DECISION GRAPHS:
Algorithms and Applications to Influence Diagram Evaluation and High-Level Path Planning Under Uncertainty

by
Runping Qi

Technical Report 94-27
October 1994

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

email: qi@cs.ubc.ca

DECISION GRAPHS:
Algorithms and Applications to Influence Diagram Evaluation and High-Level Path Planning Under Uncertainty

By

Runping Qi

B.Sc. Inner Mongolia University, China, 1982
M.Sc. Changsha Institute of Technology, China, 1986

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

April 1994

Abstract

Decision making under uncertainty has been an active research topic in decision theory, operations research and Artificial Intelligence. The main objective of this thesis is to develop a uniform approach to the computational issues of decision making under uncertainty. Towards this objective, *decision graphs* have been proposed as an intermediate representation for decision making problems, and a number of search algorithms have been developed for evaluating decision graphs. These algorithms are readily applicable to decision problems given in the form of decision trees and in the form of finite stage Markov decision processes.

In order to apply these algorithms to decision problems given in the form of influence diagrams, a stochastic dynamic programming formulation of influence diagram evaluation has been developed and a method to systematically transform a decision making problem from an influence diagram representation to a decision graph representation is presented. Through this transformation, a decision making problem represented as an influence diagram can be solved by applying the decision graph search algorithms. One of the advantages of our method for influence diagram evaluation is its ability to exploit asymmetry in decision problems, which can result in exponential savings in computation.

Some problems that can be viewed as decision problems under uncertainty, but are given neither in the form of Markov decision processes, nor in the form of influence diagrams, can also be transformed into decision graphs, though this transformation is likely to be problem-specific. One problem of this kind, namely high level navigation in uncertain environments, has been studied in detail. As a result of this case study, a decision theoretic formulation and a class of off-line path planning algorithms for the problem have been developed.

Since the problem of navigation in uncertain environments is of importance in its own right, an on-line path planning algorithm with polynomial time complexity for the problem has also been developed. Initial experiments show that the on-line algorithm can result in satisfactory navigation quality.

Contents

List of Figures

List of Tables

Acknowledgement

I thank Dr. David Poole, my thesis supervisor, for his generous friendship, continuous support and valuable advice in the past six years.

I thank Dr. Jim Little, Dr. Alan Mackworth, Dr. Maurice Queyranne and Dr. Jack Snoeyink for serving on my supervisory committee. Their constructive comments and valuable suggestions contribute a great deal to the quality of my thesis.

I thank Dr. Lianwen Zhang for his friendship and fruitful academic collaboration.

I thank Dr. Walfgang Bibel for his encouragement and help I received during my early days at UBC.

I thank Andrew Csinger for his careful reading of this thesis.

I thank my friends and colleagues in the Computer Science Department who made my stay at UBC more productive and pleasant.

I gratefully acknowledge the financial assistance I received from UBC and NSERC.

I thank my beloved wife, Ying Zhang, for her constant support and love through all these years. I cannot imagine what a poor shape I would be in without her.

I dedicate this thesis to my parents.

Chapter 1

Introduction

1.1 General Background and Thesis Objective

Decision making under uncertainty is one of the primary topics of Bayesian decision theory [25, 83, 102]. In this theory, *expected utility* is used as a normalized unit to measure the degree of desirability of various possible outcomes that can result from taking action in some situation. The prescriptive promise of this theory is that the rational action that should be taken in a situation is the one that maximizes expected utility. Bayesian decision analysis is a methodology for applying decision theory to practical decision problems [79].

Decision making under uncertainty is an active research topic in AI, decision theory and operations research. Many problems, such as diagnostic reasoning [72, 73], planning under uncertainty [29, 31, 39, 104], and path planning in uncertain environments [15, 59, 60, 77] can be abstracted as decision making under uncertainty. See [19] for a good introduction to this subject. Problems of decision making under uncertainty can be presented in various forms, such as decision trees [79], finite–stage Markov decision processes [20, 33], or influence diagrams [35]. Although much research has been carried out to address the computational issues of decision problems in particular forms, most of the algorithms are applicable only to that form and do not lend themselves to decision problems in other forms. For example, much work has been done in the operations research community (see [74] for a comprehensive survey) on the computation of Markov decision processes, but the algorithms developed for Markov decision processes are not applicable to influence diagram evaluation. Similarly, the evaluation of influence diagrams has been an active research topic in the AI community, but the algorithms developed for influence diagram evaluation, with a few exceptions [101, 106], cannot directly be used for solving Markov decision processes.

The major objective of this thesis is to study the computational issues of decision making problems in a uniform way and to develop algorithms that are general enough to apply to decision problems in various forms.

1.2 Our Methodology

To achieve our objective, we propose a simple, abstract intermediate representation for decision making problems, and develop algorithms based on this representation. To make use of these algorithms for solving decision making problems in a particular form, we map the problems into the proposed representation.

The representation we propose in this thesis is *decision graphs*. A decision graph is an AND/OR graph with an *evaluation function*. When a decision graph is used to represent a decision problem, the solution graphs of the decision graph are interpreted as the *policies* of the decision problem, while the evaluation function of the decision graph acts as a *quality measurement function* for the policies. The policies of a decision problem can be measured either in terms of costs or in terms of rewards. Accordingly, the evaluation function of a decision graph can be defined either in the form of *minimization–expectation* or in the form of *maximization–expectation*.

If the evaluation function is defined in the form of minimization–expectation, a solution graph is *optimal* if it minimizes the evaluation function. Similarly, if the evaluation function is defined in the form of maximization–expectation, a solution graph is *optimal* if it maximizes the evaluation function. Given a decision problem represented by a decision graph, we need to compute an optimal solution graph of the decision graph, which corresponds to an optimal policy of the decision problem.

1.3 Thesis Overview

In this thesis, we present a number of search algorithms for computing optimal solution graphs of decision graphs. These algorithms include a *depth–first heuristic–search algorithm*, a *best–first heuristic–search algorithm*, an *anytime algorithm* and two *iterative–deepening depth–first heuristic–search algorithms*. Similar to Ballard's *-minimax search procedures [3], the depth–first heuristic–search algorithm is developed from the alpha–beta algorithm for minimax game tree search [38]. While Ballard emphasizes the generalization of the alpha–beta algorithm to handle chance nodes in minimax trees, in our development we elaborate the pruning techniques so that domain–dependent information can effectively be used to improve search efficiency. Furthermore, we derive an anytime algorithm by integrating the anytime concept [8] into the depth–first heuristic–search algorithm. The best–first heuristic–search algorithm is obtained by modifying the AO* algorithm [50, 62, 68] for AND/OR graphs with additive costs. The iterative–deepening algorithms result from combining the iterative–deepening techniques [40] with the depth–first search techniques.

Developing algorithms for decision graph search is just the first step towards our objectives. To fully achieve our objectives, we need to show that decision problems given in other forms can be transformed into a decision graph representation. We first note that this mapping is trivial for those problems in the form of decision trees, because decision graphs are a generalization of decision trees, allowing for structure sharing. We show that it is also quite straightforward to map a finite–stage Markov decision process into a decision graph representation.

For decision problems given in the form of influence diagrams, we develop a method that transforms such a problem into a decision graph representation. By combining this transformation method with the decision graph search algorithms, we obtain a new method for influence diagram evaluation. Our method is similar to Howard and Matheson's method [35] in the sense that both methods transform an influence diagram into a decision tree (a decision graph in our case) for evaluation. However, our method is more efficient, partly because the decision graph obtained by our method is likely much smaller in size than the decision tree obtained by Howard and Matheson's for the same influence diagram. In comparison with other algorithms reported in the literature [11, 85, 88, 91, 106, 108, 109] for influence diagram evaluation, our method has three unique merits. First, it exploits asymmetry of decision problems. This is significant because (a) most practical decision problems are asymmetric [70] and (b) as it will be shown in Section 7.5, exploiting asymmetry can lead to exponential savings in computation. Second, by using heuristic search techniques, our method provides an explicit mechanism for making use of heuristic information that may be available in a domain–specific form. Finally, by using decision graphs as an intermediate representation, the value of perfect information [53] can be computed more efficiently [110]. In addition, our method, like other recent algorithms [88, 108], also provides a clean interface between influence diagram evaluation and Bayesian net evaluation [69], thus various well–established algorithms (e.g., [47]) for Bayesian net evaluation can be used in influence diagram evaluation.

Some other problems that can be viewed as decision problems with uncertainty but are given neither in the form of Markov decision processes nor in the form of influence diagrams can also be transformed into decision graphs, though this transformation is likely to be problem–specific. We have studied one problem of this kind in detail for two reasons. First, we want to illustrate how to transform a decision making problem of this kind into a decision graph representation to make use of the algorithms developed for decision graph search. Second and more importantly, the problem is of importance in its own right and deserves a thorough study.

The problem we have studied is *high–level navigation in uncertain environments*, where an autonomous agent is asked to reach a particular place in an uncertain environment, and needs to determine a plan for achieving the task. This is a challenging problem in the AI and robotic communities.

The problem of high–level navigation in uncertain environments is an important issue in the study of autonomous agents. In the navigation system of an autonomous agent, a central part is the path planning component, responsible for generating a description of a route that can guide the agent to a desired destination. However, due to the complexity and the uncertain nature of the environment, it is unreasonable to expect the path planning component to have *a priori* knowledge of every relevant detail necessary to generate an executable plan for a given navigation task. Consequently, the path planning component has to appeal to perception components for obtaining information dynamically, and must be able to dynamically elaborate plans based on the information from the perception components. In order to manage the complexity of the path planning task, various kinds of hierarchical structure are commonly employed in most navigation systems [2, 13, 55, 58, 89]. In these systems, a distinction is made between a high–level (global) path planner and a low–level (local) path planner. In the literature, the problem of low-level path planning has been intensively studied. However, considerably less attention has been paid to the problem of high-level path planning. In most navigation systems (e.g., [2, 89]) for autonomous agents, the problem of high-level path planning is usually modeled as a variation of computing the shortest distance path from a distance graph that serves as a representation of the global map. However, a major drawback of this treatment is that uncertainty is not considered in high-level path planning.

In this thesis, we address the problem of high-level path planning and navigation in uncertain environments. We propose *U-graphs*, a natural generalization of distance graphs, as a framework for representing the necessary information about uncertain environments for high-level navigation. In a U-graph, the weight of an edge can be either a constant or a random variable. An edge with a random variable as its weight is called an *uncertain edge*. The distribution of the random variable of an uncertain edge is called the *weight distribution* of the uncertain edge. Like an ordinary edge, an uncertain edge represents a connection between two places denoted by the two incident vertices. The weight distribution of the uncertain edge captures the uncertainty on the traversability of the corresponding connection. The weight distributions of uncertain edges can be dependent or mutually independent.

We use the term *U-graph based navigation* to refer to the process of an agent navigating in an uncertain environment represented by a U-graph. More specifically, an agent is given a U-graph and is asked to reach a goal vertex from a start vertex in the U-graph. We assume information on the actual weight of an uncertain edge can be determined when one of the incident vertices of the edge is first visited, or can be "purchased" at some price. Once revealed, the weight of an uncertain edge is assumed to remain the same.

We give a decision theoretic formalization of the problem of U–graph based navigation. Within this formalization, a U–graph based navigation task is represented as a decision graph. A solution graph of the decision graph for a problem corresponds to a complete navigation plan, covering all of the contingencies possibly encountered during the course of navigation. Thus the path planning task of computing a complete plan that satisfies some optimality criterion is reduced to the problem of searching an optimal solution graph from a decision graph. Because the path planning task is finished once a complete plan is computed, we refer to this paradigm as *off–line* navigation.

Since the problem is of importance in its own right, we also study another navigation paradigm, called *on–line* navigation. In the on–line paradigm, a path planner does not compute a complete plan for a given navigation task. Instead, it acts as a consultant telling the agent what to do in any situation. The planner and the executive act as co–routines. The advantage of the on–line paradigm is that we can develop polynomial time planning algorithms. We have developed an on–line planning algorithm for U–graph based navigation. Although the algorithm cannot guarantee optimal navigation[1], our experimental data show that it results in satisfactory navigation quality.

Polychronopoulos [71] has recently studied a similar problem, called *Stochastic Shortest Distance Problem with Recourse* (SSDPR), which is essentially the same as the U–graph based navigation problem. He has given a dynamic programming algorithm for the problem. Although a more thorough comparison between his algorithm and ours is yet to be carried out, we believe that our algorithm is more practical than his for the following two reasons. First, for a given navigation task, we exclude those

[1]Since the problem of optimal navigation is #P–hard [71], no polynomial time algorithm can guarantee optimal navigation, unless P is equal to #P.

"obviously non–optimal parts" (e.g., loops) from the decision graph representation at the stage of problem formulation, resulting in a smaller search space. Second, in computing an optimal plan, our algorithm uses heuristic search techniques and domain dependent information to increase the computation efficiency. Roughly, Polychronopoulos' algorithm amounts to evaluating the decision graphs in a brute–force way. Polychronopoulos also provides a simple on–line algorithm. We have simulated his on–line algorithm and ours against a few hundred randomly generated U–graphs. The results show that the navigation quality of both on–line algorithms are good and ours is closer to optimal than his.

1.4 Summary of Thesis Contributions

The contributions of this thesis are summarized as follows.

- A number of algorithms for decision graph search.
- A new method for influence diagram evaluation.
- A decision theoretic formalization of the problem of U–graph based navigation, and a general approach to the computation of optimal plans for U–graph based navigation problems.
- A polynomial–time heuristic on–line algorithm for U–graph based navigation.

In a broader context, our work can be considered as a contribution to the cross–fertilization between decision theory and AI techniques. On the one hand, we develop a number of heuristic search algorithms for the problem of decision making under uncertainty. These algorithms can be viewed as an application of AI techniques to decision analysis. On the other hand, we develop a decision theoretic formalization for a (path) planning problem. This can be viewed as an application of decision theory to planning. In the literature, much work has been reported on the applications of decision theory to AI problems. A common characteristic of these applications is that a given problem is viewed as a decision making problem and one is interested in a "good" or optimal solution, instead of an arbitrary solution, to the problem. These applications include general planning [9, 16, 29, 32, 46], diagnostic reasoning [72, 73, 98, 99], reactive systems [24, 28] and domain specific problem solving such as the monkey and bananas problem [23], the blocks world [39], and navigation problems [15, 59, 60]. Our work on U–graph based navigation belongs to the last group, applying decision theory to the domain of autonomous navigation.

1.5 Thesis Organization

The rest of this thesis consists of three parts and is organized as follows. Chapters 2 and 3 constitute the first part. In Chapter 2, the concept of decision graphs is introduced and the correspondence between decision graphs and finite–stage Markov decision processes is discussed. In Chapter 3, algorithms for decision graph search are developed.

Chapters 4 through 8 form the second part. In Chapter 4, we review some basic concepts about decision analysis, informally introduce influence diagrams, discuss the advantages and disadvantages of influence diagrams and outline our approach to handling the disadvantages. In Chapter 5, we formally introduce influence diagrams and give a formal definition of the problem of influence diagram evaluation. In Chapter 6, we review some current algorithms for influence diagram evaluation and point out their shortcomings. In Chapter 7, we present our method for influence diagram evaluation. We also show that, by exploiting asymmetry, the method leads to exponential savings in computation for typical decision problems represented by influence diagrams. In Chapter 8, we generalize the method to deal with influence diagrams with multiple value nodes.

Chapters 9 through 11 constitute the third part of the thesis, dealing with various issues of high-level navigation in uncertain environments. In Chapter 9, we first present our motivation for our study on high-level navigation and then propose U-graphs as a framework for representing uncertain environments. Next, we define the problem of U-graph based navigation. Finally, we review some related work in the area of navigation under uncertainty. In Chapter 10, we develop a decision theoretic formalization that transforms a U-graph based navigation problem into a decision graph. In Chapter 11, we discuss two navigation paradigms. We show that the algorithms developed in Chapter 3 for decision graph search can be used for off-line navigation, and present an algorithm for on-line navigation. We give some experimental data on the performance of the algorithms. Conclusions are given in Chapter 12.

Part I

ALGORITHMS FOR DECISION GRAPH SEARCH

Chapter 2

Decision Graphs and Finite–Stage Markov Decision Processes

In this chapter, we introduce the basics of decision graphs and finite–stage Markov decision processes, and show that a finite–stage Markov decision process can be naturally represented as a decision graph.

2.1 Decision Graphs

From the structural point of view, a *decision graph* is an acyclic AND/OR graph [68] with an evaluation function. More precisely, a decision graph is a directed acyclic graph whose nodes are classified into two types: *choice nodes* and *chance nodes*, which are analogous respectively to the OR nodes and the AND nodes in an AND/OR graph. Each decision graph has exactly one *root*. For simplicity of exposition, we assume that all children of a node in a decision graph are of the same type, and chance nodes and choice nodes are interleaved in a decision graph. A cost or a reward is associated with each arc emanating from a choice node. A probability is associated with each arc emanating from a chance node, and the probabilities of all the arcs from a chance node sum to unity. Leaf nodes are terminals, each with a cost or a reward.

Decision graphs are a generalization of decision trees [79, 78], allowing for structure sharing. A *solution graph* SG, with respect to a node n, of a decision graph DG is a graph with the following characteristics:

1. n is in SG;

2. if a non–terminal chance node of DG is in SG, then all of its children are in SG;

3. if a non–terminal choice node of DG is in SG, then exactly one of its children is in SG.

A solution graph with respect to the root of a decision graph is simply referred to as a solution graph of the decision graph.

A decision graph can be interpreted as a representation of a process of sequential decision making. Nodes in a decision graph represent situations. The root node represents the initial situation. A choice node represents a situation where an agent can select an action. The arcs emanating from a choice node can be viewed as the actions that the agent can take in the situation. A chance node represents an uncertain situation, resulting from taking an action in the situation represented by a choice node. The children of a chance node represent the situations that are possibly reached from the uncertain situation represented by the chance node. The probability that a particular situation will be reached is given by the number associated with the arc to the child representing the situation.

The value of a given decision graph can be defined either in terms of costs or in terms of rewards. If the value is defined in terms of costs, the decision objective is assumed to be minimizing the expected cost. If the value is defined in terms of rewards, the decision objective is assumed to be maximizing the expected reward. In the rest of this chapter and in the next chapter, we define value of decision graphs in terms of costs. However, due to the duality of costs and rewards, all the techniques and algorithms that we develop can be adjusted in a straightforward way to be applicable to decision graphs with reward–oriented values.

Let n' be one of the children of node n. We use $c(n, n')$ to denote the cost of the arc from n to n', if n is a choice node; we use $p(n, n')$ to denote the probability of the arc from n to n', if n is a chance node. We use $v(n)$ to denote the value associated with a terminal node n.

Let DG be a decision graph. A *min–exp evaluation* (in contrast to the minimax evaluation of a minimax tree [102]) of DG is a real–valued function u defined as follows:

1. If n is a terminal: $u(DG, n) = v(n)$.

2. If n is a chance node with k children $n_1, ..., n_k$ in DG:

   $u(DG, n) = \sum_{i=1}^{k} p(n, n_i) * u(DG, n_i)$.

3. If n is a choice node with k children $n_1, ..., n_k$ in DG:

   $u(DG, n) = min_{i=1}^{k}\{c(n, n_i) + u(DG, n_i)\}$.

The concepts of solution graphs and min–exp evaluation are the natural extension of those defined for decision trees in [78]. The value given by $u(DG, n)$ is called the *min–exp* value of the node n. It can be interpreted as the minimal expected cost that an agent is going to pay if it starts a sequential decision process from the situation represented by node n. Note that the above definition is applicable to a solution graph as well since a solution graph is a special decision graph.

For a decision graph DG with evaluation function u, a solution graph SG of DG is *optimal* with respect to the evaluation function if $u(SG, n) = u(DG, n)$ for every node n in SG. A general computational problem related to decision graphs is, for a given decision graph and an evaluation function defined on it, to compute an optimal solution graph (with respect to the evaluation function) of the given decision graph. In the next chapter we will present several algorithms for this problem.

Fig. 2.1 shows a simple decision graph. In the figure, boxes, circles, and dotted-line circles denote the choice nodes, chance nodes, and terminals respectively.

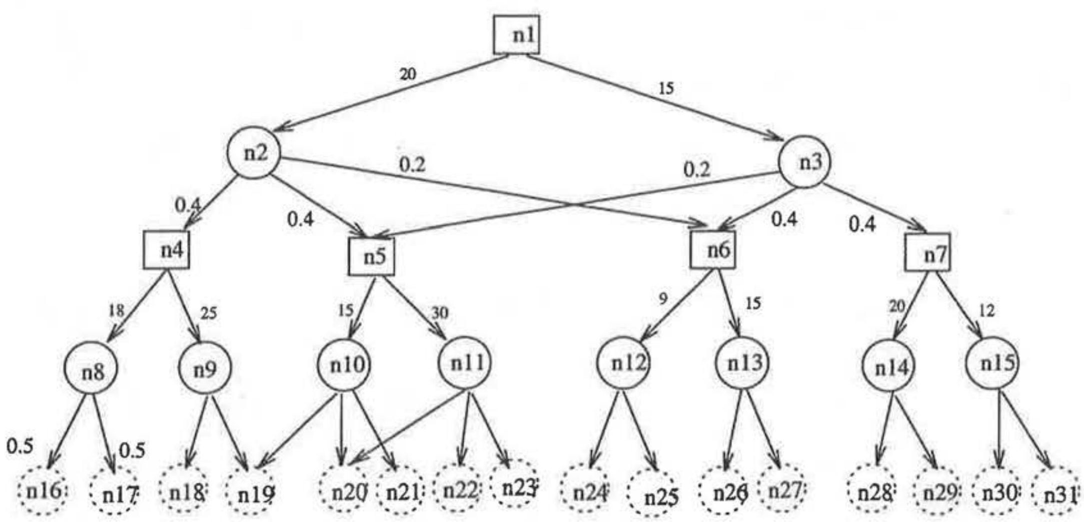

Figure 2.1: A simple decision graph

Lemma 2.1 *For a decision graph DG with a min–exp evaluation function u, we can define another min–exp evaluation function u' that is* isomorphic *to u in the following sense: 1) an optimal solution graph of the decision graph with respect to one evaluation function is also optimal with respect to the other, and 2) there exists a constant c_0 such that $u'(DG, n) = c_0 + u(DG, n)$ for every node n in the decision graph.*

Proof The evaluation function u' can be defined as follows:

1. If n is a terminal: $u'(DG, n) = v(n) + c_0$.

2. If n is a chance node with k children $n_1, ..., n_k$ in DG:

   $u'(DG, n) = \sum_{i=1}^{k} p(n, n_i) * u'(DG, n_i)$

3. If n is a choice node with k children $n_1, ..., n_k$ in DG:

   $u'(DG, n) = min_{i=1}^{k}\{c(n, n_i) + u'(DG, n_i)\}$.

The fact that the two evaluation functions are isomorphic can be proved by a simple induction on the structure of the decision graph. □

Due to this lemma, we can assume that the evaluation functions of decision graphs are all non–negative without loss of generality.

2.2 Finite–Stage Markov Decision Processes

Informally, a Markov decision process is an alternating sequence of states of, and actions on, an evolution system [20]. At each point in time, the state of the system can be observed, and an action can be taken based on the observed state. A finite–stage Markov decision process is a Markov decision process that must reach some target state within a finite number of steps. A *policy* is a prescription for taking action at each point in time.

Formally, a finite–stage Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{K}, w, v, q \rangle$, with $\mathcal{S} = \{S_t | t = 0, ..., T\}$ and $\mathcal{K} = \{K_s | s \in S\}$, where S_t denotes the space of the states observable at stage t, and $S = \cup_{t=0}^{T}$ is the total space of states, $S_i \cap S_j = \phi$ if $i \neq j$; K_s denotes the set of actions that may be taken in state s; w and v are two cost functions and q is a transition function. S_0 contains only one state s_0, called the *initial state*, and the states in S_T are called the *target states*. In this thesis, we consider only Markov decision processes with finite state sets and finite action sets.

If the system is in state $s \in S_t$ for some t, $0 \leq t \leq T$, we say the system is at stage t. The laws of motion of the system are characterized by a transition function q. Whenever the system is in state $s \in S_t$ and action $a \in K_s$ is taken, the probability of the system being in state s' at the next stage is given by $q(s, s', a)$. It is assumed that $\sum_{s' \in S_{t+1}} q(s, s', a) = 1$ for any $s \in S_t$ and $a \in K_s$, and $q(s, s', a) = 0$ if either a is not in K_s or s' is not in S_{t+1}. A cost structure is defined on the decision process by the function w. Whenever the system is in state s and action a is taken, a cost $w(s, a)$ is incurred. Whenever the system enters a target state $s \in S_T$, the system stops with cost $v(s)$.

A deterministic policy for a Markov decision process can be regarded as a function mapping from states to actions. We define the expected cost function X of a finite–stage Markov decision process with respect to a policy R as follows.

$$X(R, s) = \begin{cases} v(s) & \text{if } s \in S_T \\ w(s, R(s)) + \sum_{s' \in S_{t+1}} q(s, s', R(s)) * X(R, s') & \text{otherwise.} \end{cases}$$

The value of $X(R, s)$ is called the expected cost of the Markov decision process with respect to policy R in state s, and the value of $X(R, s_0)$ called the expected cost of the Markov decision process with respect to policy R. A policy R^* is *optimal* if $X(R^*, s_0) \leq X(R, s_0)$ for any policy R.

The finite–stage Markov decision processes that we just introduced are a special kind of general Markov decision processes [20] and are equivalent to the finite–horizon Markov decision processes in [74].

The following lemma expresses the dual results of Theorem 4.2 in [74].

Lemma 2.2 *For each* $s \in S_T$,

$$X(R^*, s) = v(s);$$

for each state $s \in S_t,\ 0 \le t < T$,

$$X(R^*, s) = min_{a \in K_s}\{w(s,a) + \sum_{s' \in S_{t+1}} q(s,s',a) * X(R^*, s')\}$$

and

$$R^*(s) = arg\ \ min_{a \in K_s}\{w(s,a) + \sum_{s \in S_{t+1}} q(s,s',a) * X(R^*, s')\}$$

where the operation arg min selects and returns an element minimizing the formula to follow.

These equations are referred to as Bellman equations in the literature [5].

2.3 Representing Finite–Stage Markov Decision Processes by Decision Graphs

Kumar and Kanal have observed a general correspondence between sequential decision processes and AND/OR graphs [44]. In this section, we show how a finite–stage Markov decision process can be represented as a decision graph. Let $M = \langle \mathcal{S}, \mathcal{K}, w, v, q \rangle$ be a Markov decision process and $DG = \langle V, A_1 \cup A_2 \rangle$ be a directed graph defined as:

$$V = S \cup \{v_{sa} | s \in S; a \in K_s\},$$

$$A_1 = \{\langle s, v_{sa} \rangle | s \in S; a \in K_s\}$$

and

$$A_2 = \{\langle v_{sa}, s' \rangle | s \in S_t, s' \in S_{t+1} \text{ for some } t \text{ with } 0 \le t \le T-1, \text{ and } a \in K_s\}$$

In such a graph, a node $s \in S$ is a *choice node*, representing an observable state in which an action can be selected and taken; a node v_{sa} is a *chance node*, representing a temporary state resulting from taking action a in state s. The next observable state after the temporary state is determined by a probability distribution $q(s, s', a)$. We can attach cost $w(s, a)$ on the arc $\langle s, v_{sa} \rangle$ and probability $q(s, s', a)$ on the arc $\langle v_{sa}, s' \rangle$. In such a graph, node s_0 is the root and the nodes $s \in S_T$ are terminals. It can be verified that DG is acyclic and is indeed a decision graph.

Let u be an evaluation function of the decision graph defined as:

- $u(DG, s) = v(s)$ if $s \in S_T$.
- $u(DG, v_{sa}) = \sum_{s' \in S_{t+1}} p(v_{sa}, s') * u(DG, s')$ for each $s \in S_t$ and each $a \in K_s$, where $p(v_{sa}, s')$ is the probability $q(s, s', a)$ on the arc $\langle v_{sa}, s' \rangle$.
- $u(DG, s) = min_{a \in K_s}\{c(s, v_{sa}) + u(DG, v_{sa})\}$ for each $s \in S - S_T$, where $c(s, v_{sa})$ is the cost $w(s, a)$ on the arc $\langle s, v_{sa} \rangle$.

Lemma 2.3 *For every state $s \in S$,*

$$X(R^*, s) = u(\mathrm{DG}, s).$$

Proof. Note that, from the construction of DG, we have: $c(s, v_{sa}) = w(s, a)$ and $p(v_{sa}, s') = q(s, s', a)$. From the definition of u, we have:

$$u(DG, s) = \begin{cases} v(s) & \text{if } s \in S_T \\ min_{a \in K_s}\{c(s, v_{sa}) + \sum_{s' \in S_{t+1}} p(v_{sa}, s') * u(DG, s') & \text{otherwise.} \end{cases}$$

By Lemma 2.2, we have $X(R^*, s) = u(DG, s)$ for every state $s \in S$. $\square$

Chapter 3

Decision Graph Search

In this chapter, we consider the problem of decision graph search. The problem is defined as follows. *For a given decision graph DG and an evaluation function, find an optimal solution graph SG (with respect to the evaluation function).*

From the definition of the evaluation function of decision graphs, two algorithms are readily available. The first one is the *folding–back–and–averaging–out* method that is commonly used in decision analysis for evaluating decision trees [79]. Another one is a recursive search algorithm. The disadvantage of these algorithms is that they need to "visit" all of the nodes in a decision graph in order to compute an optimal solution.

In this chapter, we present several algorithms for decision graph search. These algorithms need not visit every node of a decision graph in general, and in certain favorable situations, need only to visit the nodes in an optimal solution graph of the decision graph. These algorithms can be roughly classified into three categories. The first category includes a depth–first heuristic–search algorithm and its anytime version. The depth–first heuristic–search algorithm uses a branch–and–bound pruning mechanism similar to the alpha–beta technique for minimax tree search [38]. The second category includes a best–first heuristic–search algorithm, derived from AO*

[50, 62, 68]. This algorithm can also be regarded as a specialization of the general branch-and-bound algorithms described in [44, 45]. The third category includes iterative-deepening depth-first heuristic-search algorithms.

3.1 Depth-First Heuristic-Search Algorithms

In this section we discuss a depth-first heuristic-search algorithm and its variations. Like Ballard's *-minimax algorithm [3], our algorithm is derived from the alpha-beta method [38]. The difference is that our algorithm makes effective use of domain-dependent knowledge to increase search efficiency.

This algorithm was originally developed for searching decision trees with min-exp evaluation functions [78]. Since any decision graph can be viewed as a compact representation of a decision tree, these algorithms are applicable to decision graph search as well. In the rest of this section, we first present this decision tree search algorithm and its anytime version, and then discuss how to tailor the algorithms to exploit shared structures in decision graphs.

3.1.1 Algorithm DFS

In this section, we present a depth-first heuristic-search algorithm *DFS* (Depth First Search) for decision graph search. The algorithm uses an alpha-beta-like pruning mechanism.

In order to develop the pruning mechanism, we contrast a decision tree and a minimax tree. A choice node in a decision tree can be regarded as a *min* node since we want to minimize the min-exp value of it. Consequently, a chance node is analogous to a *max* node. However, a decision tree is different from a minimax tree in two major aspects. First, there is no cost or other information associated with the arcs of a minimax tree, but in a decision tree the information of this kind plays an important role in computing both the min–exp values of nodes and an optimal solution tree of the decision tree. Second and more importantly, the way to compute the minimax values in a minimax tree is different from the way to compute the min–exp values in a decision tree. These two differences make the original alpha–beta pruning rules not directly applicable to a decision tree.

Nevertheless, we still can design a similar pruning mechanism for decision trees if some *admissible heuristic functions* are available. A heuristic function for a decision tree is a function that estimates the min–exp values of the nodes of the decision tree. A heuristic function h for decision tree DG is admissible if, for every node n in DG, $h(n) \leq u(DG, n)$. For the sake of brevity, we use $h^*(n)$ to denote $u(DG, n)$, the min–exp value of the node n in the decision graph DG, when no ambiguity arises.

DFS works as follows. For each node n to be searched next with a given upper bound b, DFS tries to find out whether $h^*(n)$ is less than b. DFS returns the value of $h^*(n)$ if $h^*(n)$ is less than b, otherwise returns a value no less than b. Using the terminology in [38], we call the upper bound the "β–value" of node n.

Let h be an admissible heuristic function for DG, and b be the β–value of node n. We have the following three cases.

(1) $b \leq h(n)$. In this case, due to the admissibility of h, we have $b \leq h^*(n)$. Thus DFS need not search node n (and the subtree rooted at n).

(2) $b > h(n)$ and n is a choice node with children $n_1, ..., n_k$. In this case DFS searches the subtrees rooted at $n_1, ..., n_k$. Let

$$r_0 = b \quad \text{and} \quad r_i = min\{r_{i-1}, c(n, n_i) + h^*(n_i)\} \quad \text{for } i = 1, ..., k.$$

Here, $c(n, n_i)$ is the cost of the arc from n to n_i and r_{i-1} stands for the lowest cost obtained when the subtrees rooted at $n_1, ..., n_{i-1}$ have been searched[1] and the subtree rooted at n_i is to be searched next. We call r_{i-1} an *intermediate back-value* of node n. When searching node n_i, DFS tries to find out whether $h^*(n_i)$ is less than $r_{i-1} - c(n, n_i)$. Thus, DFS recursively applies to node n_i with $r_{i-1} - c(n, n_i)$ as the β-value. After all of the subtrees under node n have been searched, DFS returns r_k which is equal to $h^*(n)$ if $h^*(n) < b$, and is otherwise equal to b.

(3) $b > h(n)$ and n is a chance node. In this case, a series of approximations of $h^*(n)$ can be obtained as the children of n are searched. Let

$$partial_i = \sum_{j=1}^{i} h^*(n_j) * p(n, n_j) + \sum_{j=i+1}^{k} h(n_j) * p(n, n_j)$$

where $p(n, n_i)$ is the probability of the arc from n to n_i, for $i = 1, ..., k$. $partial_i$ can be considered as the approximation of $h^*(n)$ when the subtrees rooted at nodes $n_1, ..., n_i$ have been searched. From the definition of $partial_i$, we have:

$$partial_0 = \sum_{j=1}^{k} h(n_j) * p(n, n_j),$$

$$partial_i = partial_{i-1} + p(n, n_i) * (h^*(n_i) - h(n_i))$$

and

$$partial_k = \sum_{j=1}^{k} h^*(n_j) * p(n, n_j) = h^*(n).$$

Since h is admissible, $partial_{i-1} \leq partial_i$ for any i, $1 \leq i \leq k$. Thus, if for some i, $1 \leq i \leq k$, $partial_i \geq b$, then, $h^*(n) \geq b$. In this situation, DFS will stop searching the rest of the children of node n. When searching node n_i, DFS tries to find out whether

$$h^*(n_i) < (b - partial_{i-1})/p(n, n_i) + h(n_i).$$

[1]Note that the min–exp values $h^*(n_i)$ probably need not be computed for all n_i.

Thus, DFS recursively applies to node n_i with $(b - partial_{i-1})/p(n, n_i) + h(n_i)$ as the β–value. If the subtree rooted at n_k is eventually searched and $partial_k < b$, DFS returns $partial_k$ as the value of $h^*(n)$.

The pseudo code of DFS is shown in Fig. 3.1. In this algorithm, `MAXNUM` is a large positive number, representing ∞; `cost(n, i)` and `prob(n, i)` correspond to $c(n, n_i)$ and $p(n, n_i)$ respectively. Function `h` corresponds to an admissible heuristic function, and `order-d` and `order-n` correspond to two *tree ordering* functions that order the children of choice nodes and those of chance nodes respectively. These three functions are the abstraction of the domain–dependent knowledge that DFS uses.

DFS consists of two mutually recursive functions: `dnode(n, b)` and `nnode(n, b)`, for choice node search and chance node search respectively. In the algorithm, parameter `b` is the β–value for node n; variable `nb` is the β–value for the child to be searched next. In `dnode`, variable `result` denotes the intermediate back–up values of node n (corresponding to r_i). As the children of node n are being searched, variable `result` is updated, and the β–value (`nb`) for the child to be searched next is computed. If the β–value for a child is no more than the value given by the heuristic function, the child need not be searched. In `nnode`, variable `partial` represents the series of approximations of the min–exp value of node. As the children of node n are being searched, variable `partial` is updated and the β–value (`nb`) for the child to be searched next is computed. It is important to note here that `partial` will never decrease as more children of a chance node are searched, due to the admissibility of the heuristic function. Therefore, as soon as `partial` catches up with `b`, it is surely known that the min–exp value of the chance node is equal to or greater than the β–value. Thus no more children need to be searched.

The description of DFS given in Fig. 3.1 does not construct the optimal solution

```
dnode(n, b)
   if n is a terminal then
      if v(n) >= b then return MAXNUM;
                   else return v(n);
   if h(n) >= b then return MAXNUM;
   result = b;
   k = # of children of node n;
   let n1, n2, ..., nk = order-d(n);
   for (i = 1 to k) do
       nb = result -  cost(n, i);
       if nb > h(ni) then
          result = min {result; cost(n, i) + nnode(ni,nb)};
   if result >= b then return MAXNUM;
                  else return result;

nnode(n, b)
   if n is a terminal then
        if v(n) >= b then return MAXNUM;
                     else return v(n);
   if h(n) >= b then return MAXNUM;

   k = # of children of node n;
   let n1, n2, ..., nj = order-n(n);
   partial = h(n1)* prob(n, 1) + ... + h(nk) * prob(n, k);
   i = 0;
   while (partial < b) and (i < k) do
     i = i + 1;
     nb = (b - partial)/prob(n, i) + h(ni);
     partial=partial+prob(n, i)*(dnode(ni, nb)-h(ni));
   if partial >= b then return MAXNUM;
                   else return partial;
```

Figure 3.1: The pseudo code of DFS

tree. The algorithm can be easily modified to do so. Thus, for a decision tree DG and a node n, DFS can be used either to compute $h^*(n)$ or to compute an optimal subtree rooted at n in DG. Since DFS is a depth–first search algorithm, the size of the space that it needs is linear in the depth of the tree if the solution tree need not be constructed explicitly, and is otherwise linear in the size of the largest solution tree that the algorithm ever constructs in the search course.

As an illustration of this algorithm, let us consider an example. For convenience, we assume that, for a given decision tree, the algorithm orders the tree first (using its tree ordering functions) and then searches the ordered tree in the left–to–right order[2] (in contrast to integrating searching and tree ordering together). A decision tree, after being ordered by the tree ordering functions used by DFS, is shown in Fig. 3.2 where we assume that all the terminals have value 10 and all the branches of any chance node have the same probability (0.5). The optimal solution is indicated by the arrows. Its cost is 35.5. Suppose that DFS uses a heuristic function h_0 that returns zero for *every* node in the tree. Clearly, this heuristic function is admissible. The search algorithm starts from the root with ∞ as the β–value. After node n_8 is searched, the intermediate result for node n_4 is 28. Thus when node n_9 is being explored, its β–value is 3. After node n_{18} is explored, the approximation of the min–exp value of node n_9 is 5 which exceeds its β–value, thus node n_{19} is pruned[3]. Another cutoff happens right after node n_{10} is explored. The intermediate back–up value of node n_5 is 25. The β–value for node n_{11} is negative, therefore, node n_{11}, together with all the nodes below it, is pruned. The last pruned node for this problem is node n_{27}. Therefore, five nodes in total are pruned.

[2]This convention is used throughout this section.

[3]Pruning a node means that the node need not be visited at all. In the current case, even if node n_{19} is not a terminal, but is an interior node, it will still be pruned.

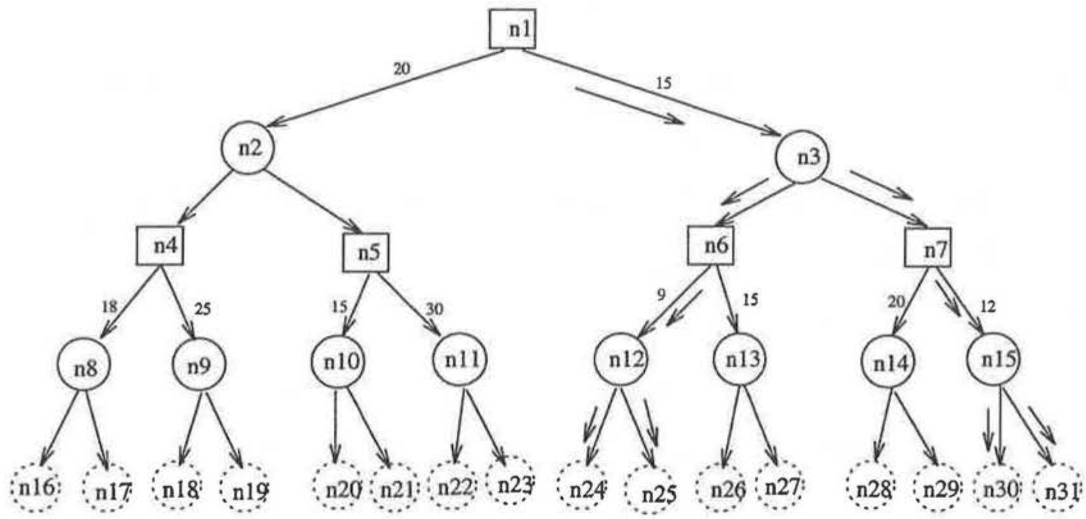

Figure 3.2: An illustration of the search algorithm

Let dt be a function defined as:

$$dt(n, b) = \begin{cases} nnode(n, b) & \text{if } n \text{ is a chance node,} \\ dnode(n, b) & \text{otherwise.} \end{cases}$$

The following theorem establishes the correctness of DFS.

Theorem 3.1 *If the heuristic function used by DFS is admissible, then:*

$$dt(n, b) = \begin{cases} h^*(n) & \text{if } h^*(n) < b, \\ \texttt{MAXNUM} & \text{otherwise} \end{cases}$$

for any node n in the decision tree and a number b.

An inductive proof of this theorem is given in Section 3.5.

Corollary 3.2 *If the heuristic function used by DFS is admissible, then $dt(n, b') \leq dt(n, b)$ for any node n in a decision tree and any two numbers $b' \geq b$. Furthermore, if $dt(n, b) < b$, then $dt(n, b) = dt(n, b')$.*

Corollary 3.3 *If the heuristic function used by DFS is admissible, then $h^*(n) = dt(n, \infty)$ for any node n in the decision tree.*

3.1.2 The effect of heuristic functions

Let h_1 and h_2 be two heuristic functions, h_1 is *more informed* than h_2 for a decision tree if $h_1(n) \geq h_2(n)$ for every node n in the decision tree. Suppose that both heuristic functions h_1 and h_2 are admissible, it is clear that the performance of DFS with h_1 will be no worse than that of DFS with h_2 for the same decision tree if h_1 is more informed than h_2.

