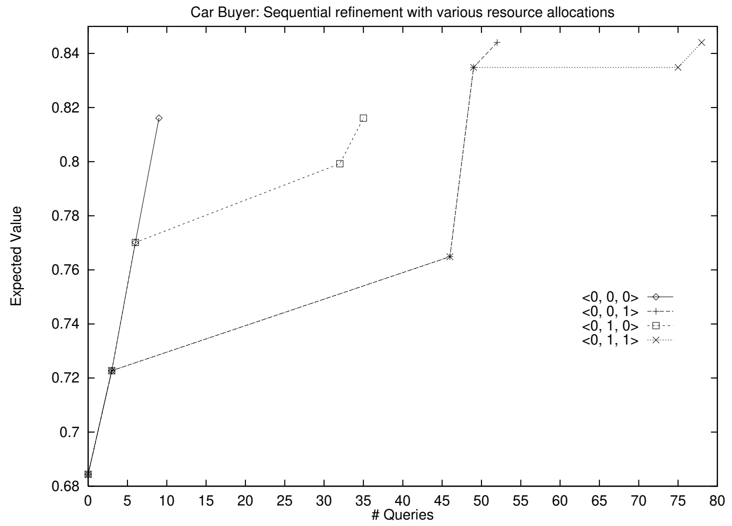


Figure 4.2: A performance profile for sequential refinement.
The performance of the sequential refinement algorithm using various resource allocations applied to the Car Buyer problem. Each dataset is labelled with three integers representing the number of refinements made to a decision node in the problem; $< a, b, c >$ *means that* a *refinements were made to* Test 1*,* b *refinements to* Test 2*, and* c *refinements to* Buy Car? *The policy which maximizes* EV_I *has an expected value of* 0.844061*, and this policy is constructed by two of these trials. The exhaustive enumeration of the information space would require more than 200 queries.*

Notably, the last decision tree to be completed in the sequential refinement approach is the first action the decision maker will have to take. A decision maker should avoid having to make random choices, especially as a first action in a sequence of action. As the results from single stage computations showed, even a simple decision function which uses no observations can make a significant difference to the object value of the decision tree. Therefore, the sequential refinement algorithm must be allowed to run to the limits of its resource allocation. Likewise, the resource allocation must be precise enough to allow the sequential refinement algorithm to complete its computation before the decision maker has to take action. In other words, it is not suitable as an anytime algorithm.

Section 4.3 shows the behaviour of the sequential refinement algorithm on larger problems. The next section presents an algorithm which has the anytime property, and does not require an *a priori* allocation of computational resources.

4.2 Random access refinement

In this section, we present an anytime algorithm for computing policies for multi-stage influence diagrams. A policy is represented by a collection of decision trees, one for each decision node in the influence diagram. As in Chapter 3, these decision trees prescribe actions for contexts which may not make use of all the information available to the decision maker. The policy is refined by choosing a leaf from one of these trees and applying a single refinement to the leaf, keeping the rest of the policy fixed.

There is no *a priori* order in which the trees are refined, which is a departure from standard dynamic programming techniques for building an optimal policy. Furthermore, the algorithm always has a current best–known policy available, refining it until the decision maker interrupts the process to act.

The refinement of a decision tree in the multi–stage problem is similar to the refinement of single stage decision functions. However, two additional elements are involved. First, the random access refinement algorithm treats the current policy as a stochastic policy, though the decision maker will not have to act stochastically; the reason and method for this is made clear in Section 4.2.1. Second, a refinement of the decision tree for D_k will have global effects on the current best policy. Section 4.2.2 explains this effect and gives a simple algorithm for updating the effects of refinement throughout the current policy.

4.2.1 Stochastic decision functions

In the random access refinement approach, we maintain a current best policy, which is to be improved by refining of one of the decision trees. If refinement is to improve the current policy, the refinement process must take into account the existing choices of the policy. In this section, we show how to represent the current best policy as a stochastic policy, so that a refinement at stage k improves the policy, without being overly constrained by it.

In single stage problems, a policy was represented by a deterministic decision tree: one action is prescribed at every leaf of the tree. This is a sufficient representation for decision functions in which a single action maximizes the decision maker's expected object utility.

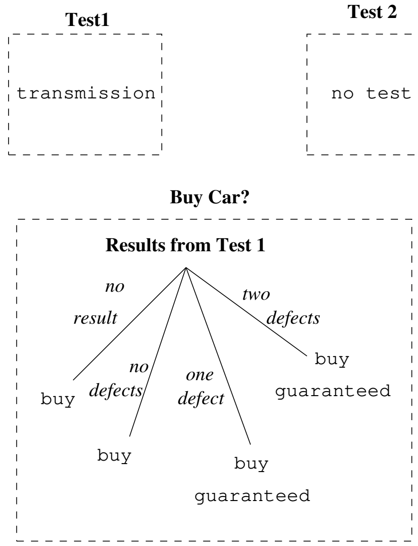


Figure 4.3: Three decision trees.
These decision trees form a policy for the Car Buyer influence diagram in Figure 2.3. There is one tree for each decision node: Test 1*,* Test 2 *and* Buy Car?.

For multi-stage problems, random access refinement keeps a current best-known policy. This policy serves two purposes. First, the policy indicates what the decision maker should do if the refinement process were to halt before more refinement occurs; for this purpose, we desire a deterministic policy, which indicates the *MEV* action for each information state. Second, a policy gives a basis upon which the refinement process can build an improved policy; for this purpose we need a stochastic policy. The reason for this can be made clear by example.

Consider Figure 4.3, which shows three decision trees, one tree for each decision node in the Car Buyer problem (Figure 2.3, page 27). The decision tree for *Test 1* is a single leaf, which tells the decision maker to perform the test on the transmission. Since there are no information predecessors for this decision node, this decision tree is complete.

The decision tree for *Test 2* tells the decision maker not to perform the test. Note that this decision node has 2 information predecessors. The decision tree does not make use of the available information; every information state is mapped to the action *no test*.

The decision tree for *Buy Car?* is a non-trivial tree, using one of four information predecessors. This decision function tells the decision maker to check the result from the first test: if any defects were found the decision maker should take the guarantee option.

These three decision trees represent a deterministic policy, which the decision maker might follow. However, this deterministic policy overly constrains the refinement process.

The problem with deterministic decision functions can be seen if the decision function for *Buy Car?* were to be refined at the context *one defect* by *Test 2*. If *Test 2* were used to refine the decision function for *Buy Car?* after the algorithm determined that *no test* should be performed at *Test 2*, the refinement could not increase the expected value of the policy. The reason is that only one value of *Test 2* is possible according to the policy which has been computed so far. The refinement using *Test 2* would create a subtree structure with only one non-zero probability context, and the action at this context would be the same as the action the refinement replaced. Thus, the refinement using *Test 2* would not result in a positive increase in expected object value for the policy, since the decision tree for *Buy Car?* would increase in size, but the decision function would not change.

In effect, a deterministic decision function is too committed to the current policy for the purposes of information refinement. If deterministic decision functions are used by the refinement process, splitting a decision function using a decision node would never change the deterministic policy.

To solve this problem, the current policy is represented using a stochastic mapping from information state to action. For each context γ in each decision tree, each available action has an associated probability, representing the belief that future refinement will endorse the action as best in all more refined contexts. We show our approach to computing this belief below. Intuitively, this belief is based on the idea that the decision maker will act according to the *MEV* action for a given context if no future refinement is made; however, the belief also acknowledges that there is some possibility that future refinement will uncover contexts in which the original *MEV* is no longer the best.

The belief $\mathrm{P}(d_i|\gamma)$ that future refinement will endorse the action d_i as best in all contexts which are refinements of γ is defined by reasoning by cases, as follows. Let q be the probability that further refinement will occur after the current refinement step. Let m_i be the probability that action d_i will be taken in any future context refined from the given context, if future refinement occurs. If no further refinement occurs, the policy should prescribe the *MEV* action for each context. Let the parameter $r_i = 1.0$ if action d_i is the *MEV* action in context γ; $r_i = 0.0$ for all other actions in the context. If there are several *MEV* actions, one of them is assigned $r_i = 1.0$. Using reasoning by cases, we define

$$\mathrm{P}(d_i|\gamma) = qm_i + (1-q)r_i$$

We assess the parameters q and m_i by the following meta-level considerations.

We argue that m_i should be close to unity if the expected value of action d_i is high relative to the value of possible outcomes. If the expected value of d_i is relatively high for a context, it is possible that a better action is available for a refinement of that context. However, d_i is probably a good choice in all refinements of the context. If there is a much better action d_j for some particular refinement of γ, the context must have relatively low probability; otherwise, the expected object value of d_i in γ would be lower. Similarly, m_i should be close to zero if the expected object value of d_i in context γ is relatively low. A simple way to realize this intuition is to use $m_i \propto E[v|d_i, \gamma]$; that is, m_i is proportional to the expected value of action d_i in context γ. We note that this is a convenient heuristic, since the value $E[v|d_i, \gamma]$ is a quantity which is computed during the construction of an extension subtree (Section 3.3.1, page 65).

The probability q is the probability that the anytime algorithm will complete at least one more refinement without being interrupted. Given that refinements consume finite portions of finite computational resources, the probability of being interrupted during the next refinement step will increase as these resources are consumed, so the rate at which q decreases is problem dependent. After a number of informal experiments, we found that it works well to decrease q at a rate of decrease which is inversely proportional to the number of refinements made so far, *i.e.*, if k refinements have been made in total, $q = \frac{1}{k+1}$. This is a very simple approach which seems to work fairly well (as shown in Section 4.3). The parameter q is similar to the learning rate parameter found in machine learning algorithms.

We end this section with a definition. A *stochastic decision tree* represents the incomplete decision functions during the random access refinement process. It differs from the decision trees discussed in Section 3.1 only at the leaf vertices. Instead of a single action (the MEV action), the stochastic decision tree labels the leaf l with a probability distribution over the actions $d \in \Omega_{D_k}$, $\mathrm{P}(d|\gamma_l)$.

When the refinement process halts, the uncertainty over action in a given context is resolved by setting $q = 0.0$.

4.2.2 The global effects of local refinement

The second complication is that the refinement process has global effects. For the purpose of refining a particular context γ within a decision tree, we keep the remainder of the policy fixed. Before refining the decision tree for D_k at context γ, the decision function currently prescribes an action d. After the refinement of γ, the decision function may indicate that actions different from d are better for the new contexts derived from γ.[1] Other decision trees may have been constructed assuming $D_k = d$; now that this decision has been changed, the value of the policy may be different, and the probabilities of the possible outcomes may have changed.

The data structures used by our algorithm are updated to account for these changes. Since our stochastic decision trees store probabilities and expected utilities, we must update them appropriately.

The expected value of each leaf in each tree must be recomputed (we store the expected value at the leaf of the decision tree). As well, we store in our decision

[1] For refinements to have a positive effect on expected value, a refinement needs to prescribe a different action for at least one of the extended contexts.

trees the probability of each vertex in every context, given the information which precedes it (from the root). These are recomputed as well.

The algorithm for the global update is quite simple. Let D_k be the decision tree which has just been refined. The algorithm recomputes the posterior probability of each internal vertex in the decision trees for $D_{k+1}, \ldots, D_n$. These can be computed most efficiently using a depth first traversal of each tree. We observe that changing these probabilities will also have a cumulative effect on the expected value of the policy.

After the internal vertices' posterior probabilities have been updated, the expected value of all the leaf vertices needs to be recomputed. This computation is necessary because if the probability of any context has changed, the expected value will change: the expectation will put more or less weight on the value of a particular action, according to the probability of the context in which that action is taken. Starting with the decision tree D_n, and working backwards to D_1: for each leaf l, we compute the value of action d_i in context γ_l.

Figures 4.5, 4.4 and 4.6 give the global update procedure which performs these two processes. The processes are illustrated in Figure 4.7.

The global update algorithm becomes more costly as the decision trees become larger. Each update requires one computation of posterior probability for each internal vertex and an expected value computation for each leaf. In the worst case all the stages have probabilities and expected values updated. The total number of leaf nodes on all the trees is $O((b-1)N + D)$, where N is the number of refinements which have been made in total, and D is the number of decision nodes in the

```
procedure update-probabilities
  Input:
    Context γ
    Decision node D

  Let δ be the decision tree for D
  Let X be the vertex at context γ in δ

  If X is a leaf
    Return
  Otherwise X is an internal vertex
    Compute P(X|γ) and store at vertex
    For each x ∈ Ω_X
      Apply update-probabilities(xγ, D)
```

Figure 4.4: A procedure to update the probabilities stored in a decision tree.

```
procedure update-expected-value
  Input:
    Context γ
    Decision node D

  Let δ be the decision tree for D
  Let X be the vertex at context γ in δ

  If X is a leaf
    Compute P(D|V = true, γ) and store at the leaf
    Compute P(V = true|, )D = d*, γ where
      d* = arg max_{d∈Ω_D} P(D|V = true, γ)
  Otherwise X is an internal vertex
    For each x ∈ Ω_X
      Apply update-expected-value(xγ, D)
```

Figure 4.5: A procedure to update the expected values of a decision tree.

procedure global-update
  Input:
    an index i for a decision node

  For $j = i + 1, \ldots, n$
    update-probabilities(ϕ,D_j)
  For $j = n, \ldots, 1$
    update-expected-value(ϕ,D_j)

Figure 4.6: The global update procedure for random access refinement.

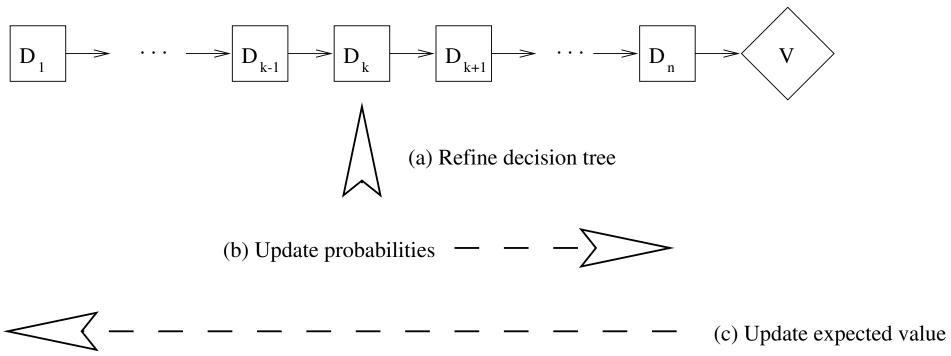


Figure 4.7: The two steps in the global update of a policy.
Following the refinement at stage k (a), the probabilities of events in later stages are updated (b). Thereafter, the expected values are updated for all stages (c).

influence diagram. The total number of internal vertices in all the decision trees is $O((b-1)N+D)$. Therefore, each global update $O((b-1)N+D)$ computations of expected value and posterior probability, in the worst case.

We apply the global update process after every refinement of the policy. An avenue for further research would investigate the behaviour of the random access refinement algorithm using the global update process in a more judicious manner. A trade-off exists here, between computation and expected value, which bears further study.

4.2.3 Computing expected value

Policies are stored as stochastic decision trees. The influence diagram models the decision problem. As in single stage problems, the algorithm will assess the value of a refinement by comparing expected value before and after the refinement. In this section, we will show how the expected value of a decision tree is computed for multi-stage problems.