As an illustration on the effect of the heuristic function, let us assume that for the decision tree shown in Fig. 3.2, we now have a more informed heuristic function h'_0 defined as follows: $h'_0(n_i) = 16$ for $i = 2, ..., 7$ and $h'_0(n_i) = 7$ for $i = 8, ..., 31$. When applying DFS with h'_0 to the decision tree, nine nodes (nodes in the subtree rooted at nodes n_9, n_{11} and n_{13}) will be pruned.

3.1.3 Tree ordering

Note that the correctness of DFS is independent of the tree ordering functions. However, like minimax tree search, the order in which the children of nodes in a decision tree are searched may have a great effect on the execution time of the algorithm. Generally speaking, we want to search first the branch of a choice node that results in the final (minimal) min–exp value of the choice node in the hope that as many other branches as possible can be pruned; and we want to search first the child of a chance node that contributes most to the min–exp value of the chance node in the hope that the partial accumulation can reach the β–value of the node as early as possible.

As an illustration on the effect of tree ordering, let us consider the decision tree shown in Fig. 3.3. This is the same decision tree as the one in Fig. 3.2 except that the orderings of the children of some nodes are different. It can be verified that when DFS with heuristic function h_0 is applied to this tree, nine nodes (nodes n_{27}, n_{29},

n_{19}, and the nodes in the subtrees rooted at nodes n_{10} and n_{11}) will be pruned, but when DFS with heuristic function h_0 is applied to the decision tree shown in 3.2, only five nodes are pruned. Similarly, when DFS with h'_0 is applied to this tree, twenty one nodes (nodes in the subtrees rooted at nodes n_{13}, n_{14}, and n_2) will be pruned, but when DFS with heuristic function h'_0 is applied to the decision tree shown in 3.2, only nine nodes are pruned.

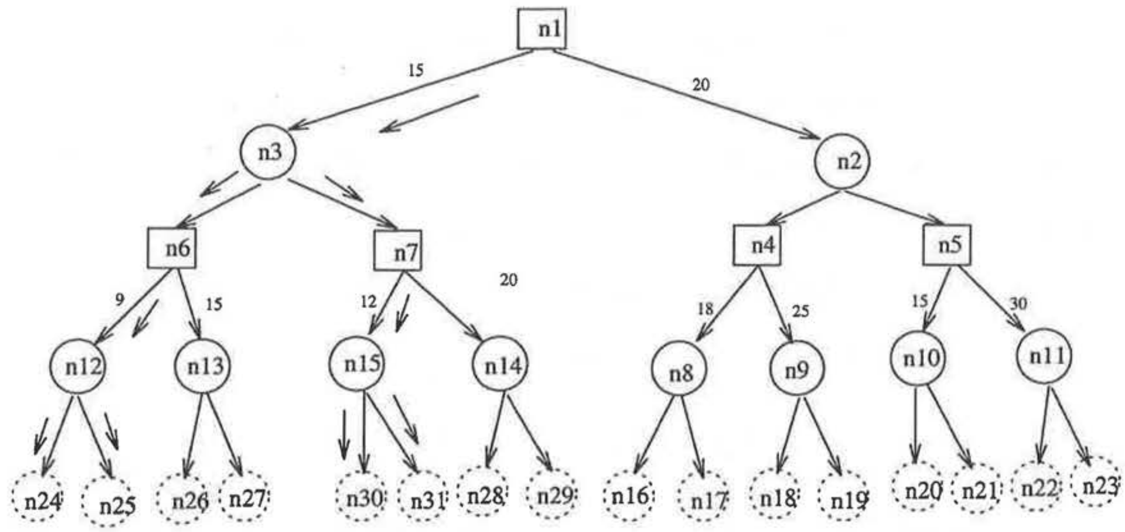

Figure 3.3: An illustration of the effect of tree ordering

Note that a heuristic function normally contains more information than a tree ordering function. In particular, we can define tree ordering functions from a heuristic function. For example, given a heuristic function h, we can define $order_d$ in such a way that if n_i and n_j are two children of a choice node n and

$$h(n_i) + c(n, n_i) \leq h(n_j) + c(n, n_j),$$

then n_i should be searched before n_j. With this definition, child n_j of a choice node

n will be pruned if there exists a child n_i of n, $i < j$, such that:

$$h^*(n_i) + c(n, n_i) \leq h(n_j) + c(n, n_j).$$

Let $e = h(n_j) + c(n, n_j) - (h(n_i) + c(n, n_i))$. The above inequality is equivalent to:

$$h^*(n_i) - h(n_i) \leq e.$$

The left hand side in the above inequality is the difference between the min–exp value of node n_i and its lower bound given by function h, and the right hand side is the difference which determines the search order between n_i and n_j. The more informed the heuristic function is, the smaller $h^*(n_i) - h(n_i)$, thus the better the chance for node n_i being pruned.

We can also define $order_p(n)$ in such a way that if n_i and n_j are two children of a chance node n and

$$h(n_i) * p(n, n_i) \geq h(n_j) * p(n, n_j),$$

then n_i should be chosen before n_j. With this ordering, children $n_{j+1}, ..., n_k$ of a chance node n will all be pruned if

$$\sum_{l=1}^{j} h^*(n_l) * p(n, n_l) + \sum_{l=j+1}^{k} h(n_k) * p(n, n_k) \geq \text{b}.$$

In Chapter 11, when we consider U–graph based navigation problems [77], we will encounter some heuristic functions for a realistic problem. Moreover, we define ordering functions in terms of heuristic functions in the same way as we did above.

3.1.4 Using inadmissible heuristic functions

The previous section required that the heuristic function h be admissible. For the A* algorithm, Harris [30] has argued that the condition of admissibility is too restrictive.

His arguments are applicable to decision tree search as well. Although it may be impractical to find a good heuristic function that never violates the admissibility condition, it is often easier to find a function that estimates the min–exp values well, but occasionally overestimates them. Like the case for A* [68], we have the following two theorems for DFS that establish the linear relationship between the maximal error of an inadmissible heuristic function and the maximal error of the min–exp value of the resulting solution.

Theorem 3.4 *Suppose DFS uses heuristic function h. If there exists a number $\delta \geq 0$ such that h satisfies $h(n) \leq h^*(n) + \delta$ for every node n in a decision tree, then for every node n in the decision tree*

$$h^*(n) + \delta \geq b \quad \textit{if} \quad dt(n,b) \geq b$$

and

$$h^*(n) + \delta \geq dt(n,b) \quad \textit{if} \quad dt(n,b) < b.$$

Theorem 3.5 *Suppose DFS uses heuristic function h. If the costs of all the arcs in a decision tree are non–negative and $h^*(n) \geq 0$ for each node n in the decision tree, and there exists a number $\delta \geq 0$ such that h satisfies:*

$$0 \leq h(n) \leq (1+\delta) * h^*(n) \textit{ for every node } n \textit{ in the decision tree,}$$

then for every node n in the decision tree and any non–negative number b,

$$h^*(n) * (1+\delta) \geq b \quad \textit{if} \quad dt(n,b) \geq b$$

and

$$h^*(n) * (1+\delta) \geq dt(n,b) \quad \textit{if} \quad dt(n,b) < b.$$

The proofs of these theorems are given in Section 3.5.

3.1.5 An anytime version of DFS

The time complexity of searching for an optimal solution tree, or a suboptimal solution tree with a bounded quality, of a decision tree is exponential in the depth of the tree. Thus for a practical problem, it may take a long time to compute such a solution tree. Note that DFS will not return anything before completing the computation of an optimal (suboptimal) solution. However, in some situations, it would be useful if an algorithm could return some (possibly non–optimal) solutions in the course of computing an optimal solution. It would be even better if the quality of those intermediate solutions improves monotonically with the computational time spent by the algorithm. Since an algorithm with this property can give an answer at any time after computing the initial solution, it is called an *anytime algorithm* [8].

We can think of an anytime algorithm as a program which generates a stream of solutions, ordered according their expected costs. For decision tree search, we can easily obtain a naive anytime algorithm from a brute–force procedure and a filter. For a given decision tree, the brute–force procedure systematically enumerates all of the possible solutions of the decision tree, and passes the solution stream to the filter. The filter maintains the minimal cost of the solutions arrived so far and discards solutions with cost no smaller than the minimal cost. Unfortunately, the performance of this algorithm can be bad.

We have developed an algorithm A–DFS (an anytime version of DFS) that incorporates the pruning mechanism we discussed in Section 3.1.1 and at the same time behaves like an anytime algorithm. A–DFS differs from DFS in the following two aspects:

- When searching a choice node, DFS will exhaust all of the children of the choice node and choose the best one. But A–DFS returns the *first* child that results in an admissible solution. Furthermore, A–DFS also sets a backtrack point so that it can continue exploring the remaining children later on.

- In the course of searching a chance node, if DFS finds that `partial` $\geq$ `b`, it reports a cutoff. But in a similar situation, A–DFS requests a backtrack.

Fig. 3.4 shows the pseudo code of A–DFS. In the figure, we have a new procedure `a-search` as the interface of the algorithm. The statement `backtracking` means to continue from the latest backtrack point. If there is no backtrack point, then just return `MAXNUM`. A Prolog version of this algorithm is given in Fig. 3.5.

3.1.6 Exploiting shared structures in decision graphs

A decision graph can be considered as a compact representation of a decision tree in which some subtrees are identical. Conversely, a decision graph can be "expanded" into a decision tree by duplicating those shared nodes in the decision graph.

The algorithms that we presented so far are based on decision trees. These algorithms are also applicable to decision graphs in the sense that they are applicable to the "expanded versions" (the corresponding decision trees) of the decision graphs. An advantage of this treatment is that the algorithms have moderate space requirements[4]. A disadvantage of the treatment is its inability to exploit shared structure of decision graphs. In other words, the algorithms may search a subgraph rooted at a shared node more than once. In order to overcome this disadvantage, we use a "cache technique." When the expected cost of a shared node is obtained, the node along with the

[4]For DFS, the space requirement is linear in the depth of the decision trees if it is not required to construct an optimal solution graph, and is otherwise linear in the size of solution graphs; the space requirement of the anytime algorithm is linear in the size of solution graphs.

```
a_search (n, b)
   if n is a choice node then result = a_dnode(n, b)
                          else result a_nnode1(n, b);
   report(result); backtracking;

a_dnode(n, b)
   if n is a terminal then
      if v(n) >= b then return MAXNUM; else return v(n);
   if h(n) >= b then return MAXNUM;
   result = b;
   k = # of children of n;
   let n1, n2, ..., nk = order_d(n);
   for (i = 1 to k) do
       nb = result -  cost(n, i);
       if nb > h(ni) then
          result1 = cost(n, i) + a_nnode(ni,nb);
       if result1 < result then
          result = result1;
          set a backtrack point; return result;
   return MAXNUM;

a_nnode(n, b)
   if n is a terminal then
        if v(n) >= b then return MAXNUM; else return v(n);
   if h(n) >= b then return MAXNUM;

   k = # of children of N;
   let n1, n2, ..., nk = order_p(n);
   partial = h(n1)* prob(n, 1) + ... + h(nk) * prob(n, k);
   i = 0;
   while (partial < b) and (i < k) do
     i = i + 1; nb = (b - partial)/prob(n, i) + h(ni);
     partial=partial+prob(n, i)*(a_dnode(ni, nb)-h(ni));
   if partial >= b then  backtracking else return partial;
```

Figure 3.4: The pseudo code of A–DFS

```
% search node N for a solution tree R with cost C such that C < B

:-dynamic stop/1.

search(N, B, term(V, N), V) :- terminal(N), v(N, V), V < B.
search(N, B, choice(C, N, R), C) :-  h(N, HV), HV < B,
             choice(N), children(N, L), searchchoice(L, B, R, C).
search(N, B, chance(C, N, R), C) :-  h(N, HV), HV < B, chance(N),
                                     children(N, L), initial(L, C0),
                                     searchchance(L, B, R, C, C0).

%search children list of a choice node
%When find a solution, we can return the solution and update the bound.

searchchoice([(LC, N)|L], B,  R, C) :- NB is B - LC,
                   search(N, NB, R1, C1), assert(stop(L)), C2 is LC + C1,
                   searchchoice_with_s([(LC, N)|L],  R1,  R, C2, C).
searchchoice([_|L], B,  R, C):- \+ stop(L),  searchchoice(L, B, R, C).

searchchoice_with_s([(LC, N)|_],  R,  (LC, R), C, C).
searchchoice_with_s([_|L],  _, R, C1, C):- searchchoice(L, C1, R, C).

%search children list of a chance node

searchchance([], B, [], C0, C0) :- C0 < B.
searchchance([(P, N)|L], B, [(P, R1)|R2], C, C0) :- C0 < B, h(N, HV),
                   NB is (B - C0)/P + HV, search(N, NB, R1, C1),
                   C2 is C0 + P*(C1 - HV), searchchance(L, B, R2, C, C2).

% An auxiliary function

initial([], 0).
initial([(P, N)|L], C) :- h(N, HV), initial(L, C1), C is C1 + P* HV.
```

Figure 3.5: A Prolog version of A–DFS

cost is stored in a cache for later use. When a node is to be searched, the algorithms first check whether the the node has been cached, and will search the node only if the node is not in the cache.

In a similar vein, we can also make use of an "adaptive" heuristic function. That is, whenever a cutoff occurs at a node n, we obtain a new lower bound on the expected cost of the node. If the new bound is greater than $h(n)$, we obtain a new heuristic function h' that agrees with h at all nodes except that $h'(n)$ equals the new lower bound. Obviously, h' is more informed than h. This could be viewed as an example of "learning from failure."

By incorporating this caching technique into DFS, we obtain DFS' as shown in Fig. 3.6. When the expected cost of a node is obtained, DFS' needs to determine whether the expected cost should be cached or not; when a cutoff occurs at a node, DFS' needs to determine whether the heuristic function should be updated accordingly. If the "caching policy" allows no node ever to be cached, the algorithm degenerates to DFS. On the other hand, if the "caching policy" allows the expected costs of all nodes to be cached, the algorithm exploits the shared structure of decision graphs to the maximum degree. However, this may lead to an exponential space requirement, defeating an advantage of DFS. Clearly, there is a tradeoff between time and space in DFS'. It is a "meta-decision" problem to decide which nodes should be cached. Generally speaking, we would like to cache those nodes that are likely to be searched again and that would take much time to search. The searching time of a node can be obtained when it is searched for the first time. The probability that a node will be searched again may be estimated based on domain dependent knowledge. Russell and Wefald's meta-reasoning mechanism [82, 81] can play a role in deciding which nodes' expected costs should be cached.

```
dnode'(n, b)
   if n is a terminal then
      if v(n) >= b then return MAXNUM; else return v(n);
   if n is cached then
      let result be the cached value;
      if result >= b then return MAXNUM else return result;
   if h(n) >= b then return MAXNUM;
   result = b;
   k = # of children of n;
   let n1, n2, ..., nk = order-d(n);
   for (i = 1 to k) do
        nb = result -  cost(n, i);
        if nb > h(ni) then
           result = min {result; cost(n, i) + nnode'(ni,nb)};
   if result >= b then
        {if function h should be updated at n then update it;
         return MAXNUM;}
      else {if n should be cached then cache n; return result;}

nnode'(n, b)
   if n is a terminal then
        if v(n) >= b then return MAXNUM; else return v(n);
   if n is cached then
      let result be the cached value;
      if result >= b then return MAXNUM else return result;
   if h(n) >= b then return MAXNUM;

   k = # of children of N;
   let n1, n2, ..., nk = order-n(n);
   partial = h(n1)* prob(n, 1) + ... + h(nk) * prob(n, k);
   i = 0;
   while (partial < b) and (i < k) do
     i = i + 1;
     nb = (b - partial)/prob(n, i) + h(ni);
     partial=partial+prob(n, i)*(dnode'(ni, nb)-h(ni));
   if partial >= b then
        {if function h should be updated at n then update it;
         return MAXNUM;}
      else {if n should be cached then cache n; return partial;}
```

Figure 3.6: The pseudo code of DFS'

A similar version can be obtained for A–DFS in the same way.

3.2 Applying AO* to Decision Graph Search

The AO* algorithm was first developed in [52] for searching AND/OR graphs with additive costs [52]. The name AO* was coined in [62]. As shown in [45], AO* is applicable to AND/OR graphs with *monotone evaluation functions* [45]. We say a function $g(x_1, ..., x_k, L)$ is *monotone* if $g(y_1, ..., y_k, L) \leq g(x_1, ..., x_k, L)$ provided $y_i \leq x_i$ for $i = 1, ..., k$. An evaluation function u is *monotone* if there is a monotone function g such that $u(n) = g(u(n_1), ..., u(n_k), L)$ for each non-terminal node n, where L represents the local information (in terms of arc costs or arc probabilities for the case of decision graphs) associated with the arcs incident from n. From the definition of the min–exp evaluation function, we can see that the min–exp evaluation function is monotone. Thus AO* is applicable to decision graph search problems as well. In this section, we illustrate how to tailor AO* so that it can apply to decision graph search, and present some results on the resulting algorithm.

We assume that the decision graph to be searched is given in the implicit form. We use h again to denote a heuristic function. The algorithm works on an explicit graph G which initially consists of the root node only and is gradually expanded. During the entire process, G is always a subgraph of the original graph.

A node in G is called a *tip* node if it has no children. A solution graph of the explicit graph is called a *potential solution graph* (psg). A psg is a solution graph of the given graph if all the tip nodes in the psg are terminals.

With the help of the heuristic function h, we define a min–exp function f on the explicit graph. Let n be any node in G, $f(n)$ is defined as follows.

- $f(n) = v(n)$ if n is a terminal.

- $f(n) = h(n)$ if n is a non-terminal tip node.
- $f(n) = min_{i=1}^{k}\{c(n, n_i) + f(n_i)\}$ if n is a choice node with children $n_1, ..., n_k$.
- $f(n) = \sum_{i=1}^{k} p(n, n_i) * f(n_i)$ if n is a chance node with children $n_1, ..., n_k$.

Due to the admissibility of h, the following result is obvious.

Lemma 3.6 *For any node n in G, $f(n) \leq h^*(n)$.*

At any moment, the explicit graph can have a number of potential solution graphs. We use *opsg* to denote an optimal potential solution graph — a potential solution with the lowest cost.

AO* can be intuitively understood as an iteration of two major operations. The first one is the node expansion that finds a non-terminal tip node in the identified optimal potential solution graph and generates the children of the node. The cost of each child is given by the heuristic function, if it is generated for the first time. The second operation is the cost updating operation that, starting from the newly expanded node, updates the costs of the ancestors of the newly expanded node, according to the cost function. In the course of the cost updating, a new optimal potential solution graph is identified. The termination condition for this process is that the optimal potential solution graph has no non-terminal tip node. The basic structure of the algorithm is as follows:

1. Initially, both G and *opsg* consist of only the root node.

2. If all of the tip nodes of *opsg* are terminals, stop and output *opsg* as the solution graph.

3. Select a non-terminal tip node of *opsg*, generate the children of the node and add them to G (if some children are already in G, just share them).

4. Set *opsg* to be an optimal potential solution graph in G.

5. Go to step 2.

The above algorithm can be further refined by using the marking technique of [52]. The following version of AO* is adapted from [10], where h denotes a heuristic function for the given decision graph satisfying $h(n) = v(n)$ for all terminals in the decision graph. In the algorithm, the marked *psg* rooted at the root of the decision graph is the same as the *opsg* in the above algorithm.

Algorithm AO*

1. Initially the explicit graph G and the potential solution graph *psg* consist solely of the root node s. Set

$$\hat{f}(s) \leftarrow h(s).$$

If s is a terminal node, then mark s SOLVED.

2. Repeat the following steps until s is marked SOLVED. Then, exit with $\hat{f}(s)$ as the solution cost.

2.1 Choose a tip node n of the marked *psg* that is not marked SOLVED. For each child n_i of n not already present in the explicit graph G, set

$$\hat{f}(n_i) \leftarrow h(n_i).$$

Mark SOLVED those children of n that are terminals.

2.2 Create a set Z of nodes containing only node n.

2.3 Repeat the following steps until Z is empty.

2.3.1 Remove from Z a node m that has no descendent in Z.

2.3.2 (i) If m is a choice node with children $m_1, ..., m_k$, then set

$$\hat{f}(m) \leftarrow min_{1 \leq i \leq k}\{\hat{f}(m_i) + c(m, m_i)\}.$$

Mark that arc (m, m_{i_0}) for which the above minimum occurs. [Resolve ties arbitrarily, but in favour of a SOLVED node.] Mark m SOLVED if and only if m_{i_0} is marked SOLVED.

(ii) If m is a chance node with children $m_1, ..., m_k$, then set

$$\hat{f}(m) \leftarrow \sum_{1 \leq i \leq k} p(m, m_i) * \hat{f}(m_i).$$

Mark all arcs (m, m_i). Mark m SOLVED if and only if every m_i is marked SOLVED.

2.3.3 If $\hat{f}(m)$ changes value at step 2.3.2 or if m is marked SOLVED, then add to Z all of the immediate predecessors of m.

The only difference between the above algorithm and the AO* algorithm given in [10] for AND/OR graphs with additive costs lies in the way of updating function $\hat{f}$ at step 2.3.2 (ii).

Lemma 3.7 *At any stage during the search process, if a node n is marked SOLVED, a solution graph with cost $\hat{f}(n)$ can be obtained by tracing down the marked arcs from n.*

Proof. By a trivial induction on the stage of the algorithm.

Lemma 3.8 *If there exists some $\epsilon \geq 0$ such that $h(n) \leq h^*(n) + \epsilon$ for every tip node n in the explicit graph G, then at any stage during the search process, we have $\hat{f}(n) \leq h^*(n) + \epsilon$ for all nodes in G.*

Proof. Similar to the proof of Lemma 1 in [52]. We prove the lemma by induction on the stage of the algorithm. The lemma is trivially true initially. Suppose that it is true at a certain stage and let us prove it is true at the next stage, after execution of the body of the outer loop (i.e., steps 2.1 — 2.3).

Since during the execution of the loop body, the $\hat{f}$ values of only those nodes that are ancestors of node n may be changed, let us consider the subgraph, G', of G obtained up to this stage, which consists of all the ancestors of node n. Since G' is acyclic, an index can be attached to each node of G', starting with $n^0 = n$, in such a way that all paths from node n^i to node n^0 contain only n^j with $j < i$.

Now, we prove by induction on the index i that the inequality still holds for each node in G' after its $\hat{f}$ value is updated.

First, we prove that this is true for n^0. Let $n_1, ..., n_k$ be the children of n. For any child, n_l, $1 \leq l \leq k$, if it has been generated before, we have $\hat{f}(n_l) \leq h^*(n_l) + \epsilon$

by the outer induction assumption, and if n_l is generated for the first time, we also have $\hat{f}(n_l) = h(n_l) \leq h^*(n_l) + \epsilon$ by the hypothesis on the heuristic function h.

If n is a choice node, we have:

$$\begin{aligned}\hat{f}(n) &= min_l\{\hat{f}(n_l) + c(n, n_l)\} \\ &\leq min_l\{h^*(n_l) + \epsilon + c(n, n_l)\} \\ &= \epsilon + h^*(n)\end{aligned}$$

Similarly, if n is a chance node, we have:

$$\begin{aligned}\hat{f}(n) &= \textstyle\sum_l\{\hat{f}(n_l) * p(n, n_l)\} \\ &\leq \textstyle\sum_l\{(h^*(n_l) + \epsilon) * p(n, n_l)\} \\ &= \epsilon + h^*(n)\end{aligned}$$

Now, let us assume that the lemma is true for all nodes n^j with $j < i$, then the inequality can be proved true for node n^i by repeating the above argument. Thus, we have proved that the lemma is true after the execution of the loop body. Therefore, the lemma holds by induction. □

Note that Lemma 1 in [52] is actually a special case of Lemma 3.8 above with $\epsilon = 0$. The above lemma will not be true for AND/OR graphs with additive costs, because if $\epsilon > 0$, the error may be accumulated in the search process for additive cost AND/OR graphs.

Theorem 3.9 *The algorithm with heuristic function h satisfying $h(n) \leq h^*(n) + \epsilon$ for every node n that is ever in the explicit graph G, where $\epsilon \geq 0$, will return a solution graph with cost less than or equal to $h^*(s) + \epsilon$, if the algorithm terminates.*

Proof. Follows from Lemma 3.7 and Lemma 3.8 immediately. □

Corollary 3.10 *The algorithm with an admissible heuristic function will return an optimal solution, if it terminates.*

Lemma 3.11 *If $h^*(n) \geq 0$ and there exists some ϵ, $\epsilon \geq 0$, $0 \leq h(n) \leq h^*(n)(1+\epsilon)$, for every node n in the explicit graph G, and the arc costs in the given graph are non–negative, then at any stage during the search process, we have $\hat{f}(n) \leq h^*(n)(1+\epsilon)$ for every node n in G.*

Proof. Similar to that for Lemma 3.8.

Theorem 3.12 *If the arc costs in the given graph are non–negative, then the algorithm with heuristic function h satisfying $0 \leq h(n) \leq h^*(n)(1+\epsilon)$ for every node that is ever in the explicit graph, where $\epsilon \geq 0$, will return a solution graph with cost less than or equal to $h^*(s)(1+\epsilon)$, if the algorithm terminates.*

Proof. Follows from Lemma 3.7 and Lemma 3.11 immediately. □

As the reader may have already noticed, Theorems 3.9 and 3.12 above do not assure that the algorithm always stops even if a *finite* solution graph exists for a given (infinite) decision graph. Thus they are weaker than Theorem 1 in [52]. However, for finite acyclic decision graphs, the algorithm is guaranteed to terminate. This weakness seems to be inevitable in general, and indeed is also shared by the depth first heuristic algorithms that we presented in the previous section, since there does exist some case where a finite optimal solution graph does exist but the algorithm will not terminate. This can be illustrated by the following example. Consider the decision tree in Fig. 3.7. Node A in the decision tree is the root of the graph. Each choice node has two children, and the child on the right side is a terminal with zero cost. The cost of the arc to the left child is 1 and the cost of the arc to the right child is 2. The left child is a chance node. Each chance node has two children, each with 0.5 probability. The subtree below each child of a chance node is isomorphic to the entire decision tree. Thus the decision tree is infinite. It is easy to prove that the min–exp value of this decision tree is 2 and an optimal solution tree consists of only two nodes: the root and its right child. However, if AO* adopts a depth–first left–to–right strategy in selecting the next tip for expansion, the algorithm will not terminate, since the $\hat{f}$ values of the marked potential solution trees will always be less than 2.

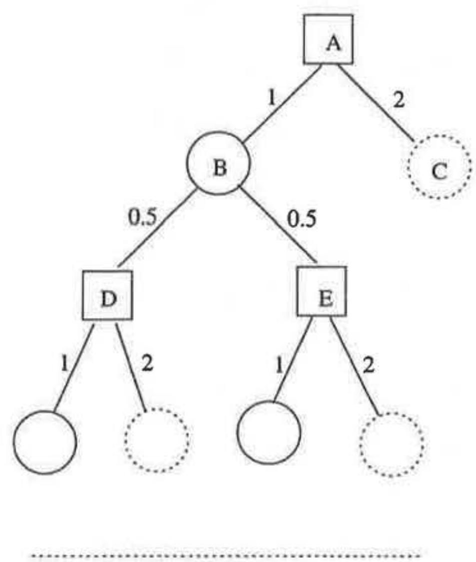

Figure 3.7: An example for which AO* may not terminate

Note that the possibility of non–termination of the algorithm stems from the arbitrariness in the selection of a tip node to be expanded next. In the case of searching into AND/OR graphs with additive costs, no matter which tip node is selected, the expansion of the tip node will increase by a certain amount the costs of the solution graphs consisting of that node, and the increasing amount can be bounded from below. However, in the case of searching into decision graphs, the increasing amount contributed by a tip node expansion can be arbitrarily small.

In order to guarantee the termination of the algorithm when a finite optimal solution graph exists, we need another heuristic function for tip node selection. Here, we propose two heuristics for tip node selection out of termination consideration.

Heuristics 1: using the breadth first strategy in tip node selection. That is, if $t_1, ..., t_k$ are the tip nodes of the marked potential solution graph, the tip node with the smallest depth should be selected for expansion.

Heuristic 2: using a best first strategy in tip node selection. Suppose $t_1, ..., t_k$ are the tip nodes of the marked potential solution graph. The tip node with the largest $P(t_i)$ value should be expanded, where $P(t_i)$ is the product of the probabilities along the path from the root to tip node t_i.

3.3 Iterative Deepening Search

The two kinds of algorithms that we discussed so far for decision graph search are complementary. A major disadvantage of AO* is that it requires exponentially large space. The advantage of AO* is that it will not stick too long to a solution graph which is apparently "bad." On the other hand, in comparison with AO*, the major advantage of the depth first search algorithms is their moderate requirement on space. However, the price for this is that they may search down to a deep layer in a solution graph that is not optimal. Thus it would be nice if we could design algorithms that combine the advantages of AO* and the advantage of the depth search algorithms together. For OR–graph search, the iterative deepening search technique [40] was proposed for such a combination, and was proved asymptotically optimal along the following three dimensions: time complexity, space complexity and the quality of the solution. In this section, we propose two iterative–deepening heuristic–search strategies for decision graph search.

3.3.1 Depth–bounded iterative deepening

The first iterative-deepening search strategy is a *depth–bounded iterative–deepening* strategy. The strategy repeatedly applies DFS to a decision graph, with increasing depth–bounds. Whenever a non–terminal node n on the depth boundary is visited, $h(n)$ is used as its min–exp value. After each iteration, a potential solution graph with the minimum cost is identified. This process terminates when the optimal potential solution graph identified this way is actually a solution graph (all tip nodes in the potential solution graph are terminals).

Unlike iterative-deepening A* [40], our algorithm uses search depth as the cutting off criterion. In this regard, our iterative–deepening strategy is similar to the iterative–deepening depth–first search algorithm (DFID) reported in [40]. However, unlike DFID, the depth–first search in each iteration in our algorithm is a kind of heuristic search. In fact, our algorithm is very much like the iterative–deepening game tree searching algorithms [93, 105].

The following result is obvious.

Theorem 3.13 *The depth–bounded iterative deepening algorithm returns an optimal solution graph if the heuristic function it uses is admissible and if it terminates.*

3.3.2 Cost–bounded iterative deepening

The second iterative deepening search strategy is a cost–bounded iterative deepening strategy, very much like iterative deepening A* (IDA*) [40]. The idea is that successive iterations correspond not to increasing depth of search, but rather to increasing

β–values for the search. The strategy maintains two values: the upper bound b_u and the lower bound b_l on the decision graph and works as follows: Initially, the lower bound b_l is set to the value given by the heuristic function, while the upper bound is set to the min–exp value of a solution graph that can be obtained by identifying an arbitrary solution graph. At each iteration, a new β–value b is set to $b_l * \alpha + b_u * (1-\alpha)$ for some $\alpha \in (0,1)$ and a depth first search (using DFS) with b as the β–value is performed. If a solution with cost less than b is returned, then the solution is an optimal solution, thus the algorithm stops. Otherwise, the lower bound b_l can be set to b. This process continues until either an optimal solution is found or the lower bound and the upper bound become close enough.

Theorem 3.14 *The cost–bounded iterative deepening search algorithm returns an optimal solution graph if the heuristic function it uses is admissible and if it terminates.*

An advantage of this algorithm over DFS is that it is less sensitive to the node ordering in a graph. This can be best illustrated by an example. Suppose the root of the decision tree is a choice node with two children n_1 and n_2. Suppose further that the subtree below n_1 is very large and so is its min–exp value, but the subtree below n_2 is very small and so is its min–exp value. Clearly, node n_1 cannot be in an optimal solution tree. However, if DFS happens to search the subtree below node n_1 first, then it will not come to node n_2 until an optimal solution tree for the subtree below node n_1 is found. This may take a lot of time. On the other hand, the cost–bounded iterative deepening algorithm will not stick to the subtree below node n_1 for too long (because the β–value can be very small).

The cost–bounded iterative deepening algorithm discussed above is analogous to the binary iterative deepening A* [67] in the sense that both the upper bound and the lower bound of the problem are maintained.

3.3.3 Generic iterative deepening

So far, we have discussed two particular iterative deepening techniques. As suggested in [18], an iterative deepening search procedure in general can be divided into two components: one for deciding which portion of the given graph should be searched next, and the other for computing an optimal solution graph in the identified sub-graph. The procedure is a simple loop alternatively calling these two components. In the previous two iterative deepening procedures, these two components are integrated. In general, it is not a trivial issue to decide which portion of the given graph should be searched. For a more detailed discussion, the reader is referred to [17].

3.3.4 Co–routines

It is interesting to note that the cost–bounded iterative deepening algorithm and the anytime algorithm A–DFS can work as co–routines in the following way. For a given problem, A–DFS gradually approaches the optimal value of the decision graph from above, and thus can be used to update the upper bound b_u of the cost–bounded iterative deepening algorithm. The co–routines stop when either algorithm reports finding an optimal solution, or when the lower bound and the upper bound become close enough. In this way, we obtain an anytime algorithm that also uses cost–bounded iterative deepening strategy.

Theorem 3.15 *The co–routines return an optimal solution graph if the heuristic function they use is admissible and if they terminate.*

Finally, we conclude this section with a result on the termination of the algorithms discussed so far.

Theorem 3.16 *All of the algorithms presented in this chapter terminate for finite acyclic decision graphs.*

Proof. A finite acyclic decision graph can be expanded into a decision tree of finite size.

The termination property of DFS and AO* can be proved by an induction on the number of nodes visited by the algorithms.

The termination property of the anytime algorithm A–DFS follows that of DFS and the fact that a decision tree of finite size can have only finite number of solution trees (graphs).

The depth–bounded iterative–deepening algorithm terminates because of two facts: (a) the number of iterations is bounded from above by the depth of the graph, and (b) each iteration just involves a call to DFS, which terminates.

The termination property of the cost–bounded iterative–deepening algorithm can be proved by establishing an upper bound on the number of iterations.

The co–routines terminate because of the termination property of A–DFS and that of the cost–bounded iterative–deepeding algorithm. □

3.4 Summary

In this chapter, we present three classes of algorithms for decision graph search. DFS and and its anytime version A–DFS belong to the first class. They are depth–first search algorithms, derived from the well–known alpha–beta search algorithm [38] for minimax tree search. These algorithms use domain dependent knowledge for increasing search efficiency. The algorithm AO* belongs to the second class, derived directly from the AO* algorithm [68, 62] for searching AND/OR graphs with additive costs. The iterative–deepening algorithms belong to the third class. These algorithms are derived by integrating the iterative–deepening search techniques [40] into the depth–first search algorithm.

Now, a natural question is still to be answered. That is, under what circumstances are these algorithms more appropriate than the folding–back–and–averaging–out method [79] for decision graph search? Our answer to the question is that it depends on the degree of node sharing in the graphs. At one extreme where there is a substantial sharing in the graph such that the number of nodes in the graph is polynomial in the depth of the graph, then the folding–back–and–averaging–out method is more appropriate, since it exhibits polynomial (in the depth of the graph) complexity while the complexity of our search–oriented algorithms is still exponential. At the other extreme where there is no sharing at all in a decision graph, the graph is essentially a tree, and our algorithms, especially those in the first and the third classes, are more appropriate.

3.5 Proofs of Theorems

Theorem 3.1 *If the heuristic function used by DFS is admissible, then*

$$dt(n,b) = \begin{cases} h^*(n) & \text{if } h^*(n) < b, \\ \texttt{MAXINT} & \text{otherwise.} \end{cases}$$

for any node n in the decision tree and a number b.

To prove this theorem, we make two observations. First, we observe that function h^* is equivalent to dt_0 defined as follows:

Case 1 n is a terminal:

$$dt_0(n) = h^*(n) = v(n). \tag{3.1}$$

Case 2 n is a chance node:

$$dt_0(n) = t_{0l}. \tag{3.2}$$

where $t_{0i}, 0 \leq i \leq l$, is recursively defined as follows:

$$\begin{aligned} t_{00} &= \textstyle\sum_{j=1}^{l} h(n_j) * p_j; \\ t_{0i} &= t_{0i-1} + p_i * (dt_0(n_i) - h(n_j)). \end{aligned} \tag{3.3}$$

If h is admissible, then $t_{0i} \leq t_{0i+1}$ for $i = 0, ..., l-1$.

Case 3 n is a choice node:

$$dt_0(n) = t_{0l} \tag{3.4}$$

where $t_{0i}, 0 \leq i \leq l$, is recursively defined as follows:

$$\begin{aligned} t_{00} &= \infty; \\ t_{0i} &= min\{t_{0i-1}, c_i + dt_0(n_i)\}. \end{aligned} \tag{3.5}$$

In the above definition, c_i denotes the cost of the edge from a choice node to its i-th child, and p_i denotes the probability associated with the i-th child of a chance node. They correspond to `cost(n, i)` and `prob(n, i)` respectively in algorithm DFS. This convention will be used in the rest of this section.

Second, we observe that, according to the structure of algorithm DFS, the definition of dt can be further refined as follows:

Case 1 n is a terminal;

$$dt(n,b) = \begin{cases} v(n) & \text{if } v(n) < b, \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{3.6}$$

Case 2 n is a chance node:

$$dt(n,b) = \begin{cases} t & \text{if } t < b, \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{3.7}$$

where $t = t_l$ and $t_i, 0 \leq i \leq l$ is recursively defined as follows:

$$\begin{aligned} t_0 &= \textstyle\sum_{j=1}^{l} h(n_j) * p_j; \\ b_i &= (b - (t_{i-1} - h(n_i) * p_i))/p_i; \\ t_i &= \begin{cases} t_{i-1} & \text{if } t_{i-1} \geq b, \\ t_{i-1} + p_i * (dt(n_i, b_i) - h(n_i)) & \text{otherwise.} \end{cases} \end{aligned} \tag{3.8}$$

Case 3 n is a choice node:

$$dt(n,b) = \begin{cases} t & \text{if } t < b, \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{3.9}$$

where $t = t_l$ and $t_i, 0 \leq i \leq l$ is recursively defined as follows:

$$\begin{aligned} t_0 &= b; \\ t_i &= \begin{cases} t_{i-1} & \text{if } t_{i-1} - c_i \leq h(n_i), \\ min\{t_{i-1}, c_i + dt(n_i, t_{i-1} - c_i)\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.10}$$

Thus, to prove the theorem, it suffices to prove

$$dt(n,b) = \begin{cases} dt_0(n) & \text{if } dt_0(n) < b, \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{3.11}$$

for every node n in the decision tree.

Based on the definition of dt_0 and the characterization of dt given above, relation (3.11) can be proved by induction on the structure of the decision graph. First, we need two intermediate results.

Lemma 3.17 *Let n be a choice node with children $n_1, ..., n_l$. Let t_{0i} and t_i be defined by equations (3.5) and (3.12) respectively. Suppose that for any number b',*

$$dt(n_i, b') = \begin{cases} dt_0(n_i) & \text{if } dt_0(n_i) < b', \\ \texttt{MAXINT} & \text{otherwise.} \end{cases}$$

for $1 \leq i \leq l$. Then, for any i, $1 \leq i \leq l$

(A) $t_i < b$ iff $t_{0i} < b$;

(B) if $t_{0i} < b$, then $t_{0i} = t_i$, otherwise. $t_i = b$.

Proof We prove this lemma by induction on i. Our observation here is that, under the given assumptions, equation (3.10) is equivalent to the following simpler one:

$$\begin{aligned} &t_0 = b; \\ &t_i = min\{t_{i-1}, c_i + dt(n_i, t_{i-1} - c_i)\} \end{aligned} \tag{3.12}$$

Basis: $i = 1$. $t_1 = min\{b, c_1 + dt(n_1, b - c_1)\}$ and $t_{01} = c_1 + dt_0(n_1)$.

If

$$t_{01} = c_1 + dt_0(n_1) \geq b$$

then,

$$dt_0(n_1) \geq b - c_1.$$

By the given assumptions, we have:

$$dt(n_1, b - c_1) = \texttt{MAXINT} > b.$$

Therefore

$$t_1 = b.$$

If

$$t_{01} = c_1 + dt_0(n_1) < b$$

then,

$$dt_0(n_1) < b - c_1.$$

By the given assumptions, we have:

$$dt_0(n_1) = dt(n_1, b - c_1);$$

$$c_1 + dt(n_1, b - c_1) = c_1 + dt_0(n_1) < b.$$

Therefore, $t_1 = c_1 + dt_0(n_1) = t_{01}$. The induction base holds.

Induction: Suppose the lemma is true for $i = k$. For $i = k + 1$, we have:

$$t_{k+1} = min\{t_k, c_{k+1} + dt(n_{k+1}, b - c_{k+1})\};$$

$$t_{0k+1} = min\{t_{0k}, c_{k+1} + dt_0(n_{k+1})\}.$$

Now, we have two cases:

(A) $t_{0k+1} \geq b$. In this case, we have $t_{0k} \geq b$ and $c_{k+1} + dt_0(n_{k+1}) \geq b$. Thus we can obtain $t_k = b$ by the induction assumption. Furthermore, since

$$dt_0(n_{k+1}) \geq b - c_{k+1},$$

then, by the given assumptions, we have:

$$dt(n_{k+1}, t_k - c_{k+1}) = dt(n_{k+1}, b - c_{k+1}) = \texttt{MAXINT}.$$

Therefore, $t_{k+1} = b$.

(B) $t_{0k+1} < b$, In this case, we need to consider three subcases:

(a) $t_{0k} \geq b$. In this subcase, we have:

$$c_{k+1} + dt_0(n_{k+1}) = t_{0k} < b;$$

$$dt_0(n_{k+1}) < b - c_{k+1};$$

$$t_{0k+1} = c_{k+1} + dt_0(n_{k+1}).$$

By the induction assumption, we have: $t_k = b$. By the given assumptions, we obtain:

$$dt(n_{k+1}, b - c_{k+1}) = dt_0(n_{k+1}).$$

Thus,

$$t_{k+1} = c_{k+1} + dt(n_{k+1}, b - c_{k+1}) = dt_0(n_{k+1}) + c_{k+1}.$$

Therefore, $t_{k+1} = t_{0k+1}$.

(b) $t_{0k} < b$ and $t_{0k} \leq c_{k+1} + dt_0(n_{k+1})$. In this subcase, we have:

$$t_{0k+1} = t_{0k}$$

$$t_{0k} = t_k \text{ (by the induction assumption).}$$

Thus,

$$t_k \leq c_{k+1} + dt_0(n_{k+1}) \leq c_{k+1} + dt_0(n_{k+1}, b - c_{k+1}).$$

Therefore, $t_{k+1} = t_k = t_{0k+1}$.