As in Chapter 3, the random access refinement algorithm uses a Bayesian network to compute posterior probabilities and expected values. The influence diagram is converted to a Bayesian network as described in Section 3.3, except that decision functions are treated differently. Each decision function is installed into the Bayesian network by constructing a conditional probability table consistent with the stochastic decision function and $\mathrm{P}(D|\gamma_l)$ at each leaf l.

Let γ_l be the context of a leaf l in decision tree δ_k. The value of an action d

in this context is defined as:

$$E[v|d,\gamma_l] = \sum_{w \in \Omega_{\Pi_V}} v(w,d)\mathrm{P}(w|\gamma_l)$$

In this definition all other decision functions are stochastic.

The expected value of a decision tree t for D_k is defined:

$$E_t = \sum_{l \in t} E[v|d_l^*, \gamma_l]\mathrm{P}(\gamma_l)$$

where d_l^* maximizes $E[v|d,\gamma_l]$ for all actions $d \in \Omega_D$. We note that E_t is defined assuming the decision maker will take the best action d_l^* for the context γ_l; the uncertainty over action does not enter into this definition.

4.2.4 The random access refinement algorithm

The high level description of the algorithm is given in Figure 4.8. The algorithm is discussed briefly step by step. We note that the policy returned by the algorithm is stochastic, in the sense that the leaf vertices contain probability distributions over the possible actions. This stochastic policy is easily made deterministic; the *MEV* action at a leaf is the one whose probability is highest.

Initialization: The initialization process considers each decision node in order $D_n, \ldots, D_1$. For each decision node, the probability distribution $P(D_i)$ is determined for the empty context. This step requires three queries to the Bayesian network for each decision node, as described in Section 3.3.1.

Choosing a decision function to refine: We maintain a priority queue of extensible leaf vertices, ordered by heuristic value. The queue contains pairs (D_i, l) where D_i is a decision node, and l is a leaf on the decision tree for D_i. Thus, the

```
procedure Random Access Refinement
  Input:
    Multi-stage influence diagram with decision nodes
      D_1, ..., D_n
  Output:
    Policy Δ = {δ_1, ..., δ_n}, a set of decision trees

  For each D_i, initialize δ_i as a single leaf
  Do {
    Choose an extensible decision tree δ_i
    Choose a leaf from δ_i
    Replace the leaf with an extension
    Update the global policy
  } Until (stopping criteria are met or policy is complete)
  Return the policy
```

Figure 4.8: The random access refinement algorithm.

heuristic value assigned to a leaf determines not only the order in which the leaf vertices for a single tree are extended, but also the the order in which the decision functions are refined. As a result, decision functions are refined in order of the heuristic importance of the refinement, rather than a predetermined sequence. The heuristics discussed in Section 3.4.2 are used for this purpose.

Replace the leaf with an extension: As in the single stage algorithm, an extension is chosen for a given leaf. This can be done by one of the strategies described briefly in Section 3.4.1. An extension is chosen from the list of possible extensions to the context of the given leaf. The leaf represents the probability distribution over the possible actions, and is computed as described in Section 4.2.1.

Updating the global policy: Each decision tree $D_{i+1}, \ldots, D_n$ has its observation probabilities updated: for each vertex X, recompute $\mathrm{P}(X|\gamma_X)$. The chance node representing the decision in the Bayesian network is changed to match the update.

Each decision tree $D_n, \ldots, D_1$ has its expected value updated. For each leaf vertex, a single query for $\mathrm{P}(D|v, \gamma)$ will provide a vector of m_i values, from which we can compute $\mathrm{P}(D_i|\gamma)$ as in Section 4.2.1. The query $\mathrm{P}(V|d^*, \gamma)$ will give the expected value of the best action. Finally, the chance node representing the decision in the Bayesian network is changed to match the update.

Returning the policy The policy is made up of a sequence of stochastic decision trees; the probability distributions at the leaf vertices of these trees can be removed, and replaced by the action which maximizes the distribution. As shown in Section 4.2.1, this action also maximizes expected utility for the context.

4.2.5 Complexity

We can analyze the cost of this procedure as follows. Suppose a decision node has n information predecessors, each with at most b values. To find a maximal extension for a single leaf requires $O(b(n-k))$ expected value computations, where k is the number of internal vertices already in the context for the leaf.

As discussed in Section 4.2.2, a single global update phase requires $O((b-1)N + D)$ computations of expected value and probability, where N is the number of refinements which have been made in total, and D is the number of decision nodes in the influence diagram. Computing a single expected value or posterior probability requires a single query to the Bayesian network.

Thus, the total cost, in terms of the number of queries to a Bayesian network, of the a single refinement and update is $O(b(n-k)+3((b-1)N+D))$.

If the algorithm is not stopped before all the policies are complete, the procedure requires $O(b^{n+1})$ queries just for the refinements. In the worst case, the updates after each refinement add $O(b^{2n})$ total queries when the decision trees are complete. This is substantially more effort than is required by an exhaustive enumeration of the state space; however, we do not intend our algorithm to be used to build a complete policy. For decision problems with large information spaces, a policy is available for use by the decision maker with much smaller cost than the limit of a complete policy. The value of such a policy is still at issue. We will first provide an example, using the Car Buyer influence diagram; in Section 4.3, the algorithm will be applied to a number of larger decision problems.

4.2.6 Example: The Car Buyer problem

The Car Buyer problem was introduced in Section 2.2.1; here we show some of the results of applying the random access refinement algorithm to it. Figure 4.9 shows the results from several heuristic/strategic variations. The data for these variations were collected using an upper limit of the number of refinements made to the policy; here we use 10 refinements only.

The x axis of the graph measures the number of queries made to the Bayesian network, and the y axis measures the expected object value. Each line in this figure connects the sequence of policies constructed by one variation of the algorithm. Each point on a line represents the expected object value of a policy. The first point is common to all lines, as it represents the value of acting randomly, with a uniform probability distribution over all actions; this is the "policy" which exists before any deliberation has occurred. The second point on each line is also common to all variations of the algorithm, since all variations initialize the decision trees in the same way.

Recall from Section 4.1 that a policy which maximizes expected object value can be constructed after initialization and a single (well chosen) refinement: observing the Results from Test 2. This particular refinement was made as the very first refinement after initialization by two of the four variations shown: Second Best Action/Maximal Extension and Post Hoc/Maximal Extension. The resulting policy is not the optimal policy, since the initialization phase initialized the first decision to prefer not performing a test. The global update phase eventually corrects this policy; this happens after nine refinements for Post Hoc/Maximal Extension.

The first few global updates have little effect on the initial policy, since the probability q that computation will end after the current refinement is finished is still fairly low. As long as q is small, commitment to buying the car is weak, and the decision function for *Test 1* is "conservative," directing the decision maker to refrain from any testing. As computational resources are consumed, *i.e.*, as q increases, this conservatism is abandoned, as it becomes clear that testing the *Fuel & Electrical Systems* will result in a better value.

The next section applies the random access refinement algorithm to some large decision problems, demonstrating that the process constructs valuable policies at a fraction of the cost of computing the optimal policy using exhaustive enumeration.

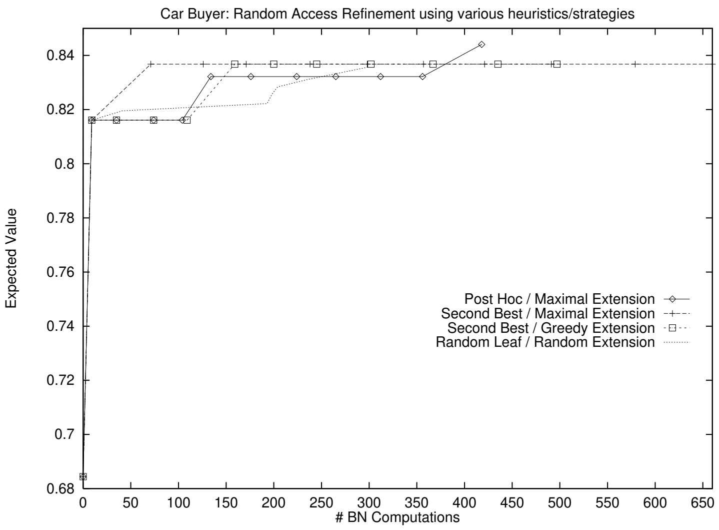


Figure 4.9: A performance profile for random access refinement.
The performance of the random access refinement algorithm using various resource allocations applied to the Car Buyer problem. The policy which maximizes EV_I *has an expected value of* 0.844061*, and is attained by the Post Hoc/Maximal Extension variation after 9 refinement operations and 420 queries. Exhaustive enumeration would compute the optimal policy after more than 200 queries.*

4.3 Empirical Results

The random access refinement process is intended to find valuable policies with a relatively small investment of computational resources. A number of large influence diagrams were constructed to demonstrate that the algorithm does achieve this intention. The influence diagrams are identical in topology, but the conditional probabilities vary. The problems have a real interpretation, in contrast to the randomly generated problems of Chapter 3. The purpose of running the algorithm on slightly varying problems is to demonstrate the effect of variations in the problem on the performance of the algorithms.

4.3.1 The problems

The decision problems are based on the model of an agent traversing a maze. The mazes consist of walls and open space, and are represented by square tiles whose size correspond to the agent's single step. See Figure 4.10. The agent has five available actions: it can move a single step in any of the four compass directions *N, S, E, W*, or stay in place. The agent has four sensors *NS, ES, SS, WS*, one in each compass direction.

The agent can only detect walls (with or without noise); the agent's position is not directly observable. The goal of the agent is to arrive at a specified location in the maze from any starting point.

The problem of choosing an action can be represented by an influence diagram; the representation imposes a finite structure on the problem, namely that the agent is limited to a fixed number of actions. A single stage is shown in Fig-

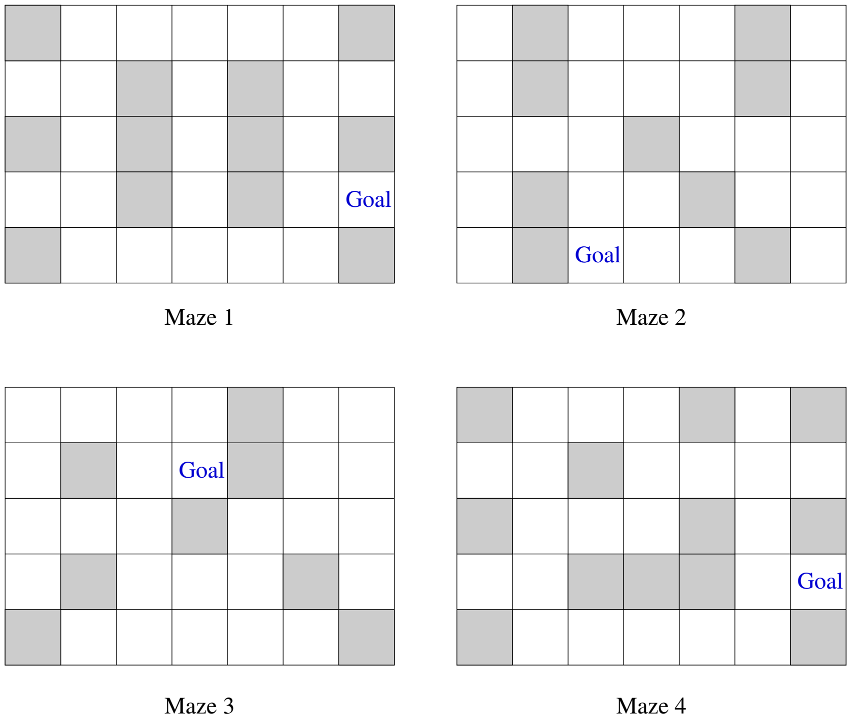


Figure 4.10: The mazes for the maze walker problem.
The shaded tiles are obstacles, and there are walls around the perimeter of the maze. The problem is to find a policy which directs the agent to the goal from any unoccupied position in the maze.

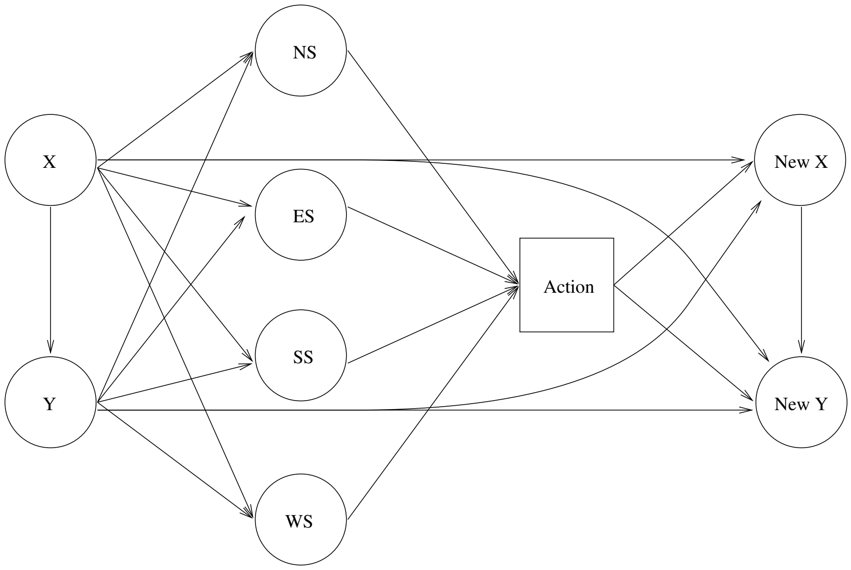


Figure 4.11: An influence diagram fragment for the Maze Walker.
This influence diagram fragment shows a single stage for variations of the maze walker problem. The problems solved in this chapter iterate this structure ten times.

ure 4.11. The four sensors are directly connected to the decision node. The two state variables affect the sensors directly, but are themselves not directly observable by the agent. In principle, the single stage can be repeated any number of times; no-forgetting arcs connect the maze walker's previous sensors and actions to the the current action. In the figure, the no-forgetting arcs have not been drawn.

The probabilistic information required by this influence diagram forms the agent model. Sensors can be modelled with the conditional probability distributions $\mathrm{P}(NS|X,Y)$, *etc.* Actuators can be modelled by the conditional probability distributions $\mathrm{P}(NewX|X,Y,Action)$ and $\mathrm{P}(NewY|X,Y,Action,NewX)$.

Four agent models were used in this test. These correspond to two sensor models: perfect and noisy; and two actuator models: perfect and noisy. The perfect sensors always detect a wall when there is one, and never detect a wall when there isn't one. The noisy sensor correctly detects walls with probability 0.9, and fails to detect a wall with probability 0.1. The noisy sensor may detect a wall when there is none, with probability 0.05, and will correctly report the absence of a wall with probability 0.95. The perfect actuators always put the agent in the correct square for a given action.