(c) $t_{0k} < b$ and $t_{0k} > c_{k+1} + dt_0(n_{k+1})$ In this subcase, we have:

$$t_{0k} = t_k \text{ (by the induction assumption);}$$

$$t_{0k+1} = c_{k+1} + dt_0(n_{k+1})$$

$$dt_0(n_{k+1}) < t_{0k} - c_{k+1} = t_k - c_{k+1}.$$

Thus, by the given assumptions, we have:

$$dt(n_{k+1}, t_k - c_{k+1}) = dt_0(n_{k+1}).$$

Thus,

$$dt(n_{k+1}, b - c_{k+1}) + c_{k+1} = dt_0(n_{k+1}) + c_{k+1} < t_k.$$

Therefore, $t_{k+1} = dt(n_{k+1}, b - c_{k+1}) + c_{k+1} = t_{0k+1}$.

In summary, the claim holds for $i = k + 1$. Consequently, the lemma holds by induction. □

Lemma 3.18 *Let n be a chance node with children $n_1, ..., n_l$. Let t_{0i} and t_i be defined by equations (3.3) and (3.8) respectively. Suppose that for any number b',*

$$dt(n_i, b') = \begin{cases} dt_0(n_i) & \text{if } dt_0(n_i) < b', \\ \texttt{MAXINT} & \text{otherwise.} \end{cases}$$

for $1 \leq i \leq l$. Then, for any i, $1 \leq i \leq l$

(A) $t_i < b$ iff $t_{0i} < b$;

(B) if $t_{0i} < b$, then $t_{0i} = t_i$.

Proof: By induction on i, similar to that for Lemma 3.18. □

Proof of Theorem 3.1. We prove relation (3.11) by induction on nodes in the decision tree.

1. n is a terminal. Then $v(n) = h^*(n) = dt_0(n)$. Thus, relation (3.11) holds trivially.

2. n is a choice node. Suppose relation (3.11) holds for all the children of n. We need to consider two cases.

   (A). $dt_0(n) \geq b$. We have the following derivation:

$t_{0l} \geq b$ (by equation (3.4))

$t = t_l \geq b$ (by Lemma 3.17)

$dt(n, b) = \texttt{MAXINT}$ (by equation (3.9)).

(B). $dt_0(n) \leq b$. We have the following derivation:

$dt_0(n) = t_{0l} < b$ (by equations (3.4))

$t_l = t_{0l} \geq b$ (by Lemma 3.17)

$dt(n, b) = t = t_{0l} = dt_0(n)$ (by equation 3.9).

In summary, relation (3.11) holds for n.

3. n is a chance node. Suppose relation (3.11) holds for all the children of n. Similarly, it can be proved that the relation holds for n as well by using Lemma 3.18.

In summary, the theorem holds in general. □

Theorem 3.4 *Suppose DFS uses heuristic function h. If there exists a number $\delta \geq 0$ such that h satisfies:*

$$h(n) \leq h^*(n) + \delta \quad \textit{for every node } n \textit{ in a decision tree}$$

then

$$h^*(n) + \delta \geq b \quad \textit{if } dt(n, b) \geq b;$$

and

$$h^*(n) + \delta \geq dt(n, b) \quad \textit{if } dt(n, b) < b$$

for every node n in the decision tree.

Theorem 3.5 *Suppose DFS uses heuristic function h. If the arc costs in a decision tree are all non-negative, $h^*(n) \geq 0$ for every node n in the decision tree, and there exists a number $\delta \geq 0$ such that h satisfies:*

$$h(n) \leq (1+\delta) * h^*(n), \textit{for every node } n \textit{ in the decision tree,}$$

then for every node n in the decision tree and any number $b \geq 0$,

$$h^*(n) * (1+\delta) \geq b \quad \textit{if } dt(n,b) \geq b;$$

and

$$h^*(n) * (1+\delta) \geq dt(n,b) \quad \textit{if } dt(n,b) < b.$$

Since h^* is equivalent to dt_0, all the occurrences of $h^*(n)$ in the above theorems can be replaced with $dt_0(n)$. Therefore, for Theorem 3.4, it suffices to prove that for every node n in the decision tree

$$dt_0(n) + \delta \geq b \quad \text{if } dt(n,b) \geq b;$$

and

$$dt_0(n) + \delta \geq dt(n,b) \quad \text{if } dt(n,b) < b$$

For Theorem 3.5, it suffices to prove that for every node n in the decision tree

$$dt_0(n) * (1+\delta) \geq b \quad \text{if } dt(n,b) \geq b;$$

and

$$dt_0(n) * (1+\delta) \geq dt(n,b) \quad \text{if } dt(n,b) < b$$

The proofs of Theorems 3.4 and 3.5 are very similar. Here we just present the proof of Theorem 3.5.

Proof of Theorem 3.5

Case 1 n is a terminal. Since $dt_0(n) = h^*(n) \geq 0$, thus, $(1+\delta) * dt_0(n) \geq dt_0(n)$.

If $dt(n,b) \geq b$, then $v(n) = h^*(n) \geq b$, therefore, $dt_0(n) * (1+\delta) \geq b$.

If $dt(n,b) < b$, then $dt(n,b)v(n) = h^*(n) = dt_0(n)$, thus $dt_0(n) * (1+\delta) \geq dt(n,b)$.

Therefore, the theorem holds for node n.

Case 2 n is a chance node.

Suppose the theorem holds for all the children of node n. We need to consider the following two cases:

(A). $dt(n,b) < b$. According to equation (3.7), we have $dt(n,b) = t = t_l < b$. Thus, $t_i < b$ for $i = 1, ..., l$. Consequently, by equation (3.8), we have: $dt(n_i, b_i) < b_i$. By the induction assumption, we have:

$$dt(n_i, b_i) \leq (1+\delta) * dt_0(n_i)$$

for $i = 1, ..., l$. According to equations (3.3) and (3.2), we obtain:

$$dt_0(n) = t_{0l} = \sum_{i=1}^{l} p_i * dt_0(n_i)$$

According to equation (3.8), we obtain:

$$t_l = \sum_{i=1}^{l} p_i * dt(n_i, b_i) \leq \sum_{i=1}^{l} p_i * dt(n_i) * (1+\delta) = t_{0l} * (1+\delta)$$

Thus, $dt(n,b) \leq dt_0(n) * (1+\delta)$.

(B). $dt(n, b) \geq b$. According to equations (3.7) and (3.8), we know that $t = t_l \geq b$. This implies that either $t_0 \geq b$ or there exists $k, 1 \leq k \leq l$ such that $t_{k-1} < b$ and $t_k \geq b$. In the former case, we have:

$$dt_0(n) * (1+\delta) = \sum_{i=1}^{l} p_i * dt_0(n_i) * (1+\delta) \geq \sum_{i=1}^{l} p_i * h(n_i) = t_0$$

Thus, $dt_0(n) * (1+\delta) \geq b$.

In the latter case, we can obtain:

$$dt(n_j, b_j) < b_j \quad \text{for} \quad 0 \leq j < k$$

$$dt(n_k, b_k) \geq b_k$$

where $b_k = (b - (t_{k-1} - h(n_k) * p_k))/p_k$. Consequently, by the induction assumption, we have:

$$dt_0(n_j) * (1+\delta) \geq dt(n_j, b_j) \quad \text{for} \quad 0 \leq j < k$$

and

$$(1+\delta) * dt_0(n_k) \geq b_k$$

Therefore:

$$p_k * (1+\delta) * dt_0(n_k) \geq b - (t_{k-1} - h(n_k) * p_k))$$

$$t_{k-1} + p_k * (1+\delta) * dt_0(n_k) - h(n_k) * p_k \geq b$$

According to equation (3.8), we have:

$$t_{k-1} = \sum_{j=1}^{k-1} dt(n_j, b_j) * p_j + \sum_{j=k}^{l} h(n_j) * p_j$$

Thus,

$$\sum_{j=1}^{k-1} dt(n_j, b_j) * p_j + p_k * (1+\delta) * dt_0(n_k) + \sum_{j=k+1}^{l} h(n_j) * p_j \geq b$$

$$\sum_{j=1}^{k}(1+\delta)dt_0(n_j) * p_j + \sum_{j=k+1}^{l}(1+\delta)dt_0(n_j) * p_j \geq b$$

Therefore,

$$(1+\delta) * dt_0(n) = \sum_{j=1}^{l}(1+\delta)dt_0(n_j) * p_j \geq b$$

Case 3 n is a choice node.

Suppose the theorem holds for all the children of node n. The theorem holds too for node n if we can prove

$$t_i \leq (1+\delta) * t_{0i} \tag{3.13}$$

for $i = 0, ..., l$, where t_{0i} and t_i are defined by equations (3.5) and (3.12) respectively. This inequality can be proved by induction on i.

Basis:: $i = 0$, trivial.

Induction:. Suppose the inequality is true for $i = k$. Consider the case when $i = k + 1$.

(A). $t_{0k} \leq c_{k+1} + dt_0(n_{k+1})$. In this case, we have $t_{0k} = t_{0k+1}$. Since $t_{k+1} \leq t_k$, by the inner induction assumption, we conclude $t_{k+1} \leq (1+\delta) * t_{0k+1}$.

(B). $t_{0k} > c_{k+1} + dt_0(n_{k+1})$. In this case, we have: $c_{k+1} + dt_0(n_{k+1}) = t_{0k+1}$. If $t_k - c_{k+1} \leq h(n_{k+1})$, then,

$$t_{k+1} = t_k \leq c_{k+1} + h(n_{k+1}) \leq c_{k+1} + (1+\delta) * dt_0(n_{k+1})$$

Since $c_{k+1} \geq 0$ and $\delta \geq 0$, we have $t_{k+1} \leq (1+\delta) * t_{0k+1}$.

If $t_k - c_{k+1} > h(n_{k+1})$, then,

$$t_{k+1} = min\{t_k, c_{k+1} + dt(n_{k+1}, t_k - c_{k+1})\}$$

When $t_k - c_{k+1} \leq dt(n_{k+1}, t_k - c_{k+1})$, by the outer induction assumption, we have:

$$(1+\delta) * dt_0(n_{k+1}) \geq t_k - c_{k+1}$$

$$t_{k+1} = t_k \leq (1+\delta) * dt_0(n_{k+1}) + c_{k+1}$$

Since $c_{k+1} \geq 0$ and $\delta \geq 0$, we have $t_{k+1} \leq (1+\delta) * t_{0k+1}$.

When $t_k - c_{k+1} > dt(n_{k+1}, t_k - c_{k+1})$, by the outer induction assumption, we have:

$$(1+\delta) * dt_0(n_{k+1}) \geq dt(n_{k+1}, t_k - c_{k+1})$$

$$t_{k+1} = c_{k+1} + dt(n_{k+1}, t_k - c_{k+1}) \leq c_{k+1} + (1+\delta) * dt_0(n_{k+1})$$

Since $c_{k+1} \geq 0$ and $\delta \geq 0$, we have $t_{k+1} \leq (1+\delta) * t_{0k+1}$.

By induction, inequality 3.13 holds for all $i = 0, ..., l$.

In summary, the theorem holds for any node n. $\square$

Part II

INFLUENCE DIAGRAM EVALUATION

Chapter 4

Decision Analysis and Influence Diagrams

In this chapter, we review some basic concepts about decision analysis and informally introduce influence diagrams — a framework for decision analysis. We motivate our work by discussing the advantages and disadvantages of this framework. In the chapters to follow, we present an extension to influence diagrams and develop a method for influence diagram evaluation. Our method, which aims to overcome the disadvantages, employs decision graphs as an intermediate representation and uses the decision graph search algorithms developed in Chapter 3 to compute optimal solutions to decision problems.

4.1 Bayesian Decision Analysis

We all make decisions every day. A decision problem often involves many variables, each having a number of possible outcomes. Some variables can be controlled in the sense that we can select one of its possible outcomes as its value. They are called *decision variables*. Other variables are beyond our control. They are called *uncertain variables*. Among uncertain variables, some are observable whereas others are not.

Variables are inter–related in one way or another. The problem is to choose appropriate values for decision variables so as to best meet our objective. A prescription on how to choose values for decision variables is called *a decision policy* or *policy*. If the non–observable uncertain variables in a decision problem are relevant to our objective, the decision problem is called a *problem of decision making under uncertainty*.

As an example, consider a simplified version of the used car buyer problem [34]. In this example, Joe has to decide whether to buy a used car. Two variables are involved in this decision problem: an uncertain variable representing the car's condition that can be either "peach" or "lemon," and a decision variable representing Joe's purchase decision. According to the current market, a peach is worth $1060 and a lemon is worth $900. The price of the car is $1000. The difficulty of Joe's decision stems from the uncertainty of the car's condition.

Bayesian decision theory [25] is concerned with those decision problems in which uncertain variables are random variables. A fundamental result of Bayesian decision theory is that any decision preference of a rational decision maker can be captured by a utility function such that his/her decision objective can be achieved by maximizing the expected utility. A policy maximizing the utility function is called an *optimal policy*.

In the used car buyer problem, if the uncertainty of the car's condition can be characterized by a probability distribution, then Bayesian decision theory is applicable. Suppose that the car is a peach with probability 0.8 and a lemon with probability 0.2. Suppose further that Joe's objective is to maximize the expected profit (in dollars). Then the policy maximizing the expected profit is to buy the car. This policy will bring Joe a profit of $60 with probability 0.8 and a loss of $100 with probability 0.2. Thus, the expected profit is $28.

Bayesian decision analysis (or decision analysis for short) [79, 96] is a discipline about the application of Bayesian decision theory to decision problems. There are two fundamental issues: how to model a decision problem, and how to compute an optimal policy.

4.2 Decision Trees

Decision trees were used as a simple tool both for problem modeling and optimal policy computation in the early days of decision analysis [79]. A decision tree can represent the order in which decisions are made and the information available to the decision agent when he/she makes a decision. It explicitly depicts all scenarios of the problem and specifies the "utility" the agent can get in each scenario.

In the literature of decision analysis, little attention has been paid to the computation of optimal policies of decision trees. It is commonly assumed that the so-called "average-out-and-fold-back" method [79, 96] is used for the computation.

From the computational point of view, a decision tree is just a decision graph without node sharing. Thus, the algorithms presented in Chapter 3 can be used for computing an optimal policy for decision trees.

In this section, we illustrate by an example some basic concepts about, and the advantages and the disadvantages of, decision trees.

A decision tree for the used car buyer problem is shown in Fig. 4.1. In this tree, the choice node corresponds to the purchase decisions and the chance nodes correspond to the car's condition. The choice node is followed by chance nodes, indicating that the car's condition is not known to Joe when he makes the purchase decision. Each path from the root to a leaf node represents a possible scenario. For example, the path on the top of the decision tree in Fig 4.1 corresponds to the decision scenario in which Joe decides to buy the car and the car turns out to be a peach. The probability for this scenario is 0.8 (the product of probabilities on the arcs along the path). The value for this scenario is 60.

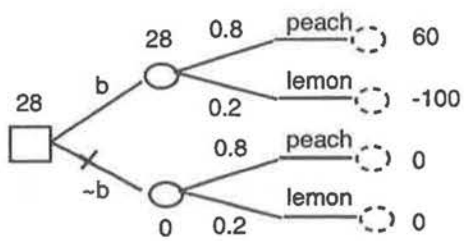

Figure 4.1: A decision tree for the used car buyer problem

In general, a decision problem may involve many decision and random variables. For such a problem, the order of nodes in the decision tree is important; it must be consistent with the decision making order and the constraints of information availability. More explicitly, a choice node N comes before another choice node N' if and only if the decision corresponding to N will be made before the decision corresponding to N'; a chance node N comes before a choice node N' if and only if the value of the random variable corresponding to the chance node N is known to the decision maker when the decision corresponding to the choice node N' is to be made.

In order to illustrate the above point, let us continue the used car buyer problem [34]. Suppose Joe knows that, of the ten major subsystems in the car, a peach has a defect in only one subsystem whereas a lemon has a defect in six subsystems (that is why Joe likes a peach and does not like a lemon). Joe also knows that a car without defect is worth \$1100 in market. Finally, Joe knows that it will cost him \$40 to repair one defect and \$200 to repair six defects. Observing Joe's concern about the possibility that the car may be a lemon, the dealer offers an "anti–lemon guarantee" option. For an additional \$60, the anti–lemon guarantee will cover the full repair cost if the car is a lemon, and cover half of the repair cost otherwise. At the same time, a mechanic suggests that some mechanical examination may help Joe have a better idea of the car's condition. In particular, the mechanic offers three alternatives: test the steering subsystem alone at a cost of \$9; test the fuel and electrical subsystems at a total cost of \$13; a two–test sequence in which the transmission subsystem will be tested first at a cost of \$10 and, after knowing the test result, Joe can decide whether to test the differential subsystem at an additional cost of \$4. All tests are guaranteed to detect a defect if one exists in the subsystem(s) being tested (in other words, the tests provide perfect information about the subsystems being tested.

In this modified example, Joe has to make three decisions: two decisions about whether to perform mechanical tests and one decision about whether to buy the car. There are two new random variables: one for the result of the first test and the other for the result of the second test. The decision making order is: the first test, the second test and then the purchase decision. Furthermore, when deciding whether to do the second test, Joe remembers his choice for the first test and knows the test results; when deciding whether to buy the car, Joe remembers his choices for both tests and knows the results for the tests. A complete decision tree for this problem is shown in Fig. 4.2.

In the figure, the choice node labeled $\mathtt{T}_1$ corresponds to the first test decision. It has four possible outcomes: `nt` for no test, `st` for testing the steering subsystem alone, `f&e` for testing the fuel and electrical subsystems, and `tr` for testing the transmission subsystem first. The chance nodes labeled $\mathtt{R}_1$ correspond to the random variable of the first test result, which has three possible outcomes: `zero` for no defect being detected, `one` for one defect being detected and `two` for two defects being detected.

The choice nodes labeled T_2 correspond to the second test decision, which needs to be made only when the choice for the first test decision is tr. It has two possible outcomes: nt for no test, and diff for testing the differential subsystem. The chance nodes labeled R_2 correspond to the random variable of the second test result, which has two possible outcomes: zero for no defect being detected and one for one defect being detected. The choice nodes labeled B correspond to the purchase decision, which has three possible outcomes: b for buying the car without the anti-lemon guarantee, g for buying the car with the anti-lemon guarantee and $\tilde{b}$ for not buying the car.

Though conceptually simple, decision trees have a number of drawbacks.

First, the dependence/independence relationships among the variables in a decision problem cannot be represented in a decision tree. For example, in the used car buyer problem, the profit is conditionally independent of the test results, given the test decisions, the purchase decision and the car condition. However, this conditional independence cannot be represented in the decision tree.

Second, a decision tree specifies a particular order for the assessment of the probability distributions of the random variables in the decision problem. This order is in most cases not a natural assessment order. For example, in the used car buyer problem, we need the probabilities of the first test results given the first test decision, and the conditional probabilities of the car condition given the test decisions and the test results. These probabilities are harder to assess than the prior distribution of the car condition, the conditional distributions of the test results conditioned on the test decisions and the car condition. In the process of constructing the complete decision tree for the used car buyer problem, we have to compute these conditional distributions using Bayes Theorem.

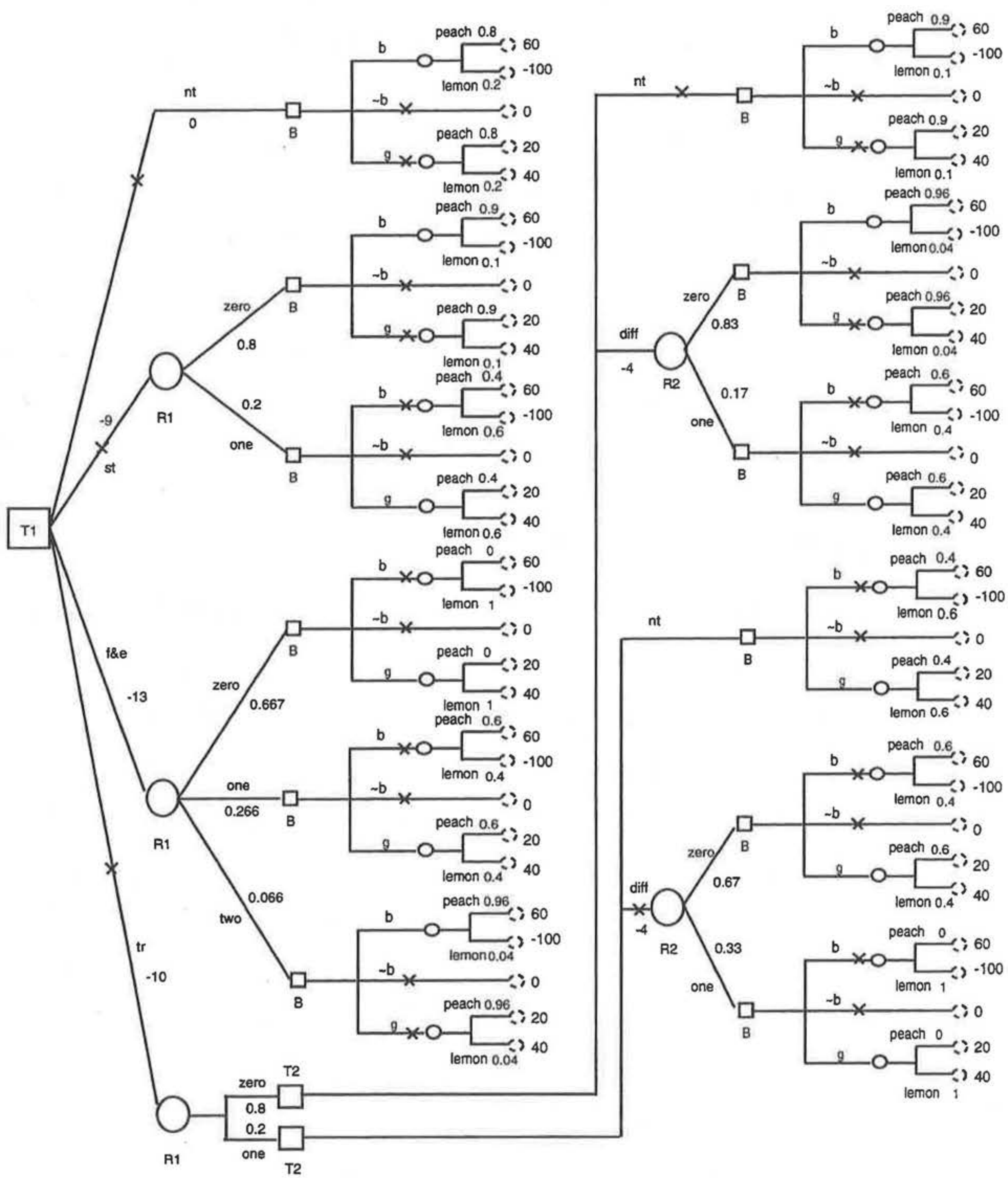

Figure 4.2: A complete decision tree for the used car buyer problem

Third, the size of a decision tree for a decision problem is in general exponential in the number of variables in the decision problem. This makes decision trees appropriate only for decision problems with a small number of variables.

Finally, a decision tree is not easily adaptable to changes in a decision problem. If a slight change is made in a problem, one may have to draw a new decision tree. This point can be illustrated by an example. Suppose that in the used car problem, Joe has \$1060 disposable money and somebody offers Joe a lottery at cost of \$50. The lottery may return \$120 or nothing, with equal probability. Joe has to decide whether to buy the lottery before making any decisions about the used car. The decision tree for this variation will be very different from the one shown in Fig. 4.2.

4.3 Influence Diagrams

Influence diagrams were first proposed as an alternative representation framework for decision analysis [35, 56].

Within the framework of influence diagrams, a decision problem can be specified at three levels: the level of relation, the level of function and the level of number [35].

At the level of relation, a decision problem is depicted by a directed acyclic graph with three types of nodes: *random nodes*, *decision nodes* and *value nodes*. Each random node represents a random variable whose value is dictated by some probability distribution. Each decision node represents a decision variable whose value is to be chosen by the decision maker. A value node represents a component of the utility function. In this thesis, we use the terms decision (random) variables and decision (random) nodes interchangeably. At the level of function, all the possible outcomes of the random variables and all the possible alternatives of the decision variables are identified. The set of all possible outcomes of a random variable is called the *frame* of the random variable. The set of all possible alternatives of a decision variable is called the *frame* of the decision variable. At the level of number, (conditional) distributions are assigned to the random variables.

In an influence diagram, an arc from a variable x to a decision variable d indicates that the value of the variable x is known to the decision maker at the time when the decision d is to be made. The Cartesian product of the frames of a decision variable's parents is the set of *information states* for the decision variable. An information state denotes the information available to the decision maker when the decision is to be made. An arc from a node x to a random node (or a value node) y indicates that the random variable (resp. the value function) y is probabilistically dependent on the variable x.

For example, at the level of relation, the used car buyer problem can be represented by the diagram as shown in Fig. 4.3. By convention, boxes denote decision variables, circles denote random variables and the diamond denotes a value function (the utility function to be maximized).

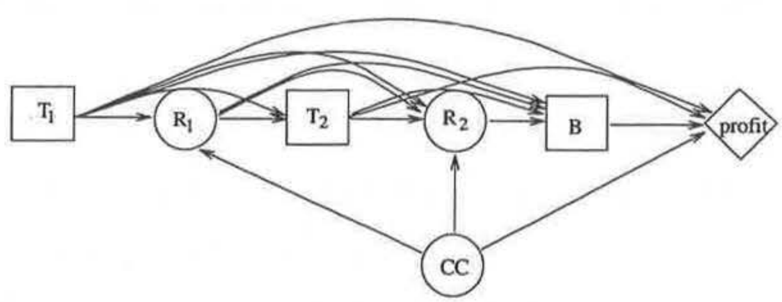

Figure 4.3: An influence diagram for the used car buyer problem

The diagram can be read as follows. There are three decision variables (T_1, T_2 and B), three random variables (R_1, R_2 and CC) and one value function (`profit`) in the decision problem. The first test decision T_1 is to be made first. The first test result

Table 4.1: The prior probability distribution of the car's condition $P\{\texttt{CC}\}$

CC	prob
peach	0.8
lemon	0.2

$\texttt{R}_1$ depends on this decision and the car's condition CC. This dependence is indicated by the two arcs to node $\texttt{R}_1$. The decision maker remembers his choice for the first test decision and knows the test result when he makes the second test decision $\texttt{T}_2$. This is indicated by the two arcs to node $\texttt{T}_2$. The second test result $\texttt{R}_2$ depends on the first test, the result of the first test, the second test and the car condition. When making the purchase decision, the decision maker remembers his choices for the first two decisions and knows the test results. The value function depends on the three decisions and the car condition.

To complete the influence diagram representation, we need to specify the frames of the variables and the (conditional) probability distributions for the random variables.

The frame for the random variable CC contains two elements: **peach** and **lemon**. This variable has no parent in the graph. Its prior probability distribution is given in Table 4.1.

The frame for the decision variable $\texttt{T}_1$ contains four elements: **nt, st, f&e** and **tr**, representing respectively the alternatives of performing no test, testing the steering subsystem alone, testing the fuel and electrical subsystems, and testing the transmission subsystem first with an option of testing the differential subsystem next.

The frame for the random variable $\texttt{R}_1$ contains four elements: **nr, zero, one** and **two** representing respectively the four possible outcomes of the first test: no result, no defect, one defect and two defects. The probability distribution of the variables, conditioned on $\texttt{T}_1$ and CC, is given in Table 4.2.

Table 4.2: The probability distribution of the first test result $P\{R_1|T_1, CC\}$

T_1	CC	R_1	prob
nt	–	nr	1.0
nt	–	others	0
st	–	nr	0
st	–	two	0
st	peach	zero	0.9
st	peach	one	0.1
st	lemon	zero	0.4
st	lemon	one	0.6
f&e	–	nr	0
f&e	peach	zero	0.8
f&e	peach	one	0.2
f&e	peach	two	0
f&e	lemon	zero	0.13
f&e	lemon	one	0.53
f&e	lemon	two	0.33

Table 4.3: The probability distribution of the second test result $P\{R_2|T_1, R_1, T_2, CC\}$

T_1	R_1	T_2	CC	R_2	prob
nt	–	–	–	nr	1.0
nt	–	–	–	others	0
st	–	–	–	nr	1.0
st	–	–	–	others	0
f&e	–	–	–	nr	1.0
f&en	–	–	–	others	0
tr	nr	–	–	nr	1.0
tr	nr	–	–	others	0
tr	two	–	–	nr	1.0
tr	two	–	–	others	0
tr	–	nt	–	nr	1.0
tr	–	nt	–	others	0
tr	zero	diff	peach	zero	0.89
tr	zero	diff	peach	one	0.11
tr	zero	diff	lemon	zero	0.67
tr	zero	diff	lemon	one	0.33
tr	one	diff	peach	zero	1.0
tr	one	diff	peach	one	0
tr	one	diff	lemon	zero	0.44
tr	one	diff	lemon	one	0.56

The frame for the decision variable T_2 contains two elements: nt and diff, denoting the two options of performing no test and testing the differential subsystem.

The frame for the random variable R_2 contains three elements: nr, zero and one, representing respectively the three possible outcomes of the second test: no result, no defect and one defect. The probability distribution of the variable, conditioned on T_1, R_1, T_2 and CC, is given in Table 4.3.

The frame for the decision node B contains three elements: $\tilde{b}$, b and g, denoting respectively the alternatives of not buying, buying without the anti–lemon guarantee, and buying with the anti–lemon guarantee.

From the above example, we observe that the influence diagram representation has a number of advantages in comparison with decision trees: it is expressive enough to explicitly describe the dependence/independence relationships among the variables involved in the decision problem; it allows a more natural assessment order on the probabilities of the random variables and it is compact. In addition, influence diagrams are easy to adapt to changes in problems. To illustrate this point, let us consider again the variation of the used car problem given at the end of the previous section. In that variation, Joe has to decide whether to buy a lottery before considering the decisions about the used car. That variation can be represented by an influence diagram as shown in Fig. 4.4, in which node B' represents the decision of lottery purchase, node L represents the lottery and node R_0 represents the actual return Joe gets from the lottery. This influence diagram is derived from the one shown in Fig. 4.3 by the addition of three new nodes and ten arcs.

A *decision function* for a decision variable is a mapping from the set of the information states of the decision variable to the decision variable's frame, prescribing a choice for the decision for each information state. A *decision policy* (or a *policy*

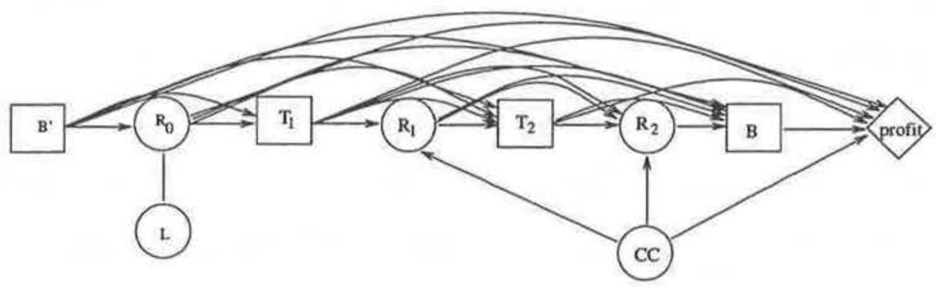

Figure 4.4: An influence diagram for a variation of the used car buyer problem

for short) for an influence diagram is a collection of decision functions, one for each decision variable. Each policy determines an expected value of the influence diagram. A policy is *optimal* if it maximizes the expected value of the influence diagram. We will give a formal definition of these concepts in the next chapter.

Influence diagrams were first proposed as a "front–end" for the automation of decision making process [56, 35]. The actual analysis of a decision problem was carried out in two phases. An influence diagram was first transformed into a decision tree and then the decision tree was evaluated. The idea of evaluating influence diagrams directly was proposed in [64]. The first complete algorithm for influence diagram evaluation was developed by Shachter [85]. We will give a review on related research efforts in Chapter 6.

4.4 Disadvantages of Influence Diagrams

While influence diagrams have many advantages as a representation framework for decision problems, they have a serious drawback in handling asymmetric decision problems [12, 26, 70, 85, 95]. Decision problems are usually asymmetric in the sense that the set of possible outcomes of a random variable may vary depending on different conditioning states, and the set of legitimate alternatives of a decision variable may vary depending on different information states. For example, in the used car buyer problem, if the choice for the first test is nt (no test), the first test result can only be nr. On the other hand, if the choice for the first test is st or tr, the test result can be one of the two possible outcomes: zero and one, and if the choice for the first test is f&e, the test result can be one of the three possible outcomes: zero, one and two. Furthermore, the alternative of testing the differential subsystem for the second test decision is possible only if the choice for the first test is tr (testing the transmission subsystem).

To represent an asymmetric decision problem as an influence diagram, the problem must be "symmetrized" by using common frames and assuming degenerate probability distributions [95]. In the used car buyer problem, the frame of the variable T_1 is a common set of outcomes for all the three cases. The impossible combinations of the test choices and the test results are characterized by assigning them zero probability in Table 4.2. The frame of the second test is also a common set of all alternatives in various states, while the fact that the second test option is not available in some states is characterized by assigning unit probability to outcome nr of the variable R_2 conditioned on these states.

This symmetrization results in two problems. First, the number of information states of decision variables is increased. Among the information states of a decision variable, many are "impossible" (having zero probability). For example, the information state corresponding to T_1=st,R_1=nr for the second test decision variable T_2 is an impossible state. The optimal choices for these impossible states need not be computed at all. However, they are computed by conventional influence diagram evaluation algorithms [95]. Second, for each information state of a decision variable, because the legitimate alternatives may constitute only a subset of the frame of the decision variable, an optimal choice is chosen from only a subset of the frame, instead of the entire frame. However, conventional influence diagram algorithms have to consider each alternative in order to compute an optimal choice for a decision in any of its information states.

From the above analysis, we observe that conventional influence diagram evaluation algorithms involve unnecessary computation. We show in Section 7.5 that, for a typical decision problem, the amount of unnecessary computation can be exponential in the number of decision variables.

4.5 Previous Attempts to Overcome the Disadvantages

Several researchers have recently proposed alternative representations that attempt to combine the strengths of influence diagrams and decision trees. Fung and Shachter [26] propose a representation, called *contingent influence diagrams*, for explicitly expressing the asymmetric aspects of decision problems. In that representation, each variable is associated with a set of contingencies, and associated with one relation for each contingency. These relations collectively specify the conditional distribution of the variable.

Covaliu and Oliver [12] propose a different representation for representing decision problems. This representation uses a *decision diagram* and a *formulation table* to specify a decision problem. A decision diagram is a directed acyclic graph whose directed paths identify all possible sequences of decisions and events in a decision problem. In a sense, a decision diagram is a degenerate decision tree in which paths having a common sequence of events are collapsed into one path [12]. Numerical data are stored in the formulation table. Covaliu and Oliver [12] also give a backward algorithm to compute optimal policies from decision diagrams by using the formulation table.

Shenoy [92] proposes a "factorization" approach for representing degenerate probability distributions. In that approach, a degenerate probability distribution over a set of variables is decomposed into several factors over subsets of the variables such that the their "product" is equivalent to the original distribution.

Smith *et al.* [95] observe that the computation of various probabilities involved in influence diagram evaluation can be sped up if the degenerate probability distributions in influence diagrams are used properly. Their philosophy is analogous to the one behind various algorithms for sparse matrix computation. In their proposal, a conventional influence diagram is used to represent a decision problem at the level of relation. In addition, they propose to use a decision tree–like representation to describe the conditional probability distributions associated with the random variables in the influence diagram. The decision tree–like representation is effective for economically representing degenerate conditional probability distributions. They propose a modified version of Shachter's algorithm [85] for influence diagram evaluation, and show how the decision tree–like representation can be used to increase the efficiency of *arc reversal*, a fundamental operation used in Shachter's algorithm. However, their algorithm cannot avoid computing optimal choices for decision variables with respect to impossible information states, because it, like other influence diagram evaluation methods, also takes a reduction approach. We will come back to this point in Section 6.2.3.

4.6 Our Solution

In this thesis, we develop an approach for overcoming the aforementioned disadvantages of influence diagrams. Our approach consists of two independent components: a simple extension to influence diagrams and a top down method for influence diagram evaluation.

Our extension allows the explicit expression of the fact that some decision variables have different frames in different information states. We achieve this by introducing a *framing* function for each decision variable. The framing function characterizes the available alternatives for the decision variable in different information states. With the help of framing functions, our solution algorithm can effectively ignore the unavailable alternatives when computing an optimal choice for a decision variable in any information state. Our extension is inspired by the concepts of *indicator valuations* and *effective frames* proposed by Shenoy [92]. Note that our extension is orthogonal to Smith's. In our influence diagram representation, we can also use their decision–tree like representation to describe conditional distributions of random variables. Furthermore, we can also use their method to exploit asymmetry in computing conditional probabilities.

A novel aspect of our method is that it avoids computing optimal choices for decision variables in any impossible states. This method works for influence diagrams with or without our extension.

Our method, similar to Howard and Matheson's [35], evaluates an influence diagram in two conceptual steps: it first generates a decision graph from the influence diagram and then evaluates an optimal policy from the decision graph. In such a decision graph, a choice node corresponds to an information state of some decision variable. The decision graph is generated in such a way that an optimal solution graph of the decision graph corresponds to an optimal policy of the influence diagram. Thus, the problem of computing an optimal policy is reduced to the problem of searching for an optimal solution graph of a decision graph, which can be accomplished by the algorithms presented in Chapter 3. Our method successfully avoids unnecessary computation by pruning those choice nodes in the decision graph that correspond to impossible states, and by ignoring those children of choice nodes that correspond to unavailable alternatives for the decision variables.

By using heuristic search techniques, our method provides an explicit mechanism for making use of heuristic information that may be available in a domain-specific form. The combined use of heuristic search techniques and domain-specific heuristics may result in even more computational savings. Furthermore, since our method provides a clean interface between influence diagram evaluation and Bayesian net evaluation, various well-established algorithms for Bayesian net evaluation can be used in influence diagram evaluation. Finally, by using decision graphs as an intermediate representation, the value of perfect information [53] can be computed more efficiently [110].

Note that our method is more efficient than Howard and Matheson's, partly because our method generates a much smaller decision graph for the same influence diagram.

Chapter 5

Formal Definition of Influence Diagrams

In this chapter, we give a formal definition of influence diagrams and present our extension. The definition is borrowed from [109].

5.1 Influence Diagrams

An influence diagram $\mathcal{I}$ is a quadruple $\mathcal{I} = (X, A, \mathcal{P}, \mathcal{U})$ where

- (X, A) is a directed acyclic graph with X partitioned into random node set C, decision node set D and value node set U, such that the nodes in U have no children.

  Each decision node or random node has a set, called the *frame*, associated with it. The frame of a node consists of all the possible outcomes of the (decision or random) variable denoted by the node. For any node $x \in X$, we use $\pi(x)$ to denote the parent set of node x in the graph. For any $x \in C \cup D$, we use Ω_x to denote the frame of node x. For any subset $J \subseteq C \cup D$, we use Ω_J to denote the Cartesian product $\prod_{x \in J} \Omega_x$.

- $\mathcal{P}$ is a set of probability distributions $P\{c|\pi(c)\}$ for all $c \in C$. For each $o \in \Omega_c$ and $s \in \Omega_{\pi(c)}$, the distribution specifies the conditional probability of event $c = o$, given that $\pi(c) = s$. Throughout this thesis, we use $J = e$ to denote the set of assignments that assign an element of e to the corresponding variable in J for any variable set J and any element $e \in \Omega_J$.

- $\mathcal{U}$ is a set $\{g_v : \Omega_{\pi(v)} \rightarrow \mathcal{R} | v \in U\}$ of *value functions* for the value nodes, where $\mathcal{R}$ is the set of the reals.

For a decision node d_i, a mapping $\delta_i : \Omega_{\pi_{d_i}} \rightarrow \Omega_{d_i}$ is called a *decision function* for d_i. The set of all the decision functions for d_i, denoted by Δ_i, is called the *decision function space* for d_i. Let $D = \{d_1, ..., d_n\}$ be the set of decision nodes in influence diagram $\mathcal{I}$. The Cartesian product $\Delta = \prod_{i=1}^{n} \Delta_i$ is called the *policy space* of $\mathcal{I}$.

5.2 Our Extension

We extend the definition of influence diagrams given in the previous section by introducing *framing functions.*

An influence diagram $\mathcal{I}$ is a tuple $\mathcal{I} = (X, A, \mathcal{P}, \mathcal{U}, \mathcal{F})$ where $X, A, \mathcal{P}, \mathcal{U}$ have the same meaning as before, and $\mathcal{F}$ is a set $\{f_d : \Omega_{\pi(d)} \rightarrow 2^{\Omega_d}\}$ of *framing functions* for the decision nodes.

The framing functions express the fact that the legitimate alternative set for a decision variable may vary in different information states. More specifically, for a decision variable d and an information state $s \in \Omega_{\pi(d)}$, the set $f_d(s)$ contains the legitimate alternatives the decision maker can choose for d in information state s. Following Shenoy [92], we call $f_d(s)$ the *effective frame* of decision variable d in information state s.

In the used car buyer problem, the frame for the decision variable $\mathtt{T}_2$ has two elements: `nt` and `diff`, denoting the two alternatives of performing no test and testing the differential subsystem. However, the `diff` alternative is available only in the information states where the choice for the first test is transmission. This can be captured by a framing function f_{T_2} defined as follows:

$$f_{\mathrm{T}_2}(s) = \begin{cases} \{\mathtt{nt},\ \mathtt{diff}\} & \text{if } \sigma_{\mathrm{T}_1}(s) = \mathtt{tr} \\ \{\mathtt{nt}\} & \text{otherwise} \end{cases}$$

where $\sigma_{\mathrm{T}_1}(s)$ denotes the projection of s on $\mathtt{T}_1$ (the value of the first test in information state s).

Similarly, we define a decision function for a decision node d_i as a mapping $\delta_i : \Omega_{\pi_{d_i}} \rightarrow \Omega_{d_i}$. In addition, δ_i must satisfy the following constraint: For each $s \in \Omega_{\pi_{d_i}}$, $\delta_i(s) \in f_{d_i}(s)$. In words, the choice prescribed by a decision function for a decision variable d in an information state must be a legitimate alternative.

5.3 Influence Diagram Evaluation

Given a policy $\delta = (\delta_1, \delta_2, ..., \delta_n)$, each decision node d_i can be regarded as a random node whose probability distribution is given as follows:

$$P_\delta\{d_i = a | \pi(d_i) = s\} = \begin{cases} 1 & \text{if } \delta_i(s) = a, \\ 0 & \text{otherwise.} \end{cases}$$

Policy δ can be considered as a collection of probability distributions for the decision variables.