The noisy actuator model is as follows. Taking a given action from its current position, the agent may move to the correct square, may fail to move from the current position, or may move to one of the squares adjacent to the current position. No movement is possible to squares which are not adjacent to the current position. If there are no walls adjacent to the current position, the noisy actuator model indicates that it is ten times more likely that the agent moves to the correct square than to stay in the current position; the agent moves to an incorrect adjacent square with one-tenth the probability of staying in the current position. For example, if the agent is in a square with no adjacent walls, and attempts to move north, there are 5 possible outcomes. The agent ends up in the right place for a given action with a probability of about 0.885 (100/113), and with probability about 0.0885 (10/113), the agent fails to move from its current position; there is a very small probability of about 0.00885 (1/113) of moving to one of the three incorrect adjacent squares. The model changes slightly when there are walls in adjacent squares. The probability of moving to occupy a wall square is reduced to zero, and the probability distribution above is renormalized. Thus, if the agent tries to move in the direct of an adjacent wall, it is likely that it will stay in its current position.

The value function is not shown in the ID fragment. It depends only on the position of the agent in the final stage, and puts full value (1.0) on being at the goal, and zero elsewhere. In terms of $vsize$, presented in Chapter 3, the value function is quite large, even though its dependence on the state is quite simple (*i.e.*, it can be described by a decision tree which uses the final X and Y position variables only). This is because the state is "hidden" from the decision maker; if these hidden variables are summed out by expectation (say, using Shachter's algorithm to eliminate the chance nodes $NewX$ and $NewY$), the value function becomes a function of all the sensor variables, and the decision nodes.

The mazes used in our experiments are shown in Figure 4.10; Maze 1 is an example from [28]. In our experiments, the agent is allowed ten stages to reach the goal, which makes it possible to reach the goal from each starting position. Using 10 stages, the tenth decision node has 49 direct predecessors, and an information space of about 2^{60}.

Maze 1 has a simple policy which guides the perfect agent to the goal from each possible starting position. The policy guides the agent south whenever possible, or otherwise east whenever possible. If neither south nor east is possible, the agent moves west, if possible, and otherwise stays in place. This decision function is repeated for the first 8 stages. The final two steps of the policy direct the agent north one step and east one step. This policy has an expected value of 1.0, and can be represented by 8 decision trees which use 3 internal vertices each, followed by two decision trees which need no internal vertices.

Optimal policies for the perfect Maze Walker in Mazes 2, 3 and 4 are not as obvious, but it turns out that all the there exist policies which guide the perfect agent to the goal from all starting positions; applications of the our algorithms have constructed policies with expected value of 1.0 (these results are included in the next section).

The optimal policies for the agents with imperfect sensors or actuators are unknown; the value of the optimal policy depends in part on the difficulty of the maze, and in part on the abilities of the agent.

4.3.2 Results for sequential refinement

The sequential refinement algorithm, using the Second Best Action/Greedy Extension combination, was applied to these sixteen problems. For this experiment, three different resource allocations were made. The first scheme allocated two refinement steps to each decision tree; the second scheme allocated three steps, and the third scheme allocated four steps. The purpose of this experiment is to demonstrate how allocation affects the value of the policies.

Figure 4.12 shows the results of the four agents in Maze 1, using 2 refinement steps per stage. Each dataset in the graph corresponds to one of the agents navigating the maze. The x-axis of the graph measures computational costs in terms of queries to the Bayesian network. The y-axis measures expected value. Each mark on a line represents the state of a policy at a particular time. The initial point represents the completely random policy, and is a common starting point for each problem. Each subsequent point represents a refinement step, according to the allocation. The policy is given a value as discussed in Section 4.1.1: if no decision

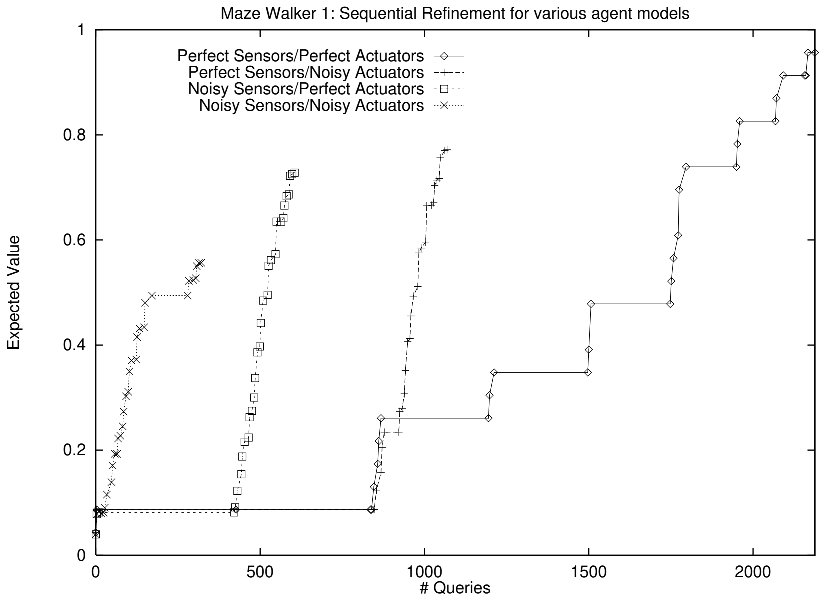


Figure 4.12: Sequential refinement with 2 refinements per stage.
The performance of the sequential refinement algorithm using various agent models for the Maze Walker problem (Maze 1). The Second Best Action/Greedy Extension combination was used to construct policies for the problems. The resource allocation scheme allowed two refinements per decision node.

tree has yet been constructed at a decision node, the policy is valued as if an action were to be chosen at random using a uniform probability distribution.

The graph shows how the policies' value increases with expenditure of computational resources. The perfect agent's dataset is marked with plateaus. These plateaus are the result of the algorithms' search through the possible extensions, but finding none which can improve the policy. The plateaus get shorter because the set of possible extensions decreases for each decision node: the first decision node has only 4 information predecessors; the tenth decision node has 49 information predecessors. Recall that the refinements are applied in reverse of the execution ordering; that is, the first refinement occurs at the last decision node in the execution ordering, so the largest plateaus will occur earlier.

The first of these plateaus is a consequence of the layout of Maze 1. In order for the agent to reach the goal in the last step, the agent must be in the goal position already, or must be one step away from it. In Maze 1, there is only one position adjacent to the maze. Choosing to step west will get the agent from the adjacent square to the goal. However, there are no observations which will improve the policy significantly, since there are no other positions adjacent to the goal. Therefore none of the observations which could be made (neither the current sensor readings, nor any of the readings or actions in the past) will improve the policy. All four agents have this plateau; the agents with perfect sensors are longest, and the agent with noisy sensors and noisy actuators is shortest.

The remaining plateaus in the policy of the perfect agent are consequences of its sensor model. For example, a perfect agent will not be able to improve a policy by using corroborating evidence: if a pair of sensor readings is sufficient to determine the position of the agent, a third sensor reading will only tell the agent what it already has inferred from its first two readings. In this case, the policy cannot improve by making another observation. On the other hand, if imperfect sensors are used, a third sensor reading may corroborate the readings of the first two. In this case, the third sensor reading may be able to improve the expected object value of the policy.

The remaining agents' policies increase in value with fewer plateaus. This is a consequence of the fact that a noisy sensor reading can usually corroborate other noisy readings, bringing at least a small increase in expected value to the policy. Recall that the Greedy Extension strategy stops looking through the possible extensions when a positive increase in value is found. Because it is likely that a sensor reading will increase the expected value a little, the number of queries is substantially smaller for the agents with noisy sensors than for the agents with perfect sensors. However, because the sensors are noisy, the probability of correctly identifying the current location is smaller than for agents with perfect sensors.

The graph in Figure 4.12 shows the results for Maze 1; the results for the remaining mazes are summarized in Table 4.3.2. Given a resource allocation of 3 refinements per stage, the sequential refinement algorithm constructed a policy with expected value of 0.9565 for the perfect agent after 2167 queries. The difference from the optimal policy is about 5%. The other agents' policies had lower value, due to the noise in sensors or actuators. We do not know the optimal policies for the remaining agents. The expected value of the policies constructed by sequen-

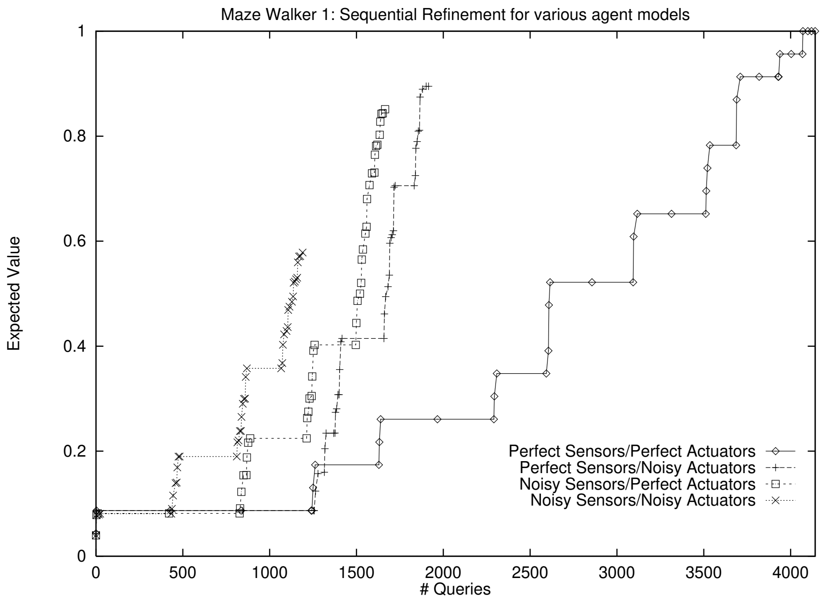


Figure 4.13: Sequential refinement with 3 refinements per stage.
The performance of the sequential refinement algorithm using all agent models for the Maze Walker problem (Maze 1). The Second Best Action/Greedy Extension combination was used to construct policies for the problems. The resource allocation scheme allowed three refinement steps per decision node in addition to the initialization.

tial refinement depends on the agent model: the perfect agent's policy is higher in expected value than the others' policies. The perfect agent's policy required more queries because of the plateaus described above.

The average runtime of the algorithm on these 16 problems was 299.6 seconds (standard deviation: 258.9) on a SPARC Ultra-2.

Figure 4.13 shows the results of applying sequential refinement using three refinements per stage to the agents in Maze 1; this is one more refinement per stage than Figure 4.12. The policies constructed are of higher value, but with a fairly substantial increase in the number of queries used. In particular, the policy constructed for the perfect agent is optimal, after 4071 queries. The relative increase in expected value is 4.5% and the the number of queries increased by a factor of 1.9 over the policy constructed using only 2 refinements per stage. The expected value for this agent increases with plateaus, as in Figure 4.12, but the plateaus are longer.

The other agents' policies were improved from Figure 4.12, but the graph shows a number of plateaus which were not present with the smaller allocation. For the agent with noisy sensors and noisy actuators, the relative increase in expected value was 3.9%; the number of queries used increased by a factor of 3.7. The extra refinement has helped find better policies, but not all of the refinements resulted in positive changes in expected value.

The average runtime of the algorithm on these 16 problems was 673.2 seconds (std: 423.9) on a SPARC Ultra-2.

Figure 4.14 shows the results of using four refinements per stage for the agents in Maze 1. The number of plateaus has increased from Figure 4.13 for all agent models. The policies, however, have not improved for all agent models. The perfect agent's policy is optimal; the extra refinement step was not needed. The policy for the agent with noisy sensors and noisy actuators improved by 17.2% over the policy constructed with three refinements per stage. For this agent, the number of queries increased by a factor of 1.7. This increase is due to the fact that the additional refinement step allowed by the allocation causes the algorithm to search for a refinement with positive increase in expected value. However, this search fails frequently, since the sensor information is perfect, and additional information may be redundant.

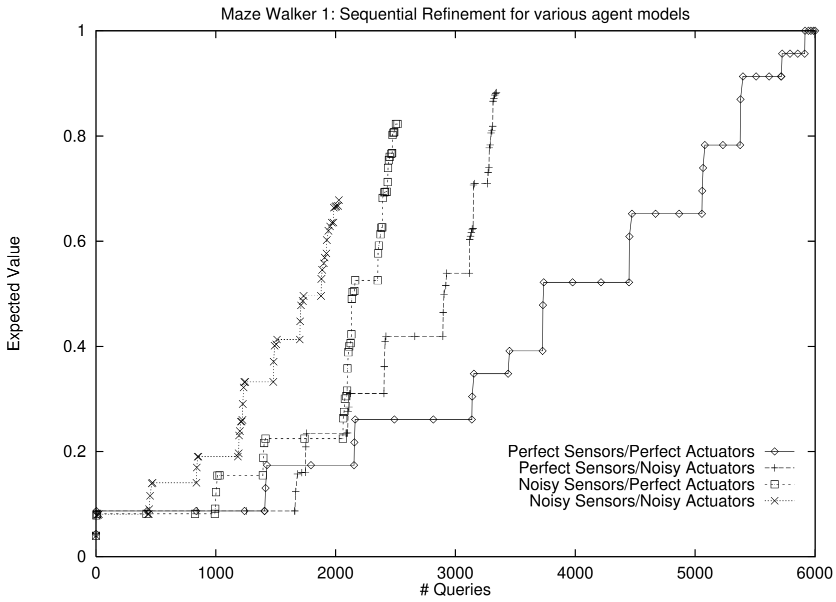


Figure 4.14: Sequential refinement with 4 refinements per stage.
The performance of the sequential refinement algorithm using all agent models for the Maze Walker problem (Maze 1). The Second Best Action/Greedy Extension combination was used to construct policies for the problems. The resource allocation scheme allowed four refinement steps per decision node in addition to the initialization.

The policies for the remaining two agents actually decreased as a result of the extra refinement per stage. The expected value of the policy for the agent with perfect sensors and noisy actuators decreased by 1.4%; for the agent with noisy sensors and perfect actuators, the decrease in value was 3.4%. The decrease is a result of Greedy Extension strategy. The extra refinement step has not made any possible extension less valuable; rather, the extra refinement step has caused the Greedy Extension strategy to choose a less valuable possible extension.

The average runtime of the algorithm on these 16 problems was 1112.9 seconds (standard deviation: 653.5) on a SPARC Ultra-2.

Table 4.3.2 summarizes the results for all four agents in all four mazes, and the three allocations. The expected value of of the best policy constructed is listed, together with the number of queries used to construct it. The table shows how the allocation of refinement steps affects the expected object value of the policies.

The expected value of policies constructed using 3 refinements per stage increased by 0% to 18% of the value of policies constructed using only 2 allocations per stage. The corresponding increase in number of queries ranged from 1.7 to 6.9 times. The expected value of policies constructed using 4 refinements per stage changed by -11% to 17%, with a corresponding increase in the number of queries used ranging from 0.5 to 3.2 times. As described above, the decreases in the expected value are a result of the Greedy Extension strategy choosing less valuable extensions.