Let $\mathcal{I} = (X, A, \mathcal{P}, \mathcal{U}, \mathcal{F})$ be an influence diagram, and let $Y = C \cup D$. Let A_Y be the set of all the arcs of A that lie completely in Y. Then the triplet $(Y, A_Y, \mathcal{P} \cup P_\delta)$ is a Bayesian net, referred to as the *Bayesian net induced from $\mathcal{I}$ by the policy δ*, and is written as $\mathcal{I}_\delta$. The prior joint probability $P_\delta\{Y\}$ is given by

$$P_\delta\{Y\} = \prod_{x \in C} P\{x|\pi(x)\} \prod_{x \in D} P_\delta\{x|\pi(x)\}. \tag{5.1}$$

Because the value nodes do not have children, for any value node v, $\pi(v)$ contains no value nodes. Hence $\pi(v) \subseteq Y$. The expectation $E_\delta[v]$ of the value function g_v under P_δ is given by

$$E_\delta[v] = \sum_{o \in \Omega_{\pi(v)}} P_\delta\{\pi(v) = o\} g_v(o).$$

The *expected value* $E_\delta[\mathcal{I}]$ of $\mathcal{I}$ under the policy δ is defined by

$$E_\delta[\mathcal{I}] \;=\; \sum_{v \in U} E_\delta[v] \tag{5.2}$$

The *optimal expected value* $E[\mathcal{I}]$ of $\mathcal{I}$ is defined by

$$E[\mathcal{I}] = max_{\delta \in \Delta} E_\delta[\mathcal{I}]. \tag{5.3}$$

The optimal value of a decision network that does not have any value nodes is zero. An *optimal policy* δ^o is one that satisfies

$$E_{\delta^o}[\mathcal{I}] = E[\mathcal{I}]. \tag{5.4}$$

In this thesis we will only consider variables with finite frames. Hence there are only a finite number of policies. Consequently, there always exists at least one optimal policy. *To evaluate an influence diagram* is to find an optimal policy, and to compute the optimal expected value.

5.4 No–Forgetting and Stepwise Decomposable Influence Diagrams

In this section, we introduce two classes of influence diagrams, which are the focus of the literature.

An influence diagram is *regular* [35, 85] if there is a directed path containing all of the decision variables. Since the diagram is acyclic, such a path defines an order for the decision nodes. This is the order in which the decisions are made. An influence diagram is "no-forgetting" if each decision node d and its parents are also parents of those decision nodes that are descendants of d [35, 85]. Intuitively, the "no-forgetting" property means that a decision maker remembers all the information that was earlier available to him and remembers all the previous decisions he made. The lack of an arc from a node a to a decision node d in a no-forgetting influence diagram means that the value of the variable a is not known to the decision maker when decision d is to be made. A regular and "no-forgetting" influence diagram represents the decision maker's view of the world.

An influence diagram is called *stepwise solvable* [108, 109] if its optimal policy can be computed by considering one decision node at a time. A necessary and sufficient condition for the stepwise solvability of influence diagrams, called *stepwise-decomposability*, is provided in [106, 108, 109].

Stepwise-decomposability is defined in terms of graph separation. Informally, an influence diagram is *stepwise decomposable* if the parents of each decision node divide the influence diagram into two parts. In order to define the property formally, we need some notation and concepts.

We use $nond(x)$ to denote the set of nodes that are not descendants of x in the influence diagram. Thus, $nond(d) \cap D$ is the set of decision nodes that are not descendants of node d. For a node set Z, let $\pi(Z) = \cup_{z \in Z} \pi(z)$ and $\pi^*(Z) = Z \cup \pi(Z)$.

The *moral graph* [47] of a directed graph G is an undirected graph $m(G)$ with the same node set such that there is an edge between node x and node y in $m(G)$

if and only if either there is an arc $x \rightarrow y$ or $y \rightarrow x$ in G, or there are two arcs $x \rightarrow z$ and $y \rightarrow z$ in G and $x \neq y$. A node x is *m–separated* from a node y by a node set Z in a directed graph G if every path between x and y in the moral graph $m(G)$ contains at least one node in the set Z. Because the "m–separated" relation is symmetric, we sometimes simply say that two nodes x and y are m–separated by Z if x is m–separated from y by Z. Two sets X and Y are m–separated by set Z if x is m–separated from y by set Z for each $x \in X$ and each $y \in Y$.

Let d be a decision node in G, let $m(G)$ be the moral graph of G and let G_d be the undirected graph obtained from $m(G)$ by removing all the nodes in $\pi(d)$. The *downstream* Y_d of d is the set of all the nodes that are connected to d in G_d, with d excluded. The *upstream* X_d of d is the set of all the nodes that are not connected to d in G_d. The upstream X_d and the downstream Y_d of d are m–separated by $\pi(d)$. This property is important for influence diagrams because m–separation implies conditional independence [108]. This property is used in Section 7.2 when we establish a stochastic dynamic programming formulation for influence diagram evaluation.

An influence diagram is *stepwise decomposable* if, for each decision node d and for any node $x \in \pi^*(nond(d) \cap D)$, the following holds: $\{x\} \cup \pi(x) \subseteq X_d \cup \pi(d)$. This definition implies that for each decision node d and any node $x \in \pi^*(nond(d) \cap D)$, $\{x\} \cup \pi(x)$ Y_d are m–separated by $\pi(d)$.

Note that a no–forgetting influence diagram is stepwise decomposable. In a stepwise decomposable influence diagram, an arc into a decision node indicates both information availability and functional dependence. More precisely, for any decision node d and any other node x in a stepwise decomposable influence diagram, the presence of an arc $x \rightarrow d$ implies that the value of variable x is available at the time when decision d is to be made, and it is *not known* that the information is *irrelevant* to the decision. On the other hand, the absence of an arc $x \rightarrow d$ in a stepwise decomposable influence diagram implies that either the value of variable x is not available at the time when decision d is to be made, or it is *known* that the information is *irrelevant* to the decision. Thus, one advantage of stepwise decomposability over no-forgetting is that stepwise decomposable influence diagrams can represent the knowledge that a piece of information (carried by a no-forgetting informational arc to a decision node) is irrelevant to the optimal decision function of the decision.

Our method is applicable to regular stepwise decomposable influence diagrams with multiple value nodes. For simplicity of exposition, however, we will first consider regular influence diagrams with exactly one value node. We come to regular influence diagrams with multiple value nodes in Chapter 8.

Chapter 6

Review of Algorithms for Influence Diagram Evaluation

In this chapter, we review related research efforts on influence diagram evaluation, and classify them into two categories. Those in the first category use an intermediate representation and evaluate an influence diagram in two conceptual steps: first transforming the influence diagram into its intermediate representation and then computing an optimal policy from the intermediate representation. Those in the second category compute optimal policies directly from influence diagrams. Our method to be presented in the next chapter belongs to the first category.

6.1 Howard and Matheson's Two–Phase Method

Howard and Matheson's method belongs to the first category. Howard and Matheson [35] discuss a way to transform a regular no–forgetting influence diagram into a decision tree. The transformation involves two steps. An influence diagram is first transformed into a *decision tree network* and then a decision tree is constructed from the decision tree network. An influence diagram is a *decision tree network* if it is regular and no–forgetting, and if all predecessors of each decision node are direct parents of the decision node [35]. The basic operation for transforming a regular, no–forgetting influence diagram into a decision network is *arc reversal* [35, 85].

The *arc reversal* operation is illustrated in Fig. 6.1. Suppose $a \rightarrow b$ is an arc in an influence diagram such that both a and b are random nodes and there is no other directed path from node a to node b. The direction of the arc can be reversed and both nodes inherit each other's parents. This operation is an application of Bayes Theorem. In Fig. 6.1, we begin with conditional probability distributions $P\{b|a,\cdot\}$ and $P\{a|\cdot\}$, and end up with conditional probability distributions $P\{a|b,\cdot\}$ and $P\{b|\cdot\}$. Formally, we have:

$$P\{b|x,y,z\} = \sum_a P\{a,b|x,y,z\} = \sum_a P\{b|a,y,z\} * P\{a|x,y\}$$

$$P\{a|b,x,y,z\} = \frac{P\{a,b|x,y,z\}}{P\{b|x,y,z\}} = \frac{P\{b|a,y,z\} * P\{a|x,y\}}{P\{b|x,y,z\}}.$$

As an example, consider the influence diagram shown in Fig. 6.2, which represents the oil wildcatter problem from [79]. In the problem, an oil wildcatter must decide whether to drill or not to drill an oil well on a particular site. He is not certain whether the site is `dry`, `wet` or `soaking`. Before making this decision, he can order a test on the seismic structure. If he does, the test result will be available to him at the time when the drill decision is to be made. The profit depends on the cost of the test, the amount of oil and the cost of drilling, which can be either low, medium, or high.

In the influence diagram, `T` and `D` are decision variables, corresponding to the test decision and the drill decision. `O`, `S`, `R` and `CD` are random variables, representing the amount of oil, the seismic structure, the test result, and the cost of drilling, respectively.

The two decision variables have the same frame with two alternatives: `yes` and `no`.

The frame of random variable O has three values: dry, wet and soaking. The frame of random variable S has three values: ns for no–structure, cs for close–structure and os for open–structure. The frame of random variable R has four values: ns for no–structure, cs for close–structure, os for open–structure, and nobs for no observation. The frame of random variable CD has three values: l for low, m for medium and h for high.

The influence diagram in Fig. 6.2 is regular and no–forgetting, but is not a decision tree network, since node S and O are two predecessors of node D but they are not parents of D. In order to transform that influence diagram into a decision network, we first reverse the arc $O \rightarrow S$ and then reverse the arc $S \rightarrow R$. The resulting decision network is shown in Fig. 6.3. In the course of this transformation, we have also to compute the new conditional probability distributions for nodes O and S. More specifically, when we reverse the arc $O \rightarrow S$, we need to compute probability distributions $P\{O|S\}$ and $P\{S\}$ from probability distributions $P\{S|O\}$ and $P\{O\}$; when we reverse the arc $S \rightarrow R$, we need to compute probability distributions $P\{R|T\}$ and $P\{S|T, R\}$ from probability distributions $P\{S\}$ and $P\{R|T, S\}$. This operation introduces the arc from T to S.

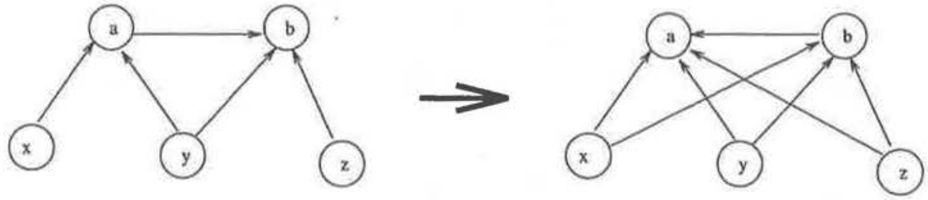

Figure 6.1: An illustration of the arc reversal operation: reversing arc $a \rightarrow b$

A decision tree is constructed from a decision tree network as follows. First, define a total order $\prec$ over the set $C \cup D \cup \{v\}$ satisfying the following three conditions:

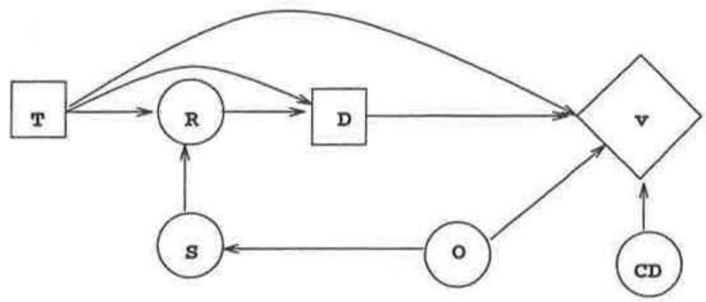

Figure 6.2: An influence diagram for the oil wildcatter's problem

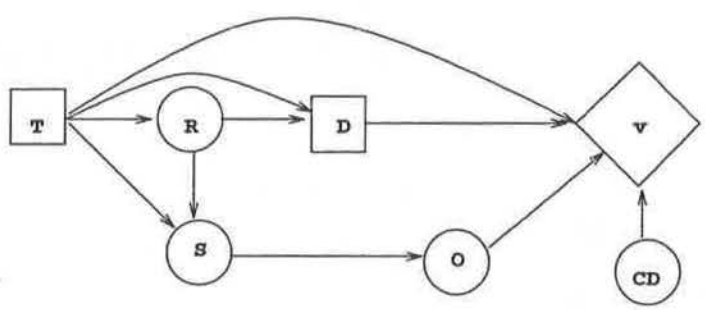

Figure 6.3: A decision tree network derived from the influence diagram for the oil wildcatter's problem

(1) $a \prec b$ if b is the value node; (2) $a \prec b$ if there is a directed path from a to b; (3) $a \prec b$ if a is a decision node and there is no directed path from b to a. Then, construct a decision tree by considering variables one by one in the order. Each layer in the decision tree corresponds to a variable. For the decision tree network in Fig. 6.3, we obtain the following order:

$$\mathrm{T} \prec \mathrm{R} \prec \mathrm{D} \prec \mathrm{S} \prec \mathrm{O} \prec \mathrm{CD} \prec \mathrm{v}$$

Using Howard and Matheson's notation [35], the decision tree for the oil wildcatter problem is shown in Fig. 6.4. In the figure, the boxes correspond to decision variables, circles to random variables, and diamonds to the value node. This is a compact representation of a full decision tree. A layer of the decision tree is indicated by the corresponding variable and its possible values (alternatives). In the case of a random variable, its probability distribution is also included in the layer. The full decision tree can be obtained by systematically expanding each layer and adding necessary connections in the expanded graph. Fig. 6.5 shows a partial decision tree resulting from the expansion of the first two layers, and Fig. 6.6 shows a partial decision tree resulting from the expansion of the first three layers of the compact representation.

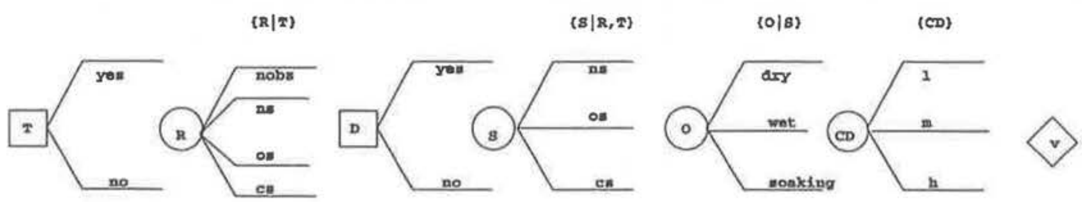

Figure 6.4: A compact representation of the decision tree derived for the oil wildcatter problem

Note that the decision tree here is slightly different from the definition of decision trees (graphs) in Chapter 2. In this decision tree, chance nodes and choice nodes

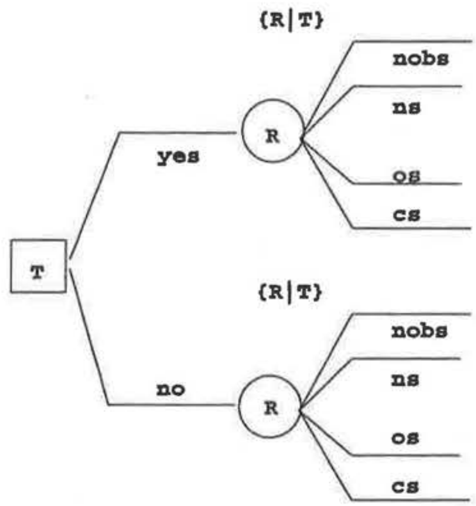

Figure 6.5: A partial decision tree after the expansion of the first two layers

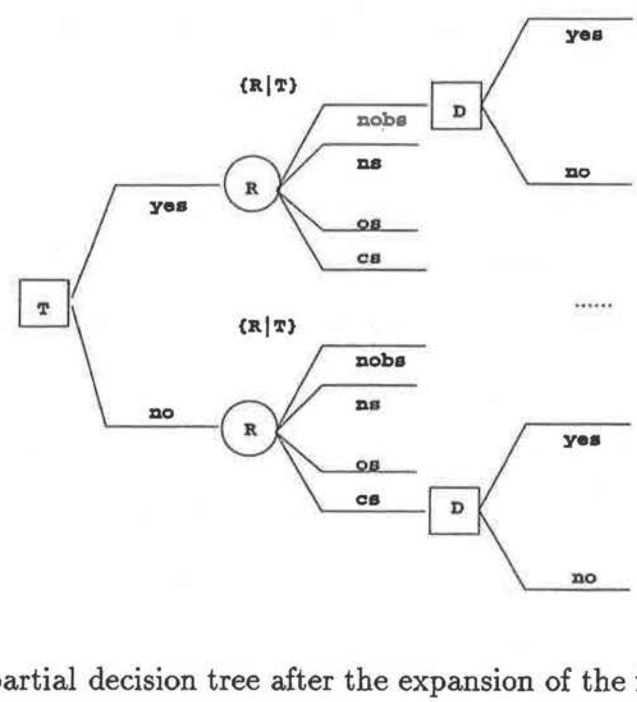

Figure 6.6: A partial decision tree after the expansion of the first three layers

are not strictly interleaved. Nevertheless, the algorithms discussed in Chapter 3 can be easily adjusted for such cases. Moreover, since the algorithms can work on implicit representations of decision trees, it is not necessary to expand the compact representation into a full decision tree. The algorithms can work on the compact representation directly.

The major problem with this approach is that the resultant decision tree tends to be large. The depth of the decision tree so obtained from an influence diagram is equal to the number of variables in the influence diagram. Thus, the size of the decision tree is exponential in the number of variables in the influence diagram.

6.2 Methods for Evaluating Influence Diagrams Directly

The idea of evaluating influence diagrams directly was proposed in [64]. The first complete algorithm for influence diagram evaluation was developed by Shachter [85].

6.2.1 Shachter's algorithm

Shachter's algorithm takes a reduction approach. The algorithm evaluates an influence diagram by applying a series of *value-preserving reductions*. A value-preserving reduction is an operation that can transform an influence diagram into another one with the same optimal expected value.

Shachter identifies four basic value-preserving reductions, namely, *barren node removal, random node removal, decision node removal* and *arc reversal*. The arc reversal operation has been illustrated in the previous section. The other reductions are illustrated as follows.

Barren node removal. A node in an influence diagram is called a *barren node* if it has no child in the diagram. The barren node removal reduction states that any barren node that is not a value node can be removed together with its incoming arcs.

Random node removal. If the value node is the only child of a random node x in an influence diagram, then the node x can be removed by conditional expectation. As a result of this operation, the random node x is removed and the old value node is replaced with a new one that inherits all of the parents of both the old value node and the random node. The reduction is illustrated in Fig. 6.7 where the value function g' of the new value node v' in the resultant influence diagram is given by:

$$g'(a,b,c) = \sum_x g(x,b,c) * P\{x|a,b\}.$$

Decision node removal. A decision node is called a *leaf* decision node if it has no decision node descendant. If a leaf decision node d has the value node v as its only child and $\pi(v) \subseteq \{d\} \cup \pi(d)$, then the decision node can be removed by maximization. The reduction is illustrated in Fig. 6.8 where the value function g' of the new value node v' in the resultant influence diagram is given by:

$$g'(b) = max_d g(d,b).$$

The maximizing operation also results in an optimal decision function δ_d for the leaf decision node through

$$\delta_d(b) = arg\ \ max_d g(d,b).$$

Note that the new value node has the same parent as the old value node. Thus, some of the parents of d may become barren nodes as a result of this reduction. In Fig. 6.8, node a becomes a barren node. The arc from such a node represents information available to the decision maker, but the information has no effect on either the optimal expected value or the optimal policy of the influence diagram. This kind of arc (such as $a \rightarrow d$ in Fig. 6.8) is called an *irrelevant* arc. Irrelevant arcs can be identified and removed in a pre–processing step [106, 109, 111].

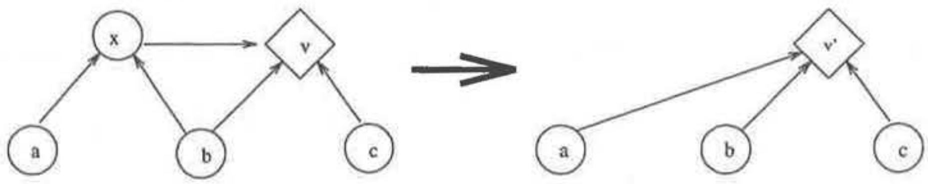

Figure 6.7: An illustration of random node removal: x is removed by expectation

Figure 6.8: An illustration of decision node removal: d is removed by maximization

6.2.2 Other developments

After Shachter's algorithm, research on influence diagram evaluation has advanced in two directions: *making use of Bayesian net evaluation methods*, and *exploiting separability of the value function*. In this section, we discuss the first direction. We come to the second direction in Chapter 8.

Influence diagrams are closely related to Bayesian nets [69]. Quite a few algorithms have been developed in the literature [36, 47, 69, 87, 107] for computing marginal probabilities and posterior probabilities in Bayesian nets. Thus, it is natural to ask whether we can make use of these Bayesian net algorithms for influence diagram evaluation. This problem is examined in [11, 61, 86, 88, 90, 91, 108, 109], and the answer is affirmative.

Recall that a decision function for decision node d in an influence diagram is a mapping from $\Omega_{\pi(d)}$ to Ω_d. It is observed in [11, 86, 88, 108] that given a regular, no-forgetting influence diagram, the optimal policy can be computed by sequentially computing the optimal decision functions for decision nodes, one at a time, starting from the last one backwards. The computation of the optimal decision function of a decision node is independent of those decision nodes that precede the decision node.

Cooper [11] gives a recursive formula for computing the maximal expected values and optimal policies of influence diagrams. To some extent, the formula serves as a bridge between the evaluation of Bayesian nets and that of influence diagrams.

Shachter and Peot show in [88] that the problem of influence diagram evaluation can be reduced to a series of Bayesian net evaluations. To this end, the value node of an influence diagram is first replaced with an "observed" probabilistic *utility* node v' with frame $\{0,1\}$ and a normalized probability distribution. The optimal decision function δ_n for the last decision node d_n can be computed as follows: for each element $e \in \Omega_{\pi(d_n)}$,

$$\delta_n(e) = arg\ \ max_{a \in \Omega_{d_n}} P\{d_n = a, \pi(d_n) = e | v' = 1\}.$$

The optimal decision function δ_i of the decision node d_i is computed after the optimal decision functions δ_{i+1} ..., δ_n have been obtained. The decision nodes d_{i+1}, ..., d_n are first replaced with their corresponding deterministic random-nodes in the influence diagram. The decision function δ_i is then computed as follows: for each element $e \in \Omega_{\pi(d_i)}$,

$$\delta_i(e) = arg\ \ max_{a \in \Omega_{d_i}} P\{d_i = a, \pi(d_i) = e | \delta_{i+1}, ..., \delta_n, v' = 1\}.$$

The problem of influence diagram evaluation is then reduced to a series of problems of computing posterior probabilities in Bayesian nets. Shachter and Peot [88] also point out that an influence diagram can be converted into a *cluster tree*, which is similar to the clique trees [47] of Bayesian nets, and that the problem of evaluating the influence diagram can thus be reduced to evaluating the cluster tree. A similar approach has also been used by Shenoy for his *valuation based systems* [90] and by Ndilikilikesha for evaluating *potential influence diagrams* [61].

Zhang and Poole [108] propose a divide–and–conquer method for evaluating stepwise decomposable influence diagrams. This method is further studied in [109, 106]. Like Shachter and Peot's algorithm, Zhang and Poole's method also deals with one decision node at a time. Unlike Shachter and Peot's algorithm, Zhang and Poole's method takes a reduction approach. Suppose node d is a leaf decision node of a stepwise decomposable influence diagram $\mathcal{I}$. $\pi(d)$ separates the influence diagram into two parts, namely a *body* and a *tail*. The tail is a simple influence diagram with only one decision node (d). The body's value node is a new node whose value function is obtained by evaluating the tail. A reduction step with respect to the decision node d transforms $\mathcal{I}$ to the body. The main computation involved in a reduction step, however, is for evaluating the tail.

Since the tail is a simple influence diagram with only one decision node, its evaluation can be directly reduced to a problem of computing posterior probabilities in a Bayesian net, as suggested in [88, 109]. The result of the evaluation consists of two parts: a value function $g' : \Omega_{\pi(d)} \rightarrow \mathcal{R}$, and an optimal decision function $\delta_d : \Omega_{\pi(d)} \rightarrow \Omega_d$. The same reduction is applicable to the resulting body.

6.2.3 Some common weaknesses of the previous algorithms

One common weakness of the influence diagram evaluation algorithms that we have reviewed is that they fail to provide any explicit mechanism to make use of domain dependent information (e.g., a heuristic function estimating the optimal expected values of influence diagrams), even when it is available for some problems.

Another notable and common shortcoming of these algorithms is inherited from the disadvantage of conventional influence diagrams for asymmetric decision problems. This was also observed in [85].

To be represented by an influence diagram, an asymmetric decision problem must be "symmetrized." This symmetrization results in many "impossible" information states (they have zero probability). The optimal choices for these impossible states need not be computed at all.

Current algorithms for directly evaluating influence diagrams have two weaknesses, stemming from the symmetrization. The first one is that, for each decision node d, they will perform a maximization operation conditioned on each information state in $\Omega_{\pi(d)}$, even though the marginal probabilities of some states are zero. This weakness arises from the fact that these algorithms compute the decision functions in the reverse order of the decision nodes in the influence diagram. At the time of computing the decision function for decision d, the marginal probabilities of the information states in $\Omega_{\pi(d)}$ are not computed yet. The second weakness is that optimal choices for a decision node are chosen from the whole frame Ω_d, instead of from the corresponding effective frames. Thus, it is evident that these algorithms involve unnecessary computation. In the next chapter, we develop an influence diagram evaluation method that avoids such unnecessary computation.

Chapter 7

A Search–Oriented Algorithm

In this chapter, we present our method for influence diagram evaluation. We first formulate influence diagram evaluation as a stochastic dynamic programming problem [80]. Then we give a graphical depiction of the computational structure of the optimal expected value by mapping an influence diagram into a decision graph. Next, we point out how to avoid unnecessary computation in computing the optimal policy and propose a search–oriented approach to influence diagram evaluation. Finally, we analyze how much our method can save in evaluating an influence diagram.

7.1 Preliminaries

A decision node d *directly precedes* another decision node d' if d precedes d' and there is no other decision node d'' such that d precedes d'' and d'' precedes d'. In a regular influence diagram, a decision node can directly precede at most one decision node.

A decision node that is preceded by no other decision node is called a *root* decision node.

Let $\mathcal{I}$ be a regular, stepwise decomposable influence diagram with a single value node v. Suppose the decision nodes in $\mathcal{I}$ are d_1, ..., d_n in the regularity order, then d_1 is the only root decision node and d_n is the only leaf decision node. For each k with $1 \leq k < n$, d_k directly precedes d_{k+1}. Let Y_{d_k} denote the downstream of d_k. Let $\mathcal{I}(d_k, d_{k+1})$ denote the subgraph consisting of d_k, $\pi(d_k)$, $\pi(d_{k+1})$ and those nodes in Y_{d_k} that are not m–separated from d_{k+1} by $\pi(d_{k+1})$. Procedurally, $\mathcal{I}(d_k, d_{k+1})$ can be obtained from $\mathcal{I}$ as follows: (1) remove all nodes that are m–separated from d_k by $\pi(d_k)$, excluding the nodes in $\pi(d_k)$; (2) remove all descendants of d_{k+1}; (3) remove all barren nodes not in $\pi(d_{k+1})$; (4) remove all arcs among nodes in $\pi(d_k) \cup \{d_k\}$ and assign uniform distributions to the root nodes[1] in $\pi(d_k) \cup \{d_k\}$.

$\mathcal{I}(d_k, d_{k+1})$ is called the *sector*[2] of $\mathcal{I}$ from d_k to d_{k+1}. The sector $\mathcal{I}(-, d_1)$ contains only the nodes in $\pi(d_1)$ and those nodes that are not m–separated from d_1 by $\pi(d_1)$. The sector $\mathcal{I}(d_n, -)$ contains those nodes in $\pi(d_n) \cup \{d_n\}$ and those nodes in the downstream Y_{d_n} of d_n.

Note that the sector $\mathcal{I}(-, d_1)$ is a Bayesian net. Furthermore, because $\mathcal{I}$ is stepwise decomposable, d_k is the only decision node in the sector $\mathcal{I}(d_k, d_{k+1})$ that is not in $\pi(d_k)$. Similarly, d_n is the only decision node in the sector $\mathcal{I}(d_n, -)$ that is not in $\pi(d_n)$.

As an example, consider again the oil wildcatter problem. The sector $\mathcal{I}(-, \mathtt{T})$ is empty. The sectors $\mathcal{I}(\mathtt{T}, \mathtt{D})$ and $\mathcal{I}(\mathtt{D}, -)$ are shown in Fig. 7.1.

Let $\delta = (\delta_1, ..., \delta_n)$ be any policy for $\mathcal{I}$. We have the following results on $\mathcal{I}_\delta$.

Lemma 7.1 *For any j, k with $1 \leq j < k \leq n$, the set $\pi(v)$ is independent of $\pi(d_j)$*

[1]The assignment of uniform distributions to the root nodes in $\pi(d_k) \cup \{d_k\}$ is only to make $\mathcal{I}(d_k, d_{k+1})$ a Bayesian network. Since we will only be considering probabilities conditioned on $\pi(d_k) \cup \{d_k\}$, the distributions of those nodes are irrelevant.

[2]Previously, such a part of an influence diagram has been called a *section* [109].

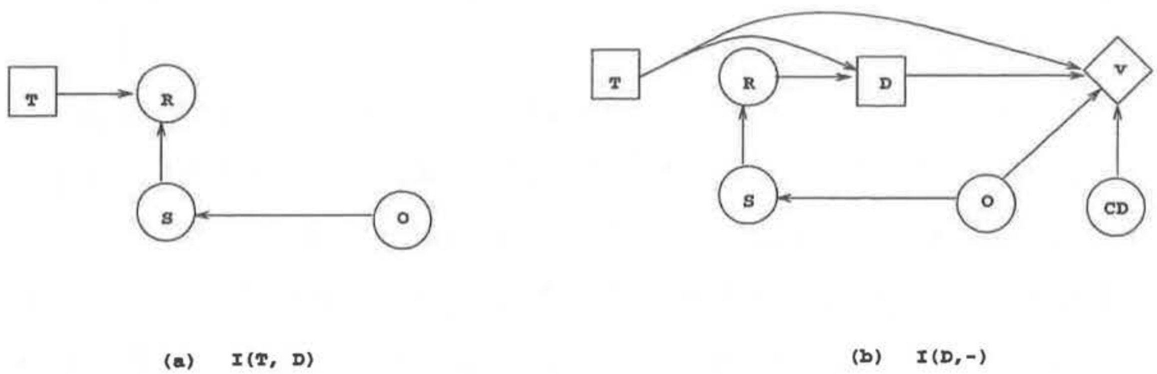

Figure 7.1: Two sectors of the influence diagram for the oil wildcatter problem.

and d_j, *given* $\pi(d_k)$. *Formally, the following relation holds:*

$$P_\delta\{\pi(v) = o|\pi(d_k) = y\} = P_\delta\{\pi(v) = o|\pi(d_k) = y, \pi(d_j) = x, d_j = a\}$$

for any $o \in \Omega_{\pi(v)}$, $y \in \Omega_{\pi(d_k)}$, $a \in \Omega_{d_j}$, *and for any* $x \in \Omega_{\pi(d_j)}$ *consistent with* y *(i.e., the projections of* x *and* y *on the common variable are the same).*

Proof Immediately follows from the m-separation property of a stepwise decomposable influence diagram.

Lemma 7.2 *For any* k *with* $1 \leq k \leq n$, *and any* $o \in \Omega_{\pi(v)}$ *and* $x \in \Omega_{\pi(d_k)}$,

$$P_\delta\{\pi(v) = o|\pi(d_k) = x, d_k = \delta_k(x)\} = P_\delta\{\pi(v) = o|\pi(d_k) = x\}.$$

Proof. Recall that, with respect to a policy $\delta = (\delta_1, ...\delta_n)$, the decision node d_i, for $i = 1, ..., n$, is characterized by the following probability distribution:

$$P\{d_i = x|\pi(d_i) = c\} = \begin{cases} 1 & \text{if } \delta_i(c) = x, \\ 0 & \text{otherwise.} \end{cases}$$

Thus,

$$P_\delta\{\pi(v) = o|\pi(d_k) = x\}$$

$$= \sum_{a \in \Omega(d_k)} P_\delta\{\pi(v) = o | \pi(d_k) = x, d_k = a\} * P_\delta\{d_k = a | \pi(d_k) = x\}$$
$$= P_\delta\{\pi(v) = o | \pi(d_k) = x, d_k = \delta_k(x)\}$$

□

Lemma 7.3 *For any $x \in \Omega_{\pi(d_1)}$, the probability $P_\delta\{\pi(d_1) = x\}$ depends only on nodes in the sector $\mathcal{I}(-, d_1)$, and is independent of δ. Consequently, for any other policy δ',*

$$P_\delta\{\pi(d_1) = x\} = P_{\delta'}\{\pi(d_1) = x\}.$$

Proof. Since all nodes not in $\mathcal{I}(-, d_1)$ are non–ancestors of the nodes in $\pi(d_1)$, they are irrelevant to the marginal probabilities of $\pi(d_1)$. Since there is no decision node in $\mathcal{I}(-, d_1)$, then $P_\delta\{\pi(d_1) = x\}$ is independent of δ. □

Lemma 7.4 (1) *For any $o \in \Omega_{\pi(v)}$, $x \in \Omega_{\pi(d_n)}$ and $a \in \Omega_{d_n}$, the conditional probability $P_\delta\{\pi(v) = o | \pi(d_n) = x, d_n = a\}$ depends only on those nodes in the sector $\mathcal{I}(d_n, -)$, and is independent of δ. In other words, for any other policy δ',*

$$P_\delta\{\pi(v) = o | \pi(d_n) = x, d_n = a\} = P_{\delta'}\{\pi(v) = o | \pi(d_n) = x, d_n = a\}.$$

(2) *For any $y \in \Omega_{\pi(d_{k+1})}$, $x \in \Omega_{\pi(d_k)}$ and $a \in \Omega_{d_k}$, the conditional probability $P_\delta\{\pi(d_{k+1}) = y | \pi(d_k) = x, d_k = a\}$ depends on only those nodes in the sector $\mathcal{I}(d_k, d_{k+1})$, and is independent of δ. In other words, for any other policy δ',*

$$P_\delta\{\pi(d_{k+1}) = y | \pi(d_k) = x, d_k = a\} = P_{\delta'}\{\pi(d_{k+1}) = y | \pi(d_k) = x, d_k = a\}.$$

(3) *Suppose $\delta' = (\delta'_1, ..., \delta'_n)$ is another policy for $\mathcal{I}$ such that $\delta'_1 = \delta_1, ..., \delta'_{k-1} = \delta_{k-1}$ for some k, $1 \le k \le n$, then, for any j, $1 \le j \le k$, and any $x \in \Omega_{\pi(d_j)}$,*

$$P_\delta\{\pi(d_j) = x\} = P_{\delta'}\{\pi(d_j) = x\}.$$

Proof. Follows from the definition of sectors and the m-separation property of a stepwise decomposable influence diagram. □

Lemmas 7.3 and 7.4 indicate that some conditional probabilities in an influence diagram are independent of policies for the influence diagram, and can be computed in a sector of the influence diagram. The computation can be carried out by any well-established algorithms for Bayesian nets. This fact facilitates a clean interface between influence diagram evaluation and Bayesian net evaluation.

7.2 Influence Diagram Evaluation via Stochastic Dynamic Programming

In this section, we establish a stochastic dynamic programming formulation for influence diagram evaluation by studying the relationship among the conditional expected values of influence diagrams.

Let e be any event in $\mathcal{I}_\delta$ and let $E_\delta[v|e]$ be defined as follows:

$$E_\delta[v|e] = \sum_{o \in \Omega_{\pi(v)}} g(o) * P_\delta\{\pi(v) = o|e\}.$$

For each k with $1 \leq k \leq n$, let U_k be a function defined as follows.

$$U_k(x, \delta) = E_\delta[v|\pi(d_k) = x] \tag{7.1}$$

Informally, $U_k(x, \delta)$ is the expected value of the influence diagram with respect to policy δ, conditioned on $\pi(d_k) = x$.

Lemma 7.5 *The expected value of the influence diagram with respect to policy δ can be expressed in terms of U_k as:*

$$E_\delta[v] = \sum_{x \in \Omega_{\pi(d_k)}} U_k(x, \delta) * P_\delta\{\pi(d_k) = x\}.$$

Proof By the definition of $E_\delta[v]$, we have:

$$E_\delta[v] = \sum_{o \in \Omega_{\pi(v)}} g(o) * P_\delta\{\pi(v) = o\}.$$

Since

$$P_\delta\{\pi(v) = o\} = \sum_{x \in \Omega_{\pi(d_k)}} P_\delta\{\pi(v) = o | \pi(d_k) = x\} * P_\delta\{\pi(d_k) = x\},$$

thus,

$$E_\delta[v] = \sum_{o \in \Omega_{\pi(v)}} g(o) * \sum_{x \in \Omega_{\pi(d_k)}} P_\delta\{\pi(v) = o | \pi(d_k) = x\} * P_\delta\{\pi(d_k) = x\}.$$

By changing the order of the two summations, we have:

$$E_\delta[v] = \sum_{x \in \Omega_{\pi(d_k)}} P_\delta\{\pi(d_k) = x\} * \sum_{o \in \Omega_{\pi(v)}} g(o) * P_\delta\{\pi(v) = o | \pi(d_k) = x\}.$$

By the definition of U_k, we have:

$$E_\delta[v] = \sum_{x \in \Omega_{\pi(d_k)}} U_k(x, \delta) * P_\delta\{\pi(d_k) = x\}.$$

□

Lemma 7.6 *The following relation between functions U_k and U_{k-1} holds.*

$$U_{k-1}(x, \delta) = \sum_{y \in \Omega_{\pi(d_k)}} U_k(y, \delta) * P_\delta\{\pi(d_k) = y | \pi(d_{k-1}) = x\}$$

for any $x \in \Omega_{\pi(d_{k-1})}$.

Proof By the definition of U_{k-1}, we have:

$$U_{k-1}(x, \delta) = \sum_{o \in \Omega_{\pi(v)}} g(o) * P_\delta\{\pi(v) = o | \pi(d_{k-1}) = x\}.$$

Since

$$P_\delta\{\pi(v) = o|\pi(d_{k-1}) = x\} =$$
$$\sum_{y\in\Omega_{\pi(d_k)}} P_\delta\{\pi(v) = o|\pi(d_{k-1}) = x, \pi(d_k) = y\} * P_\delta\{\pi(d_k) = y|\pi(d_{k-1}) = x\},$$

then by Lemma 7.1, we have:

$$U_{k-1}(x,\delta) = \sum_{o\in\Omega_{\pi(v)}} g(o) * \sum_{y\in\Omega_{\pi(d_k)}} P_\delta\{\pi(v) = o|\pi(d_k) = y\} * P_\delta\{\pi(d_k) = y|\pi(d_{k-1}) = x\}.$$

By reordering the two summations, we obtain:

$$U_{k-1}(x,\delta) = \sum_{y\in\Omega_{\pi(d_k)}} P_\delta\{\pi(d_k) = y|\pi(d_{k-1}) = x\} * \sum_{o\in\Omega_{\pi(v)}} g(o) * P_\delta\{\pi(v) = o|\pi(d_k) = y\}.$$

By the definition of U_k, we have:

$$U_{k-1}(x,\delta) = \sum_{y\in\Omega_{\pi(d_k)}} U_k(y,\delta) * P_\delta\{\pi(d_k) = y|\pi(d_k) = x\}.$$

□

Lemma 7.7 *Let $\delta' = (\delta'_1, ..., \delta'_n)$ be another policy for $\mathcal{I}$ such that $\delta'_k = \delta_k, ..., \delta'_n = \delta_n$, for some k, $1 \leq k \leq n$. Then, $U_j(x,\delta) = U_j(x,\delta')$ for each j, $k \leq j \leq n$ and each $x \in \Omega_{\pi(d_j)}$.*

Proof By induction.

Basis: Consider U_n. By the definition of U_n, we have:

$$U_n(x,\delta) = \sum_{o\in\Omega_{\pi(v)}} g(o) * P_\delta\{\pi(v) = o|\pi(d_n) = x\}$$

and

$$U_n(x,\delta') = \sum_{o\in\Omega_{\pi(v)}} g(o) * P_{\delta'}\{\pi(v) = o|\pi(d_n) = x\}.$$

By Lemma 7.2, we have:

$$P_{\delta'}\{\pi(v) = o|\pi(d_n) = x\} = P_{\delta'}\{\pi(v) = o|\pi(d_n) = x, d_n = \delta'_n(x)\}$$

and

$$P_{\delta}\{\pi(v) = o|\pi(d_n) = x\} = P_{\delta}\{\pi(v) = o|\pi(d_n) = x, d_n = \delta_n(x)\}.$$

Since $\delta_n = \delta'_n$, then by Lemma 7.4-(1), we have:

$$P_{\delta}\{\pi(v) = o|\pi(d_n) = x, d_n = \delta_n(x)\} = P_{\delta'}\{\pi(v) = o|\pi(d_n) = x, d_n = \delta'_n(x)\}.$$

Thus, $U_n(x,\delta) = U_n(x,\delta')$. Therefore, the basis holds.

Induction: Suppose $U_i(x,\delta) = U_i(x,\delta')$ for all i, $k < i \leq n$. By Lemma 7.6, we have:

$$U_{i-1}(x,\delta) = \sum_{y \in \Omega_{\pi(d_i)}} U_i(y,\delta) * P_{\delta}\{\pi(d_i) = y|\pi(d_{i-1}) = x\}$$

and

$$U_{i-1}(x,\delta') = \sum_{y \in \Omega_{\pi(d_i)}} U_i(y,\delta') * P_{\delta'}\{\pi(d_i) = y|\pi(d_{i-1}) = x\}.$$

By the induction hypothesis, we have:

$$U_i(y,\delta) = U_i(y,\delta').$$

By Lemma 7.2, we have:

$$P_{\delta}\{\pi(d_i) = y|\pi(d_{i-1}) = x\} = P_{\delta}\{\pi(d_i) = y|\pi(d_{i-1}) = x, d_{i-1} = \delta_{i-1}(x)\}$$

and

$$P_{\delta'}\{\pi(d_i) = y|\pi(d_{i-1}) = x\} = P_{\delta'}\{\pi(d_i) = y|\pi(d_{i-1}) = x, d_{i-1} = \delta'_{i-1}(x)\}.$$

Since $\delta_{i-1} = \delta'_{i-1}$, then by Lemma 7.4-(2), we obtain:

$$P_{\delta'}\{\pi(d_i) = y|\pi(d_{i-1}) = x, d_{i-1} = \delta'_{i-1}(x)\} = P_{\delta}\{\pi(d_i) = y|\pi(d_{i-1}) = x, d_{i-1} = \delta_{i-1}(x)\}.$$

Thus,

$$P_{\delta}\{\pi(d_i) = y|\pi(d_{i-1}) = x\} = P_{\delta'}\{\pi(d_i) = y|\pi(d_{i-1}) = x\}.$$

Therefore,

$$U_{i-1}(x,\delta) = U_{i-1}(x,\delta').$$

Therefore, the lemma holds in general. □

So far, we have developed some results on the expected values of an influence diagram with respect to an arbitrary policy. Let us now examine the properties of an optimal policy. Let $\delta^0 = (\delta_1^0, ..., \delta_n^0)$ be an optimal policy for influence diagram $\mathcal{I}$ and let V_k be a function defined as

$$V_k(x) = U_k(x,\delta^0).$$

Intuitively, $V_k(x)$ is the optimal expected value of the influence diagram $\mathcal{I}$ conditioned on $\pi(d_k) = x$. In other words, $V_k(x)$ is the expected value a decision maker can obtain if he starts to make optimal decisions in the situation represented by the event $\pi(d_k) = x$.

Let $V_k'(x,a)$ be an auxiliary function defined as:

$$V_n'(x,a) = \sum_{y\in\Omega_{\pi(v)}} g_v(y) * P_{\delta^0}\{\pi(v) = y|\pi(d_n) = x, d_n = a\} \tag{7.2}$$

$$V_k'(x,a) = \sum_{y\in\Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x, d_k = a\} \tag{7.3}$$

Intuitively, $V_k'(x,a)$ is the optimal expected value of the influence diagram $\mathcal{I}$ conditioned on $\pi(d_k) = x, d_k = a$. In other words, $V_k'(x,a)$ is the expected value a decision maker can obtain if he starts to make decisions in the situation represented by the event $\pi(d_k) = x$, and first chooses a for d_k (in this situation) and then follows an optimal policy for the remaining decisions.

The next two lemmas characterize the relationship between V_k and V_k'.

Lemma 7.8 *For all* $k = 1, ..., n$,

$$V'_k(x, \delta^0_k(x)) = V_k(x).$$

Proof

$$\begin{aligned}
&V'_k(x, \delta^0_k(x)) \\
&\quad = \sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta^0}\{\pi(d_{k+1}) = y | \pi(d_k) = x, d_k = \delta^0_k(x)\} \\
&\quad = \sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta^0}\{\pi(d_{k+1}) = y | \pi(d_k) = x\} \quad \text{by Lemma 7.2} \\
&\quad = \sum_{y \in \Omega_{\pi(d_{k+1})}} U_{k+1}(y, \delta^0) * P_{\delta^0}\{\pi(d_{k+1}) = y | \pi(d_k) = x\} \quad \text{by the definition of } V_k \\
&\quad = U_k(x, \delta^0) \quad \text{by Lemma 7.6} \\
&\quad = V_k(x) \quad \text{by the definition of } V_k.
\end{aligned}$$

□

Lemma 7.9 *For all* $k = 1, ..., n$,

$$V'_k(x, \delta^0_k(x)) \geq V'_k(x, a) \quad \textit{for each } x \in \Omega_{\pi(d_k)} \textit{ and each } a \in \Omega_{d_k}.$$

Proof Suppose the inequality does not hold. Thus, there exist $x_0 \in \Omega_{\pi(d_k)}$ and $a_0 \in \Omega_{d_k}$ such that

$$V'_k(x_0, a_0) > V'_k(x_0, \delta^0_k(x_0)).$$

Construct a policy $\delta' = (\delta'_1, ..., \delta'_n)$ such that $\delta'_i = \delta^0_i$ for all i, $1 \leq i \leq n$, $i \neq k$ and $\delta'_k(x_0) = a_0$, and $\delta'_k(x) = \delta^0_k(x)$, for all $x \in \Omega_{\pi(d_k)}, x \neq x_0$. For any $x \in \Omega_{\pi(d_k)}$, any $a \in \Omega_{d_k}$ and any $y \in \Omega_{\pi(d_{k+1})}$, by Lemma 7.7, we have

$$U_{k+1}(x, \delta') = U_{k+1}(x, \delta^0) = V_{k+1}(x);$$

by Lemma 7.3, we have

$$P_{\delta'}\{\pi(d_k) = x\} = P_{\delta^0}\{\pi(d_k) = x\};$$

by Lemma 7.4, we have

$$P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x, d_k = a\} = P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x, d_k = a\}.$$

We can prove $E_{\delta'}[v] > E_{\delta^0}$ by the following derivation:

$$\begin{aligned}
&E_{\delta'}[v] \\
&\quad = \sum_{x \in \Omega_{\pi(d_k)}} U_k(x, \delta') * P_{\delta'}\{\pi(d_k) = x\} \\
&\quad = \sum_{x \in \Omega_{\pi(d_k)}} (\sum_{y \in \Omega_{\pi(d_{k+1})}} U_{k+1}(y, \delta') * P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x\}) * P_{\delta'}\{\pi(d_k) = x\} \\
&\quad = (\sum_{y \in \Omega_{\pi(d_{k+1})}} U_{k+1}(y, \delta') * P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0\}) * P_{\delta'}\{\pi(d_k) = x_0\} \\
&\qquad + \sum_{x \in \Omega_{\pi(d_k)}, x \neq x_0} (\sum_{y \in \Omega_{\pi(d_{k+1})}} U_{k+1}(y, \delta') * P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x\}) * P_{\delta'}\{\pi(d_k) = x\} \\
&\quad = (\sum_{y \in \Omega_{\pi(d_{k+1})}} U_{k+1}(y, \delta^0) * P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0\}) * P_{\delta^0}\{\pi(d_k) = x_0\} \\
&\qquad + \sum_{x \in \Omega_{\pi(d_k)}, x \neq x_0} (\sum_{y \in \Omega_{\pi(d_{k+1})}} U_{k+1}(y, \delta^0) * P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x\}) * P_{\delta^0}\{\pi(d_k) = x\} \\
&\quad = (\sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0, d_k = \delta'(x_0)\}) * P_{\delta^0}\{\pi(d_k) = x_0\} \\
&\qquad + E_{\delta^0}[v] - (\sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0\}) * P_{\delta^0}\{\pi(d_k) = x_0\} \\
&\quad = (\sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta'}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0, d_k = a_0\}) * P_{\delta^0}\{\pi(d_k) = x_0\} \\
&\qquad + E_{\delta^0}[v] \\
&\qquad -(\sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0, d_k = \delta_0(x_0)) * P_{\delta^0}\{\pi(d_k) = x_0\} \\
&\quad = (\sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x_0, d_k = a_0\}) * P_{\delta^0}\{\pi(d_k) = x_0\}
\end{aligned}$$

$$+E_{\delta^0}[v] - V_k'(x_0, \delta^0(x_0)) * P_{\delta^0}\{\pi(d_k) = x_0\}$$

$$= V_k'(x_0, a_0) * P_{\delta^0}\{\pi(d_k) = x_0\} - V_k'(x_0, \delta^0(x_0)) * P_{\delta^0}\{\pi(d_k) = x_0\} + E_{\delta^0}[v]$$

$$= (V_k'(x_0, a_0) - V_k'(x_0, \delta^0(x_0)) * P_{\delta^0}\{\pi(d_k) = x_0\} + E_{\delta^0}[v]$$

$$> E_{\delta^0}[v]$$

This contradicts the optimality assumption of δ^0. □

The results we have obtained so far are summarized by the following theorem.

Theorem 7.10 *Let $\mathcal{I}$ be a regular and stepwise decomposable influence diagram, let functions V_k and V_k' be defined as before, and let $\delta^0 = (\delta_1^0, ..., \delta_n^0)$ be an optimal policy for $\mathcal{I}$. For any k with $1 \leq k < n$, and $x \in \Omega_{\pi(d_k)}$ and $a \in \Omega_{d_k}$, the following relations hold:*

$$V_k'(x, a) = \sum_{y \in \Omega_{\pi(d_{k+1})}} V_{k+1}(y) * P\{\pi(d_{k+1}) = y | \pi(d_k) = x, d_k = a\} \tag{7.4}$$

$$V_k(x) = V_k'(x, \delta_k^0(x)) = max_{a \in \Omega_{d_k}} V_k'(x, a) \tag{7.5}$$

$$\delta_k^0(x) = arg\ \ max_{a \in \Omega_{d_k}} \{V_k'(x, a)\} \tag{7.6}$$

$$E_{\delta^0}[v] = \sum_{x \in \Omega_{d_1}} V_0(x) * P\{\pi(d_1) = x\}. \tag{7.7}$$

where $P\{\pi(d_1) = x\} = P_{\delta^0}\{\pi(d_1) = x\}$ can be computed in the sector $\mathcal{I}(-, d_1)$ and $P\{\pi(d_{k+1}) = y|\pi(d_k) = x, d_k = a\} = P_{\delta^0}\{\pi(d_{k+1}) = y|\pi(d_k) = x, d_k = a\}$ can be computed in the sector $\mathcal{I}(d_k, d_{k+1})$, both independently of δ^0.

Equations 7.4, 7.5 and 7.6 establish the computational structure of influence diagram evaluation in the form of finite-stage stochastic dynamic programming [80]. They essentially describe an expectation-maximization iteration for computing the optimal policy and the optimal expected value. Equations 7.4 and 7.5 collectively form a variation of Bellman's optimality equation [5] for stochastic dynamic programming.

Theorem 7.10 shows that functions $V_1, ..., V_n$ and $\delta_1^0, ..., \delta_n^0$, as well as $E_{\delta^0}[v]$ can be computed from V_n'. The computation process is similar to the one implied in the recursive formula given in [11]. For an influence diagram, the computation can be roughly divided into two parts: one for computing conditional probabilities in the sectors of the influence diagram and one for the summations and maximizations as specified in Equations 7.4 and 7.5. The definition of V_n' is given in Equation 7.2. The computation of V_n' can be computed from the sector $\mathcal{I}(d_n, -)$.

As shown in the previous example, the sectors of an influence diagram can have some overlap. Consequently, some redundancy may be involved in computing conditional probabilities in different sectors. This problem does not arise for *smooth* influence diagrams [109]. A non–smooth influence diagram can be transformed into an equivalent smooth influence diagram by a series of arc reversal operations [106, 109]. Therefore, we can either deal with an influence diagram directly or first transform the influence diagram into a smooth influence diagram and deal with the smooth one.

It is now fairly clear that the amount of computation involved in the above mentioned process is comparable to that involved in the other algorithms such as those in [88, 108].

In the above mentioned process, the value $\delta_i^0(x)$ will be computed for each d_i and each $x \in \Omega_{\pi(d_i)}$. Thus, unnecessary computation has not been avoided.

7.3 Representing the Computational Structure by Decision Graphs

In this section, we use *decision graphs* to depict the computational structures of the optimal expected values of influence diagrams. For each influence diagram, we can construct a decision graph and define a *max–exp* evaluation function for the decision graph in such a way that the solution graphs of the decision graph correspond to the policies for the influence diagram, and the optimal solution graphs correspond to the optimal policies.

Before we discuss how to construct decision graphs for influence diagrams, we need some terminology.

Let d be a decision variable in an influence diagram. For each $x \in \Omega_{\pi(d)}$, we call assignments of the form $\pi(d) = x$ a *parent situation* for the decision variable d. For each alternative $a \in \Omega_d$, we call assignments of the form $\pi(d_i) = x, d = a$ an *inclusive situation* for the decision variable d. Two situations are *consistent* if the variables common to the two situations are assigned the same values.

For an influence diagram, we define a decision graph in terms of situations. In the graph, a choice node represents a parent situation and a chance node represents an inclusive situation. The following is a specification of such a decision graph.

- A chance node denoting the empty situation is the *root* of the decision graph.
- For each information state $x \in \Omega_{\pi(d_1)}$, there is a choice node, denoting the parent situation $\pi(d_1) = x$, as a child of the root in the decision graph. The arc from the root to the node is labeled with the probability $P\{\pi(d_1) = x\}$.
- Let N be a choice node in the decision graph denoting a parent situation $\pi(d) = x$ for some decision variable d and some $x \in \Omega_{\pi(d)}$. Let $f_d(x)$ be the effective

frame for the decision variable d in the information state x. Then, N has $|f_d(x)|$ children, each being a chance node corresponding to an alternative[3] in $f_d(x)$. The node corresponding to alternative a denotes the inclusive situation $\pi(d) = x, d = a$.

- The node denoting an inclusive $\pi(d_n) = x, d_n = a$ is a terminal in the decision graph, having value $V'_n(x, a)$.

- Let N be a chance node denoting an inclusive situation $\pi(d_{i-1}) = x, d_{i-1} = a$, and let $\mathcal{A}$ be the subset of the parent situations for decision variable d_i which are consistent with $\pi(d_{i-1}) = x, d_{i-1} = a$. Node N has $|\mathcal{A}|$ children, each being a choice node denoting a parent situation in $\mathcal{A}$. The arc from N to the child denoting a parent situation $\pi(d_i) = y$ is labeled with the conditional probability[4] $P\{\pi(d_i) = y | \pi(d_{i-1}) = x, d_{i-1} = a\}$.

In such a decision graph, a choice node represents a situation of the form $\pi(d_i) = x$ for some i and $x \in \Omega_{\pi(d_i)}$. In such a situation, the decision agent needs to decide which alternative among $f_{d_i}(x)$ should be selected for d_i. Thus, the choice node has $|f_{d_i}(x)|$ children, each for an alternative in $f_{d_i}(x)$. The child corresponding to alternative a is a chance node, representing the probabilistic state $\pi(d_i) = x, d_i = a$. From this probabilistic state, one of the information states of d_{i+1} may be reached. The probability of reaching information state y is $P\{\pi(d_{i+1}) = y | \pi(d_i) = x, d_i = a\}$.

As an example, consider the oil wildcatter problem. The probability distributions and the value functions are given in Tables 7.1–7.5. A complete decision graph for the oil wildcatter problem is shown in Fig. 7.2. As we know, a probability is associated

[3]Note that here we are using the effective frame $f_d(x)$ instead of the frame Ω_d.

[4]Note that probabilities of this kind on arcs are not computed unless necessary. This point will be clear in Section 7.4.1.

Table 7.1: The function of the value node

T	D	O	CD	V
no	no	--	--	0
no	yes	dry	l	-40
no	yes	dry	m	-50
no	yes	dry	h	-70
no	yes	wet	l	80
no	yes	wet	m	70
no	yes	wet	h	50
no	yes	soaking	l	230
no	yes	soaking	m	220
no	yes	soaking	h	200

T	D	O	CD	V
yes	no	--	--	-10
yes	yes	dry	l	-50
yes	yes	dry	m	-60
yes	yes	dry	h	-80
yes	yes	wet	l	70
yes	yes	wet	m	60
yes	yes	wet	h	40
yes	yes	soaking	l	220
yes	yes	soaking	m	210
yes	yes	soaking	h	190

Table 7.2: The conditional probability distribution of R

T	S	R	prob
no	--	nobs	1.0
no	--	others	0
yes	--	nobs	0
yes	ns	ns	1.0
yes	cs	cs	1.0
yes	os	os	1.0

with each arc from a chance node to a choice node. These probabilities can be computed in the sector $\mathcal{I}(\texttt{T, D})$ as shown in Fig. 7.1.

In Fig. 7.2, those arcs without labels are associated with zero probability. Non-zero probabilities are computed as follows.

$$P\{\texttt{T=yes,R=ns|T=yes}\}$$
$$= P\{\texttt{R=ns|T=yes}\}$$
$$= P\{\texttt{R=ns|T=yes,S=ns}\} * P\{\texttt{S=ns}\}$$

Table 7.3: The conditional probability distribution of CD

CD	prob
l	0.2
m	0.7
h	0.1

Table 7.4: The prior probability distribution of O

O	prob
dry	0.5
wet	0.3
soaking	0.2

Table 7.5: The conditional probability distribution of S

O	S	prob
dry	ns	0.6
	cs	0.1
	os	0.3
wet	ns	0.3
	cs	0.3
	os	0.4
soaking	ns	0.1
	cs	0.5
	os	0.4

$$+P\{\texttt{R=ns|T=yes,S=os}\} * P\{\texttt{S=os}\}$$
$$+P\{\texttt{R=ns|T=yes,S=cs}\} * P\{\texttt{S=cs}\}$$
$$= 1 * P\{\texttt{S=ns}\} + 0 * P\{\texttt{S=os}\} + 0 * P\{\texttt{S=cs}\}$$
$$= P\{\texttt{S=ns}\}$$

Similarly, we have:

$$P\{\texttt{T=yes,R=os|T=yes}\} = P\{\texttt{S=os}\}$$

and

$$P\{\texttt{T=yes, R=cs|T=yes}\} = P\{\texttt{S=cs}\}.$$

The marginal probabilities of S are computed as follows.

$$\begin{aligned} P\{\texttt{S=ns}\} &= P\{\texttt{S=ns|O=dry}\} * P\{\texttt{O=dry}\} \\ &\quad + P\{\texttt{S=ns|O=wet}\} * P\{\texttt{O=wet}\} \\ &\quad + P\{\texttt{S=ns|O=soaking}\} * P\{\texttt{O=soaking}\} \end{aligned}$$

$$= \quad 0.6 * 0.5 + 0.3 * 0.3 + 0.1 * 0.2 = 0.41$$

Similarly, we can obtain that $P\{\texttt{S=os}\} = 0.35$ and $P\{\texttt{S=cs}\} = 0.24$.

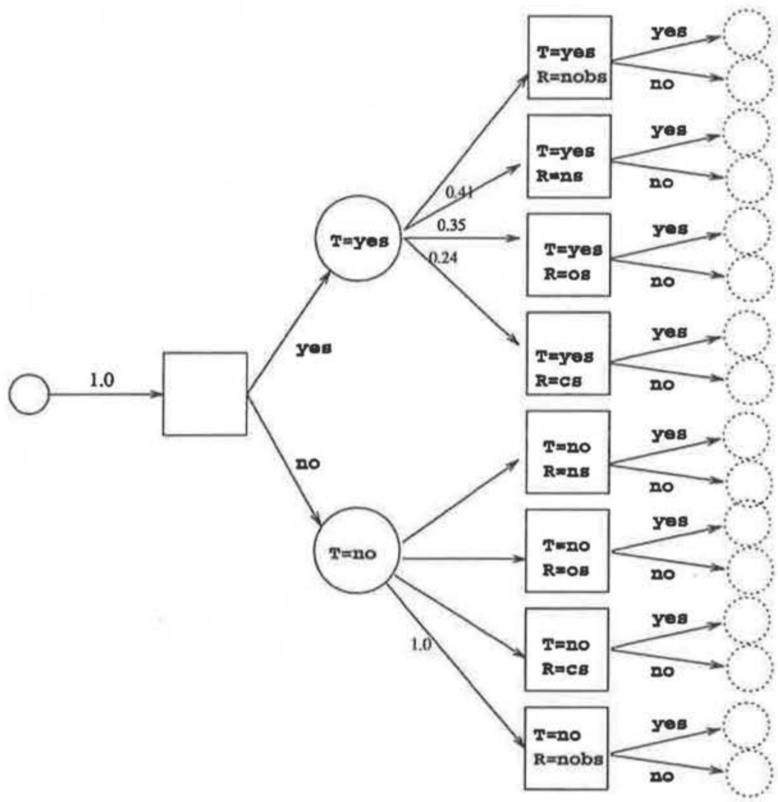

Figure 7.2: A complete decision graph for the oil wildcatter problem

Let DG be a decision graph derived from a regular stepwise-decomposable influence diagram with a single value node. We can define an evaluation function, u_1, on DG as follows:

- If N is a terminal representing a situation $\pi(d_n) = x, d_n = a$, then

$$u_1(DG, N) = V_n'(x, a)$$

- If N is a chance node with children $N_1, ..., N_l$, then

$$u_1(DG, N) = \sum_{i=1}^{l} p(N, N_i) * u_1(DG, N_i)$$

where $p(N, N_i)$ is the probability on the arc from node N to node N_i.

- If N is a choice node with children $N_1, ..., N_l$, then

$$u_1(DG, N) = max_{i=1}^{l}\{u_1(DG, N_i)\}.$$

The following lemma can be easily proved by induction on nodes in the decision graph.

Lemma 7.11 *(1) If N is a choice node representing a parent situation $\pi(d_k) = x$, then $u_1(N) = V_k(x)$.*

(2) If N is a chance node representing an inclusive situation $\pi(d_k) = x, d_k = a$, then $u_1(N) = V'_k(x, a)$.

(3) If N is the root, then $u_1(N)$ is equal to the optimal expected value of the influence diagram.

The correspondence between the optimal policies of the influence diagrams and the optimal solution graphs becomes apparent. As a matter of fact, an optimal solution graph of the decision graph can be viewed as a representation of decision tables in which all the unreachable situations are removed [111].

7.4 Computing the Optimal Solution Graph

As we discussed in Chapter 3, an optimal solution graph of a decision graph can be computed either "bottom-up" or "top-down." If a bottom-up approach is taken, the values of the leaves are computed first. The max-exp values of interior nodes can be computed when the max-exp values of all children of the node are available. The max–exp value of a choice node denoting a parent situation $\pi(d) = x$ is computed by a maximization operation ranging over the children of the choice node, each corresponding to an alternative in the effective frame $f_d(x)$. As a result of this operation, an optimal choice for the decision variable d is also determined for the information state x. The computational complexity of this process is linear in the size of the decision graph[5]. This method cannot avoid computing optimal choices for decision nodes with respect to impossible states.

7.4.1 Avoiding unnecessary computation

We observe that the asymmetry of an influence diagram is reflected by arcs with zero probability in the corresponding decision graph. As we know, the value of a chance node in a decision graph is the weighted sum of the values of its children. If the probability on the arc to a child is known in advance to be zero, then there is no need to compute the value of the child (as far as this chance node is concerned). In case the probabilities on the arcs to a choice node are all zero, the value of the node will never be required. Thus, we need not compute the max–exp value of, and the optimal choice for, the node. In this case, the probability of the information state denoted by the choice node is zero. Thus, it is equivalent to say we need not compute the optimal choice for the corresponding decision variable for the impossible information state. One way to avoid such computation for the impossible states is by pruning those choice nodes corresponding to impossible states. The following procedure for generating a decision graph for an influence diagram effectively realizes this objective:

(1) generate the root node and put it into set G;

[5]Note that the size of the decision graph is normally exponential in the size of the decision node's parents. See the analysis in the next section.

(2) if G is empty then stop;

(3) pick a node N from G and set G to $G - \{N\}$;

(4) if N is a terminal node then go to (2);

(5) generate the child set $C(N)$ of node N; if N is a choice node, set G to $G \cup C(N)$, otherwise let $C'(N)$ be the subset of $C(N)$ such that the probabilities of the arcs from N to the nodes in $C'(N)$ are non-zero; set G to $G \cup C'(N)$;

(6) goto (2);

The above procedure will not expand a choice node unless its probability is not zero. Thus, the procedure effectively ignores subtrees rooted at nodes corresponding to impossible states. Thus, the computation (for computing optimal choices for choice nodes and for computing the conditional probabilities) involved in the subtrees is totally avoided.

Consider again the decision graph for the oil wildcatter problem shown in Fig. 7.2. In the figure, those arcs without labels have zero probability. If we apply the above procedure to the influence diagram of the problem, we obtain the simpler decision graph shown in Fig. 7.3. With this decision graph, we no longer need to compute optimal choices for decision D with respect to impossible states such as `T=no,R=ns`.

Values for the other terminals can be computed locally in the sector $\mathcal{I}(\mathtt{D}, -)$ (as shown in Fig. 7.1) as follows. The solution graph in Fig. 7.4 corresponds to the policy of no test and drill. The expected value of the influence diagram with respect to the policy is 40, which is the optimal expected value of the influence diagram. Thus, the solution graph corresponds to an (the) optimal policy of the influence diagram.

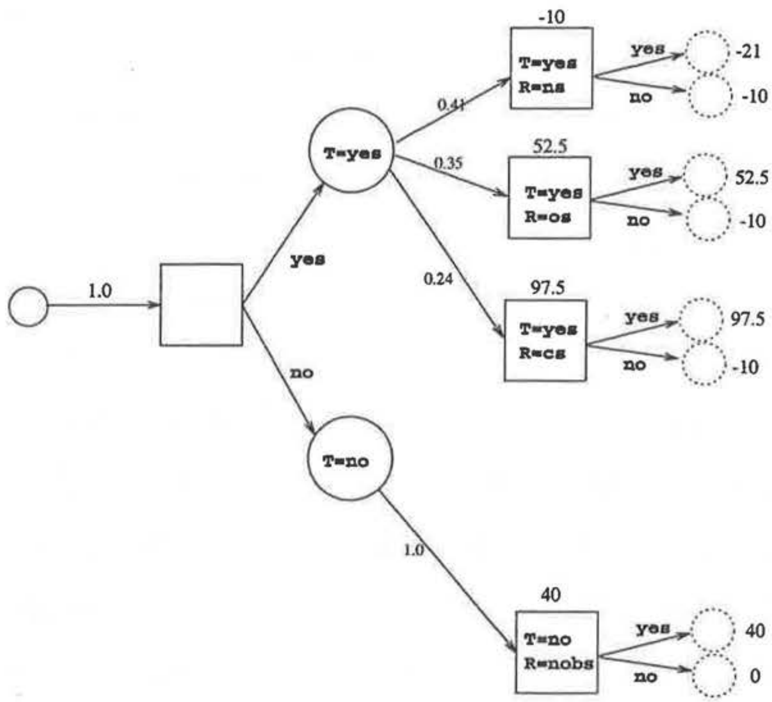

Figure 7.3: A simplified decision graph

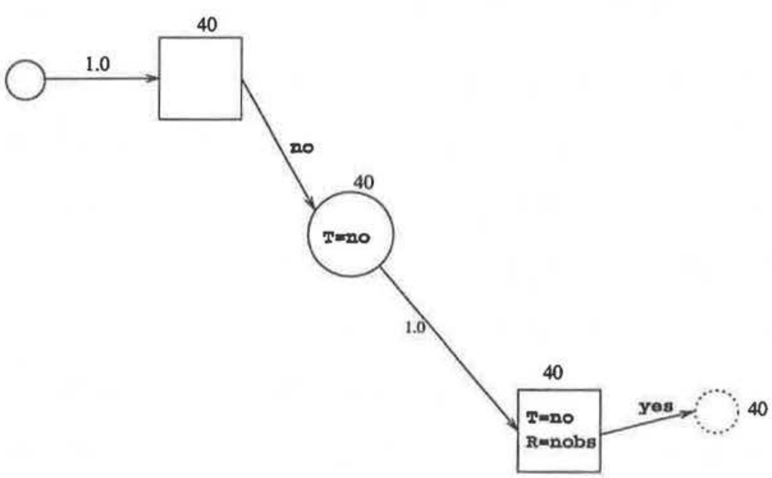

Figure 7.4: The optimal solution graph for the oil wildcatter problem

7.4.2 Using heuristic algorithms

The method just described effectively exploits asymmetry in decision problems. Better performance will be achieved if the algorithms presented in Chapter 3 are used for computing an optimal solution graph. By using these algorithms, some subgraphs may not be processed at all.

To use these algorithms, we need a heuristic function that gives admissible estimation on $u_1(s)$ for any situation s. Note that the notion of admissibility for a heuristic function here is different from the traditional one. Because we are maximizing merit instead of minimizing cost, we define a heuristic function to be admissible if it never under–estimates for any situation s. Formally, a function h is admissible if $h(s) \geq u_1(s)$ for any situation s. An obvious admissible heuristic function for an influence diagram evaluation problem is the one returning $+\infty$ for each situation. Performance can be further enhanced if we can obtain a more *informed* heuristic function by using domain–dependent knowledge.

7.4.3 A comparison with Howard and Matheson's method

The relationship between our method and Howard and Matheson's should now be clear. In both methods, an influence diagram is first transformed into a secondary representation from which an optimal policy is computed. However, there are a few notable differences between the two methods.

First, Howard and Matheson's method works only for no–forgetting influence diagrams while ours is applicable to regular stepwise decomposable influence diagrams (we handle influence diagrams with multiple value nodes in the next chapter).

Second, the sizes of the secondary representation generated by the two methods are different. For a given influence diagram, the depth of the decision tree obtained by

Howard and Matheson's method is equal to the number of variables in the influence diagram, while the depth of the decision graph obtained by our method is $2n$, where n is the number of decision variables in the influence diagram. Typically, there are more random variables than decision variables in a decision problem, thus the depth of the decision tree obtained by Howard and Matheson's method is larger than the depth of the decision graph obtained by ours for the same influence diagram. Furthermore, the number of nodes in the decision tree obtained by Howard and Matheson's method from an influence diagram is exponential in the depth of the tree, but this is not necessarily true for the decision graph obtained by our method. In fact, the number of nodes in a decision graph obtained by our method is:

$$1 + |\Omega_{\pi(d_n)}| + \sum_{i=1}^{n-1}(|\Omega_{\pi(d_i)}| + |\Omega_{\pi(d_i)}| * |\Omega_{d_i}|).$$

Third, our method provides a clean interface to those algorithms developed for Bayesian net evaluation.

7.5 How Much Can Be Saved?

In this section, we first give a general analysis of how much can be saved by exploiting asymmetry in a typical decision problem and then examine the used car buyer problem. Since the number of optimal choices to be computed is a relative measure on the time an algorithm takes for evaluating an influence diagram, we compare the number of optimal choices to be computed by our method against other methods.

7.5.1 An analysis of a class of problems

In this subsection, we analyze our algorithm against the following generalized buying problem:

> Suppose we have to decide whether to buy an expensive and complex item. Before making the decision, we can have n tests, denoted by $T_1, ..., T_n$, on the item. Suppose test T_i has k_i alternatives and j_i possible outcomes for $i = 1, ..., n$.. Among the k_i alternatives for test T_i, one stands for the option of no–testing. Correspondingly, among the j_i possible outcomes of test T_i, one stands for no observation, resulting from the no–testing alternative.

An influence diagram for this problem is shown in Fig. 7.5. In this influence diagram, decision variable T_i has a frame of size k_i, including an alternative `nt` for not testing, and random variable R_i has a frame of size j_i, including an outcome `nr` for no observation. Let $H_i = k_i j_i$. The size of $\Omega_{\pi(T_i)}$ is $\Pi_{l=1}^{i-1} H_l$. Thus, the decision T_i has $\Pi_{l=1}^{i-1} H_l$ information states, the decision B has $\Pi_{l=1}^{n} H_l$ information states. If we do not exploit the asymmetry of the problem, we will have to compute an optimal choice for every decision and each of its information states. The total number is $\sum_{i=1}^{n+1} \Pi_{l=1}^{i-1} H_l$.

Let us consider for how many information states our method will compute optimal choices for each decision variable. To do so, we simply count the number of choice nodes in the decision graph (actually, decision tree) generated by our method from the influence diagram.

Before counting, let us first make some notes on the conditional probabilities we will use in the process of decision graph construction. During the process, we need the conditional probabilities $P\{\pi(d_{i+1})|\pi(d_i), d_i\}$, where d_i denotes T_i for $i = 1, ..., n$ and d_{n+1} denotes B, the purchase decision. First, because $\pi(d_{i+1}) = \pi(d_i) \cup \{d_i, R_i\}$, we have:

$$P\{\pi(d_{i+1})|\pi(d_i), d_i\} = P\{\pi(d_i), d_i, R_i|\pi(d_i), d_i\} = P\{R_i|\pi(d_i), d_i\}.$$

Second, for any given information state $y \in \Omega_{\pi(d_i)}$ with $1 \leq i \leq n$, the total number of test-choice/test-result combinations is $k_i j_i$. Among these combinations, some have zero probability, as illustrated by the partial decision tree shown in Fig. 7.6, where shadowed boxes correspond to the zero probability combinations. The number of zero probability condition/outcome combinations determines the degree of asymmetry of the problem. Let $J_{ir}(y)$ denote the set of possible outcomes of R_i if the choice for T_i in situation $\pi(d_i) = y$ is $a_r \in \Omega_{T_i}$. In terms of probability distributions, this is equivalent to saying that, for all $y \in \Omega_{\pi(d_i)}$, for all $a_r \in \Omega_{d_i}$ and for all $x \in \Omega_{R_i} - J_{ir}(y)$

$$P\{R_i = x | \pi(d_i) = y, d_i = a_r\} = 0$$

and for all $y \in \Omega_{\pi(d_i)}$ and for all $a_r \in \Omega_{d_i}$

$$\sum_{x \in J_{ir}(y)} P\{R_i = x | \pi(d_i) = y, d_i = a_r\} = 1.$$

Thus, the total number of non-zero probability combinations of test-choice/test-result, conditioned on $\pi(d_i) = y$, is bounded from above by $\sum_{r=1}^{k_i} |J_{ir}(y)|$. Let $h_i(y) = \sum_{r=1}^{k_i} |J_{ir}(y)|$ and $h_i = \max_y h_i(y)$.

At the most conservative extreme, we can assume that for any $y \in \Omega_{\pi(d_i)}$, there exist some $a \in \Omega_{d_i}$ and some $x \in \Omega_{R_i}$ such that

$$P\{R_i = x | \pi(d_i) = y, d_i = a\} = 0.$$

In this case, the total number of non-zero probability combinations of test-choices/test-results is bounded from above by $k_i j_i - 1$.

A reasonable assumption we can make about a practical decision problem is that, for any decision variable, the choice of do-nothing will always lead to no result while a choice other than do–nothing will always lead to some results. We call this assumption the "no action, no result" assumption. For the generalized buying problem, this assumption means that no-test choice nt for decision node T_i will lead to no observation nr as the only possible outcome of R_i, and other test choices will not lead to no observation nr. The partial decision tree in Fig. 7.6 depicts this case, where shaded nodes correspond to the states with zero probability. In terms of probability distributions, we have:

$$P\{R_i = x|\pi(d_i) = y, d_i = \mathtt{nt}\} = 0$$

$$P\{R_i = \mathtt{nr}|\pi(d_i) = y, d_i = \mathtt{nt}\} = 1$$

for any $x \in \Omega_{R_i}, x \neq \mathtt{nr}$, and any $y \in \Omega_{\pi(d_i)}$, and

$$P\{R_i = \mathtt{nr}|\pi(d_i) = y, d_i = a\} = 0$$

for any $a \in \Omega_{d_i}, a \neq \mathtt{nt}$, and $y \in \Omega_{\pi(d_i)}$. In this case, the total number of non–zero probability combinations of test–choices/test–results is bounded from above by $1 + (k_i - 1)(j_i - 1)$.

Now, let us count the number of choice nodes in the decision graph that correspond to information states with non–zero probabilities. Based on the above analysis of the probability distributions, we have the following: T_1 has one parent situation; T_2 has at most h_1 parent situations with non–zero probabilities; T_2 has at most $h_1 h_2$ parent situations with non–zero probabilities; etc., and B has at most $\Pi_{l=1}^{n} h_l$ parent situations with non–zero probabilities. Thus, the total number of choice nodes in the decision graph is bounded from above by $\sum_{i=1}^{n+1} \Pi_{l=1}^{i-1} h_l$.

Let $\rho_i = H_i/h_i$. We call ρ_i the "savings factor" with respect to decision T_i. Since $h_i \leq 1 + (k_i - 1)j_i \leq H_i$, we have $\rho_i \geq (k_i j_i)/(1 + (k_i - 1)j_i) \geq 1$.

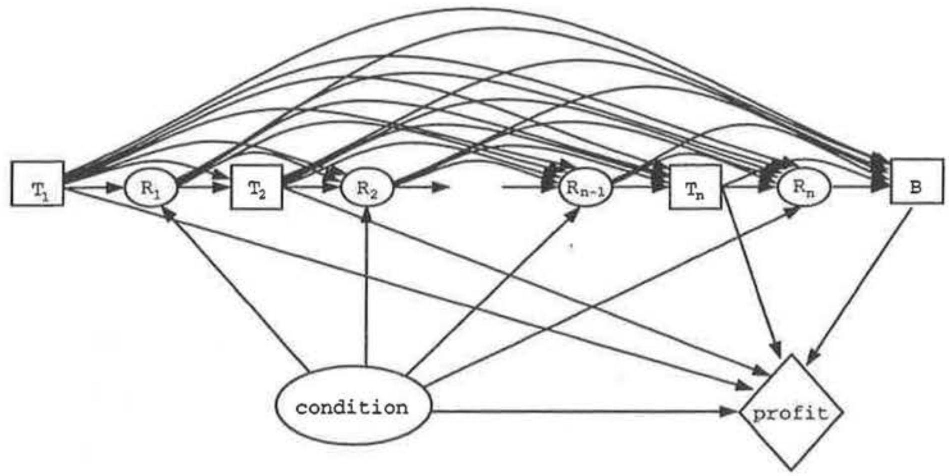

Figure 7.5: An influence diagram for a generalized buying problem

The overall savings factor is given by

$$\rho = \frac{\sum_{i=1}^{n+1} \Pi_{l=1}^{i-1} H_l}{\sum_{i=1}^{n+1} \Pi_{l=1}^{i-1} h_l} \geq \frac{\Pi_{l=1}^{n} H_l}{\sum_{i=1}^{n+1} \Pi_{l=1}^{i-1} h_l} = \frac{\Pi_l^n \rho_l}{\sum_{i=1}^{n+1} \frac{1}{\Pi_{l=i}^{n} h_l}}.$$

It is reasonable to assume that $h_i \geq 2$ for $i = 1, ..., n$. Then we have $\rho \geq \frac{\Pi_{i=1}^{n} \rho_i}{2}$.

For example, suppose $k_i = 4$ (each test has three alternatives plus the alternative of no test) and $j_i = 4$ for $i = 1, ..., n$. Then we have $\rho_i \geq 16/15$ in the most conservative extreme, and $\rho_i \geq 16/10$ under the "no action, no result" assumption. Then, the overall savings factor is bounded from below by $\frac{(16/15)^n}{2}$ in the most conservative extreme, and by $\frac{(16/10)^n}{2}$ under the "no action, no result" assumption.

In the above analysis, we assume that test T_i has exactly k_i alternatives in every information state. In fact, the set of legitimate alternatives for test decision T_i in an information state s is $f_{T_i}(s)$ and may have fewer than k_i elements. Thus, the actual overall savings factor could be much greater than $\Pi_{i=1}^{n} \rho_i / 2$.

Finally, let us note that the above analysis is applicable to any decision problem: as long as it satisfies the "no action, no result" assumption, exploiting asymmetry

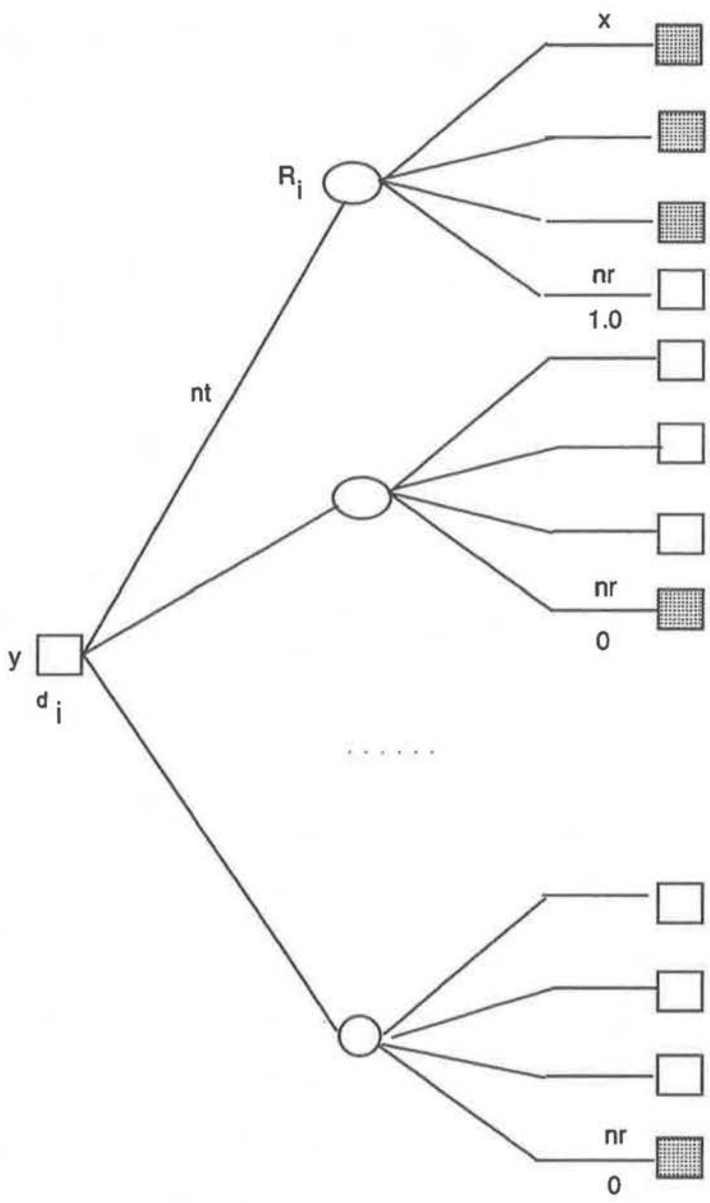

Figure 7.6: An illustration of the asymmetry of a decision variable

will lead to a savings factor that is exponential in the number of asymmetric decisions in the decision problem.

It is worth pointing out that an exponential savings factor for an exponential problem may not change the exponential nature of the problem. More specifically, suppose that problem P will take algorithm A $O(c^n)$ time units and take algorithm B $O(c/\alpha)^n$ time units with $\alpha > 1$. We say that algorithm B has an exponential savings factor in comparison algorithm A because algorithm B runs α^n times faster than algorithm A. Although algorithm B is still an exponential algorithm, for a fixed time resource, algorithm B may be able to handle larger problems than algorithm A. Let n_a and n_b be the size of the largest problems algorithms A and B can handle, respectively. Then n_a and n_b satisfy:

$$c^{n_a} = (c/\alpha)^{n_b}$$

or

$$\frac{n_b}{n_a} = \frac{\log c}{\log c - \log \alpha}.$$

For example, let us consider again the generalized buying problem. Suppose that each decision in the problem has two alternatives and each random node has three outcomes, i.e., $k_i = 2$ and $j_i = 3$ for $i = 1, .., n$. It takes those algorithms that do not exploit asymmetry $O((2*3)^n)$ time units and it takes our algorithm $O(3^n)$ time units. In this case, we have $c = 6$ and $\alpha = 2$. Thus, we have $n_b = n_a \log 6/\log 3 = 1.63 n_a$. Note that for the same α, the bigger the value of c, the smaller the ratio n_b/n_a.