	2 Refinements per stage		3 Refinements per stage		4 Refinements per stage	
Agent Model (Sensor/Actuator)	Best Policy	Queries	Best Policy	Queries	Best Policy	Queries
Perfect/Perfect	0.9565	2167	1.0	4071	1.0	5918
Perfect/Noisy	0.7719	1069	0.8950	1915	0.8822	3342
Noisy/Perfect	0.7280	605	0.8514	1665	0.8226	2503
Noisy/Noisy	0.5568	321	0.5785	1190	0.6780	2027

Maze 1

Perfect/Perfect	0.8462	1889	1.0	3257	1.0	5115
Perfect/Noisy	0.5175	184	0.6488	699	0.6440	1869
Noisy/Perfect	0.5349	237	0.5873	455	0.6379	1489
Noisy/Noisy	0.4176	673	0.5021	1797	0.5141	2124

Maze 2

Perfect/Perfect	0.9259	2011	1.0	3754	1.0	5607
Perfect/Noisy	0.6208	233	0.6224	1371	0.6559	2286
Noisy/Perfect	0.5503	302	0.6330	780	0.6799	1462
Noisy/Noisy	0.4445	212	0.5008	1478	0.4455	850

Maze 3

Perfect/Perfect	1.0	2112	1.0	3845	1.0	5921
Perfect/Noisy	0.7585	584	0.8381	1334	0.9095	2093
Noisy/Perfect	0.7852	1034	0.8580	1885	0.8646	3208
Noisy/Noisy	0.6105	177	0.6463	699	0.7086	1939

Maze 4

Table 4.1: Sequential refinement summary.
A summary of results of applying sequential refinement to the Maze Walker problems. Three allocation schemes were used. The expected value of the best policy is listed, together with the number of queries to the Bayesian network used by the algorithm.

4.3.3 Discussion

The experiment showed that increasing the allocation of refinements resulted in an increase in expected value of a policy. As well, the increase in expected value is accompanied by an increase in the number of queries needed to construct the policy. The increase in the number of plateaus, as shown for Maze 1, suggests that there is some value in knowing when to stop refining at a particular stage, or how many refinements to make at a stage. For example, in the case of Maze 1, there is only one square from which the goal position is accessible. This information could have been used to avoid the long plateaus which were the result of refining the last decision tree (it gets refined first). This would have decreased the number of queries needed substantially. In the case of Maze 1, this could have been predicted. In general, it may be harder to know when extra refinements are not going to improve the expected object value of a decision function.

The increase in refinement allocations sometimes brought a decrease in expected value. This is a result of the Greedy Extension strategy. It is likely that the Maximal Extension strategy would have constructed more valuable policies than the Greedy Extension strategy, but at a much higher cost, in terms of queries to the Bayesian network. To get a picture of the potential cost of the Maximal Extension strategy, consider that each increase in expected value is accompanied by an increase in the number of queries roughly the size of the closest plateau. Because of this, no experiments were performed using the Maximal Extension strategy with sequential refinement for these problems.

The allocation of refinement steps was intended to be a rough allocation of computational resources. The results of this experiment are positive, in the sense that a small allocation of resources resulted in valuable policies. The variation in the actual resources used is a benefit of the Greedy Extension strategy. The resources were allocated uniformly across all stages. The Greedy Extension strategy was able to make positive contributions to the value of the policy, often with a small cost in terms of queries.

The Random Extension strategy was not used. The maze walker problems have the property that the information available from previous stages is less valuable than the sensor information available at the current stage. That is, the past sensor information does not, on its own tell as much about the agent's current state as the current sensor readings. Thus, choosing 4 random extensions from the set of information predecessors at each stage is likely to result in a poor policy.

4.3.4 Results for random access refinement

The random access refinement algorithm was applied to the Maze Walker problems, using two variations. The Second Best Action heuristic was used to select leaf vertices to extend, combined with the Maximal Extension and Greedy Extension strategies to extend each leaf. The algorithm had 30 extensions in total allocated for each problem. This allocation provided a resource constraint for the algorithm, but one in which it seemed possible that an optimal policy could be found for the perfect agent.

Figure 4.15 shows 4 datasets, each dataset corresponding to all agent models navigating Maze 1. The x-axis measures computational costs, in terms of the num-

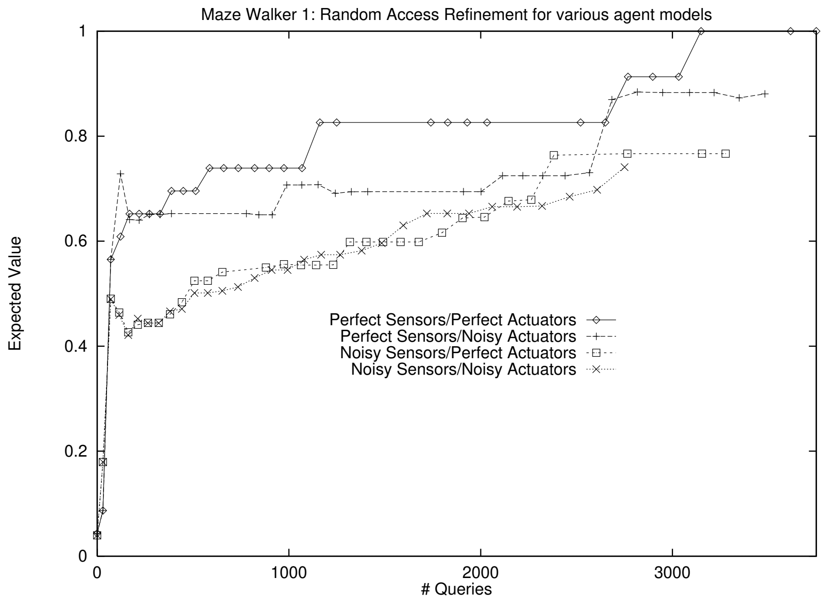


Figure 4.15: Random access refinement using the Greedy Extension strategy. *The performance of the random access refinement algorithm using various agent models for the Maze Walker problem (Maze 1). The Second Best Action/Greedy Extension combination was used to construct policies for the problems.*

ber of posterior probabilities and expected values computed (queries to the Bayesian network). The y-axis measures expected value of each policy. Each point on a curve represents the expected value of a policy in the sequence of policies constructed by the algorithm. Here we show the expected value of the deterministic version of the policy. The first policy is the same for each of the problems, and represents the value of acting randomly (before any deliberation has occurred). The data in Figure 4.15 was collected using the Second Best Action/Greedy Extension combination.

The average run time on a SPARC Ultra-2 for these four problems was 54.1 minutes. Three of the agents had run times of less than 25 minutes, and the problem with the agent with perfect sensors and noisy actuators required 156 minutes.

For the perfect agent, the algorithm does not find the optimal policy using the allotted resources, but levels off at an expected value of 0.9130 after 2373 queries. The policy guides the agent to the goal from 21 of the 23 starting positions. The error here is 8.7% from optimal.

Also of note is the fact that the algorithm is able to find a policy for the agent with perfect sensors and noisy actuators which exceeds the value of the best policy for the perfect agent. This behavior is due to the heuristics used by the algorithm. The different probability distribution in the problems will give rise to different heuristic values for the possible refinements.

For some of the agent models, the algorithm produces policies which decrease in value (for example, in the range of 122 to 219 queries in Figure 4.15). This behaviour is the result of making refinements when the commitment to the current policy is weak. The refinement takes advantage of the relatively high probability of a non-MEV action. When the effects of the refinement are made global, the non-MEV action drops in probability, and any action which was chosen based on the probability that a non-MEV action might be taken will drop in value. This drop in value is temporary, and further refinement, stronger commitment, and global updates correct for the decrease.

The curves in Figure 4.15 give an indication of how the conditional probabilities underlying the agent model affect the performance profile. When the probabilities are very sharp, and a few states contain most of the probability mass (as in the case of the perfect agent), the increases tend to be steep and plateaus are common. As the probability mass of is distributed over many more states (as in the agent with noisy sensors and noisy actuators), the increases tend to be less steep, and the plateaus shorter.

The curve for the agent with perfect sensors and noisy actuators also shows decreases in expected value: in the range of 122 to 169 queries, the expected value drops by about 12%; in the range of 1152 to 1242 queries, the expected value drops by about 2% These decreases have the same explanation as the decreases mentioned above. The effect is smaller as the number of queries increases, since the commitment to the *MEV* action is stronger.

These curves are typical (see Appendix B for all the graphs for all the agents in all four mazes).

Table 4.2 summarizes the performance of the various agents in the various mazes for the Second Best Action/Greedy Extension combination. The expected value of the best policy is shown, with the number of queries used to compute this policy is given, and the number of refinement operations after which the policy was available. Where the number of refinement steps is less than the limit of 30, the refinements which followed did not increase the expected value of the policy. The algorithm was able to find good policies for the perfect agent in Mazes 1 and 4; the optimal policy for the perfect agent in Maze 4 was constructed after 16 refinement steps, and 1267 queries. For Maze 2, information refinement using the Second Best Action/Greedy Extension combination constructed a policy for the perfect agent worth only 0.5769, which is just more than half of the optimal policy. For Maze 3, the best policy constructed by the algorithm for the perfect agent had an expected value of 0.7407; the difference between this policy and optimal is about 26%.

The average runtime for these 16 problems was 33.2 minutes; thirteen of these problems required less than 25 minutes. Dynamic programming using exhaustive enumeration would require more than 2^{60} queries to compute complete policies.[2]

The Second Best Action/Maximal Extension combination was applied to these problems. The performance of the agents in Maze 1 is shown in Figure 4.16. The optimal policy for the perfect agent was found after 7447 queries and 29 refinement steps. The best policies constructed for the other agents in this maze were higher than the best policies for these agents using the Greedy Extension strategy. Note that the scale of the Figure 4.16 is higher than in Figure 4.15; on average, the Maximal Extension strategy requires more queries than the Greedy Extension strategy to determine an extension to a given leaf.

The results for the Second Best Action/Maximal Extension combination are summarized in Table 4.3. The optimal policy for the perfect agent in Maze 4 was constructed after 9678 queries, and 30 refinement steps. For mazes 2 and 3, the best policies for the perfect agent were 0.6923 and 0.7037, a difference of about 30% from the optimal policy. The other agents' best policies are somewhat higher here than the policies constructed using the Greedy Extension strategy. The relative changes ranged from -5% to 38%. The number of queries needed to compute the best policies is substantially larger for the Maximal Extension strategy than for the greedy strategy, from 2.2 to 7.6 times larger.

[2] To get an idea of the scale of this number: the figure is about 10 cm wide; at this scale, 2^{60} queries is approximately 32 light-hours to the right. It would take about 587 million years to compute according to the average reported above.

	Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Maze 1	Perfect/Perfect	1.0	3149	28
	Perfect/Noisy	0.8805	3482	30
	Noisy/Perfect	0.7666	2765	28
	Noisy/Noisy	0.7409	2750	30

	Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Maze 2	Perfect/Perfect	0.6923	1976	24
	Perfect/Noisy	0.4534	3005	30
	Noisy/Perfect	0.5294	2825	30
	Noisy/Noisy	0.3983	3206	30

	Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Maze 3	Perfect/Perfect	0.7407	1716	20
	Perfect/Noisy	0.5851	3420	30
	Noisy/Perfect	0.6347	2872	24
	Noisy/Noisy	0.4724	3009	30

	Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Maze 4	Perfect/Perfect	1.0	1267	17
	Perfect/Noisy	0.8216	2398	28
	Noisy/Perfect	0.8377	2202	26
	Noisy/Noisy	0.6966	3339	30

Table 4.2: A summary of results for random access refinement.
The value of the best policies for the maze walkers, found using the Second Best Action/Greedy Extension combination of random access refinement and a resource limit of 30 refinements. The expected value of the best policy constructed is listed; The column labelled "Steps" indicates the number of refinement steps which were performed to reach the best policy; if the number of steps is fewer than 30, the remaining steps did not improve the policy. The optimal policy for the perfect agents is known to have expected value 1.0 for all mazes. Dynamic programming would require about 2^{60} *queries to compute an optimal policy.*

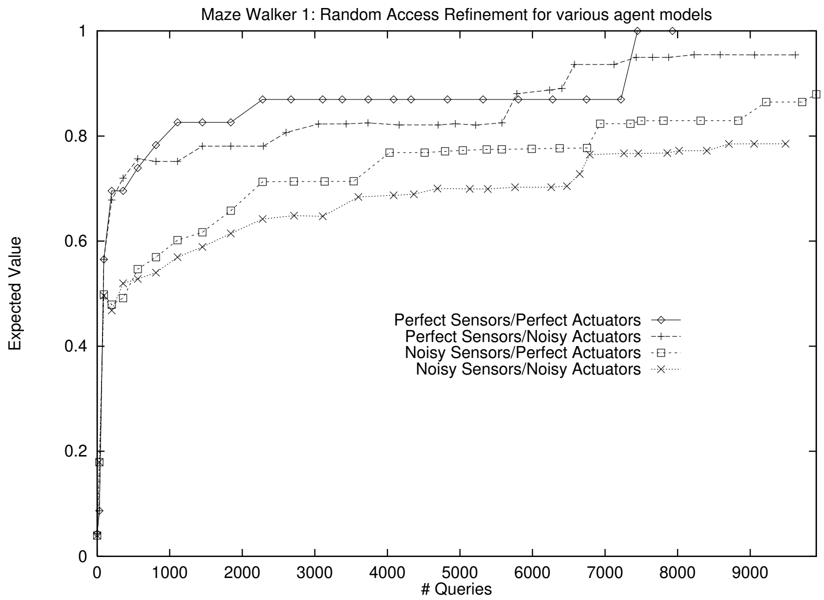


Figure 4.16: Random access refinement using the Maximal Extension strategy. *The performance of the random access refinement algorithm using various agent models for the Maze Walker problem (Maze 1). The Second Best Action/Maximal Extension combination was used.*

Maze 1

Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Perfect/Perfect	1.0	7447	29
Perfect/Noisy	0.9544	9622	30
Noisy/Perfect	0.8795	9911	30
Noisy/Noisy	0.7853	9054	29

Maze 2

Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Perfect/Perfect	0.6923	5435	29
Perfect/Noisy	0.6260	9336	29
Noisy/Perfect	0.6203	10677	30
Noisy/Noisy	0.5230	9470	30

Maze 3

Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Perfect/Perfect	0.7037	4522	16
Perfect/Noisy	0.6236	7741	25
Noisy/Perfect	0.6468	9939	30
Noisy/Noisy	0.5719	8753	28

Maze 4

Agent Model (Sensor/Actuator)	Best Policy	Queries	Steps
Perfect/Perfect	1.0	9678	30
Perfect/Noisy	0.8189	9718	30
Noisy/Perfect	0.8954	10103	30
Noisy/Noisy	0.7147	9938	30

Table 4.3: A summary of results for random access refinement.
The value of the best policies for the maze walkers, found using the Second Best Action/Maximal Extension combination of random access refinement and a resource limit of 30 refinements. The expected value of the best policy constructed is listed, along with the number of queries used, and the number of refinement operations used. The optimal policy for the perfect agents is known to have expected value 1.0 for all mazes. Dynamic programming would require about 2^{60} *queries to compute an optimal policy.*

4.4 Discussion

This chapter has presented two procedures for solving multi-stage decision problems using the information refinement technique of Chapter 3. The first technique allocates a fixed amount of computational resource towards the solution of each individual decision node in the network. The difficulty with this procedure is in assigning the resource allocation *a priori*.

A second procedure was motivated by the difficulties of the first. An "anytime" algorithm was developed which could make refinements to the policy anywhere in the decision sequence. The procedure is very expensive asymptotically, but has been shown to construct valuable policies with relatively small amounts of computational resources.

The sequential refinement algorithm generally produced policies with higher expected value and lower computational costs than the random access refinement algorithm. However, the random access refinement algorithm uses the results of its previous computation to guide future refinement, whereas the sequential refinement algorithm requires that the available resources be allocated prior to deliberation.

For the larger of the problems "solved" in this chapter, no optimal policy is known. These problems are too large to enumerate the information space exhaustively.

It is possible to construct an influence diagram for which the optimal policy can only be expressed as a set of complete decision trees. Nevertheless, the data shown in this chapter demonstrate that information refinement constructs reasonably valuable policies using reasonable allocations of computational resources. The

Car Buyer problem and the Maze Walker (perfect sensors and actions) both have a relatively large number of impossible information states; policies which summarize a large subspace of the information set can exploit these asymmetries, never refining impossible contexts. Druzdzel [10] argues that it is common for a few states to cover a large portion of the total probability mass in a joint probability distribution. Thus, while states with small but non-zero probability are likely to be (non-optimally) abstracted together with more likely states, if these states are sufficiently improbable, this approximation will not result in too great a reduction in expected value.

Finally, it is important to acknowledge that the space of IDs is very large, and the set of problems treated in this section is a small sample from a highly restricted subclass of IDs. The evidence in this section shows that there exist large problems for which random access refinement can find policies which are reasonably valuable policies using reasonable amounts of computational resources. These problems are too large to solve using traditional methods.

Chapter 5

Conclusions

The goal of this dissertation was to understand how to make good decisions. A good decision depends on the the way the world works, the way choices affect the world, and on the computational resources available to the decision maker. Our approach is based on the idea that computational actions are choices which must be considered when computational resources are finite, and computational costs are not negligible [18, 42].

We have presented a flexible approach for constructing policies for sequential decision problems expressed as influence diagrams. This approach allows the decision maker to choose a trade-off between the computational costs and object value.

We have proposed and studied three heuristic algorithms which allow a decision maker make such a trade-off, based on the idea of information refinement. These algorithms can construct valuable policies for very large decision problems, or more generally, situations in which the decision maker does not have the computational resources available to compute the optimal policy. These algorithms can be used in decision analysis and artificial intelligence, particularly in the areas of planning under uncertainty and medical informatics.

Information refinement is a process which iteratively improves the value of a decision function (a mapping from available information to action). The initial decision function does not make use of any of the information available to the decision maker; incremental improvements to the decision function are made by including available information as refinements to the contexts in which actions might be taken. This approach is similar to the machine learning technique of learning decision trees (also called classification trees) from examples [38], and has parallels in reinforcement learning [3, 30].

We have proposed and studied a number of heuristics which are used to select information to be included in the decision function. These heuristics are domain independent in the sense that they have been devised without explicit consideration of any particular domain; they make use of domain information provided in the influence diagram representation, and have been shown to be superior to random selection. We feel that the domain independence of these heuristics is valuable as a starting point for understanding how to incorporate domain dependent heuristics into the approach.

The information refinement approach was implemented in three algorithms. A simple anytime algorithm for single stage decision problems was presented. The algorithm was shown to build policies which are valuable to a decision maker over a range of computational expenditures. Asymptotically, this algorithm was shown to require $O(b^n)$ computations of expected value and posterior probability ("queries"), where n is the number of information predecessors, and b is the maximum number of values taken by any of these nodes. This is a constant factor difference from exhaustive enumeration of the state space performed by algorithms which maximize expected object value. The benefit of the anytime approach is that a valuable policy is available to the decision maker with smaller computational cost than the optimal policy.

Two novel algorithms for sequential decision making under uncertainty were developed for multi–stage problems. The first of these, sequential refinement, applies the single stage algorithm to each decision node in the traditional "backwards induction" sequence; decision trees are constructed according to an *a priori* allocation of computational resources. The resource allocation was specified in terms of the size of each decision tree. With this approach, we were able to investigate the effects of allocating resources, without assuming particular resource constraints for a particular decision maker. The drawback of this approach is that it may be difficult to determine *a priori* which decision functions should be allocated more (or less) of the available computational resources.

The second multi–stage algorithm, random access refinement, is an anytime algorithm; that is, a policy is always available, and computational resources are expended to improve the current policy. The decision maker can interrupt the decision making process at any time, and use the current policy. The algorithm takes a novel approach to decision making, in that the refinement process can be applied to the decision functions in any order. This approach requires the decision making process to determine which of the decision functions to refine. This algorithm is a partial response to the problem of how to allocate resources among the decision functions. We used the same domain independent heuristics which guide the refinement of a given decision function to select which decision function to refine. Each iteration of the random access refinement algorithm has two phases: the refinement of a decision function, followed by a global update phase. The global update phase is necessary to maintain consistency between the decision functions, and as the decision trees get larger, this phase becomes more expensive.

We applied the multi–stage algorithms to a number of large problems for which exhaustive enumeration of the information space would be impossible. The sequential refinement algorithm was shown to construct valuable policies using an allocation scheme which constructed decision trees of equal depth for all decision nodes. We showed empirically that a collection of very small decision trees (relative to the size of the information space) can form the basis of valuable policies. We expect that more informed allocations, which distributes the computation less uniformly according to domain specific information, can be used to the benefit of the decision maker's comprehensive value.

The random access refinement algorithm was also shown to construct valuable policies. Initial investments of computation resulted in steep increases in value; as policies became more refined, costs increased, and the incremental values tended to level off.

5.1 Future Work

Our account of computational costs is very coarse grained. Our unit of cost is a single posterior probability computation (query) in a Bayesian network. This is sufficient for purposes of comparison to approaches which enumerate the information space exhaustively, since these approaches must make similar computations. A more fine grained approach would acknowledge that some of our queries are more expensive than others. For example, the early queries in the information refinement approach are less expensive than later queries, because refinement increases the connectivity of the Bayesian network in which the query is made in our algorithms. In algorithms which enumerate the state space exhaustively, the cost of a query does not depend on when it is made, since the connectivity of the network doesn't change. The connectivity of the information refinement approach reaches a maximum when all information predecessors have been included in the policy; this maximum is the connectivity used by traditional algorithms which enumerate the information space. More fine–grained accounting of the cost of a query would give the decision maker more control over the deliberation process.

The initialization step of our algorithms puts a single action (or a probability distribution over the possible actions) as the leaf of a tree, and information refinement is used thereafter to improve the policy. There may be considerable advantage to an alternative approach, which is to initialize each decision tree using a small subset of the available information, and refine the decision trees using one of our refinement techniques. For some problems, this initial set can be determined from the domain. For example, in the maze walker problems (Chapter 4), the information space includes all the sensor readings of the past, as well as a set of new sensor readings at each stage. For this problem, the initial information set could be the set of “new” information available at the stage. Informal experiments have shown that this initialization would be extremely valuable. In general, domain information might make the choice of initial information obvious. We expect that a good, domain independent initial set may be the new information available at a decision. More experiments with multi-stage decision problems will be necessary.

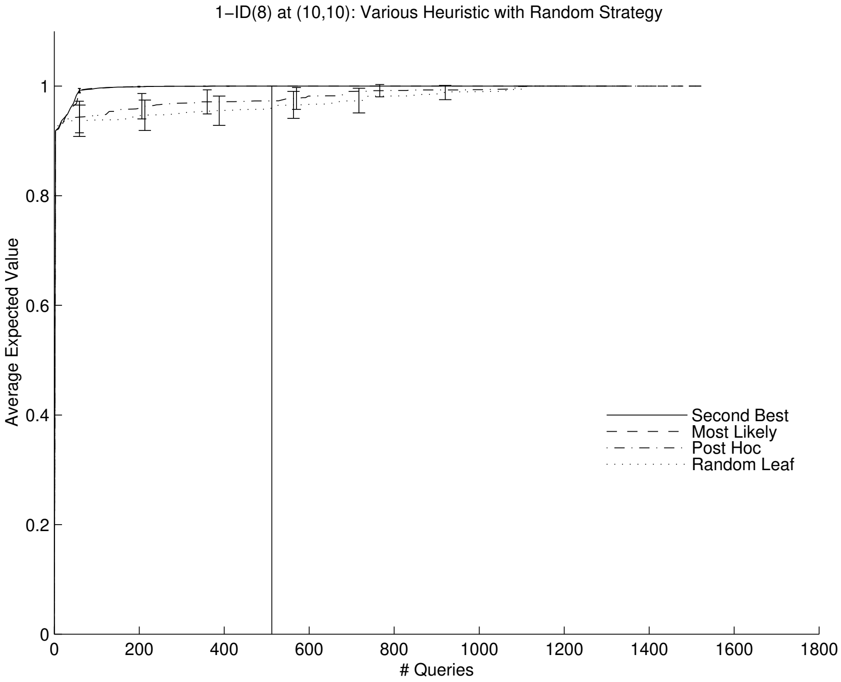
The random access refinement algorithm could be used to implement an algorithm which reasons explicitly about the costs and values associated with computation. Because the approach is myopic, and can exhibit plateau behaviour, the comprehensive value function (the combination of cost and value of computation) may have local maxima. Thus, maximization of comprehensive value is not as simple as detecting a slope of zero. Refinement need not stop at the first maximum; rather, the deliberation should end when the expectation of increase in value is less than the cost of the next refinement. One of the open problems is to find a way to estimate the value of the next refinement (our EVI estimates the value of a refinement at a given leaf using a given information predecessor). It is possible that the expectation of future value can be based on experience; that is, assessing future value based on the performance of the algorithm on other problems.

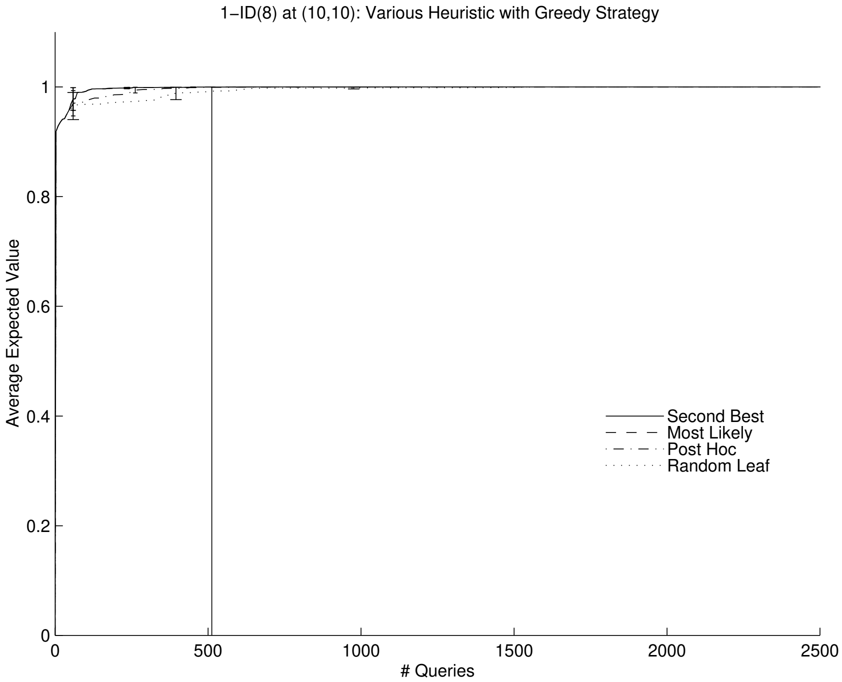
The heuristic component of our algorithms requires further development and experimentation. While we studied our heuristics used individually, it might be valuable for the algorithm to have several heuristics available, and to choose between them. For example, the Greedy Extension approach to selecting a refinement often resulted in a very cheap extension with only marginal improvement to the policy, when the Maximal Extension approach would have resulted in a very expensive extension, with more substantial improvement to the policy. A compromise approach to selecting a refinement could examine a small number of possible extensions, and choose the one which resulted in the highest increase in value.

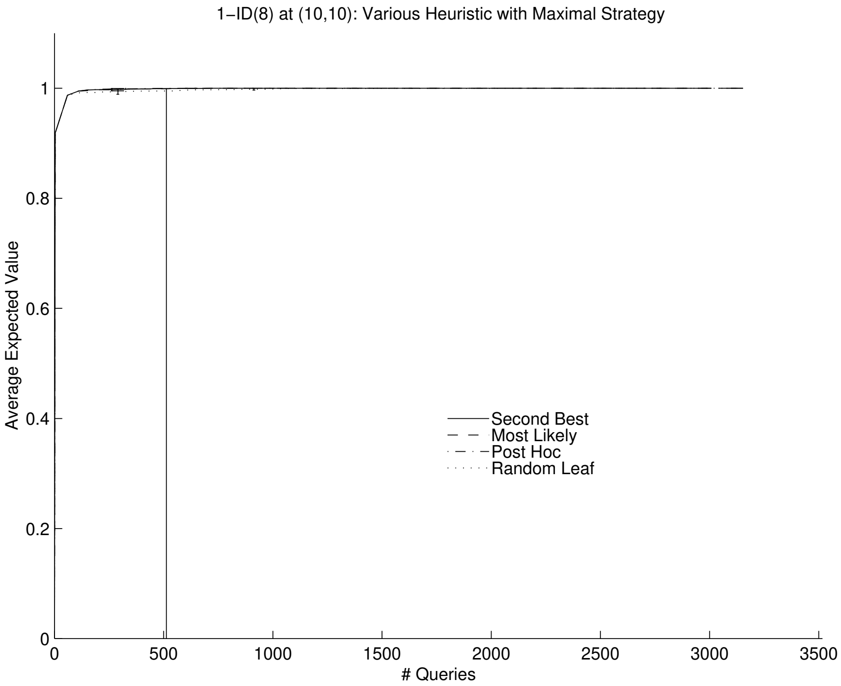
Appendix A

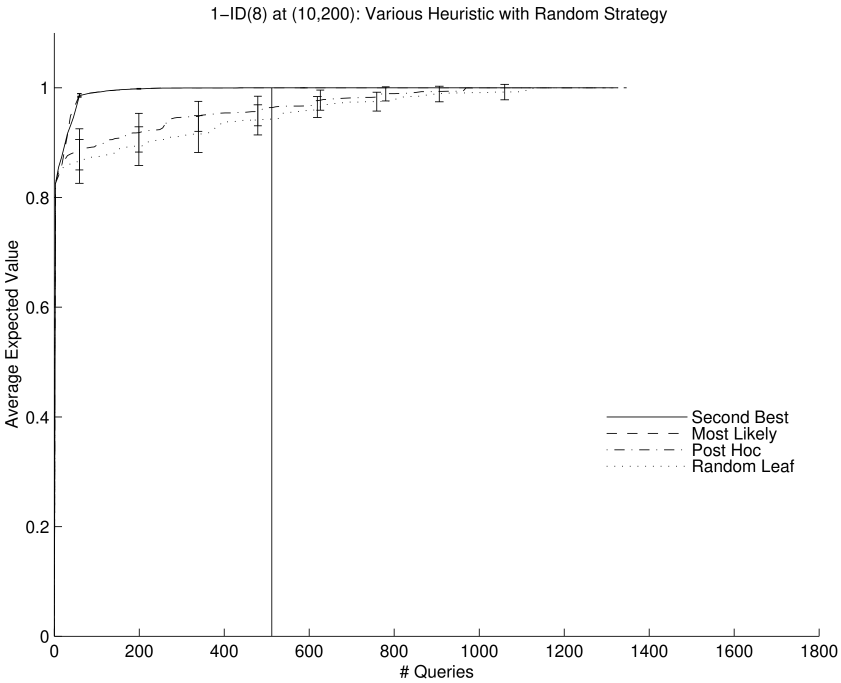
Data for 1-ID(8)