7.5.2 A case analysis of the used car buyer problem

When applying our algorithm to the used car buyer problem, a decision tree as shown in Fig. 7.7 will be generated. In the tree, the leftmost box represents the only state in which the first test decision is to be made. The boxes in the middle column correspond to the information states in which the second test decision is to be made. Similarly, the boxes in the right column correspond to the information states in which the purchase decision is to be made. From the figure we can see that among those nodes corresponding to the information states of the second test, all but two have only one child because the second test in the corresponding information states has only a single legitimate alternative — no–test. Making use of the framing function this way is equivalent to six prunings, each cutting a subtree under a node corresponding to an information state of the second test. The shadowed boxes correspond to the impossible states. Our algorithm effectively detects these impossible states and prunes them when they are created. Each pruning amounts to cutting a subtree under the corresponding node. Consequently, our algorithm does not compute optimal choices for a decision node for those impossible states. For the used car buyer problem, our algorithm computes optimal choices for the purchase decision for only 12 information states, and optimal choices for the second test for only 8 information states (among which six can be computed trivially). These constitute the minimal information state set one has to consider in order to compute an optimal policy for the used car buyer problem. This suggests that, as far as decision making is concerned, our method exploits asymmetry to the maximum extent. In contrast, those algorithms that do not exploit asymmetry will compute the optimal choices for the purchase decision for 96 ($4 \times 4 \times 2 \times 3$) information states and will compute the optimal choices for the second test for 16 information states. The overall savings factor is 5.5.

By applying the analysis results we obtained in the previous section to this problem, we have: $H_1 = 16$, $H_2 = 6$, $h_1 = 8$, and $h_2 = 3$. Thus, $\rho_1 = 2$ and $\rho_2 = 2$. $\rho = (1 + 16 + 96)/(1 + 8 + 24) = 3.7$.

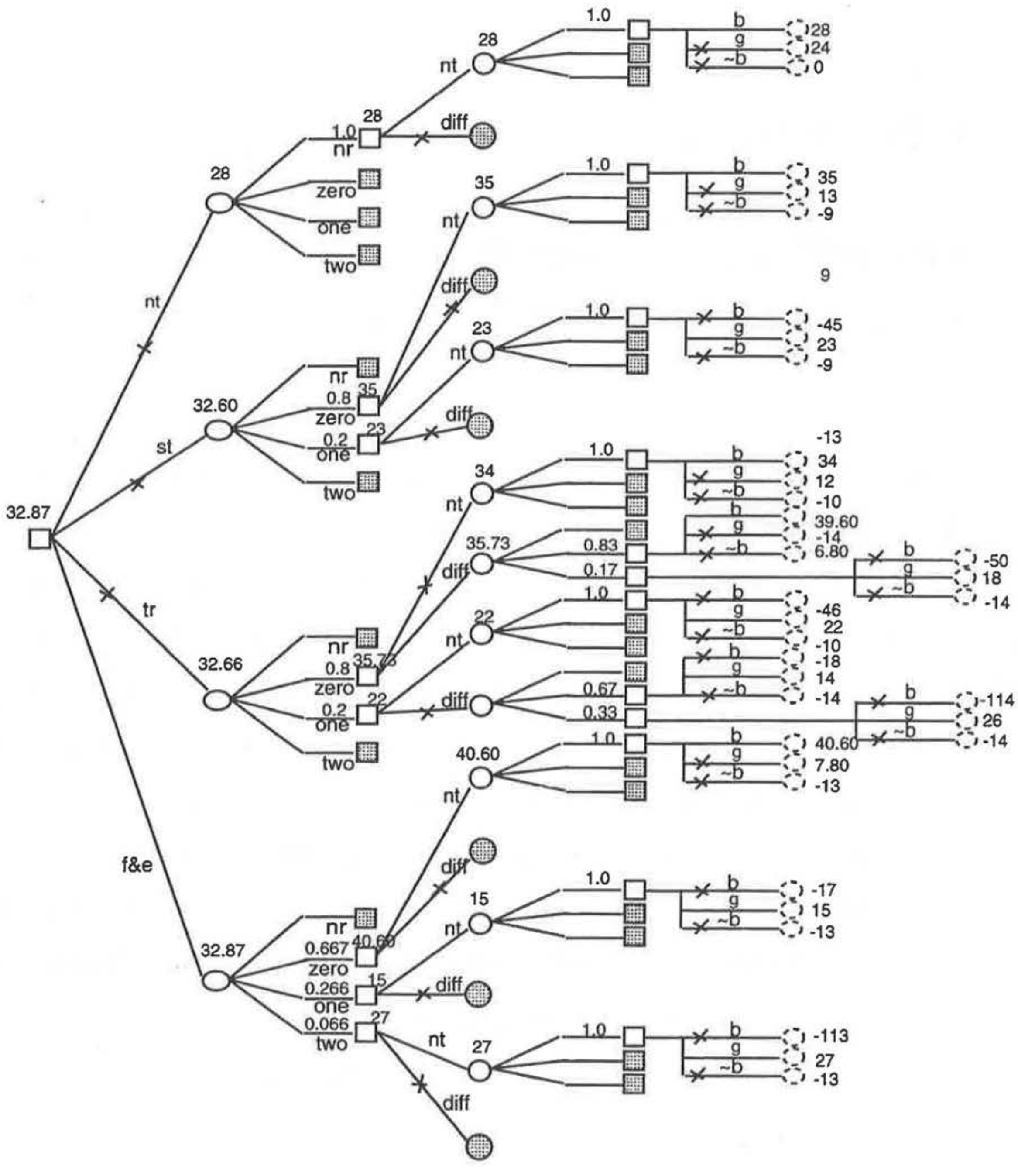

Figure 7.7: A decision graph (tree) generated for the used car buyer problem

Chapter 8

Handling Influence Diagrams with Multiple Value Nodes

In the previous chapter, we developed a method for influence diagram evaluation. We assumed that the influence diagrams were regular and had one value node. As pointed out in [101], if the value function of an influence diagram is separable, then the separable nature can be exploited to increase the efficiency of influence diagram evaluation. The separability of a value function can be represented by multiple value nodes. Thus, it is desirable to develop algorithms that can be used to evaluate influence diagrams with multiple value nodes.

A generalization of Shachter's algorithm [85] has been developed in [101] that can exploit the separability of value functions. Zhang and Poole's algorithm [108] and the later study [109, 106] is developed for influence diagrams with multiple value nodes. In this chapter, we generalize the method presented in the previous chapter so that it applies to regular influence diagrams with multiple value nodes as well.

8.1 Separable Value Functions

If the value function of the value node in an influence diagram can be expressed as the sum of two or more functions with fewer variables, we say the value function is *separable*. More precisely, let $g(Z)$ be a value function with variable set Z, let g_1, ..., g_q be functions with variable sets Z_1 ... Z_q respectively, where Z_1 ... Z_q are all proper subsets of Z. Function g is separable if

$$g(Z) = \sum_{i=1}^{q} g_i(Z_i).$$

Consider again the oil wildcatter problem. The value node depends on four variables: T, D, O and CD. The value function can be separated into three parts: a function g_1 on the cost of the seismic structure test, a function g_2, on the drilling cost, and a function g_3 on the value of the oil. Formally, this can be expressed as

$$g(\mathtt{T},\ \mathtt{D},\ \mathtt{O},\ \mathtt{CD}) = g_1(\mathtt{T}) + g_2(\mathtt{D},\ \mathtt{CD}) + g_3(\mathtt{D},\ \mathtt{O}).$$

With this separation of the value function, the value node can be *split* into three value nodes, v_1, v_2 and v_3, with g_1, g_2 and g_3 as their value functions respectively. This separation results in a new influence diagram as shown in Fig. 8.1.

The semantics of influence diagrams with multiple value nodes is the same as that of influence diagrams with a single value node except that, for an influence diagram with multiple value nodes, we interpret the sum of the values of all individual value nodes as the value of the influence diagram. Therefore, all the terminology we have used before can be used here for influence diagrams with multiple value nodes.

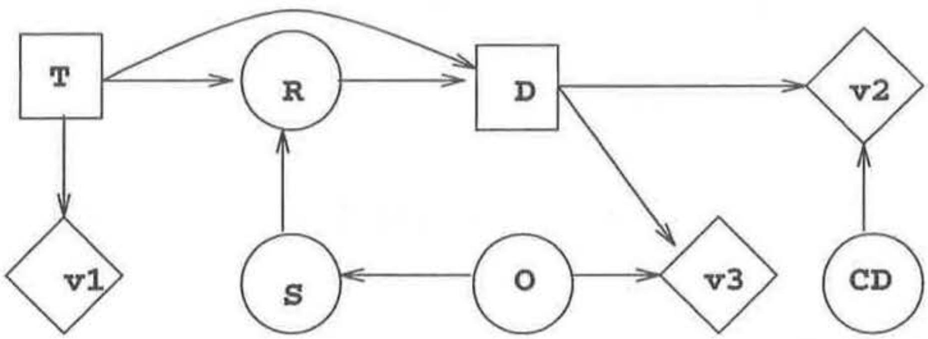

Figure 8.1: A new representation of the oil wildcatter problem by an influence diagram with multiple value nodes

8.2 Decision graphs for influence diagrams with multiple value nodes.

As for influence diagrams with a single value node, the computational structure of the optimal expected value of an influence diagram with multiple value nodes can also be represented as a decision graph. The difference is that in the case of a single value node, the values are associated only with the terminals in the decision graph while in the case for influence diagrams with multiple value nodes, values can be associated with arcs as well. In order to illustrate this, consider the influence diagram shown in Fig. 8.1, which is an influence diagram with three value nodes for the oil wildcatter problem. Since the value node v_1 depends only on node T, its value can be determined in any situation where the variable T is instantiated. Thus, at the nodes representing the situations T=yes and T=no, the value of v_1 can be determined. In particular, the value of g_1 for the situation T=yes is -10 and the value of g_1 for the situation T=no is 0. The value -10 can be viewed as the value resulting directly from performing the test action, while the value 0 can be viewed as the value resulting directly from performing the no-test action.

In order to deal with general cases, we introduce a new concept. Let $\pi(d_k) = x$ be a parent situation for d_k, $\pi(d_k) = x, d_k = a$ be an inclusive situation for d_k and let $\mathcal{I}(d_k, d_{k+1})$ be the sector of $\mathcal{I}$ from d_k to d_{k+1}. Without loss of generality, suppose nodes $v_i, \ldots, v_j$ are the value nodes in $\mathcal{I}(d_k, d_{k+1})$. For $i \leq l \leq j$, let $E_{\mathcal{I}(d_k,d_{k+1})}[v_l|\pi(d_k) = x, d_k = a]$ denote the expected value of the value node v_l, conditioned on $\pi(d_k) = x, d_k = a$, in sector $\mathcal{I}(d_k, d_{k+1})$. We call the sum

$$\sum_{l=i}^{j} E_{\mathcal{I}(d_k,d_{k+1})}[v_l|\pi(d_k) = x, d_k = a]$$

the *value of the inclusive situation* $\pi(d_k) = x, d_k = a$. This value can be computed by a method in [109]. Intuitively the value can be viewed as the utility directly resulting from selecting a as the choice for decision node d_k in the parent situation $\pi(d_k) = x$. Using the terminology for decision graphs, the value can be associated with the arc from the choice node representing the parent situation $\pi(d_k) = x$ to the chance node representing the inclusive situation $\pi(d_k) = x, d_k = a$. The following is a new specification of the decision graph of an influence diagram with multiple value nodes.

- A chance node denoting the empty situation is the *root* of the decision graph.
- For each information state $x \in \Omega_{\pi(d_1)}$, there is a choice node, denoting the parent situation $\pi(d_1) = x$, as a child of the root in the decision graph. The arc from the root to the node is labeled with the probability $P\{\pi(d_1) = x\}$.
- Let N be a choice node in the decision graph denoting a parent situation $\pi(d) = x$ for some decision variable d and some $x \in \Omega_{\pi(d)}$. Let $f_d(x)$ be the effective frame for the decision variable d in the information state x. Then, N has $|f_d(x)|$ children, each being a chance node corresponding to an alternative

in $f_d(x)$. The node corresponding to alternative $a \in f_d(x)$ denotes the inclusive situation $\pi(d) = x, d = a$. The arc from the choice node to the chance node denoting the inclusive situation $\pi(d) = x, d = a$ is labeled with the value of the inclusive situation.

- The node denoting an inclusive $\pi(d_n) = x, d_n = a$ is a terminal in the decision graph having value 0.
- Let N be a chance node denoting an inclusive situation $\pi(d_{i-1}) = x, d_{i-1} = a$, and let $\mathcal{A}$ be the subset of the parent situations for decision variable d_i which are consistent with $\pi(d_{i-1}) = x, d_{i-1} = a$. Node N has $|\mathcal{A}|$ children, each being a choice node denoting a parent situation in $\mathcal{A}$. The arc from N to the child denoting a parent situation $\pi(d_i) = y$ is labeled with the conditional probability $P\{\pi(d_i) = y | \pi(d_{i-1}) = x, d_{i-1} = a\}$.

As an example, consider again the influence diagram as shown in Fig. 8.1. A decision graph for the influence diagram is shown in Fig. 8.2.

Let DG be a decision graph derived from an influence diagram with multiple value nodes, we can define an evaluation function, u_2, on it as follows:

- If N is a terminal, then
$$u_2(DG, N) = 0$$
- If N is a chance node with children $N_1, ..., N_l$, then
$$u_2(DG, N) = \sum_{i=1}^{l} p(N, N_i) * u_2(DG, N_i)$$
where $p(N, N_i)$ is the probability on the arc from node N to node N_i.

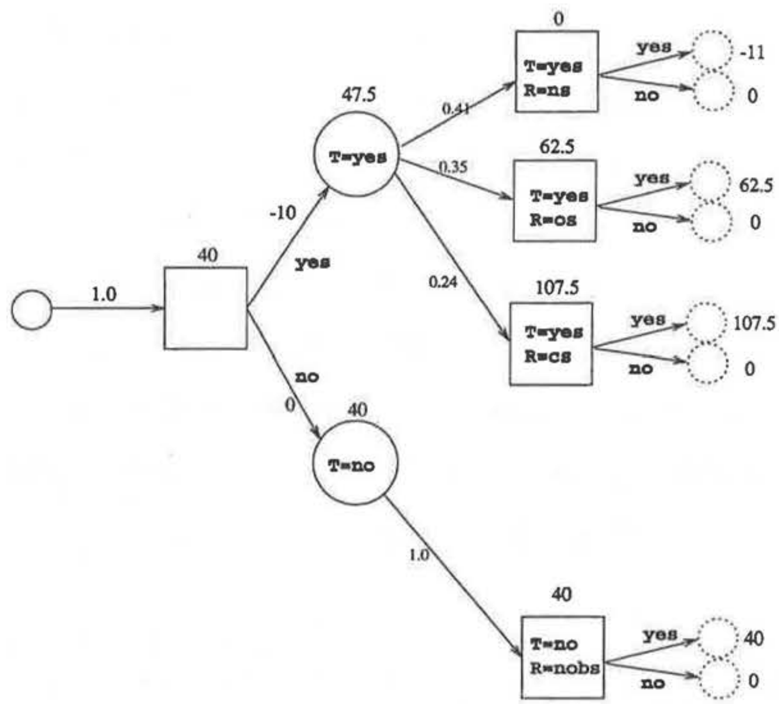

Figure 8.2: A decision graph for the influence diagram shown in Fig. 8.1

- If N is a choice node with children $N_1, ..., N_l$,then

$$u_2(DG, s) = max_{i=1}^{l}\{c(N, N_i) + u_2(DG, N_i)\}$$

where $c(N, N_i)$ is the value on the arc from node N to node N_i.

The reader may have noticed that the decision graphs corresponding to the examples we have considered so far are decision trees. This need not be true in general. Here we consider another example whose decision graph has shared structure.

Consider the following variation of the oil wildcatter problem. In the previous examples, we implicitly assumed that the amount of oil the oil wildcatter can obtain is equal to the amount of the oil underground. Now we replace this assumption by a more realistic one, namely, the amount of oil the oil wildcatter can obtain depends also on the equipment status. Thus, the oil wildcatter needs also to decide whether to upgrade his equipment. Furthermore, suppose the profit by selling oil also depends on market information and the sale policy. This more elaborate problem can be represented by the influence diagram shown in Fig. 8.3.

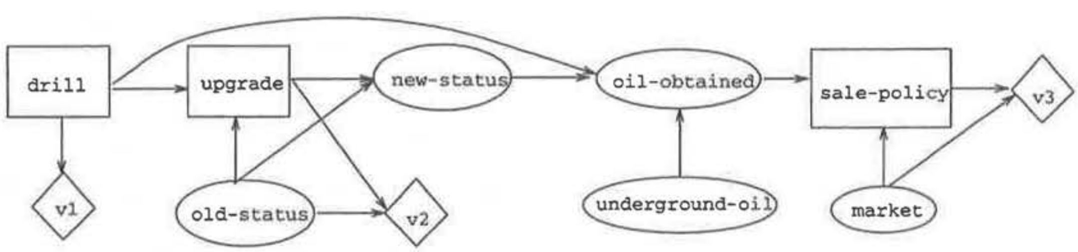

Figure 8.3: A variation of the oil wildcatter problem

The amount of obtained oil can be zero, low, medium or high. The decision graph corresponding to the problem is shown in Fig. 8.4. The decision problem is asymmetric in the following sense: if the drill decision is yes, then the amount of obtained oil must not be zero, and if the drill decision is no, the amount of obtained oil must be zero. Therefore, some of the arcs to the nodes representing the situations for `oil-obtained` are labeled with zero probability. After removing these zero–probability arcs, the decision graph becomes the one in Fig. 8.5, which is indeed a graph (not a tree).

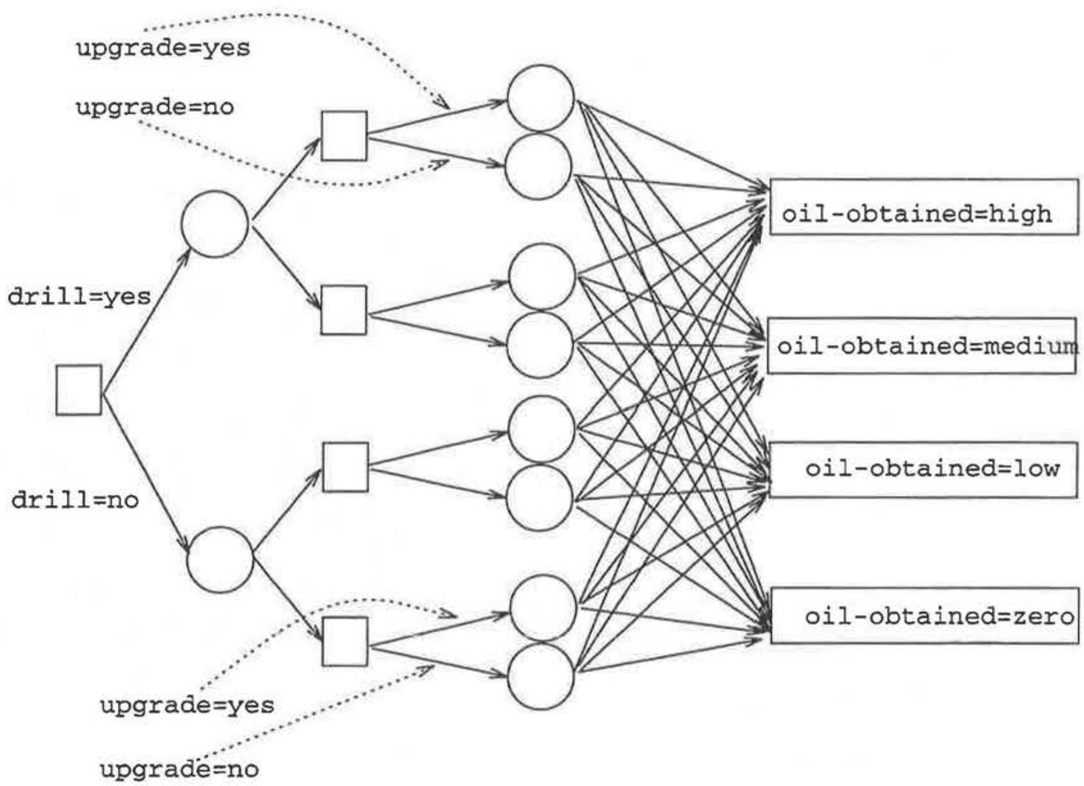

Figure 8.4: A decision graph for the influence diagram in Fig. 8.3

A possible heuristic function for this problem can be defined as follows: For any node N, if N represents a situation in which the drill decision is no, return zero, otherwise, return M where M is a large integer. Obviously, if M is large enough, this heuristic function is admissible. Suppose we use Algorithm DFS presented in Chapter 3 with this heuristic function to search the decision graph in Fig. 8.5. If the

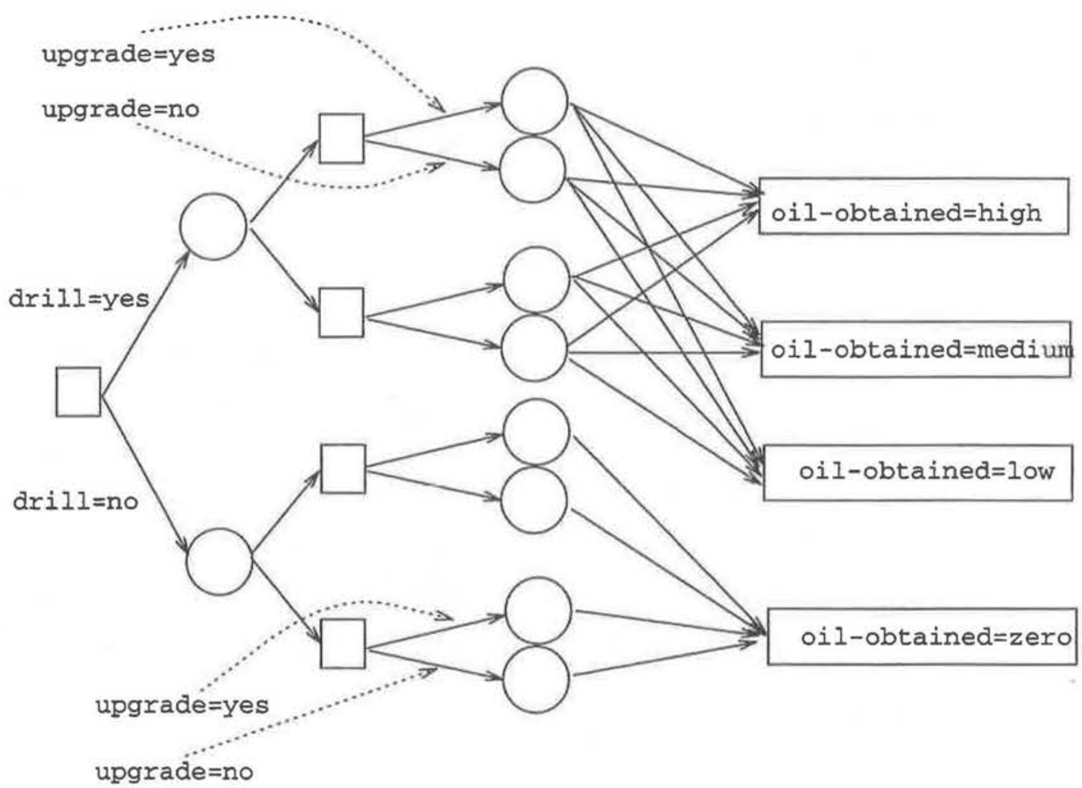

Figure 8.5: A decision graph, with zero probability arcs removed, for the influence diagram in Fig. 8.3

algorithm searches the branch corresponding to `drill=yes` first, then the subgraph under the branch corresponding to `drill=no` could be pruned altogether, provided that, according to the actual numerical setting, it is profitable to drill. Furthermore, if we have a stronger heuristic function asserting that the optimal expected value for the node representing the situation `drill=no` *equals* zero, then the lower half of the decision graph in Fig. 8.5 need not be expanded at all. As a matter of fact, when building a decision tree for the decision problem, one will use exactly the same heuristic information to avoid expanding the branch corresponding to `drill=no`.

8.3 Summary

We have developed a new method for influence diagram evaluation. The basic idea of the method is to transform an influence diagram into a decision graph in such a way that the optimal policies of the influence diagram correspond to the optimal solution graphs of the decision graphs. In this respect, our method is similar to Howard and Matheson's original approach to influence diagram evaluation. However, our method is more sophisticated and efficient than theirs.

To the best of our knowledge, our method is the only one enjoying all of the following merits simultaneously.

(1) It is applicable to a class of influence diagrams that is more general than the class of no–forgetting influence diagrams.

(2) It provides an interface to algorithms for Bayesian net evaluation.

(3) It can make use of heuristic search techniques and domain dependent knowledge.

(4) It exploits asymmetry in decision problems.

Part III

NAVIGATION IN UNCERTAIN GRAPHS

This part of the thesis is concerned with the problem of *navigation in uncertain environments*. We propose *U–graphs* (uncertain graphs) as a framework for uncertain environment representation. A U–graph is a distance–graph in which edge (arc) weights are not constants, but are random variables. We consider this problem for two reasons. First, the problem is of importance in its own right and there has been in the literature a growing interest in it. Second, as we will show, such a navigation task can be represented by a decision graph whose optimal solution graphs correspond to the optimal plans for the navigation task. Thus it provides an ideal application domain for the decision graph search algorithms discussed in Chapter 3.

In the next chapter, we first outline two domains where the problem of navigation in uncertain graphs arises and show how to represent uncertain environments by U–graphs, then define the problem of U–graph based navigation, and finally discuss some related work. In Chapter 10, we develop a decision theoretic formalization for the problem of U–graph based navigation. In Chapter 11, we discuss two algorithmic approaches to the problem: the *off–line* approach and the *on–line* approach. In the off–line approach, an agent computes a complete navigation plan for the task and then follows the plan. The algorithms presented in Chapter 3 are used for computing complete plans. In the on–line approach, the agent needs to decide where to go next in each encountered situation. We present a polynomial–time algorithm for on–line navigation. We also give some experimental data on the performance of some off–line algorithms and the on–line algorithm.

Chapter 9

U–graph based navigation

In this chapter, we introduce a problem of decision making under uncertainty — navigation in uncertain environments. We first discuss some motivations to the problem, and then introduce U–graphs as a framework for representing uncertain environments. Next, we define the problem of U–graph based navigation. Finally, we discuss the related work in this area.

9.1 Motivation

In this section we outline two major domains where the problem of navigation in uncertain environments arises. One is autonomous agent navigation and the other is computer network communication.

9.1.1 High–level navigation for autonomous agents

Robotics has been an active research area in the past decade. One of the ultimate goals of robotics research is to make robots capable of autonomous navigation in practical and large environments. A large environment is an environment whose structure is at a significantly larger scale than the observation range of the agent [42]. Practical environments often change over time. A central part of the navigation system of a robot is a path planning component. The primary goal of the path planning component is to generate a description of a route which can guide the robot to the desired destination. However, due to the complexity and the uncertainty of the environment, it is unreasonable to expect the path planning component to have *a priori* knowledge of every relevant detail necessary to generate an executable plan for a given task. Consequently, the path planning component has to appeal to some perception components for obtaining information dynamically.

In order to strike a balance between the use of prior knowledge and dynamic information, various kinds of hierarchical structure are commonly employed in most navigation systems [2, 13, 55, 58, 89]. In these systems, a distinction is made between a high–level (global) path planner and a low–level (local) path planner (as shown in Fig. 9.1).

The basic motivation for making this distinction between the low–level and high–level path planners is to isolate a component that can make maximum use of information known about the environment. Mitchell *et al.* [58] classifies path planning techniques into three categories: *deterministic*, *stochastic* and *servo* control, based on the nature of available information and the way in which a goal is decomposed into subgoals. A deterministic planning process is one in which the available information is relatively static and is assumed to be complete. With complete data, a deterministic planner can generate a plan which bridges the gap between the initial state and the goal state. A stochastic planning process is one in which the available information is not complete, but may be augmented as a result of plan execution. A stochastic planning process typically consists of an infinite loop of three steps: planning a primitive subgoal based on available information, achieving the subgoal and perceiving the environment. A servo control planning process is characterized by its

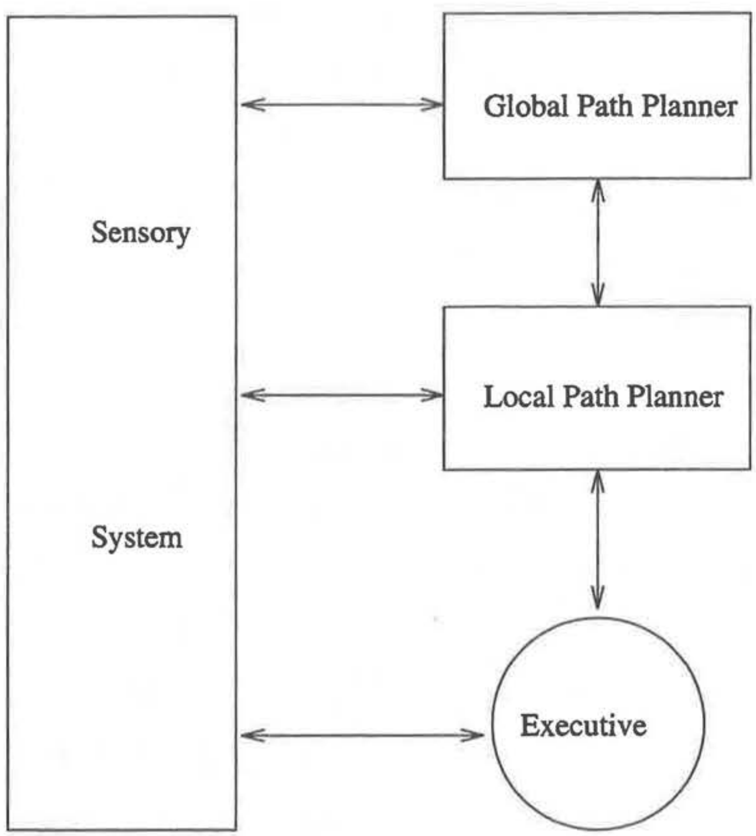

Figure 9.1: The block diagram of a typical autonomous navigation system

inherent use of feedback as a mechanism for control. Servo control processes may be used to achieve a goal which can be expressed in terms of a measurable state variable.

In most navigation systems, the high–level path planner takes a deterministic approach while the low–level path planner takes a stochastic approach. For a given navigation task in an environment, the high–level path planner generates a plan consisting of an ordered set of subgoals based on a high–level representation, usually called *a global map*, of the environment. In order to be relatively static and complete, the granularity of the global map must be coarse. Consequently, the granularity of the plan generated by the high–level path planner is also coarse. In order to carry out these subgoals, the low–level path planner needs to further elaborate them. This elaboration may involve a stochastic loop as described above.

With this distinction, the low–level path planner, along with the executive, can be regarded as a highly reactive system with limited reasoning capability [1, 7, 13, 58, 63], but able to perform local navigation, while the high–level path planner provides the source of global rationale, but is limited in its ability to react quickly to changes in the environment [55, 58].

In most navigation systems for autonomous robots, primary attention is focused on the low–level path planner and the executive, while little attention is paid to problems related to high–level path planning. This is largely due to the fact that, based on current technology, the development of a quality agent that can navigate locally is still a challenge to researchers and practitioners. The problem of high–level path planning is usually modeled as a variation of computing the shortest distance path from a distance–graph which serves as a representation of the global map.

However, a major drawback of using distance–graphs as global maps is that the uncertainty arising from practical environments cannot be modeled. For example, suppose we are in a region where traffic jams may occur along some roads. The traveling cost (time) along a road in the event of a traffic jam can be different from that without a traffic jam. Furthermore, we do not know in advance whether there is a traffic jam along a road, though we can estimate the "probability" that such a jam may happen. As another example, we know that there is a route between two places, and that the route may be blocked; we are not sure whether the route is blocked now, though we know that, according to past experience, the probability for the route being traversable is about 0.8. Clearly, distance–graphs are not sufficient to precisely model such situations, and it is highly desirable to take this uncertainty into consideration when we decide whether to go to a destination via an uncertain route. Therefore, some natural questions arise: how do we model this kind of uncertainty? how do we define the path planning problem? how do we characterize the quality of a navigation plan? how do we compute a "good" plan for a navigation task in an uncertain environment?

9.1.2 Packet routing in computer networks

In a computer network, a key issue for the network layer is how messages (packets) are routed from source nodes to destinations. This issue is referred to as the *network routing problem*.

A commonly used approach for network routing is called *packet switching* [97, 100]. In this approach, each packet is routed independently. A packet going from a source to a destination in a network is analogous to an autonomous agent traveling from the source to the destination in an environment. Unlike an autonomous agent, a packet is not able to decide where to go. Instead, whenever a packet arrives at an intermediate node, the node is responsible for deciding where the packet goes next (which link it goes through or which neighbour it reaches next). This approach is used widely in modern computer networks.

A class of commonly used routing algorithms in the computer network community is based on the concept of shortest distance paths (e.g., [54]). In these routing algorithms, a distance–graph is used to represent the topology of a network. In such a graph, nodes represent the computer nodes and arcs represent communication links. The weight of an arc from node A to node B represents the time that it will take for a packet to be forwarded from node A to node B through the communication link represented by the arc. Based on this model, a simple shortest path algorithm is used to make an optimal decision for every node–packet pair. Furthermore, if the network is static (i.e., neither the network topology nor the arc weights ever change), a fixed optimal routing table, indexed by all possible destinations, can be constructed for each node. When a packet arrives at a node, the node needs only to look up the destination of the packet in the table in order to decide through which link the packet should be forwarded.

Despite its simplicity, the routing model described above suffers from an obvious problem arising from the impracticality of the *stasis assumption.* In a network, the time for a packet to go through a link can be roughly divided into three components: the transmission time, the CPU processing time, and the time the packet has to wait in a buffer. Although the first two components are comparatively stable, the third component varies depending on the traffic load in the network.

A standard way to alleviate this problem is "routing–information updating" [100]. That is, each node periodically measures the lengths of the queues of those arcs emanating from it and broadcasts this information to the whole network. Upon receiving new routing information, each node updates its graph representation of the network and recomputes a new routing table based on the updated graph.

Experience [6, 103] shows that a routing–information updating approach can result in good network performance when the general traffic load in the network is light. However, when a network is under heavy traffic load, the solution does not work well. Since the weight of each arc in the graph is measured at some earlier time, the distance–graph represents only a "snapshot" of the network at a particular moment. As the traffic load becomes heavy, the waiting times for links tend to change quickly. Thus, the weight information in the graph tends to be out–of–date quickly. Therefore, the shortest paths based on the representation of a snapshot are likely different from the real shortest paths.

One way to compensate for this deviation is to increase the frequency of routing–information updating. Unfortunately, this leads to new problems. First, frequent routing–information updating requires frequent re–computation of routing tables, therefore, the requirement on CPU resource is increased. Second, since the routing–information is distributed across the network by "flooding," frequent routing updating consumes an unacceptable portion of network bandwidth. This in turn leads to even heavier traffic loads. A possible consequence of frequent updating can be a wild oscillation and performance degradation of the network [6, 103]. In practice, the frequency of routing information updating is chosen to be quite low to ensure the stability of the network. For example, in ARPANET, the typical interval between routing information updating ranges from 10 to 50 seconds.

Another way to ensure that the weight information in the graph remains up–to–date for a longer time is to use probabilistic information. Since the traffic load in a computer is dynamic, the lengths of the waiting queues for links change over time. Therefore, it is no surprise that constant estimates of the link delays will be out–of–

date quickly. On the other hand, the changes of the queue lengths can often exhibit some kind of probabilistic pattern (this is supported by queuing theory [37]). It seems reasonable to use probability distributions to represent the dynamic changes of the queues.

If computation of the routing tables is based on probabilistic information, then the optimality of the routing tables can be robust against changes in link delays. Routing information updating will not be necessary as long as the link delay changes conform statistically to the corresponding probability distributions. Thus, it can be hoped that the frequency of routing information updating could be substantially decreased (say, to a few times per day).

As in the case of autonomous navigation, the immediate questions are: how do we represent and use probabilistic information for network routing?

In the next section, we propose a framework, called *U–graphs*, for uncertain environment representation. In Section 9.3, we illustrate how to use U–graphs to represent uncertain environments. The rest of this chapter and the chapters to follow are devoted to problems of *U–graph based navigation*. By U–graph based navigation, we mean the process of navigating in an uncertain environment represented by a U–graph. Although this abstract treatment is applicable to both "real" navigation in natural environments and to packet routing in a computer network alike (see [75]), we assume in the rest of this thesis that our application context is the former when we describe the intuitive meanings of various concepts.

9.2 U–graphs

A U–graph is an extension of an ordinary distance–graph. The weights of some edges (arcs) in a U–graph are not constants, but are random variables. Clearly, probabilistic information on the uncertainty with respect to travel costs in a natural environment or link delays in a computer network can be represented in a U–graph.

A U–graph can be either directed or undirected. In this thesis, we consider undirected U–graphs only, because (1) this makes the presentation less cumbersome; (2) undirected U–graphs are sufficient in most cases encountered in navigation; and (3) the techniques and the algorithms developed for undirected U–graphs can easily be adapted to directed U–graphs.

If the weight of an edge is a random variable, the edge is referred to as an *uncertain edge*. Otherwise, the edge is referred to as an *ordinary edge*, or simply as an edge if no confusion arises. An uncertain edge between vertices A and B represents the knowledge that there exists a connection between the locations denoted by the vertices A and B, and that the weight of the connection is not certain and is given by a random variable.

A formal definition of a U–graph is as follows.

Definition 9.1 *A* U–graph *is quadruple* $\langle V, E, U, P \rangle$*, where:*

- V *is a finite set of vertices;*
- E *is a set of ordinary edges over* V *with constant weights;*
- U *is a set of uncertain edges over* V *whose weights are discrete non–negative random variables;*
- P *specifies a joint probability distribution over* U.

This definition is a generalized version of the one presented in [76]. Let $G = \langle V, E, U, P \rangle$ be a U–graph. For each $u \in U$, let Ω_u denote the set of possible weight values of u. We call Ω_u the *frame* of u. For each subset $U' \subseteq U$, let $\Omega_{U'} = \prod_{u \in U'} \Omega_u$.

Since P is a joint distribution over U, for each subset $U' \subseteq U$, we can talk about the marginal distribution over U', which is defined as follows:

$$P\{U'\} = \sum_{U-U'} P\{U\}.$$

Suppose U is indexed such that $U = (u_1, ..., u_m)$. The distribution P can be regarded as a mapping from Ω_U to $[0,1]$. For any $a = (a_1, ..., a_m) \in \Omega_U$, we let $U = a$ denote the event that each uncertain edge u_i takes weight a_i, for each $i = 1, ..., m$, and let $P\{U = a\}$ denote the probability of that event. Let $G(a)$ be a graph obtained from G by assuming that uncertain edge u_i has weight a_i, $i = 1, ..., m$. $G(a)$ is called a possible *realization* of G. The probability that $G(a)$ is the actual realization is given by $P\{U = a\}$.

The joint distribution P can be given in various forms such as a table or a Bayesian net. In the rest of this thesis, we assume that there are algorithms to compute marginal distributions $P\{U'\}$ from a joint distribution for any subset $U' \subseteq U$, and to compute a posterior distribution from a joint distribution and a piece of evidence.

As an example, a U-graph is shown in Fig. 9.2. The solid lines are edges and the dotted lines are uncertain edges. Each edge has a positive weight. The frames of the uncertain edges are given as follows: $\Omega_{s_1} = \Omega_{s_2} = \{10, 25\}$ and $\Omega_{s_3} = \Omega_{s_4} = \{20, 45\}$. The joint distribution is given in Table 9.1.

This U-graph of Fig. 9.2 can be viewed as a representation of a set of locations connected in a ring structure by highways. The uncertain edges model those highways which may be jammed. The uncertain edges carry the following meaning: It will cost 10 (in some normalized unit) to traverse the highways denoted by s_1 and s_2 if they are not jammed and cost 25 if they are jammed; similarly, it will cost 20 to traverse

Table 9.1: The weight distribution P

s_1	s_2	s_3	s_4	prob
10	10	20	20	0.0096
10	10	20	45	0.0864
10	10	45	20	0.0024
10	10	45	45	0.0216
10	25	20	20	0.0144
10	25	20	45	0.1296
10	25	45	20	0.0036
10	25	45	45	0.0324
25	10	20	20	0.0224
25	10	20	45	0.2016
25	10	45	20	0.0056
25	10	45	45	0.0504
25	25	20	20	0.0336
25	25	20	45	0.3024
25	25	45	20	0.0084
25	25	45	45	0.0756

the highways denoted by s_3 and s_4 if they are not jammed and cost 45 if they are jammed.

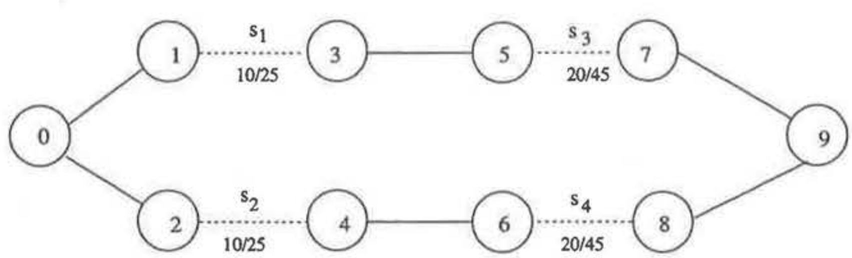

Figure 9.2: An example U–graph

In a navigation process, the actual states of some uncertain edges may be discovered. We distinguish two ways for discovering the states of uncertain edges. One way is to "buy" the information. The other way is to observe.

A question related to information purchasing is: for which edges is information available? and at what price, in what situation? The answer to this question depends on the concrete problem setting. Another related question is: if a piece of information

Table 9.2: The weight distribution P' after observing $s_1 = 10$

s_2	s_3	s_4	prob
10	20	20	0.0320
10	20	45	0.2880
10	45	20	0.0080
10	45	45	0.0720
25	20	20	0.0480
25	20	45	0.4320
25	45	20	0.0120
25	45	45	0.1080

on the actual state of an uncertain edge is available at some price in some particular situations, should an agent buy this information at the price? This problem is addressed in Section 9.4.3.

If the state of an uncertain edge is observable, the agent can discover the actual state of the uncertain edge at no cost. A related question is: which uncertain edges are observable in what situations? It is plausible to assume that an agent can observe the state of an uncertain edge when it is at a vertex of the uncertain edge[1]. Of course, an uncertain edge may also be observable in some other situations. For example, when an agent is at a location with high altitude, the uncertain edges in the nearby area may be observable as well. However, this depends on the concrete problem setting and we treat this case as a special kind of information purchase.

Whenever an agent obtains some information I regarding the actual state of some uncertain edges, a new U–graph is needed to represent the updated knowledge about the environment. The difference between the new U–graph and the old one lies only in their joint probability distributions. Thus, in order to derive the new U–graph, we only need to compute a posterior probability distribution $P' = P\{\cdot|I\}$

[1]For the case of directed U–graphs, it is reasonable to assume that an uncertain edge is observable in the situation when the agent is at the tail of the arc (at the entry of a one way road). This is also the case for network routing where it is assumed that a computer node knows the buffer state of the outgoing channel.

from the joint distribution and the new information I. If the joint distribution is represented as a Bayesian net [69], then the computation can be carried out by any well–established Bayesian net evaluation algorithm. The updating would be trivial if the weight variables are mutually independent.