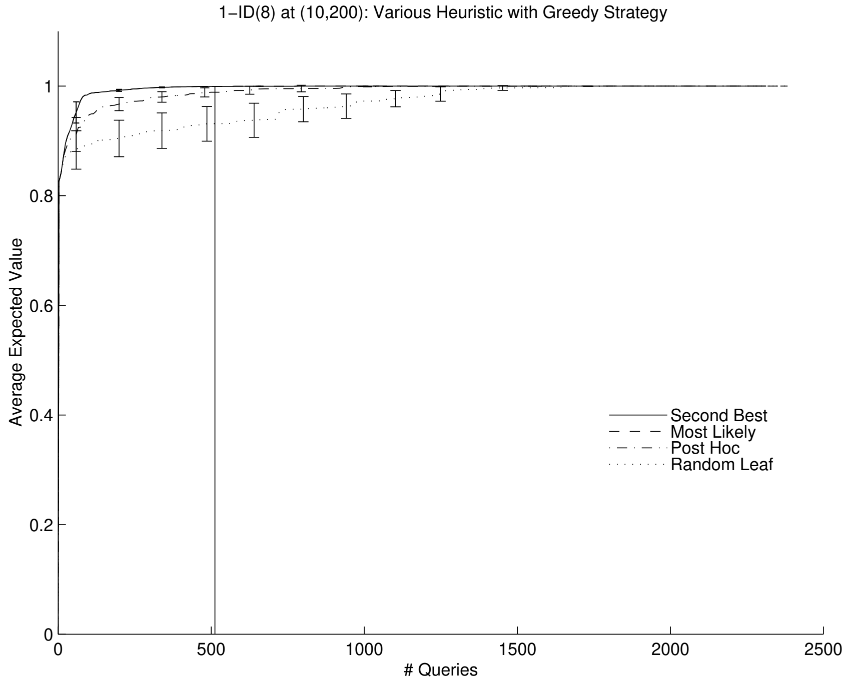
This appendix contains summaries of the data collected by applying information refinement to the 1-ID(8) problems in Chapter 3. The graphs presented here show how the expected object value of the anytime policies change as resources are consumed. Each graph summarizes the data collected for a particular strategy (*i.e.*, Random Extension, Maximal Extension or Greedy Extension), combined with each of the four heuristics (Second Best Action, Post Hoc, Most Likely Context, Random Leaf). There are four sets of influence diagrams to which the information refinement algorithm was applied: (10,10), (10,200), (200,10) and (200,200). The vertical line in each graph shows the point at which exhaustive enumeration would have an optimal policy.

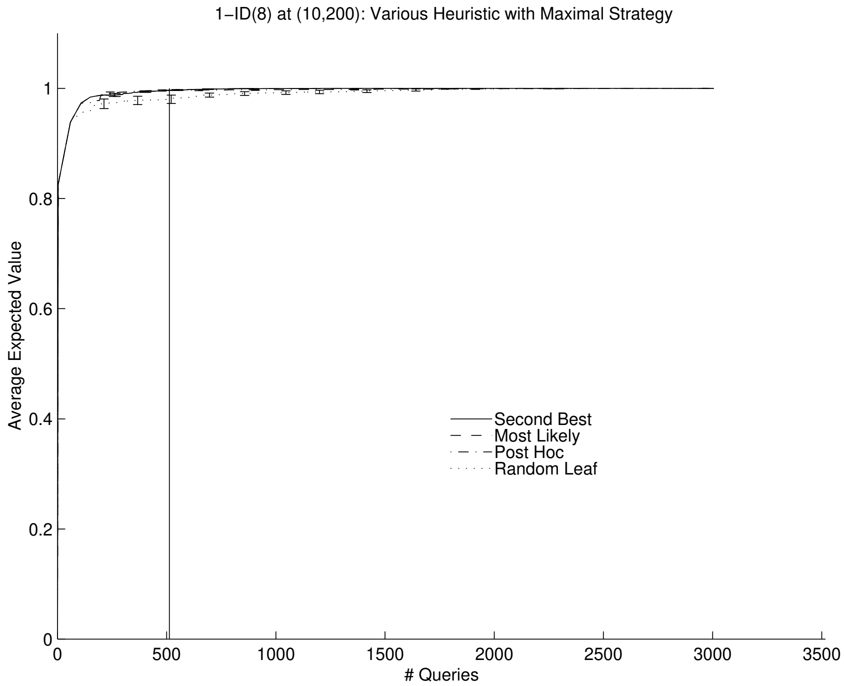

1−ID(8) at (10,10): Various Heuristic with Random Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf

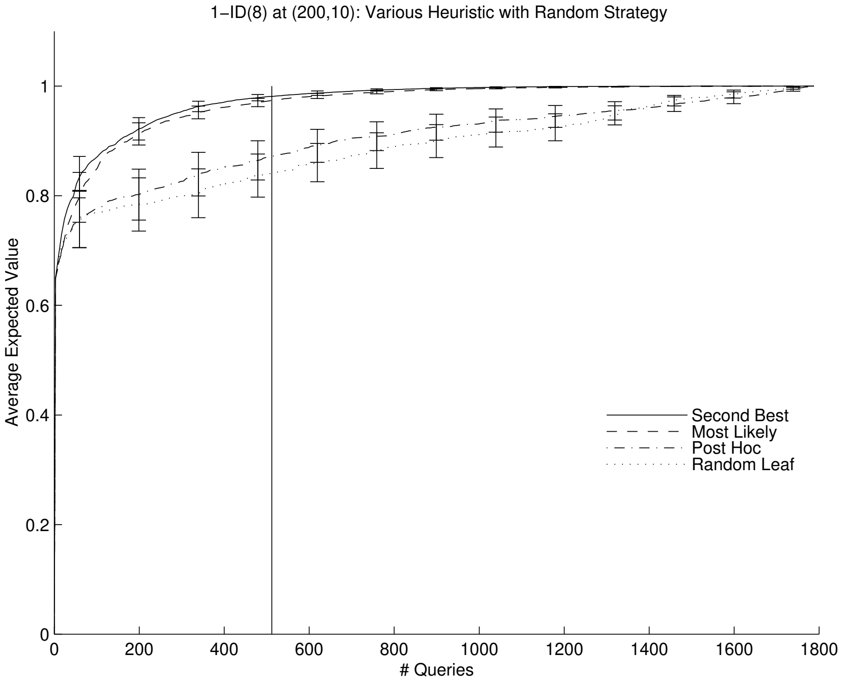

1−ID(8) at (10,10): Various Heuristic with Greedy Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf

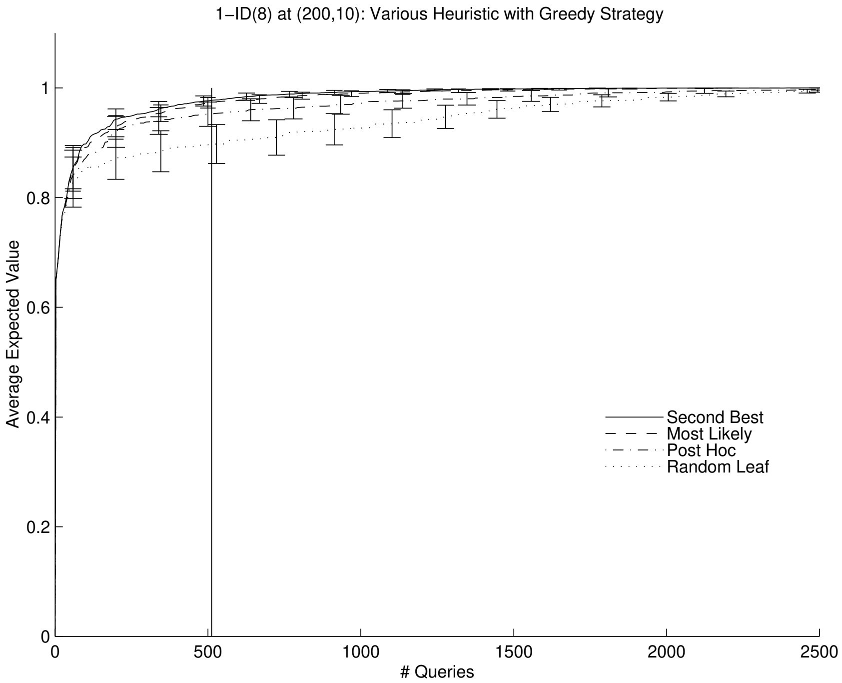

1−ID(8) at (10,10): Various Heuristic with Maximal Strategy
Average Expected Value
Queries
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
3000
3500
Second Best
Most Likely
Post Hoc
Random Leaf

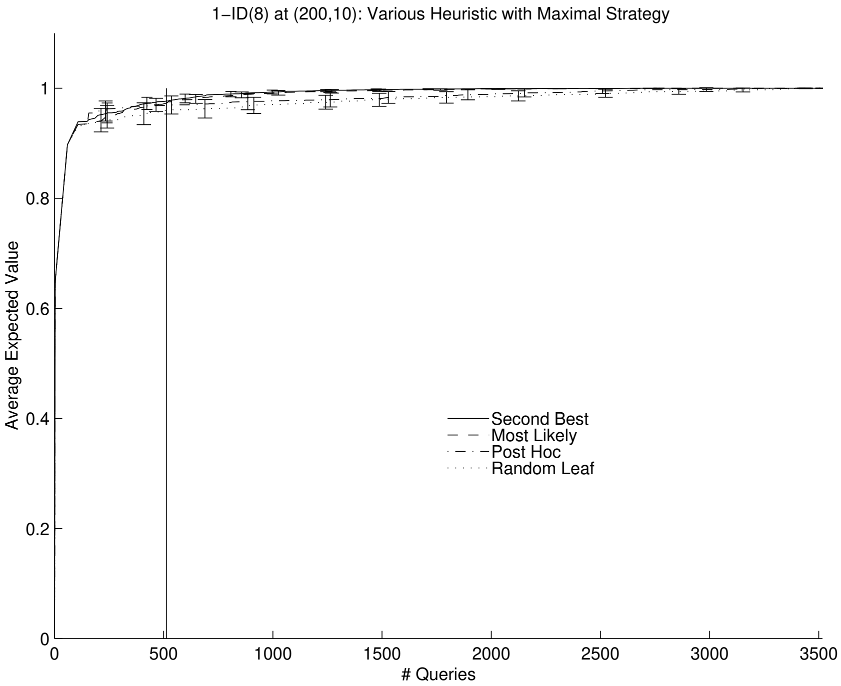

1−ID(8) at (10,200): Various Heuristic with Random Strategy
Average Expected Value
Queries
0
0.2
0.4
0.6
0.8
1
0
200
400
600
800
1000
1200
1400
1600
1800
Second Best
Most Likely
Post Hoc
Random Leaf

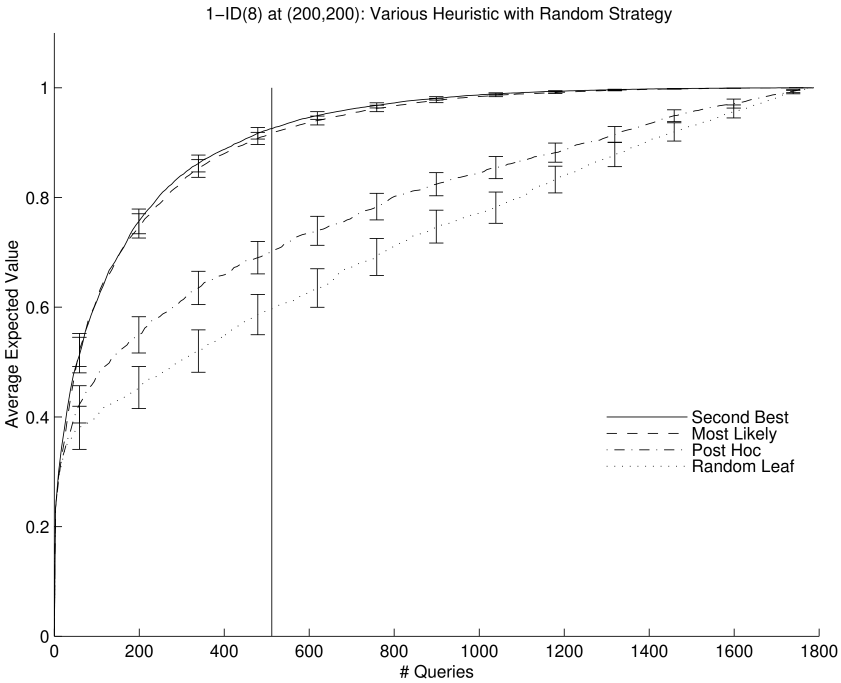

1−ID(8) at (10,200): Various Heuristic with Greedy Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500

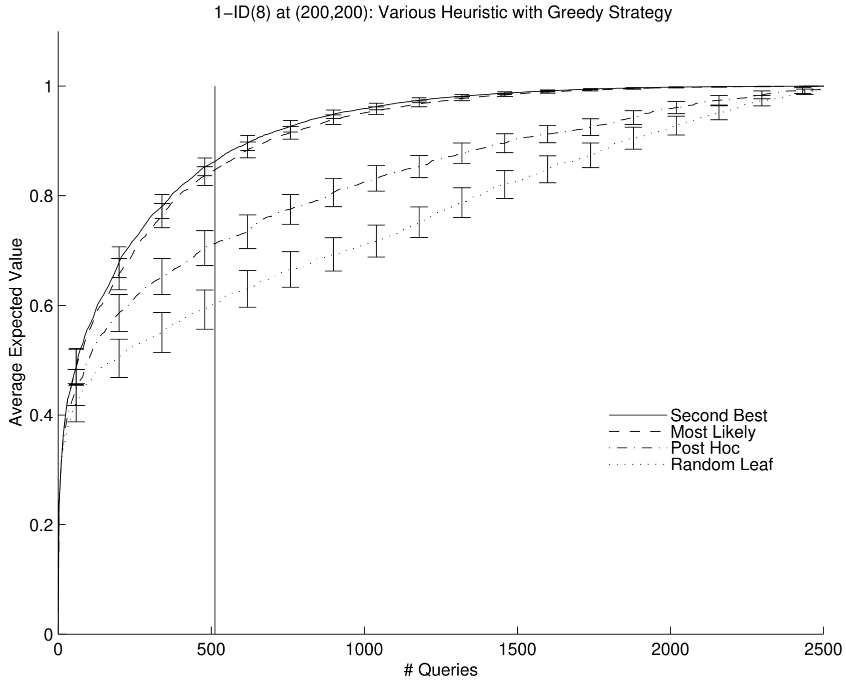

1−ID(8) at (10,200): Various Heuristic with Maximal Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
3000
3500

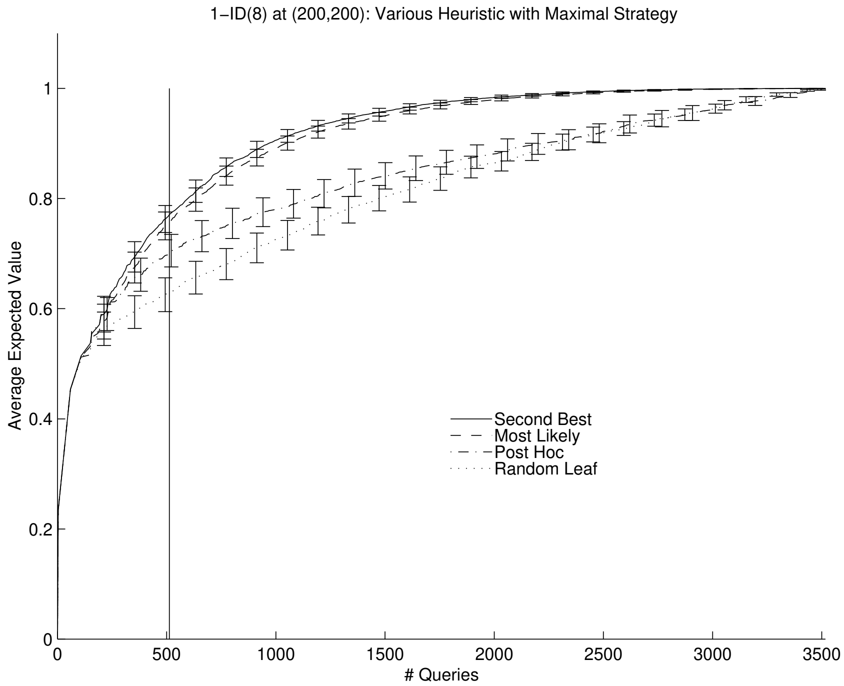

1−ID(8) at (200,10): Various Heuristic with Random Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf


1−ID(8) at (200,10): Various Heuristic with Greedy Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf


1-ID(8) at (200,10): Various Heuristic with Maximal Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf


1-ID(8) at (200,200): Various Heuristic with Random Strategy
Average Expected Value
Queries
Second Best
Most Likely
Post Hoc
Random Leaf


1−ID(8) at (200,200): Various Heuristic with Greedy Strategy
Average Expected Value
Queries
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
Second Best
Most Likely
Post Hoc
Random Leaf

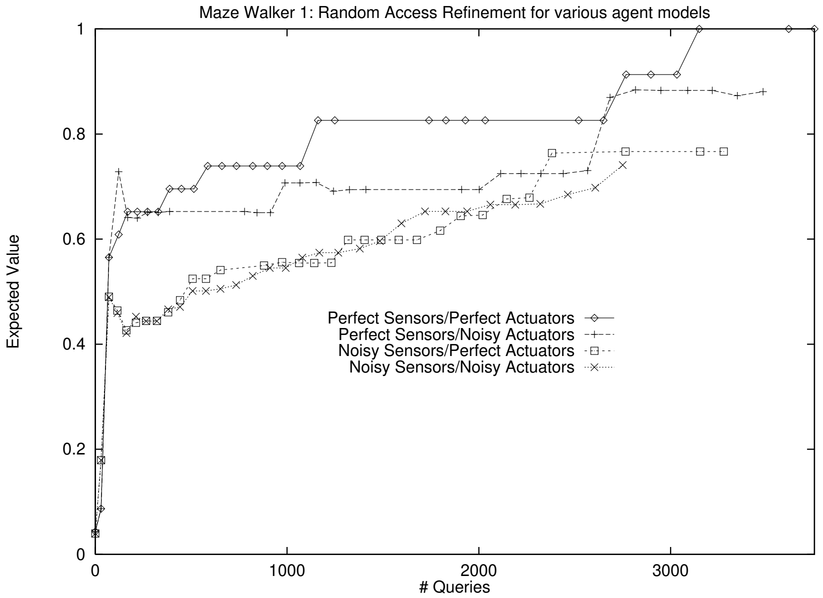

1−ID(8) at (200,200): Various Heuristic with Maximal Strategy
Average Expected Value
Queries
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
3000
3500
Second Best
Most Likely
Post Hoc
Random Leaf

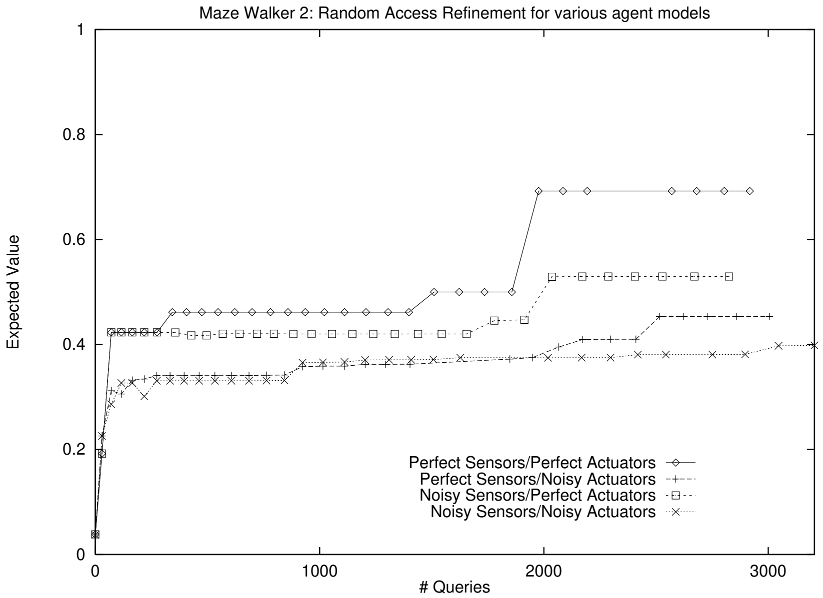
Appendix B

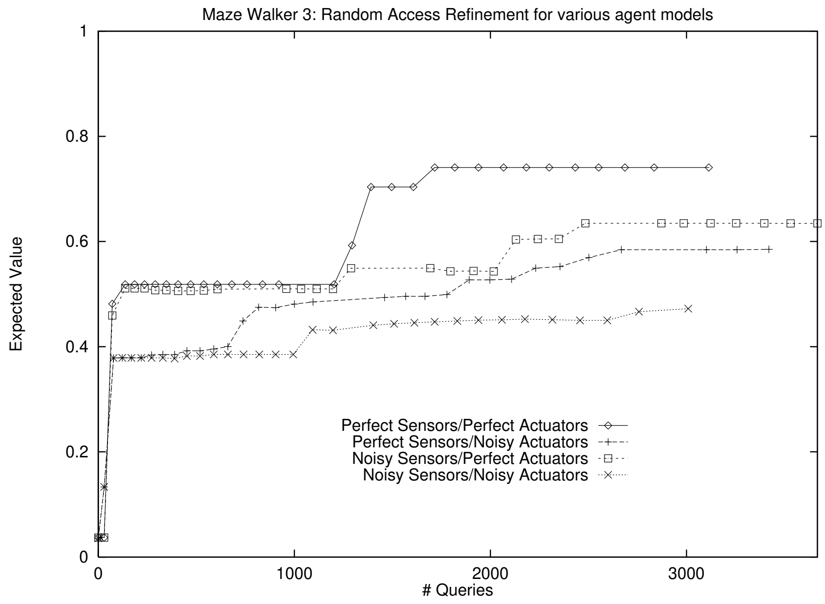
Data for Maze Walker

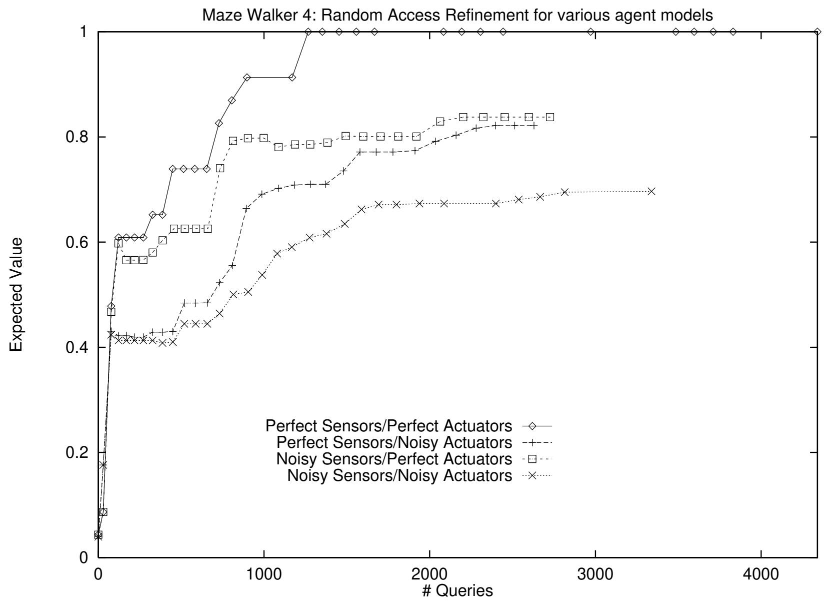
This appendix contains the data reported in Chapter 4 for the random access refinement algorithm. The graphs show the expected value of the policies as a function of the computational resource consumption for the maze walker influence diagrams. Each graph contains data for the four agent models in one of the mazes. Each point represents an anytime policy. The first set of four graphs was collected using the Second Best Action/Greedy Extension combination; the next set of four graphs was collected using the Second Best Action/Maximal Extension combination. The last set of four graphs compare how the various combinations performed on the influence diagram representing the agent with noisy sensors and noisy actuators.

B.1 Second Best Action/Greedy Extension

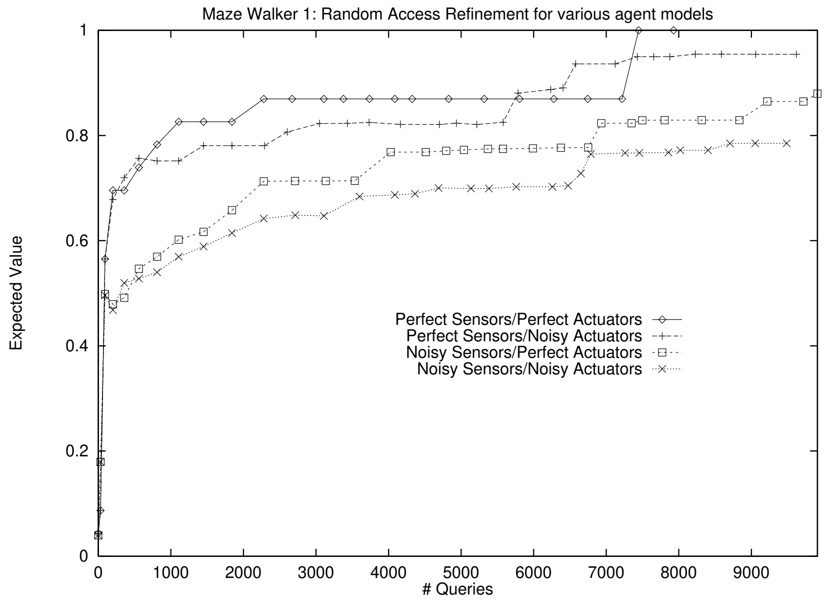


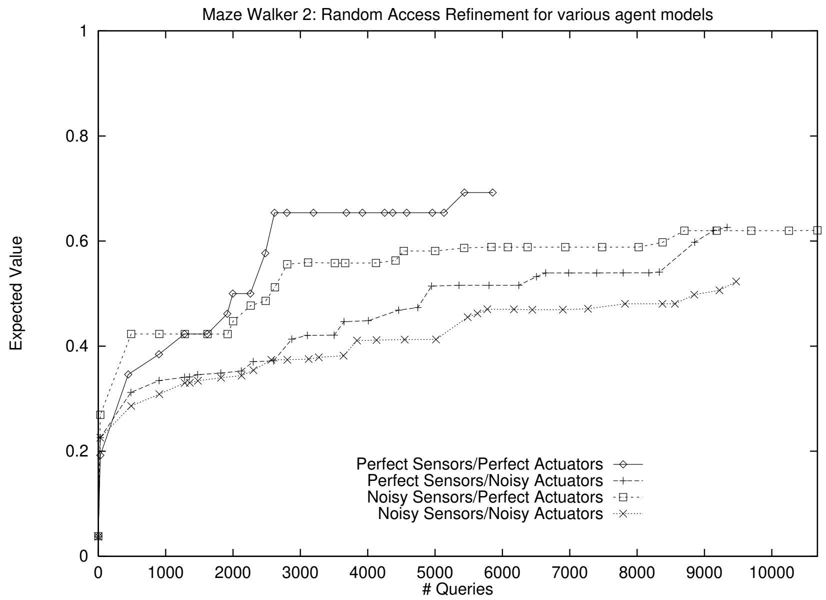


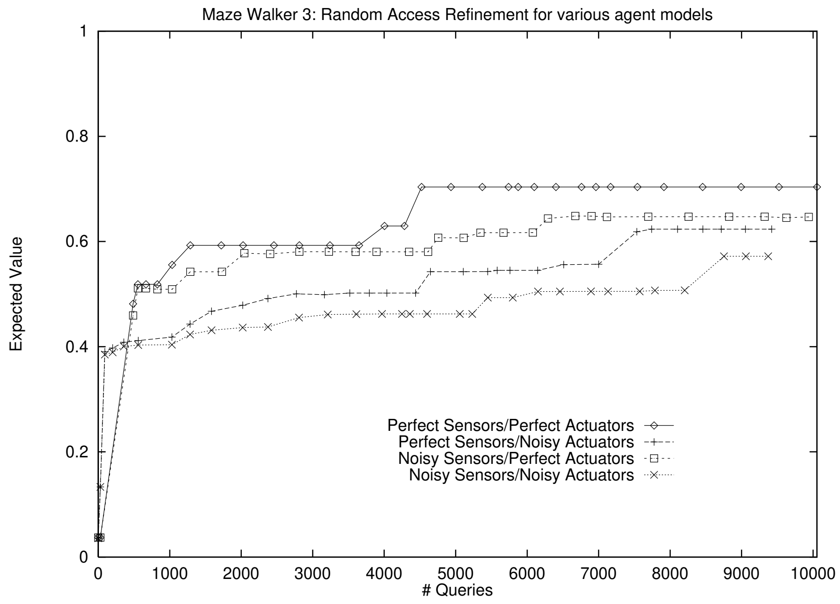

Maze Walker 3: Random Access Refinement for various agent models
Expected Value
Queries
Perfect Sensors/Perfect Actuators
Perfect Sensors/Noisy Actuators
Noisy Sensors/Perfect Actuators
Noisy Sensors/Noisy Actuators
0
0.2
0.4
0.6
0.8
1
0
1000
2000
3000


Maze Walker 4: Random Access Refinement for various agent models
Expected Value
Queries
Perfect Sensors/Perfect Actuators
Perfect Sensors/Noisy Actuators
Noisy Sensors/Perfect Actuators
Noisy Sensors/Noisy Actuators
0
0.2
0.4
0.6
0.8
1
0
1000
2000
3000
4000