For example, suppose that we discovered that the actual weight of uncertain arc s_1 in the previous example is 10. Let I denote this evidence. Then, we need to compute a posterior probability $P' = P\{\cdot|I\}$ such that $P'\{e\} = P\{e|I\}$ for any event e. The joint distribution P' is given in Table 9.2.

In Definition 9.1, we make an explicit distinction between edges and uncertain edges. One may immediately argue that this distinction is not necessary because an edge is a special uncertain edge. This argument is valid. Theoretically, it is not necessary to make such a distinction. We do this for two reasons. First, the number of uncertain edges turns out to be a crucial complexity parameter — the time complexity of some of the algorithms to be developed is exponential in the number. Second, by making this distinction, we encourage the user to use ordinary edges where they are sufficient, in order to obtain better computational performance.

In Definition 9.1, we make the following assumption.

Assumption 1: The weight distribution of any uncertain edge in a U–graph is discrete.

While this assumption facilitates our formulation of U–graph based navigation, it does not limit much the expressive power of U–graphs, since any continuous distribution can, at least in theory, be approximated by a discrete distribution.

9.3 Representing Uncertain Environments in U–graphs

The usefulness of U–graphs can be demonstrated from two aspects. First, as a natural extension of distance–graphs, U–graphs inherit the expressive power of distance-graphs, in addition to the capability of expressing a kind of uncertainty. Thus, U–graphs can be useful in situations where distance–graphs can be used.

Second, we can think of various practical situations where distance–graphs are insufficient and U–graphs can be useful. We illustrate this below by considering some examples in high–level navigation.

The essential knowledge that a path planner needs to have about an environment for high–level path planning is the environment's topological structure, which can be represented by a set of "interesting places", and the connectivity relation among these places. The places can be abstracted as vertices, and the connectivity relation abstracted by edges in a U–graph. If it is uncertain how much it costs an agent to traverse the route between two places, this uncertainty can be represented by an uncertain edge. Here are some examples.

Example 1: Fords.

Suppose an agent is at place A in an environment as shown in Fig. 9.3-(a) and wants to reach place F. In this environment there are two fords (BD and CE) along a river. Suppose that the agent is not sure whether the fords are traversable or not, though it knows the joint probability for the traversability state of the two fords. Suppose further that the agent knows the distances between the place pairs (the numbers in the figure). Based on all this information, we can construct a U–graph as shown in Fig. 9.3-(b) to model the environment, where the directed graph is a

Bayesian net representing the joint distribution of the weights of the uncertain edges.

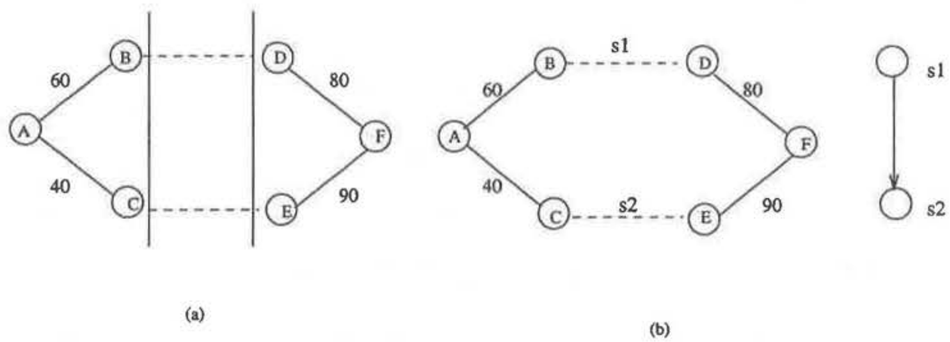

Figure 9.3: Modeling an uncertain environment by a U–graph

Example 2: Traffic Jams.

Consider the situation shown in Fig. 9.4. There may be a heavy traffic jam, or a light traffic jam along segment AB of a highway; and the probabilities for the events of heavy traffic jam, light traffic jam and no traffic jam along the segment are p_1, p_2 and p_3 ($p_1 + p_2 + p_3 = 1$), respectively. Let c_1, c_2 and c_3 be the costs that an agent needs to spend to go through the segment in light of the events heavy traffic jam, light traffic jam and no traffic jam, respectively. This segment can be modeled by an uncertain edge s with weight distribution that takes value c_1, c_2 or c_3 with probabilities p_1, p_2 and p_3 respectively.

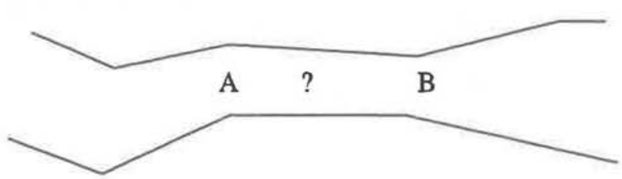

Figure 9.4: A segment of road that may have traffic jams

Example 3: Rooms connected by doors

In a typical indoor environment, doors are a source of uncertainty since they may be locked from time to time. This kind of environment can be modeled by U-graphs as well. As an example, a simple map of the LCI lab area in the old building of the computer science department of UBC is shown in Fig. 9.5. Suppose this map is constructed for a mobile robot in the lab whose responsibility is to bring coffee from the coffee machine in Room 300. For each given task, the robot must go to the coffee machine from the lab and then come back with coffee. To accomplish such a mission, the robot needs to plan a return route to Room 300. Since the doors shown in the map are not always open and the robot is unable to go through a locked door, the robot should take into account the uncertainty arising from these doors.

For high-level path planning, the map can be represented by the U-graph shown in Fig. 9.6, where each uncertain edge models a door in the map. The frame of each uncertain edge consists of two values: the small one representing the effort needed to go through the corresponding door when it is open, and the larger one representing that a locked door is not traversable.

Example 4: Choke Regions.

Linden and Glicksman [49] partition an environment into three kinds of regions: *passable* regions, *impassable* regions, and *choke* regions. A region is passable if the agent concerned can travel through it, and is otherwise impassable. A choke region is a relatively narrow traversable region between impassable regions, where some kind of blockage (hard or soft) that cannot easily be detected remotely may happen. Roads through dense forests and passes through mountains are examples of choke regions. Clearly, if the probabilities of choke regions being traversable can be estimated in some way, the uncertainty induced by choke regions can be readily modeled by uncertain edges.

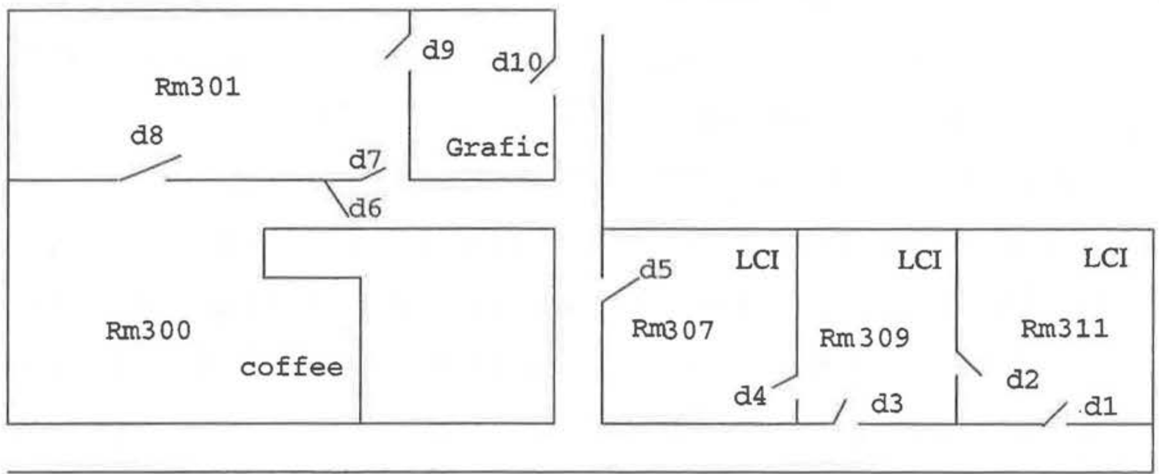

Figure 9.5: A simple map of the LCI lab area

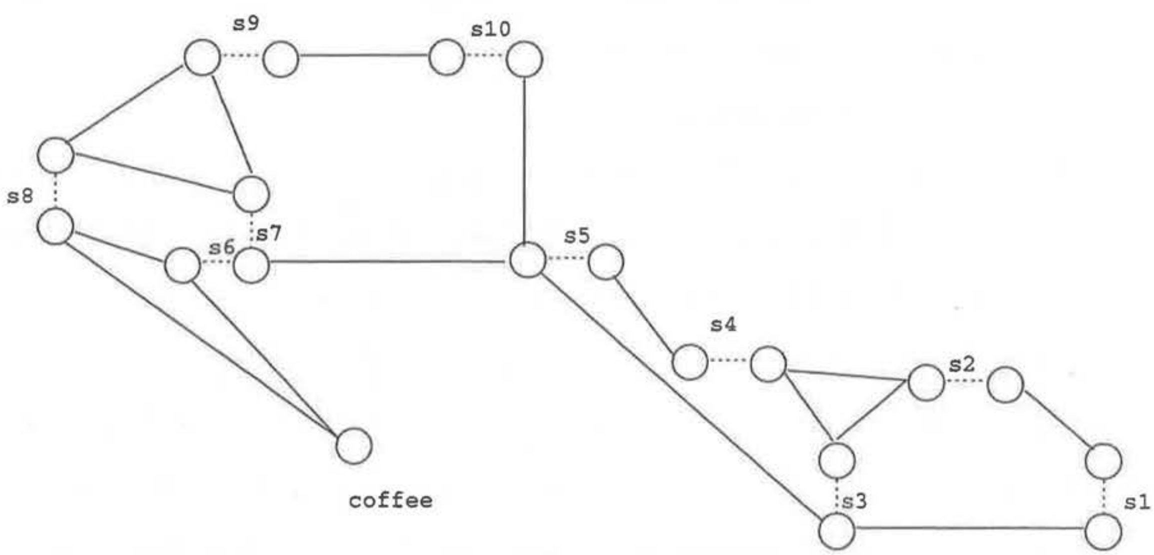

Figure 9.6: A U-graph representing the LCI lab area

An example environment is shown in Fig. 9.7, where the dark areas are impassable regions and those narrow areas between the dark areas are choke regions. The topological structure of the environment can be depicted by a U–graph like the one embedded in the figure.

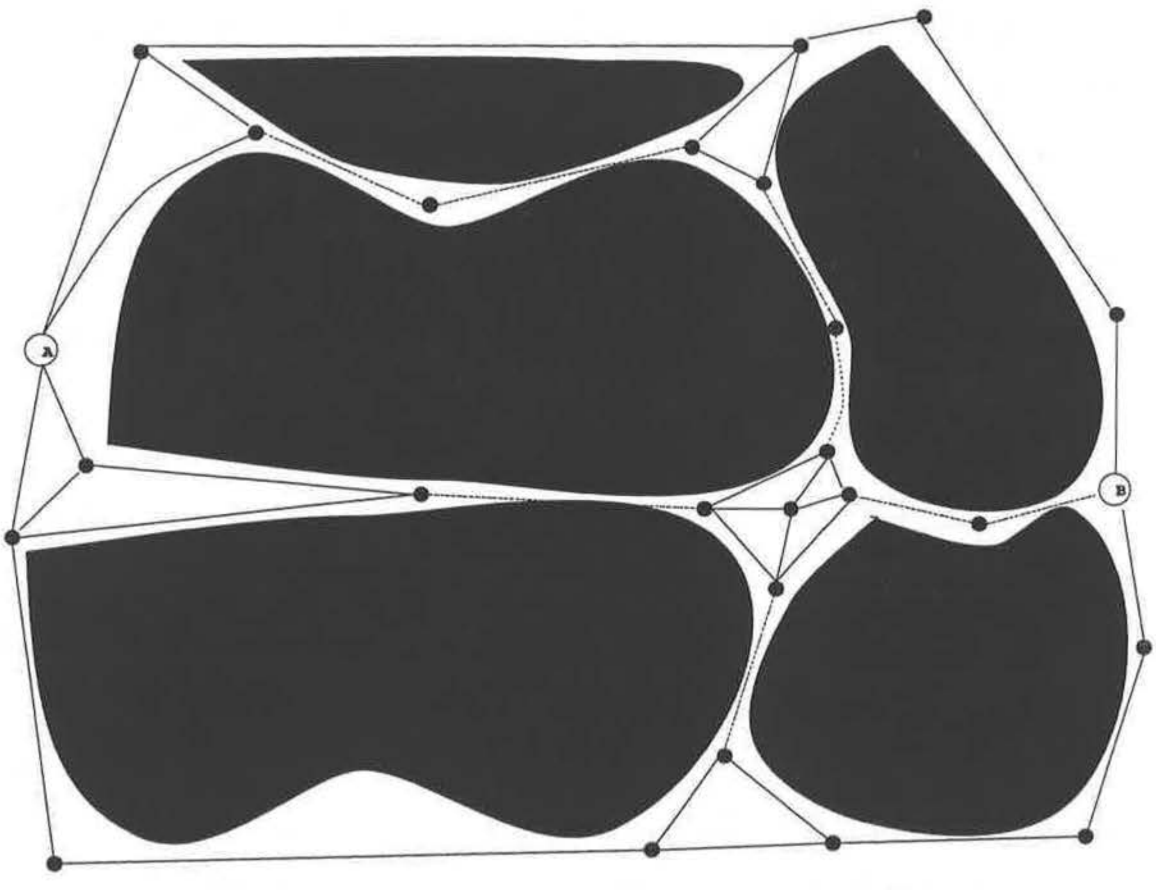

Figure 9.7: A map of an environment with choke regions

9.4 U–graph Based Navigation

When we refer to a navigation task, we assume that we are given a U–graph representation of an environment. We do not consider the problem of how to obtain a U–graph representation. There is a large body of research on map acquisition by exploring environments (e.g., [14, 22, 41, 57]).

9.4.1 Distance–graph based navigation vs. U–graph based navigation

There are several differences between distance–graph based navigation and U–graph based navigation. First, the plan structures are different. Since it is assumed that no contingent event occurs during distance–graph based navigation, the navigation plan generated for a distance–graph based navigation task is a simple path in the distance–graph between the start vertex and the goal vertex. On the other hand, for U–graph based navigation, since it is impossible to predict the actual weight of an uncertain edge in advance, some contingent events may occur (e.g., a plan specifies traversal an uncertain edge only if its weight is less than some constant). A plan for a U–graph based navigation task specifies what to do in some or all of the contingencies encountered during the course of a traversal.

Second, the interactions between the executives and the environments are different. For distance–graph based navigation, the executive need not interact with the environment. However, for U–graph based navigation, the executive needs to examine the status of an uncertain edge when it is at a vertex of the edge. Based on the examination result, the environment model (i.e., the U–graph) is updated and the next move is selected.

Third, the interactions between path planners and the executives are different. In distance–graph based navigation, the path planner computes a shortest path between the start vertex and the goal vertex and passes the description of the path to the executive. The executive simply follows the path with no interaction with the path planner. But in U–graph based navigation, if the executive encounters a situation for which no action is specified in the plan, the executive has to appeal to the path planner for a new plan. The practice of interleaving planning and execution is usually called *reactive planning* [27]. For the extreme case where the plans generated by the planner cover all the contingencies possibly encountered during the course of navigation, no replanning is ever needed. Schoppers called these plans Universal Plans [84]. In this thesis, we call these plans "complete plans." We will discuss the computational issues of navigation both in the paradigm where a path planner is used to generate complete plans and in the paradigm where a path planner is used to perform reactive planning.

9.4.2 The optimality issue

As we mentioned in the previous section, the plan generated for a given distance–graph based navigation task is a simple path between the start vertex and the goal vertex in the distance graph. Thus the optimality criterion for distance–graph based navigation is simply to minimize the path distance. On the other hand, for U–graph based navigation, some contingencies may arise. It is impossible to predict the exact navigation trace for a given task. Therefore, the optimality criterion for distance–graph based navigation is not applicable to U–graph based navigation. Actually, U–graph based navigation can be thought of as a game against nature [65]. At any moment, the concerned agent faces some uncertainty about the actual weights of those uncertain edges, and needs to decide where to go next. Thus the actual trace (and the actual cost) of a navigation cannot be determined based purely on the plan, but is also dependent on the actual state of the environment.

An important question about U–graph based navigation is what optimality criteria to use. In the literature, three criteria [4] have been used for this kind of problem:

minimizing the *competitive ratio* [94], minimizing the worst–case cost and minimizing the expected cost.

In this thesis, we use the criterion of minimizing the expected cost for U-graph based navigation. However, the formalization we will develop is general enough for other optimality criteria as well. We will discuss this in Section 10.4.

In order to illustrate the optimality criterion of minimizing the expected cost, let us consider a navigation task in an environment represented by the U–graph shown in Fig. 9.8-(a) where the weight distribution of the uncertain edge takes values d_4 and M with probabilities p and $(1-p)$ respectively, where M is a very large number. Intuitively, this uncertain edge denotes a segment of route that may be traversable with probability p. Suppose that an agent is at vertex A and is asked to go to vertex B.

There are two plans for the task[2]. The first one is to go to B through edge AB. Another one is as follows: go to C first; if the weight of CD is d_4, go to D then to B; otherwise go back to A, then go to B through edge AB. The expected cost for the first plan is d_1, the weight of edge AB. The expected cost for the second plan is given by the following formula:

$$d_2 + p * (d_4 + d_3) + (1-p) * (d_2 + d_1).$$

The agent can choose between the two options based on the expected costs.

9.4.3 The possibility of information purchase

In the previous example, we assumed that the status of an uncertain edge can be determined when an agent reached either vertex of the uncertain edge. In many

[2]Actually, there are many "silly" plans such as repeatedly going back and forth between node A and C for an arbitrary number of times and then to node B. Here, we are not interested in these silly plans and just ignore them.

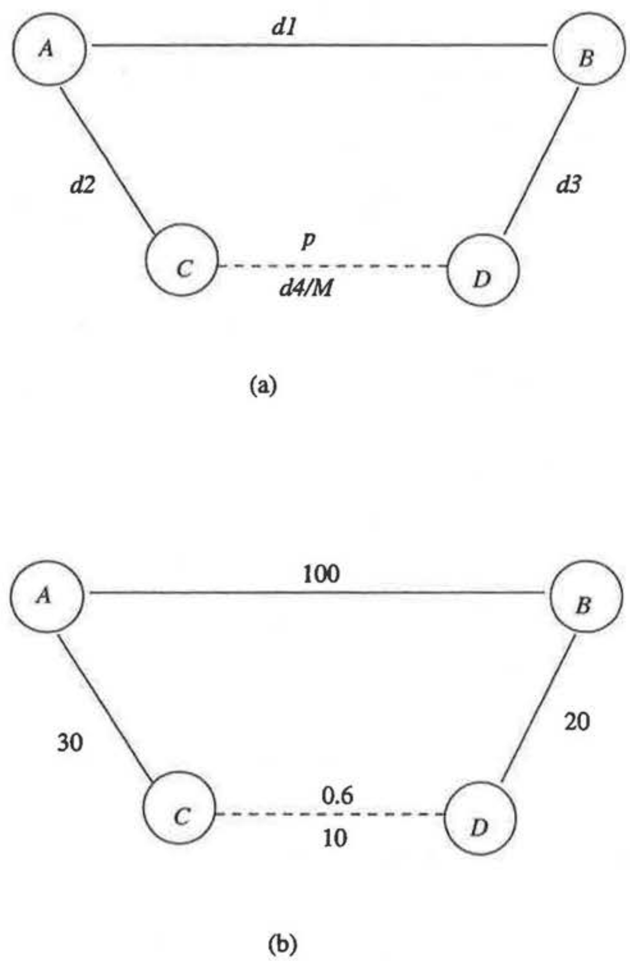

Figure 9.8: A simple path planning case with uncertainty

situations, the status of an uncertain edge can be determined remotely, possibly at some cost. For example, there may be a telephone service that can tell an agent whether or not there is a traffic jam along some road segment.

If we assume that an agent can "buy" information on the actual status of one or more uncertain edges, a natural problem arises: "at what price should an agent buy a particular piece of information?" In order to illustrate this problem, consider again the navigation situation shown in Fig. 9.8-(a). Suppose that the actual status of the uncertain edge is available to the agent at cost c. Then, what is the range of cost c such that the agent would be willing to buy the information?

As we know, if the agent does not buy information about the status of the edge, the minimal expected cost is:

$$min\{d_1, d_2 + p*(d_4 + d_3) + (1-p)*(d_2 + d_1)\}.$$

On the other hand, if the agent buys the information at cost c, the expected cost is:

$$c + p*(d_2 + d_4 + d_3) + (1-p)*d_1.$$

Thus, the agent should buy the information if and only if:

$$c + p*(d_2 + d_4 + d_3) + (1-p)*d_1 < min\{d_1, d_2 + p*(d_4 + d_3) + (1-p)*(d_2 + d_1)\}$$

Or

$$c < min\{p(d_1 - d_2 - d_4 - d_3), 2(1-p)d_2\}.$$

For the situation shown in Fig. 9.8-b, the agent should buy the information if and only if $c < 24$.

9.5 Related Work

9.5.1 Related work in path planning

A number of approaches to path planning and navigation in uncertain environments have been proposed and many navigation systems for mobile robots have been built (e.g., [2, 13]). However, in most of these systems, attention is focused primarily on the problem of how to make a robot capable of moving around in relatively small environments. Some notable exceptions are Kuipers's TOUR model [41], Kuipers and Levitt's COGNITIVE MAP [42] for the representation of spatial knowledge of large scale environments, and Levitt and Lawton's qualitative approach [48] to navigation in large scale environments.

The emphasis of Kuipers's TOUR model is on its power to express and assimilate knowledge gradually acquired over time. The emphasis of Levitt and Lawton's work on qualitative navigation is on techniques for an agent to locate itself relative to the goal position and various remote landmarks. The emphasis of our work on U-graph based navigation is, however, on the precise formulation of optimal navigation in environments with uncertainty, as well as algorithmic solutions to optimal navigation.

Dean *et al* [15] describes a navigation system that makes use of utility theory in navigation planning. They emphasize the coordination of task-achieving activities and map-building activities so as to efficiently accomplish a group of navigation tasks in an uncertain environment. In contrast, our planner is concerned primarily with finding optimal plans for accomplishing given U-graph based navigation tasks.

Linden and Glicksman's work on contingency planning [49] is also concerned with computing optimal plans for navigation in large and uncertain environments. They use *uncertain grids* to represent uncertain environments. A cell of an uncertain grid has value 0 or 1 or is a binary random variable. A cell with value 1 means the point is free; a cell with value 0 means the point is occupied. A cell with a binary random variable means that it is uncertain whether the point is free or not, and the actual state of the point is determined by the random variable. A grid is partitioned into three regions: *passable regions*, *impassable regions* and *choke regions*. It is assumed that all the cells in a passable region have value 1, all the cells in an impassable region have value 0, and the values of all the cells in a choke region are determined by random variables. Linden and Glicksman propose a path planning algorithm for this model. The algorithm is an extension of A*.

Mobasseri has studied the problems of path planning in uncertain environments from a decision theoretic point of view [59]. He also uses uncertain grids to represent uncertain environments, and formalizes the problem in a dynamic programming style.

9.5.2 Canadian Traveler Problems

Papadimitriou and Yannakakis [66] first named the problem of traveling with uncertainty the *Canadian Traveler Problem* (CTP). In their formulation, a traveler is given an unreliable graph (map); some of the edges of the graph may disappear at certain times. They also assume that the traveler cannot know whether an edge is actually there unless he/she reaches an adjacent vertex of the edge, and that the status of an edge will not change after being revealed. The problem is to devise a travel plan that results in optimal travel (according to some predefined measure) from one vertex to another.

Papadimitriou and Yannakakis define their optimality criteria in terms of *competitive ratios*. Competitive ratios are used in the literature to measure the quality of on-line algorithms [51, 94]. The competitive ratio of an on-line algorithm with respect to a problem can be informally defined as the ratio of the performance of the on–line algorithm to the best performance that may be achieved for any problem instance consistent with the given problem and with the information discovered during the operation [4]. Papadimitriou and Yannakakis show that devising an on–line travel plan with a bounded competitive ratio is PSPACE-complete. They also show that the problem of devising a plan that minimizes the expected ratio, provided that each edge in a graph has associated with it a given probability of being present, remains hard (#P–hard and solvable in polynomial space).

Bar–Noy and Schieber have studied several interesting variations of the CTP [4]. One variation is the *k–Canadian Traveler Problem*, which is a CTP with k as the upper bound on the number of blocked roads (disappeared edges). They give a recursive algorithm to compute a travel plan that guarantees the shortest worst–case travel time. The complexity of the algorithm is polynomial for any given constant k. They also prove that the problem is PSPACE-complete when k is non–constant.

Another variation Bar–Noy and Schieber have studied is the *Stochastic Recoverable CTP*. In this problem, it is assumed that blocked roads can be reopened in certain time. They present a polynomial algorithm for devising a travel plan that minimizes expected travel time, under the assumption that for any edge e in a given graph, the *recover time* of e is less than the weight of any edge adjacent to it in the graph. It is unclear how to relax this assumption.

Polychronopoulos studies in his Ph.D. thesis [71] a problem called *Stochastic Shortest Distance Problem with Recourse* (SSDPR). There are two common aspects between his study on the SSDPR and our study on U–graph based navigation problem. First, the problem statement is very similar. In a SSDPR, an agent is asked to reach a goal vertex from a start vertex in an uncertain graph. It is assumed that the actual weight of an uncertain edge is determined only when an adjacent vertex is visited for the first time. The problem is to find a navigation plan minimizing expected cost. Second, Polychronopoulos also considers off–line and on–line navigation paradigms, and develops algorithms for both paradigms.

There are four major differences between Polychronopoulos' study and ours.

First, Polychronopoulos considers two versions of the problem, one with and another without an independence assumption on the weight distributions of uncertain edges. In the version without the independence assumption, an uncertain graph is represented by a graph along with the set of possible realizations of the edges in the graph. In contrast, our formulation treats both cases uniformly. In both cases, a U–graph is represented by a graph and a joint probability distribution over the uncertain edge set. The joint probability distribution can be given in various forms such as a Bayesian net. Within our framework, the case with the independence assumption is naturally a special case of the one without the independence assumption.

Second, our algorithms are different. His off–line planning algorithm is a dynamic programming version while ours are search oriented. In the development of our algorithms, we emphasize the possibility of pruning the non–optimal part of the search space both at problem formulation stage and at plan computing stage. For on–line planning, our algorithm is more sophisticated than his. In his on–line planning algorithm, the weight of a substituting edge is simply the mean value of the weight distribution of the corresponding uncertain edge. Thus, only the information local to an uncertain edge is taken into account in computing the substituting weight for the uncertain edge. However, in our on–line algorithm, in addition to the local information, the goal position and the global structure of the U–graph of the given navigation task are also taken into account in computing the substituting weights for uncertain edges. Initial simulation results show that our on-line algorithm results in better navigation quality.

Third, Polychronopoulos discusses navigation problems with both directed and undirected graphs, while our attention is focused on problems with undirected graphs, by noting that algorithms developed for undirected graphs can be adapted to directed graphs.

Fourth, Polychronopoulos presents some theoretical results about the complexity of the problem. In particular, he shows that SSDPR is #-P hard and is solvable in P-SPACE.

9.5.3 Related work in AI and decision theory

In some sense, our study on U-graph based navigation can be viewed as an application of decision theory to planning. In the literature, much work has been reported on applications of decision theory to AI problems. A common characteristic of these applications is that a given problem is viewed as a decision making problem and one is interested in a "good" or optimal solution, instead of an arbitrary solution to the problem. These applications include general planning [9, 16, 29, 46], diagnostic reasoning [72, 73], reactive systems [24, 28] and domain specific problem solving such as the monkey and bananas problem [23], the blocks world [39] and navigation problems [15, 59, 60]. Our work on U-graph based navigation belongs to the last group, i.e., applying decision theory to the domain of autonomous navigation.

Chapter 10

A Formalization of U–Graph Based Navigation

In this chapter, we present a decision theoretic formalization for U–graph based navigation. With this formalization, a U–graph based navigation problem is represented by a decision graph whose optimal solutions represent the optimal plans. This formalization reduces the problem of computing optimal plans for navigation tasks to a decision graph search problem, for which we have developed various algorithms. Moreover, the formalization is general enough to deal with various variations of U-graph based navigation.

In the previous chapter, we made an assumption about the weight distributions of U–graphs. Before starting our formalization, we make two additional assumptions for U–graph based navigation.

Assumption 2: The agent can determine the status of an uncertain edge of a U–graph when and only when it is at either vertex of the uncertain edge.

This assumption rules out the possibility of "buying" information on the status of uncertain edges. The sole motivation for making this assumption is to simplify our presentation. Our formulation can easily be extended to handle decisions by agents to buy or decline information. We come to this point in Section 10.2.

Assumption 3: The status of any uncertain edge does not change after being revealed.

This assumption is *crucial* to the formalization and the algorithms to be developed in this and the next chapters. An immediate consequence of this assumption is that when the agent determines the actual weight of an uncertain edge, the uncertain edge can be regarded as an edge with this weight.

With Assumptions 2 and 3, we can precisely specify how an agent updates a U–graph after observing the states of some uncertain edges. Let $G = \langle V, E, U, P\rangle$ be a U–graph. Suppose the agent has just arrived at vertex $v \in V$. Let $U(v) = \{u_{i_1}, ..., u_{i_k}\}$ be the set of the uncertain edges adjacent to v. Then, according to Assumption 2, the agent can observe the weights of these uncertain edges and compute a new U–graph based on the observation. Suppose the observation is that the actual weights of uncertain edges $u_{i_1}, ..., u_{i_k}$ are $w_1, ..., w_k$ respectively. Let $a = (w_1, ..., w_k)$. We use $U(v) = a$ to denote the observation. The new U–graph $G(a) = \langle V, E, U, P'\rangle$ is the same as G except the joint distribution is the posterior distribution P' obtained from P based on evidence $U(v) = a$. More precisely, P' is a distribution over U such that $P'\{X\} = P\{X|U(v) = a\}$ for any event X about the uncertain edges in U.

Let $\Omega = \prod_{u \in U(v)} \Omega_u$. For each element $a \in \Omega$, $G(a)$ is one of the possible new U–graphs that the agent may have after the observation. The probability that $G(a)$ is the actual one is given by the marginal probability

$$P\{U(v) = a\} = P\{u_{i_1} = w_1, ..., u_{i_k} = w_k\}.$$

10.1 Preliminaries

Let $G = \langle V, E, U, P\rangle$ be a U-graph. An uncertain edge $u \in U$ is *deterministic* if there exists a value $w \in \Omega_u$ such that $P\{u = w\} = 1$. An uncertain edge is *random* if it is not deterministic.

Let $U^d = \{u \in U | u \text{ is deterministic}\}$ and $U^r = \{u \in U | u \text{ is random}\}$. U^d and U^r partition U.

For each $v \in V$, let $U^r(v) = \{u \in U^r | u \text{ is adjacent to } v\}$. $U^r(v)$ is the set of random edges adjacent to vertex v. Let $V^r = \{v \in V | U^r(v) \neq \phi\}$ and $V^d = V - V^r$. V^r is the set of vertices to which at least one random edge is adjacent and V^d is the set of vertices to which no random (uncertain) edge is adjacent.

If the agent comes to a vertex v in V^r, it can discover the actual weights of the random edges in $U^r(v)$. Thus, the vertices in V^r are called *information collecting* vertices.

For each $u \in U$, we say w is a *possible realization* of u if $w \in \Omega_u$. Let $u^o = min\{w \in \Omega_u | P\{u = w\} > 0\}$ and $u^p = max\{w \in \Omega_u | P\{u = w\} > 0\}$. We call u^o the *optimistic realization* of u and u^p the *pessimistic realization* of u. Notice that u is deterministic *iff* $u^p = u^o$.

An *induced graph* of G is a variation of G in which each uncertain edge has a possible realization as its weight. Note that an induced graph is a distance graph. Each element $a \in \Omega_U$ determines an induced graph, denoted as $G(a)$, of G. The probability that $G(a)$ is the actual realization of G is given by the marginal probability $P\{U = a\}$.

The *pessimistic induced graph* G^p of G is the induced graph of G in which the weight of each uncertain edge is equal to the corresponding pessimistic realization.

The *optimistic induced graph* G^o of G is the induced graph of G in which the weight of each uncertain edge is equal to the corresponding optimistic realization.

We use *configurations* to represent the situations that an agent may encounter during navigation. A *configuration* is a quadruple $\langle G, v_c, v_g, c\rangle$, where G is a U-graph representing the current knowledge that an agent has about the environment; v_c and v_g are two vertices of the U-graph representing the current position and the goal position, respectively; c is a parameter representing the cost that the agent has incurred so far when it reaches this configuration. For a given navigation task of going from v_s to v_g in G, the configuration $\langle G, v_s, v_g, 0\rangle$ is called the *initial configuration*. A *terminal* is a configuration $\langle G, v_c, v_g, c\rangle$ such that the shortest distance from v_c to v_g in G^p is equal to that in G^o. Obviously, $\langle G, v_g, v_g, c\rangle$ is a terminal.

These concepts about induced graphs can be generalized to configurations. We simply refer to the pessimistic (or optimistic) induced graph of the U-graph of a configuration as the *pessimistic (or optimistic) induced graph of the configuration*.

A configuration $C = \langle G, v_c, v_g, c\rangle$ is called an *uncontrolled configuration* if v_c is an information collecting vertex (i.e., $v_c \in V^r$). A configuration $C = \langle G, v_c, v_g, c\rangle$ is a *controlled configuration* if v_c is not an information collecting vertex (i.e., $v_c \in V^d$).

When an agent is in a non-terminal controlled configuration $C = \langle G, v_c, v_g, c\rangle$ with $k\,(k > 0)$ neighbours $v_1, ..., v_k$, the agent has the freedom to move to any one of these neighbours. Should the agent decide to move to a particular neighbour, the agent incurs cost equal to the weight of the edge between the current vertex and the neighbour[1]. Such a move leads to a new configuration. In order to model such moves, we define k *controlled transitions*, each corresponding to a move to one of the k

[1]In case there are multiple edges between the two vertices, the cost is equal to the minimum of the weights. In the discussion to follow, we assume that there is at most one edge connecting two vertices, without loss of generality.

neighbours. We use $trans(C,t)$ to denote the configuration to which the transition t can lead from configuration C. Formally, we have:

$$trans(\langle G, v_c, v_g, c\rangle, t) = \langle G, v', v_g, c + c(t)\rangle$$

where t is the transition corresponding to neighbour v', and $c(t)$ is the cost of transition t, equal to the weight of edge $\langle v_c, v'\rangle$. Since the agent chooses which transition will happen, these transitions are called *controlled transitions*.

When in a non-terminal uncontrolled configuration $C = \langle G, v_c, v_g, c\rangle$ with $U^r(v_c) = \{u_{i_1}, ..., u_{i_k}\}$ ($k > 0$), the agent can observe the states of the random edges in $U^r(v_c)$ and update the U-graph based on the observation. For each $a \in \Omega_{U^r(v_c)}$, $G(a)$ is a possible new graph and $\langle G(a), v_c, v_g, c\rangle$ is a possible next configuration. Depending on the actual state of the random edges in $U^r(v_c)$, the current configuration will change to one of the possible next configurations. Such configuration changes incur no cost. To model such configuration changes, we define a set of *uncontrolled transitions* for configuration C, each corresponding to a configuration change from C to one of its possible next configurations. We refer to these transitions as uncontrolled transitions because which transition will actually happen is not controlled by the agent, but determined by the probability distribution over the possible next configurations of the uncontrolled configuration. The probability distribution is the same as that of the new U-graphs. More specifically, the probability that $C' = \langle G(a), v_c, v_g, c\rangle$ is the actual next configuration is $P\{U^r(v_c) = a\}$. This probability is associated with the transition from C to C'.

With these definitions, we can think of U-graph based navigation as a sequence of decision making and configuration transitions. When an agent is in a non-terminal controlled configuration, it can select a controlled transition and then reach a new configuration. The new configuration can be either a terminal, an uncontrolled configuration or a controlled configuration. In the case that the new configuration is a controlled configuration, the agent can repeat this transition selection. If we take a longer perspective on the transition selection, we observe that, from a controlled configuration, the agent will eventually either come back to the same configuration or reach an uncontrolled configuration or reach a terminal by taking a sequence of controlled transitions. The first possibility corresponds to a case where an agent takes a circular trip without discovering any new information. Obviously, any navigation plan resulting in this kind of behavior must not be optimal. Thus, we want to exclude such circular behaviors. To do so, we use ***composite transitions*** to model "non–circular" controlled transition sequences and ignore the "circular" transition sequences.

A sequence of controlled transitions is called a *composite transition* if it leads to the destination v_g or to an uncontrolled configuration. Formally, a sequence of controlled transitions $T = t_1, ..., t_k$ with $k \geq 1$, is a composite transition from controlled configuration $C = \langle G, v_c, n_g, c\rangle$ to configuration $C' = \langle G, v', v_g, c'\rangle$ if $C' = trans(...trans(C, t_1), ..., t_k)$ and if $v' = v_g$ or C' is an uncontrolled configuration. We use $ctrans(C, T)$ to denote the configuration resulting from *taking* the composite transition T in C. For the above case, we have $ctrans(C, T) = C'$. The cost for a composite transition T, denoted by $c(T)$, is defined as the sum of the costs of the constituent transitions of T. Intuitively, starting from any controlled configuration, the next intermediate vertex the agent will reach is either the goal vertex or an information collecting vertex. Note that, for a given controlled configuration C and a configuration C', there may exist zero, one or many composite transitions from C to C'. A composite transition from C to C' is *optimal* if it has the least cost. A configuration C' is called a ***composite successor*** of a controlled configuration C if there is a composite transition from C to C'. From now on, whenever we refer to a successor of a controlled configuration we mean a composite successor of the configuration. Similarly, whenever we refer to the transition set associated with a controlled configuration, we mean the set of optimal composite transitions of the configuration.

The composite successor set of a controlled configuration $C = \langle G, v_c, v_g, c \rangle$ can be computed as follows. Let $V_c = V^r \cup \{v_g\}$. V_c is called the set of *candidate vertices* for the next move. If the shortest path from v_c to a vertex $v \in V_c$ in the pessimistic induced graph G^p contains no vertex in V_c other than v, then v is a vertex the agent can reach next by a (composite) transition. Vertex v is called a *next vertex* of C. Formally, let V' denote the set of all next vertices of C and let $d_s(G^p, v_c, v)$ denote the shortest distance from v_c to v in G^p. The successor set of C is given by:

$$\{\langle G, v, v_g, c + d_s(G^p, v_c, v) \rangle | v \in V'\}.$$

The cost of the composite transition from C to $\langle G, v, v_g, c{+}d_s(G^p, v_c, v) \rangle$ is $d_s(G^p, v_c, v)$. This successor set can be computed by Dijkstra's shortest distance algorithm [21].

Configuration C' is *reachable* from configuration C if C' is a successor of C or if there exists a configuration C'' such that C'' is a successor of C and C' is reachable from C''. The configurations reachable from the initial configuration along with the successor relationship among them form a directed graph. Such a graph is called the *representing graph*, and can be specified as follows.

- The initial configuration is in the representing graph.
- If a non–terminal controlled configuration C is in the graph, so are all of its (composite) successors, which form its children in the graph. The arc from C to each child configuration C' corresponds to the (optimal composite) transition

from C to C' and is labeled with the cost of the transition.

- If a non-terminal uncontrolled configuration C is in the graph, so are all of its successors, which are the children of C. The arc from C to each child configuration C' corresponds to the uncontrolled transition from C to C' and is labeled with the probability of C'.

We call such a graph the *representing graph* of the navigation because it represents all the possible courses of the navigation.

Lemma 10.1 *The representing graph of a U-graph based navigation task is acyclic. The longest path in the representing graph is bounded by $2k+2$ where $k = min\{m, n\}$, m is the number of uncertain edges and n is the number of vertices in the U-graph.*

Proof. The representing graph is acyclic due to the following four facts: (a) a child of a controlled configuration is either a terminal or an uncontrolled configuration, a child of an uncontrolled configuration is either a terminal or a controlled configuration; (b) a terminal has no child; (c) the number of random edges in a controlled configuration is equal to the number of random edges in any of its successors; (d) the number of random edges in an uncontrolled configuration is strictly greater than that in any of its successors. Because of facts (a) and (b), controlled configurations and uncontrolled configurations must be interleaved along any path $C_1, ..., C_i$ of length $i > 2$. Without loss of generality, let us assume that i is even and $C_2, C_4, ..., C_i$ are all uncontrolled configurations and $C_1, C_3, ..., C_{i-1}$ are all controlled configurations. Let $u(C)$ denote the number of random edges in configuration C. According to facts (c) and (d), we have $u(C_j) \geq u(C_{j+1})$ and $u(C_j) > u(C_{j+2})$, for $j = 1, ..., i-2$. Suppose the lemma is false, then there must be a cycle in the graph containing at least one uncontrolled configuration C. Thus, we can find a path $C_1, ..., C_i$ such that $C_{j_1} = C_{j_2} = C$, and $j_1 + 2 \leq j_2 \leq i$. Therefore, we have $u(C) = u(C_{j_1})$, $u(C) = u(C_{j_2})$, and $u(C_{j_1}) > u(C_{j_2})$, a contradiction.

From the above analysis, we can conclude that the longest path in the representing graph is bounded by $2m + 2$. Furthermore, note that any two uncontrolled configurations along a path must have different current vertices, thus there are at most n uncontrolled configurations along any path. Therefore, the longest path in the representing graph is also bounded by $2n + 2$. □

By the above lemma, we know that the representing graph of a navigation is a DAG with the initial configuration as the root of the graph.

As an example, let us consider the U-graph shown in Fig. 10.1 in which CD is the only uncertain edge. The weight of CD is d_4 with probability p and M with probability $(1-p)$ where M is a very large number. Suppose an agent is asked to reach vertex B from vertex A in the U-graph. Note that if $d_1 \leq d_2+d_3+d_4$, then the initial configuration is a terminal. Suppose $d_1 > d_2 + d_3 + d_4$. The representing graph of this task is shown in Fig. 10.2, where solid boxes represent controlled configurations, dotted circles represent terminals, solid circles represent uncontrolled configurations, arcs represent transitions, the labels for controlled transitions are their costs and the labels for uncontrolled transitions represent their probabilities. In the figure, node 1 is the initial configuration, node 2 is the configuration when the agent reaches vertex B through edge AB, node 3 is the (uncontrolled) configuration when the agent reaches vertex C through edge AC.