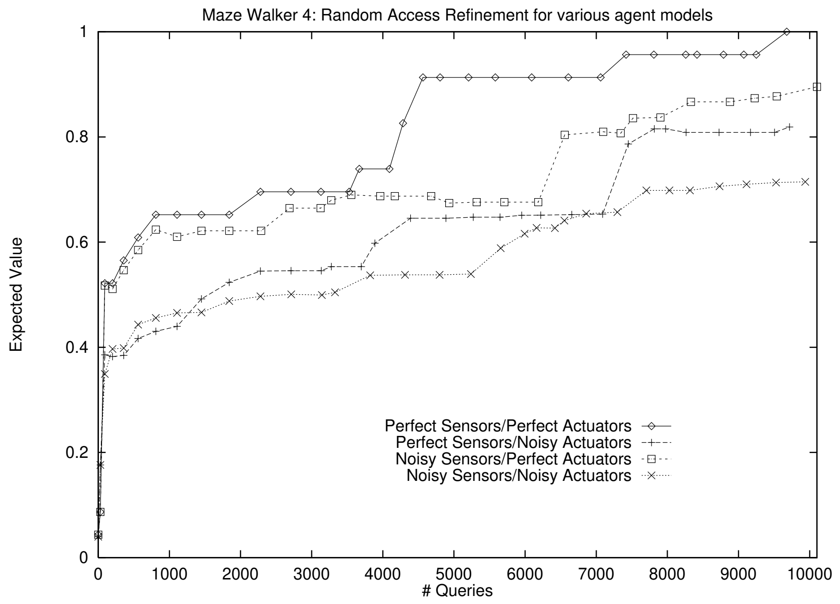
B.2 Second Best Action/Maximal Extension







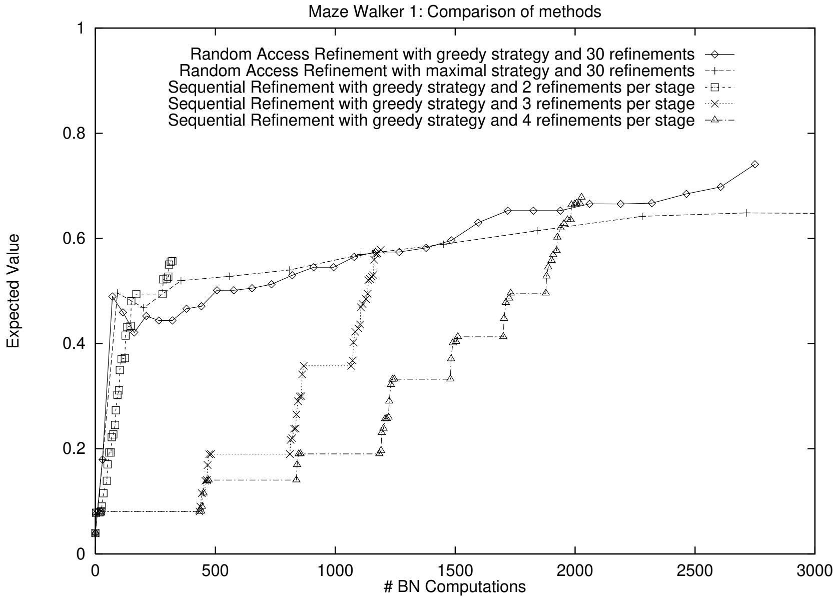
Maze Walker 3: Random Access Refinement for various agent models
Expected Value
Queries
Perfect Sensors/Perfect Actuators
Perfect Sensors/Noisy Actuators
Noisy Sensors/Perfect Actuators
Noisy Sensors/Noisy Actuators
0
0.2
0.4
0.6
0.8
1
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

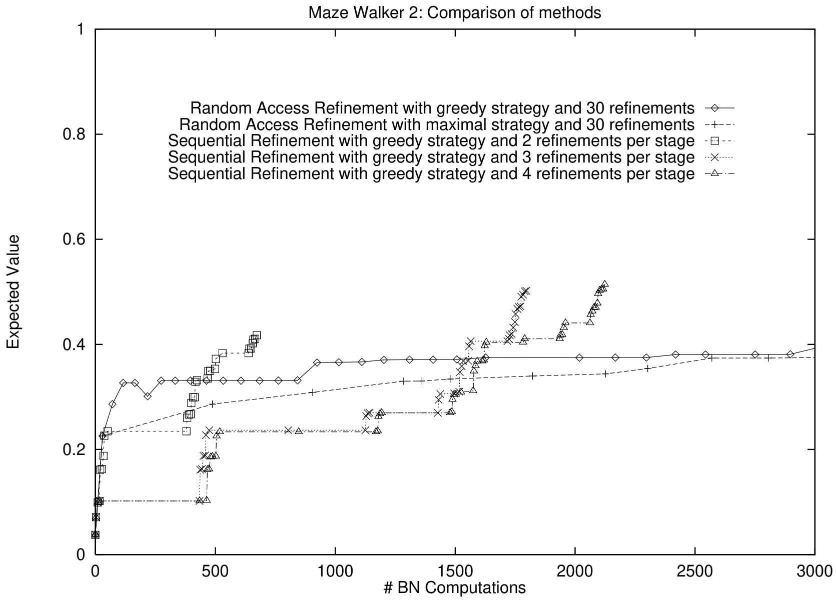


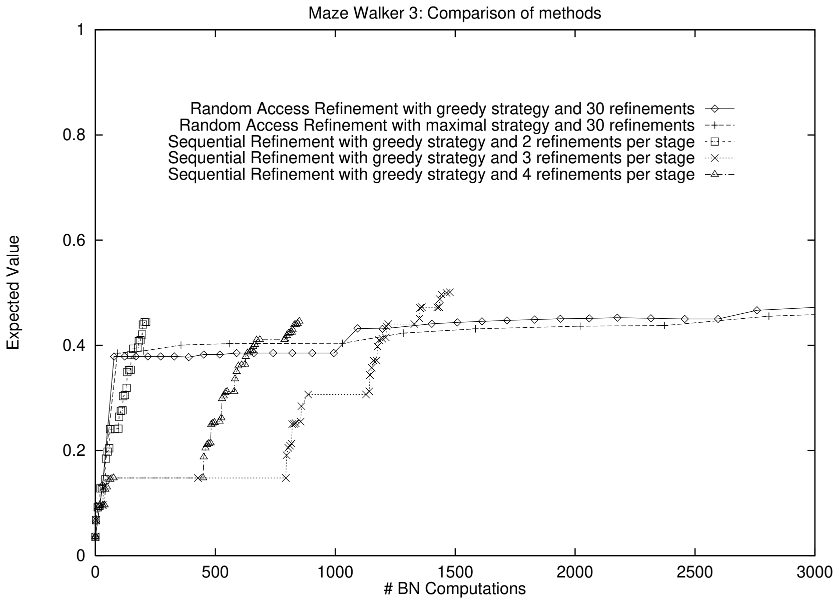
Maze Walker 4: Random Access Refinement for various agent models
Expected Value
Queries
Perfect Sensors/Perfect Actuators
Perfect Sensors/Noisy Actuators
Noisy Sensors/Perfect Actuators
Noisy Sensors/Noisy Actuators
0
0.2
0.4
0.6
0.8
1
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

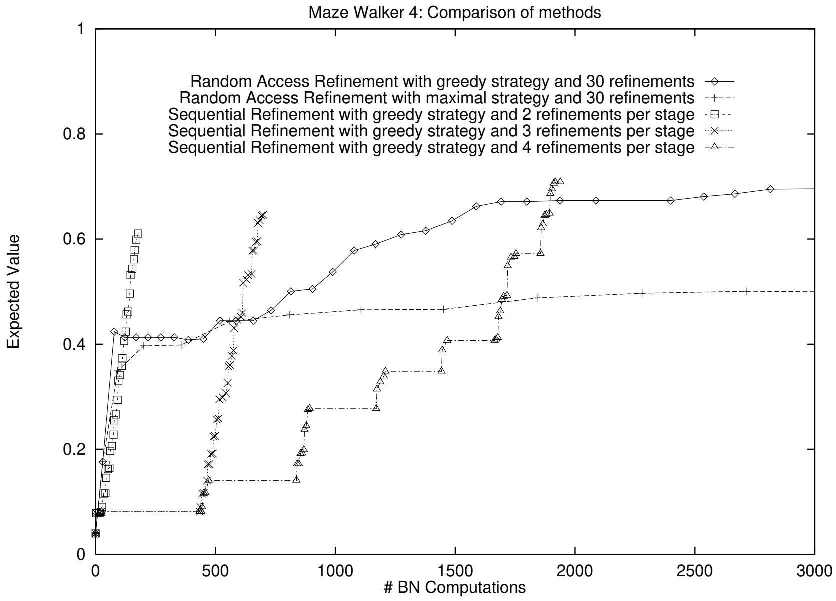
B.3 A comparison of methods

These graphs make a direct comparison for the two algorithms proposed for multi-stage influence diagrams. Each graph shows five data sets, one for each of the combinations reported in Chapter 4. Each graph corresponds to one of the mazes, and the agent model used here is the one with noisy sensors and noisy actuators.




Maze Walker 2: Comparison of methods
Random Access Refinement with greedy strategy and 30 refinements
Random Access Refinement with maximal strategy and 30 refinements
Sequential Refinement with greedy strategy and 2 refinements per stage
Sequential Refinement with greedy strategy and 3 refinements per stage
Sequential Refinement with greedy strategy and 4 refinements per stage
Expected Value
BN Computations
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
3000


Maze Walker 3: Comparison of methods
Random Access Refinement with greedy strategy and 30 refinements
Random Access Refinement with maximal strategy and 30 refinements
Sequential Refinement with greedy strategy and 2 refinements per stage
Sequential Refinement with greedy strategy and 3 refinements per stage
Sequential Refinement with greedy strategy and 4 refinements per stage
Expected Value
BN Computations
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
3000


Maze Walker 4: Comparison of methods
Random Access Refinement with greedy strategy and 30 refinements
Random Access Refinement with maximal strategy and 30 refinements
Sequential Refinement with greedy strategy and 2 refinements per stage
Sequential Refinement with greedy strategy and 3 refinements per stage
Sequential Refinement with greedy strategy and 4 refinements per stage
Expected Value
BN Computations
0
0.2
0.4
0.6
0.8
1
500
1000
1500
2000
2500
3000

Bibliography

[1] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, 1976.

[2] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[3] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

[4] R. Chavez and G. Cooper. An empirical evaluation of a randomized algorithm for probabilistic inference. In *Proceedings of the Fifth Conference on Uncertainty in Artificial Intelligence*, pages 60–70, 1989.

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

[6] Paul Dagum and Michael Luby. Approximating proabilistic inference in Bayesian belief networks is NP–hard (research note). *Artificial Intelligence*, 60(1):141–153, 1993.

[7] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[8] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Deliberation scheduling for time–critical sequential decision making. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 309–316, 1993.

[9] Richard Dearden and Craig Boutilier. Integrating planning and execution in stochastic domains. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 162–169, 1994.

[10] Marek J. Druzdzel. Some properties of joint probability distributions. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 187–194, 1994.

[11] I. J. Good. Twenty-seven principles of rationality. In V. P. Godambe and D.A. Sprott, editors, *Foundations of Statistical Inference*, pages 108–141. Holt,Rinehart,Winston, Toronto, 1972.

[12] 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.

[13] D. E. Heckerman, E. J. Horvitz, and B. N. Nathwani. Update on the pathfinder project. In *Proceedings of the Thirteenth Symposium on Computer Applications in Medical Care*, pages 203–207, 1989.

[14] Max Henrion. Propagation of uncertainty by probabilistic logic sampling in Bayes' networks. In *Uncertainty in Artificial Intelligence 2*, pages 149–164. North–Holland, 1988.

[15] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63–91, 1993.

[16] Michael C. Horsch and David Poole. Flexible policy construction by information refinement. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 315–324, 1996.

[17] Michael C. Horsch and David Poole. An anytime algorithm for decision making under uncertainty. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 246–255, 1998.

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

[19] Eric J. Horvitz and Adrian C. Klein. Utility–based abstraction and categorization. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 128–135, 1993.

[20] R.A. Howard. The used car buyer problem. In *Readings on the Principles and Applications of Decision Analysis*. 1984.

[21] R.A. Howard and J.E. Matheson, editors. *Readings on the Principles and Applications of Decision Analysis*. Strategic Decisions Group, CA, 1984.

[22] 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.

[23] 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.

[24] Jin H. Kim and Judea Pearl. A computational model for causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 190–193, 1983.

[25] 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.

[26] 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.

[27] Zhaoyu Li and Bruce D'Ambrosio. Efficient inference in Bayes networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 11:55–81, 1994.

[28] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370, 1995.

[29] James E. Matheson. Using influence diagrams to value information and control. In *Infuence Diagrams, Belief Nets and Decision Analysis*, pages 25–63. John Wiley & Sons, 1990.

[30] Andrew Kachites McCallum. Learning to use selective attention and short-term memory in sequential tasks. In *From Animals to Animats: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour*, 1996.

[31] Judea Pearl. Fusion, propagation and structuring in belief networks. *Artificial Intelligence*, 29(3):241–288, 1986.

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

[33] Kim Leng Poh and Eric J. Horvitz. Reasoning about the value of decision–model refinement: Methods and application. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 174–182, 1993.

[34] David Poole. Probabilistic conflicts in a search algorithm for estimating posterior probabilities in Bayesian networks. *Artificial Intelligence*, 88:69–100, 1996.

[35] Martin L. Puterman. Dynamic programming. In *Encyclopedia of Physical Science and Technology*, volume 4, pages 438–463. Academic Press, Inc., 1987.

[36] Runping Qi. *Decision Graphs: Algorithms and Applications to Influence Diagram Evaluation and high-Level Path Planning Under Uncertainty*. PhD thesis, Department of Computer Science, University of British Columbia, 1994. Technical Report 94-27.

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

[38] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[39] J.R. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann Publishers, Los Altos, CA, 1992.

[40] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices under Uncertainty*. Random House, New York, 1968.

[41] Stuart Russell and Devika Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2:575–609, 1995.

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

[43] Leonard J. Savage. *The Foundations of Statistics*. Dover Publications, Inc., 1972.

[44] 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.

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

[46] Ross D. Shachter, Stig. K. Anderson, and Peter Szolovits. Global conditioning for probabilitistic inference in belief networks. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 514–522, 1994.

[47] Jonathan King Tash. *Decision Theory Made Tractible: The Value of Deliberation with Applications to Markov Decision Process Planning*. PhD thesis, University of California, Berkely, 1996.

[48] J. von Neuman and O. Morgenstern. *The Theory of Games and Economic Behaviour*. Princeton University Press, 1947.

[49] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[50] C.C. White III and W.T Scherer. Finite-memory suboptimal design for partially observable Markov decision processes. *Operations Research*, 42(3):440–455, 1994.

[51] Nevin Lianwen Zhang and Brent Boerlage. Information filtering for planning in partially observable stochastic domains. Technical Report HKUST-CS94-36, Department of Computer Science, Hong Kong University of Science and Technology, 1995.

[52] Nevin Lianwen Zhang and David Poole. A simple approach to Bayesian network computations. In *Proceedings of the Tenth Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.

[53] Nevin Lianwen Zhang and David Poole. Exploiting Causal Independence in Bayesian Network Inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.