The representing graph of a navigation task represents the possible traces of the task. In a representing graph, a controlled configuration represents a situation where the agent can choose the next configuration, while an uncontrolled configuration rep-

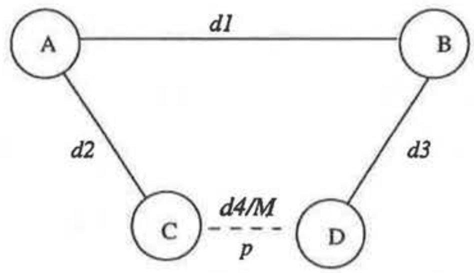

Figure 10.1: A simple U-graph

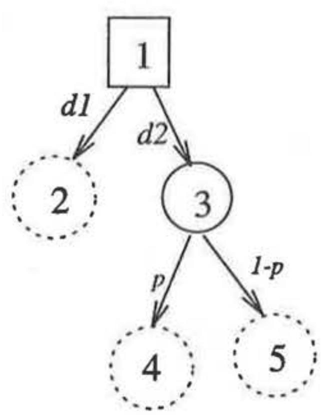

Figure 10.2: The representing graph of a navigation task

resents a situation where "nature" chooses the next configuration during a navigation process. Thus, a representing graph can be regarded as a decision graph with the controlled configurations as choice nodes and the uncontrolled configurations as chance nodes. With this analogy, a solution graph of a representing graph can be viewed as a representation of a navigation plan for the corresponding navigation task. Furthermore, the plan is *complete* in the sense that it covers all the possible configurations an agent may encounter if the agent follows the plan. The evaluation function of the decision graph will be defined in an appropriate way to reflect our optimality criterion of minimizing the expected cost. Thus, the path planning problem for U–graph based navigation is, for any given navigation task, to compute an optimal solution graph of the representing graph of the task.

10.2 Modeling Information Purchase

When information on the status of some random edges is available to the agent at some cost, the agent must decide in each controlled configuration whether or not to buy the information. This option can be modeled by a special (controlled) transition. Taking the transition will incur a cost equal to the price of the information and results in an uncontrolled configuration whose children can be determined according to the possible states of the uncertain edges. To illustrate this, consider again the task of going from vertex A to vertex B in the U–graph shown in Fig. 10.1. Suppose the agent can determine the state of uncertain edge CD at cost c. The representing graph of this task is shown Fig. 10.3, where the initial configuration has one more child (node 6 in the figure) than the one in Fig. 10.2. The new child results from the special transition modeling the option of information purchase. It has two children, one corresponding to the case where CD has weight d_4 and the other corresponding to the case where CD has weight M.

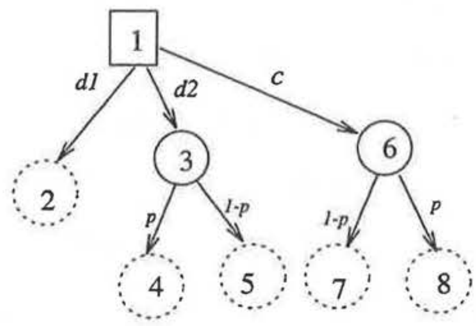

Figure 10.3: The representing graph of a navigation task with the information purchase option

10.3 The Expected Costs of U–graph Based Navigation Plans

In order to define the expected costs of navigation plans, we need to define a cost function on the solution graphs. The cost function should, for each solution graph, return the expected cost an agent incurs if the agent follows the plan corresponding to the solution graph. Let X denote a cost function with two parameters: a solution graph and a configuration in the solution graph. X can be recursively defined as follows.

$$X(sg, C) = \begin{cases} d_s(G^p, v_c, v_g) & \text{if } C \text{ is a terminal,} \\ X(sg, C') + c(C, C') & \text{if } C \text{ is a controlled configuration,} \\ \sum_j X(sg, C_j) * q(C_j) & \text{if } C \text{ is an uncontrolled configuration} \end{cases}$$

where G^p is the pessimistic induced graph of the configuration C, $d_s(G^p, v_c, v_g)$ denotes the shortest distance from the current vertex v_c to the goal vertex v_g in G^p, $q(C_j)$ denotes the probability of the successor C_j of the uncontrolled configuration C, and C' denotes the successor configuration of C in sg. Intuitively, $X(sg, C)$ is the expected cost an agent incurs if the agent follows the plan corresponding to the solution graph starting in configuration C. The above definition can be understood as a formal expression of the following intuition. If C is a terminal, the agent can complete the task by following the shortest path from the current vertex to the goal vertex in the pessimistic induced graph; if C is a controlled configuration, $X(sg, C)$ is equal to the sum of the cost of the transition from C to C' prescribed by the plan for configuration C and the expected cost the agent incurs starting in C'; if C is an uncontrolled configuration, $X(sg, C)$ is the average of the costs that the agent incurs starting in C's successor configurations.

The representing graph shown in Fig. 10.2 has two solution graphs as shown in Fig. 10.4, corresponding to two possible plans for the corresponding navigation task.

Plan (a): go to B through edge AB.

Plan (b): go to C first, if CD's weight is d_4, go to B via D, otherwise, go back to A then to B through edge AB.

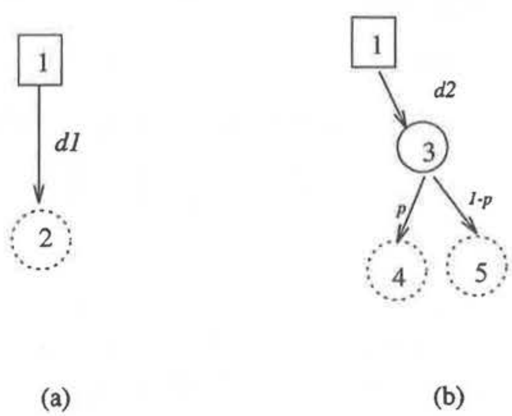

Figure 10.4: Two solution graphs of a navigation task

The expected costs for Plans (a) and (b) are: d_1 and $d_2 + p(d_4 + d_3) + (1-p)(d_2 + d_1)$, respectively.

In order to express the optimality criterion, we can define an evaluation function X' for representing graphs by extending the definition of function X. The definition of function X' on a representing graph rg is as follows.

$$X'(rg, C) = \begin{cases} d_s(G^p, v_c, v_g) & \text{if } C \text{ is a terminal,} \\ min_j\{X'(rg, C_j) + c(C, C_j)\} & \text{if } C \text{ is a controlled configuration,} \\ \sum_j X'(rg, C_j) * q(C_j) & \text{if } C \text{ is an uncontrolled configuration.} \end{cases}$$

Therefore, the problem of computing an optimal plan for a navigation task is reduced to computing an optimal solution graph of the representing graph of the task with X' as the evaluation function. This problem can be solved by applying the algorithms presented in Chapter 3. We discuss this issue in the next chapter.

10.4 Other Variations

So far we have presented a formalization for U-graph based navigation with respect to the optimality criterion of minimizing the expected cost. Our formalization is general enough to deal with other optimality criteria and/or other variations of the problem. We explore this aspect in this section.

10.4.1 Minimizing the competitive ratio

Papadimitriou and Yannakakis [66] define optimality criteria for the *Canadian Traveler Problem* (CTP) in terms of *competitive ratios*. The competitive ratio of a plan with respect to a problem instance is informally defined as the ratio of the cost of the plan to the minimum cost that may be achieved for the problem instance. The competitive ratio of a plan is the maximum of the ratios of the plan with respect to all possible problem instances. Intuitively, a plan with competitive ratio r guarantees that for any problem instance, the cost of the plan is bounded from above by r times the minimal cost for the same problem instance.

We define a function r to precisely express the competitive ratio of a plan.

$$r(sg, C) = \begin{cases} \frac{c(C)+d_s(G^o,v_c,v_g)}{d_s(G^o,v_s,v_g)} & \text{if } C \text{ is a terminal,} \\ max_j\{r(sg, C_j)|q(C_j) > 0\} & \text{if } C \text{ is an uncontrolled configuration,} \\ r(sg, C_j) & \text{if } C \text{ is a controlled configuration} \end{cases}$$

where $d_s(G^o, v_c, v_g)$ denotes the shortest distance from v_c to v_g in G^o, the optimistic induced graph of configuration C. Note that in the second clause of the above definition, the maximization is over only those successors with non-zero probabilities. To express the optimality criterion of minimizing the competitive ratio, we extend the definition of r into an evaluation function r' for the representing graphs as follows.

$$r'(rg, C) = \begin{cases} \frac{c(C)+d_s(G^o,v_c,v_g)}{d_s(G^o,v_s,v_g)} & \text{if } C \text{ is a terminal,} \\ max_j\{r'(rg, C_j)|q(C_j) > 0\} & \text{if } C \text{ is an uncontrolled configuration,} \\ min_j\{r'(rg, C_j)\} & \text{if } C \text{ is a controlled configuration.} \end{cases}$$

Note that the representing graph with this evaluation function is a minimax graph. An optimal solution graph of a representing graph with r' as its evaluation function represents an optimal plan minimizing the competitive ratio.

10.4.2 Minimizing the expected competitive ratio

Similarly, we define a function r_e to express the *expected competitive ratio* of a plan. The function is defined as follows.

$$r_e(sg, C) = \begin{cases} \frac{c(C)+d_s(G^o,v_c,v_g)}{d_s(G^o,v_s,v_g)} & \text{if } C \text{ is a terminal,} \\ \sum_j r_e(sg, C_j) * q(C_j) & \text{if } C \text{ is an uncontrolled configuration,} \\ r_e(sg, C_j) & \text{if } C \text{ is a controlled configuration.} \end{cases}$$

To express the optimality criterion of minimizing the expected competitive ratio, we extend the definition of r_e into an evaluation function r'_e for the representing graphs as follows.

$$r'_e(rg, C) = \begin{cases} \frac{c(C)+d_s(G^o,v_c,v_g)}{d_s(G^o,v_s,v_g)} & \text{if } C \text{ is a terminal,} \\ \sum_j r'_e(rg, C_j) * q(C_j) & \text{if } C \text{ is an uncontrolled configuration,} \\ min_j\{r'_e(rg, C_j)\} & \text{if } C \text{ is a controlled configuration.} \end{cases}$$

An optimal solution graph of a representing graph with r'_e as its evaluation function represents an optimal plan minimizing the expected competitive ratio.

10.4.3 Minimizing the worst case cost

The quality measurement of plans in terms of worst case cost can be given as a function c_w defined on solution graphs.

$$c_w(sg, C) = \begin{cases} c(C) + d_s(G^o, v_c, v_g) & \text{if } C \text{ is a terminal,} \\ max_j\{c_w(sg, C_j) | q(C_j) > 0\} & \text{if } C \text{ is an uncontrolled configuration,} \\ c_w(sg, C_j) & \text{if } C \text{ is a controlled configuration.} \end{cases}$$

To express the optimality criterion of minimizing the worst case cost, we extend the definition of c_w into an evaluation function c'_w for representing graphs as follows.

$$c'_w(rg, C) = \begin{cases} c(C) + d_s(G^o, v_c, v_g) & \text{if } C \text{ is a terminal,} \\ max_j\{c'_w(rg, C_j) | q(C_j) > 0\} & \text{if } C \text{ is an uncontrolled configuration,} \\ min_j\{c'_w(rg, C_j) & \text{if } C \text{ is a controlled configuration.} \end{cases}$$

Again, the representing graph with this evaluation function is a minimax graph.

10.4.4 Reaching one of the goal vertices

Suppose that an agent is given a U-graph G, a start vertex and a set V_g of goal vertices and is asked to reach any one of the goal vertices in V_g. The optimality criterion is to minimize the expected cost.

This variation can be reduced to a problem with a single goal vertex. To do so, we construct a new U-graph G' which is the same as G except that all the vertices in V_g are "collapsed" into a single vertex v_g. The original problem is equivalent to the problem of going to vertex v_g in G'.

10.4.5 Reaching multiple goal vertices

Suppose an agent is asked to visit all of the goal vertices in V_g^0 (in any order), instead of just one of them. In order to deal with this variation, we need to modify our formalization.

First, the definition of configurations needs to be changed as follows: a configuration is a quadruple $\langle G, v_c, V_g, c\rangle$ where G, v_c and c have the same meanings as before and V_g is a set of goal vertices yet to visit.

Next, the definition of terminals needs to be changed as follows: a *terminal* is a configuration $\langle G, v_c, \{\}, c\rangle$.

Third, the definitions of the composite successors and the composite transitions of controlled configurations are redefined as follows: Let $C = \langle G, v_c, V_g, c\rangle$ be a controlled configuration. Let $V_c = V^r \cup V_g$. V_c is called the *candidate vertex set* for the next move. If the shortest path from v_c to a vertex $v \in V_c$ in G^p contains no other vertex in V_c than v, then v is a vertex the agent can reach next by a (composite) transition. Vertex v is called a *next vertex* of C. Formally, let V' denote the set of all next vertices of C and let $d_s(G^p, v_c, v)$ denote the shortest distance from v_c to v in G^p. The successor set of C is given by:

$$\{\langle G, v, V_g - \{v\}, c + d_s(G^p, v_c, v)\rangle | v \in V'\}.$$

Finally, the function returning the expected cost of a plan is defined as follows:

$$Y(sg, C) = \begin{cases} 0 & \text{if } C \text{ is a terminal,} \\ \sum_j Y(sg, C_j) * q(C_j) & \text{if } C \text{ is an uncontrolled configuration,} \\ Y(sg, C') + c(C, C') & \text{if } C \text{ is a controlled configuration.} \end{cases}$$

Similarly, we can extend Y into an evaluation function Y' for the representing graphs.

Chapter 11

Computational Issues of U–graph Based Navigation

This chapter is concerned with computational issues of U–graph based navigation. We first discuss *off–line* and *on–line* paradigms, and then present algorithms for U-graph based navigation in both paradigms. Finally, we present some experimental data obtained for these algorithms.

11.1 Planning Paradigms

A plan for a U–graph based navigation task covers some or all contingencies which may arise during navigation. If a plan covers all possible contingencies for a navigation task, it is called a *complete plan*. The solution graphs of the representing graph of a navigation task represent complete plans.

U–graph based navigation can be carried out either in an off–line paradigm or in an on–line paradigm. In the off–line paradigm, the planner computes a complete plan for each navigation task. After computing such a complete plan, the planner will not be invoked during the navigation process.

In the on–line paradigm, the planner and the executive can be considered as co–routines. At any point during a navigation process, the current U–graph and the current position, together with the goal position and the cost incurred so far, constitute a configuration. The planner acts as an "oracle," telling the agent where to go next. Once the agent reaches an information collecting vertex, the agent can update the U–graph according to the actual status of the random edges incident from the vertex. As the result of the updating, a new configuration is reached. Note that a configuration here is essentially a controlled configuration. The following procedure, **Navigate**, captures this kind of interaction between the executive and the planner.

Navigate

Input: G: a U–graph v_s and v_g: a pair of vertices $\mathcal{P}$: an on–line planner

Output: the amount spent for the task

1. Set $v_c = v_s$ and $c = 0$.
2. If $v_c = v_g$, stop with cost c.
3. Call $\mathcal{P}$ with arguments G, v_c, and v_g to obtain a plan.
4. Follow the plan until either reaching v_g or reaching an information collecting vertex v.
5. Set $v_c = v$.
6. Increase c by the amount spent on following the plan.
7. Update G based on the new information on the uncertain edges adjacent to v_c and go to step 2.

11.2 Computing Complete Plans

In the previous chapter, we showed that a U–graph based navigation can be represented by a decision graph. A solution graph of the decision graph is a complete representation of a navigation plan. Computing a complete plan for a navigation task amounts to computing the corresponding solution graph from the decision graph. Thus, we can use the algorithms presented in Chapter 3 for off–line planning.

In order to apply those algorithms, we need to define heuristic functions to estimate minimum expected costs for configurations.

For a given navigation task, let rg denote the representing graph with the function X' defined in Section 10.3 as its min–exp evaluation function. One possible heuristic function, denoted f, is defined as follows:

$$f(C) = d_s(G^o, v_c, v_g)$$

where G^o is the optimistic induced graph of configuration C. Intuitively, $f(C)$ is the shortest distance between the current position and the goal position in the optimistic induced graph of configuration C. Thus $f(C) \leq X'(rg, C)$ for any configuration C. Therefore, f is admissible.

11.2.1 Some experimental data

In this section, we present some experimental data obtained when applying some of the algorithms discussed in Chapter 3 to U–graph based navigation. Our primary objective for carrying out these experiments is to compare the performance of those algorithms.

Problem Instances

In our experiments, we consider two classes of U–graphs. The U–graphs in Class 1 are randomly generated from grids (as shown in Fig.11.1) with parameters d_1, d_2, p_1, p_2 and two reference numbers r_1 and r_2. Here, d_1 and d_2 are two positive integers specifying the numbers of rows and columns of the grids; $0 \leq p_1 \leq 1$ specifies the probability that a connection (either an ordinary edge or an uncertain edge) exists between any pair of neighbouring vertices on the grids; p_2 specifies the probability of a connection, if it exists, being an uncertain edge. The reference numbers $r_1 > 1$ and $r_2 > 1$ are used to generate uniform distributions from which the weights of a U–graph are generated. A U–graph of this kind is an abstraction of the road layout of a city. We assume that the weight distributions of all uncertain edges are binary and independent of one another.

Given parameters d_1, d_2, p_1, p_2, r_1 and r_2, a random U–graph is generated as follows. First, $d_1 \times d_2$ vertices are generated, arranged in a d_1 by d_2 grid. Next, with probability p_1, a connection is generated between each pair of neighbouring vertices. Third, each connection is marked as an uncertain edge with probability p_2 and as an ordinary edge with probability $1 - p_2$. Finally, the weights for the connections of the graph are generated as follows. For each edge, we first randomly generate two numbers a_1 and a_2; a_1 is drawn from a uniform distribution between 1 and r_1, and a_2 from a uniform distribution between 1 and r_2. Then we construct a uniform distribution $\mathcal{D}$ with a lower bound a_1 and an upper bound $a_1 * a_2$. The weight of the edge is a random number generated from the uniform distribution $\mathcal{D}$. For each uncertain edge, we first generate a probability p from a uniform distribution between 0.01 and 0.99 and construct two uniform distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ in the same way as described above, and then generate two numbers c_1 and c_2 from the uniform distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. The weight of the uncertain edge is a binary random variable having value c_1 with probability p and c_2 with probability $1-p$. For each randomly generated U-graph, we assume the navigation task is to go from the upper-left corner to the lower-right corner on the grid. We discard those U-graphs whose optimistic induced graphs are disconnected with respect to the two corners.

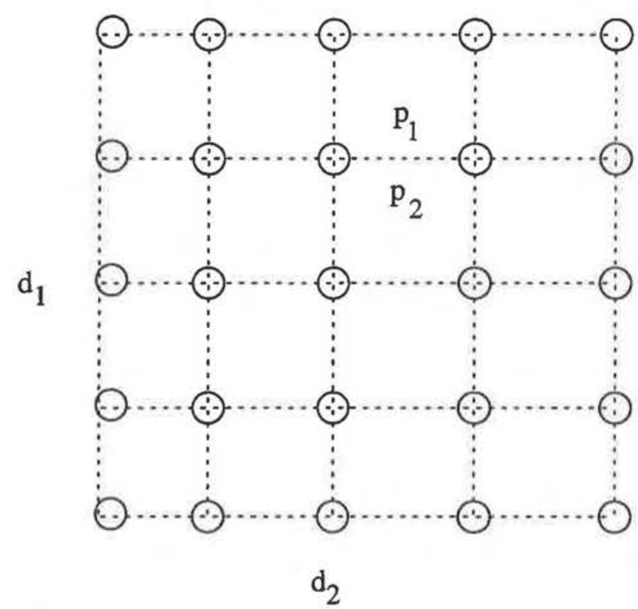

Figure 11.1: A U-graph in Class 1 — a representation of city roads

The U-graphs in Class 2 are randomly generated from a structure as shown in Fig.11.2, which is a model of two parallel-highway systems joined by a bipartite graph. The U-graphs are generated with four parameters: n, p_1, r_1 and r_2. Here r_1 and r_2 are used in the same way as in Class 1, n is the number of parallel highways, and p_1 is the probability that an ordinary edge exists between a pair of vertices, one in each half of the bipartite graph. Again, we assume the weight distributions of all uncertain edges are binary and are independent of one another.

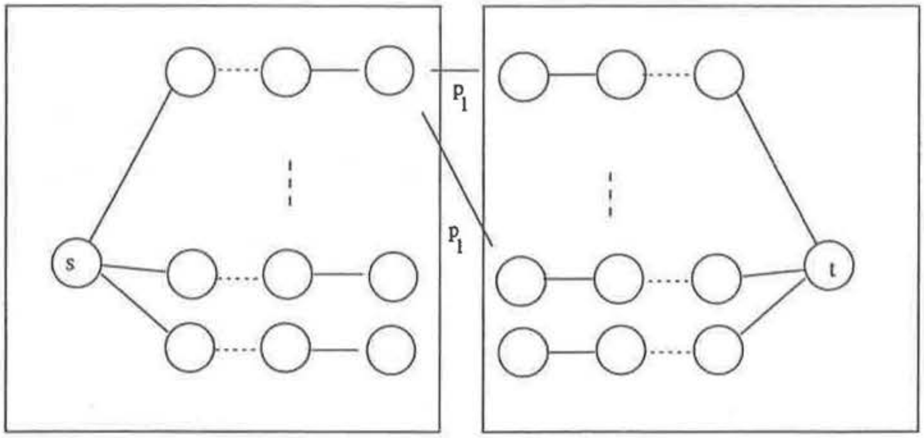

Figure 11.2: A U–graph in Class 2 — an abstraction of a parallel highway system

Given parameters n, p_1, r_1 and r_2, a random U–graph is generated as follows. First, two partial graphs as shown in the boxes of Fig.11.2 are generated. Each partial graph has $3n+1$ vertices, $2n$ edges and n uncertain edges. Next, with probability p_1, an edge is generated between each pair of vertices, one on the right boundary of the left partial graph and one on the left boundary of the right partial graph. Finally, the weights of the ordinary edges and the weight distributions of the uncertain edges are generated in the same way as for the U–graphs in Class 1. For each randomly generated U–graph, we assume that the navigation task is to go from vertex s to vertex t. Again, we discard those U–graphs whose optimistic induced graphs are disconnected with respect to vertices s and t.

Note that in the process of generating a random U–graph, the weights are generated from uniform distributions that are also randomly constructed. We expect that this treatment reduces the correlation among the connection weights in a U–graph. The Lisp functions that are used in our experiments for generating random graphs are included in Appendix A.

Experiment 1

Our first experiment compares the performance of algorithms AO* and DFS. AO* is a best–first heuristic search algorithm. DFS is a depth–first heuristic search algorithm. We mentioned in Chapter 3 that the primary advantage of DFS over AO* is that it needs only linear space (if the solution graph need not be explicitly constructed). We want to test if DFS examines fewer nodes than AO* does for randomly generated problems. Because structure sharing in a decision graph of a U–graph based navigation task is not substantial, we did not make use of structure sharing in our implementation of AO*. Thus, a decision graph is essentially treated as an unfolded tree, and we avoid high overhead for checking whether a node is already in the graph. The algorithms are implemented in Common Lisp.

In our first experiment, we made two hundred *successful trials*. A trial consists of generating a problem instance and applying DFS and AO* to the problem instance. A trial is *aborted* if it cannot be finished in a reasonable period of time (typically an hour on a Sparc–2 machine).

Of the two hundred successful trials, one hundred are generated from Class 1. The parameters used for generating U–graphs are set up in the following way:

with probability 0.5, d_1 and d_2 are both set to 5; with probability 0.5, d_1 is set to 4 and d_2 to 6;

p_1 is a random number generated from the uniform distribution between 0.6 and 0.9; p_2 is a random number generated from the uniform distribution between 0.2 and 0.5;

r_1 is a random number generated from the uniform distribution between 100 and 200; r_2 is a random number generated from the uniform distribution between 10 and 20;

Another hundred problem instances are generated from Class 2. The parameters used for generating U–graphs are set up in the following way:

$n = 5$;

p_1 is a random number generated from the uniform distribution between 0.5 and 0.8;

r_1 is a random number generated from the uniform distribution between 100 and 200; r_2 is a random number generated from the uniform distribution between 10 and 20.

We measured for each (successful) trial the number of nodes examined by DFS and by AO*. For seventy–seven problems out of one hundred in Class 1, DFS examined fewer nodes than AO* did. For fifty–two problems out of one hundred in Class 2, DFS examined fewer nodes than AO* did.

Moreover, our experiments showed that, for most of the problems, DFS spent less time, even when it examined more nodes than AO* did. This is due to the fact that DFS is a depth first algorithm, involving less overhead, and suggests that DFS is better suited for U–graph based navigation.

Experiment 2

If the admissibility of a heuristic function for a given problem cannot be assured, the solution computed by DFS may not be optimal. An algorithm with a particular heuristic function can be evaluated by two factors: its speed and the expected navigation cost it induces.

Our second experiment is for testing the effect of heuristic functions on DFS. This experiment casts some light on the tradeoff between computational time and solution quality. DFS is tested with four different heuristic functions: h_0, h_1, h_2 and h_3, defined as follows.

$h_0(C) = f(C)$,

$h_1(C) = h_0(C)/(1-\epsilon)$,

$h_2(C) = h_0(C)/(1-\epsilon)^2$ and

$h_3(C) = h_0(C)/(1-\epsilon)^3$

where $\epsilon = 0.2$. Heuristic functions h_1, h_2 and h_3 are not admissible. Theorems 3.4 and 3.5 give quality bounds for DFS with these heuristic functions.

For this experiment, we also made two hundred successful trials using the same approach to trial construction as in the first experiment. For each successful trial, we measured the cost of the solution graph and the number of nodes visited by the algorithm for each heuristic function. More specifically, for each problem instance, we measured the following data (for $i = 0, 1, 2$ and 3):

- c_i: the cost of the solution graph returned by DFS with heuristic function h_i.
- n_i: the number of nodes examined by DFS with heuristic function h_i.

From the measured data, we computed the following data:

$cr_j = c_j/c_0, \quad sr_j = n_0/n_j, \quad \text{for } j = 1, 2, 3$

Intuitively, cr_j means the *cost ratio* of the solution returned by DFS with heuristic function h_j to the cost of the optimal solution; sr_j means the *speedup ratio* of DFS with heuristic function h_j to DFS with heuristic function h_0. We use the cost ratios to measure the quality of the algorithm with inadmissible heuristic functions, and

Table 11.1: The average speedup ratios of Algorithm DFS

	h_1	h_2	h_3
Class 1	1.54	2.07	2.41
Class 2	2.77	9.71	23.04

Table 11.2: The average cost ratios of Algorithm DFS

	h_1	h_2	h_3
Class 1	1.011	1.031	1.041
Class 2	1.012	1.064	1.131

use the speedup ratios to measure the performance of the algorithm with inadmissible heuristic functions. Table 11.1 contains the average speedup ratios. Table 11.2 contains the corresponding average cost ratios.

From these tables, we observe that the average cost ratios of DFS with heuristic functions h_1, h_2 and h_3 are all quite close to one. This implies that, even though these heuristic functions are not admissible, they usually give very conservative estimates. Therefore, there is a great potential to obtain *more informed* heuristic functions.

Experiment 3

This experiment tests the effect of another heuristic function, h', on DFS. Like heuristic functions h_1, h_2 and h_3, h' is not admissible. Unlike those functions, we do not have a bound on the admissibility of heuristic function h'.

Heuristic function h' is defined as follows:

$$h'(C) = d_s(G^a, v_c, v_g)$$

where G^a is an induced graph of the U–graph G of C in which each uncertain edge is replaced with an edge whose weight is the mean value of the weight distribution of the uncertain edge.

This experiment showed that, with heuristic function h', the average cost ratio and the average speedup ratio of DFS for the problems in Class 1 were 1.006 and 39.49, respectively; the average cost ratio and the average speedup ratio of DFS for the problems in Class 2 were 1.074 and 23.18, respectively. From these data, we made the following observation:

> For the U–graphs in both classes, DFS with heuristic function h' outperformed DFS with heuristic functions h_1, h_2 and h_3. This was especially true for the U–graphs in Class 1. For the U–graphs in this class, the average cost ratio of DFS with the heuristic function h' was 1.006, less than the cost ratio of DFS with the heuristic function h_1; the average speedup ratio was almost 40, far greater than the speedup ratios of DFS with the heuristic function h_3.

The good performance of DFS with the heuristic function h' can be attributed to the fact that *more* domain–dependent knowledge is encoded in h' than in h_1 or h_2 or h_3. This is another illustration of the importance of domain–dependent knowledge for decision graph search in particular and for heuristic search in general.

11.3 On–Line Planning

In this section, we first discuss the issue of characterizing the quality of an on–line planner and then develop a polynomial algorithm for on–line planning. Finally, we present some experimental data. In our experiments, we compare the navigation quality of our algorithm with the navigation quality of another simple on–line algorithm given by Polychronopoulos [71]. The experimental data show that our algorithm results in good navigation quality and is better than Polychronopoulos's on–line algorithm in terms of navigation quality.

11.3.1 The optimality characterization of on–line planners

The navigation quality of an on–line planner can be measured in terms of the *expected cost* for given navigation tasks.

Let $J = \langle G, v_s, v_g \rangle$ be the navigation task of going from v_s to v_g in U–graph G, let $\mathcal{P}$ be an on–line planner the agent uses for the navigation task and let $C_a(\mathcal{P}, J)$ denote the *expected cost* of J induced by planner $\mathcal{P}$. Cost $C_a(\mathcal{P}, J)$ can be determined by simulating the navigation process under the guidance of $\mathcal{P}$ against all of the possible realizations of the U–graph G.

Suppose $C_a^*(J)$ is the minimal expected cost of J. The *cost ratio* of $\mathcal{P}$ with respect to task J is defined as $C_a(\mathcal{P}, J)/C_a^*(J)$. We say that the planner $\mathcal{P}$ will result in an *optimal navigation* for task J if the cost ratio is unity. We say the planner is *optimal* if $C_a(\mathcal{P}, J)/C_a^*(J) = 1$ for any navigation task J.

11.3.2 A graph transformation based algorithm

The basic idea behind this algorithm is to substitute for the uncertain edges in a U–graph by edges with "appropriate weights". In so doing, we hope that the "net effect" of the uncertain edges in a U–graph can be approximated by a set of *substituting edges*. Suppose we are given the task of navigating from vertex v_c to vertex v_g in U–graph G. Assume that the uncertain edges of U–graph G can be ordered as $u_1, u_2, ..., u_l$. Let $W = \langle w_1, w_2, ..., w_l \rangle$ be a vector of weights. We call W a *substituting weight vector*. Let $G(W)$ denote the graph obtained by replacing the uncertain edges, $u_1, u_2, ..., u_l$, in U–graph G respectively by edges $e_1, e_2, ..., e_l$ where e_i and u_i have the same incident vertices and e_i has weight w_i, for $1 \leq i \leq l$. The central problem here is to compute an "appropriate" substituting weight vector W for a given configuration. We first define ideal substituting weight vectors.

A vector $W = \langle w_1, ..., w_l \rangle$ is an *ideal substituting weight vector* for U–graph G and the goal vertex v_g, if the following two conditions are met:

1. for any vertex v in G, the shortest distance from v to vertex v_g in graph $G(W)$ is equal to the minimal expected cost for the configuration $\langle G, v, v_g, c \rangle$;
2. the initial segment of the shortest path from v to v_g is consistent with the optimal next move for the configuration $\langle G, v, v_g, c \rangle$.

Note that, in the above definition, the second condition is sufficient for optimal navigation. However, as we will see, the above definition facilitates a way to compute an "appropriate" substituting weight vector.

It should also be noted that, if we have an algorithm to compute an ideal substituting weight vector for any configuration, we in effect have an on–line navigation algorithm that always results in optimal navigation. However, since the problem of optimal navigation is #–P hard [71], it should not be surprising if the substituting weight vectors computed by a polynomial algorithm are not ideal.

Suppose we have magically obtained an ideal substituting vector $W = \langle w_1, ..., w_l \rangle$ and the corresponding substituting graph $G(W)$; and suppose we happen to forget the value of w_i for some i, $1 \leq i \leq l$. We want to make a good guess at the value of w_i on the basis of the known information.

Suppose $u_i = \langle v_1, v_2 \rangle$. Let u_i^o denote the optimistic realization and let u_i^p denote the pessimistic realization of u_i. For each $a \in \Omega_{u_i}$, let $W(a,i) = \langle w_1', ..., w_l' \rangle$ be a weight vector such that $w_i' = a$ and $w_j' = w_j$ for all j with $1 \leq j \leq l$ and $j \neq i$; let $D(G, W, i, a)$ denote the absolute difference between the shortest distances from v_1

to v_g and from v_2 to v_g in graph $G(W(a,i))$. Intuitively, $D(G,W,i,a)$ reflects the difference between the expected costs for an agent to go to vertex v_g from vertices v_1 and v_2 in the U–graph under the condition that uncertain edge u_i has weight a.

We define $e(G,W,i)$ as follows:

$$e(G,W,i) = max\{u_i^o, \sum_{a \in \Omega_{u_i}} P\{u_i = a\} * D(G,W,i,a)\}.$$

Intuitively, $e(G,W,i)$ is the weighted sum of $D(W,i,a)$ for each $a \in \Omega_{u_i}$.

Lemma 11.1 *For any i, $e(G,W,i) \leq c_a(u_i)$ where $c_a(u_i)$ is the expected value of the weight distribution of the uncertain edge u_i.*

Proof. We simply note that $D(G,W,i,a) \leq a$ in the above definition of $e(G,W,i)$. □

Function e can be extended to a vector function $\hat{e}$ such that $\hat{e}(G,W)$ is a new vector $W' = \langle w_1', ..., w_l' \rangle$ satisfying $w_i' = e(G,W,i)$. We use $\hat{e}(G,W)$ as an estimate of W. We say a weight vector W is *appropriate* if it is a fix–point of $\hat{e}$, i.e.,

$$\hat{e}(G,W) = W.$$

The above condition imposes l equations with l variables. Since these equations are not linear, it is hard to obtain an analytic solution. However, we can approximate their solution by iteration.

Practically, for any initial vector W_0, we compute a sequence $W_0, ..., W_j, W_{j+1}, ...$ satisfying $W_{j+1} = \hat{e}(G,W_j)$, $i \geq 0$. Vector W_j can be regarded as a good estimate to the solution if W_{j-1} and W_j are close enough. An iteration algorithm based on this idea for computing appropriate substituting weight vectors is given as follows.

Algorithm AA

Input: a U–graph G, a current vertex v_c and a goal vertex v_g.

Output: A substituting vector.

1. Set $j = 0$, $W_0 = (c_a(u_1), ..., c_a(u_l))$.

2. Set $W_{j+1} = \hat{e}(G, W_j)$.

3. Set $j = j + 1$.

4. If W_{j-1} and W_j are close enough, return W_j, otherwise, go back to step 2.

The condition for termination at step 4 in the above algorithm can be tailored to different situations. For example, if an agent is in an urgent situation, it may adopt a very loose condition; otherwise it can choose a more restricted one. One candidate condition could be that the sum of the absolute differences between the element pairs from W_j and W_{j-1} is less than a given threshold. Our experimental data show that a small number of iterations is good enough for the two classes of randomly generated problems. We will see this later in this section.

The time complexity of this algorithm can be roughly estimated as follows. In each iteration, we need to compute a new weight for each uncertain edge u. This computation involves $|\Omega_u|$ calls of a shortest distance algorithm. Therefore, the time complexity of AA is $O(k * l * b * s)$ where k is the number of iterations, l is the number of uncertain edges, b is the average size of the frames of the uncertain edges and s is the time complexity of the shortest distance algorithm used in AA. The size of the space required by the algorithm is $O(E + b * l)$, which is of the same order as the size of the U–graph.

With the help of algorithm AA, an on–line planning algorithm is defined as follows.

Algorithm AA1

Input: G, v_c and v_g

Output: A plan specifying where to go next

1. If $\langle G, v_c, v_g, - \rangle$ is a terminal, return a shortest path from v_c to v_g in G^p, the pessimistic induced graph of G.
2. Use algorithm AA to compute a substituting vector W.
3. Compute the substituting graph $G(W)$.
4. Compute a shortest path from v_c to v_g in $G(W)$.
5. Output the initial segment (up to the first uncertain edge) of the path.

The next issue is the size of our algorithm's average cost ratio. Unfortunately, we do not have theoretical results for general cases. In the next subsection, we present some experimental results which suggest that the cost ratio is quite low.

11.3.3 Experimental results

In this section, we present some experimental results on our on–line planning algorithm AA1 and another simple on–line planning algorithm AA2, which is given by Polychronopoulos [71].

AA2

Input: G, v_c and v_g

Output: A plan specifying where to go next

1. Compute a shortest path between v_c and v_g in G^a, where G^a is obtained from G by replacing each uncertain edge with an ordinary edge whose weight is equal to the mean value of the weight distribution of the uncertain edge.

2. Output the initial segment of the path.

AA2 has the same time complexity as that of the shortest distance path algorithm it uses.

This experiment was conducted in the same way as were the previous experiments. The difference was that for each trial, we measured the cost ratios of AA1 and AA2 for the problem instance of the trial as follows. We first generated a problem instance for each trial, then computed the optimal expected cost of the problem by calling DFS. To determine the expected cost that AA1 (AA2) will incur for the problem, we simulated AA1 (AA2) against all of the possible realizations of the U–graph of the problem instance. Each simulation may involve up to l calls of AA1 (AA2) where l is the number of uncertain edges of the U–graph. Thus, up to $l * 2^l$ calls of AA1 (AA2) may be needed in order to determine the expected cost that AA1 (AA2) will incur for the problem.

The experimental results are summarized in Table 11.3. The two rows of the table correspond to the two problem classes. The data in column AA2 are the average cost ratios of AA2 for the problems in the two classes. The data in the AA1/i columns are the average cost ratios of AA1 with iteration time i for the problems in the two classes. From these tables we observe that the average cost ratios of both algorithms are all close to one and that the average cost ratio of AA1 is lower than that of AA2.

It is not hard to explain why AA1 is better on average than AA2. In both algorithms, the initial segment of the shortest path between the current position and the goal position in a substituting graph is computed as the plan. The difference, however, lies in the ways that the weights of the substituting edges are computed. In AA2, the weight of an edge substituting for an uncertain edge is the mean value of the weight distribution of the uncertain edge, thus no information on other uncertain edges is integrated in the computation. On the other hand, in AA1, the weight of an edge substituting for an uncertain edge is computed with the following kinds of information being taken into account: the goal vertex; the weight distribution of the uncertain edge; the structure of the entire U–graph; and the information on the other uncertain edges. Although yet to be verified through real applications, the initial experimental results suggest that AA1 exhibits good quality for U–graph based navigation.

Finally, we would like to make a note on the (lack of) convergence property of the algorithm AA1. The main component of AA1 is AA, which is an iterative algorithm taking a set of non–linear equations as its input and computing an approximation of a "fix–point" of the equations. However, there is no guarantee that the outputs of AA can converge to a "fix–point" as iteration time increases. It is not known whether the equations have a fix–point. The quality of AA1 is not guaranteed to improve monotonically as the the number of iterations increases. This is also reflected in the results in Table 11.3. The average cost ratios of AA1 fluctuate as iteration time increases.

Table 11.3: The average cost ratios of the on–line algorithms

	AA2	AA1/1	AA1/2	AA1/3	AA1/4	AA1/5	AA1/6
class 1	1.060	1.044	1.042	1.042	1.041	1.032	1.042
class 2	1.082	1.014	1.011	1.008	1.010	1.009	1.010

Chapter 12

Conclusions

In this thesis, we took a uniform approach to computational issues of the problems with decision making with uncertainty. We proposed decision graphs as a simple intermediate representation, for the decision making problems, and developed some algorithms based on this representation.

These algorithms can be readily applied to decision problems given in the form of decision trees, since decision graphs are a generalization of decision trees. It is also straightforward to apply these algorithms to decision problems given in the form of finite stage Markov decision processes, since a problem in such form can be represented as a decision graph. In order to make use of these algorithms for solving decision making problems in influence diagrams, we developed a stochastic dynamic programming formulation of the problem of influence diagram evaluation and presented a method to systematically transform a decision making problem in influence diagram representation into a decision graph representation. In effect, we obtained a two–phase method for influence diagram evaluation. In comparison with other algorithms in the literature for influence diagram evaluation, our method has a few notable merits. First, it exploits asymmetry of decision problems in influence diagram evaluation, which leads to exponential savings in computation for typical decision problems.

Second, by using heuristic search techniques, it provides an explicit mechanism for making use of heuristic information that may be available in a domain–specific form. Third, because it provides a clean interface between influence diagram evaluation and Bayesian net evaluation, various well–established algorithms for Bayesian net evaluation can be used in influence diagram evaluation. Finally, by using decision graphs as an intermediate representation, the value of perfect information [53] can be computed more efficiently [110].

High–level navigation in uncertain environments can be viewed as a decision problem with uncertainty, but is given neither in the form of Markov decision processes, nor in the form of influence diagrams. We developed a decision theoretic formulation of the problem and showed how to represent such a problem in a decision graph. As a result, we can use the decision search algorithms for off–line path planning.

Since the problem is of importance in its own right, we also developed an on–line path planning algorithm with polynomial time complexity. Experimental results show that the on–line algorithm results in satisfactory navigation quality.

12.1 Contribution Summary

The contributions of this thesis are summarized as follows.

- A number of algorithms for decision graph search.
- A new method for influence diagram evaluation.
- A decision theoretic formalization of the problem of U–graph based navigation, and a general approach to the computation of optimal plans for U–graph based navigation tasks.

- A polynomial time heuristic on–line algorithm for U–graph based navigation.

12.2 Future Research

Future research can be carried out in several directions. For decision graph search, we would like to have a more thorough examination of the (absolute and/or comparative) performances of the decision graph search algorithms presented in Chapter 3. A theoretical examination, as has been taken for minimax tree search [38, 68], would be very interesting, but is likely to be challenging. An experimental examination may also be valuable for a better understanding on the performances of the algorithms. Another possible research direction is, as for other search problems [43], to investigate parallel algorithms for decision graph (tree) search.

Our work on influence diagram evaluation can be extended in at least two aspects. First, our method can be generalized to handle *irregular* stepwise decomposable influence diagrams as well. To do so, we need to introduce a new kind of node to our decision graphs: *sum nodes* to capture parallel decision threads. Second, an influence diagram evaluation system using our method is yet to be built. With such a system, we can experimentally compare the performance of our method with other algorithms. This would also provide a platform for experimentally comparing various decision graph search algorithms.

The analysis we performed in Section 7.5.1 on potential savings by exploiting asymmetry of decision problems is quite conservative. We expect that an average case based analysis would reveal greater potential of savings.

Some interesting future work related to U–graph based navigation would be to apply the approach to a practical autonomous navigation system. By experimenting with such a practical system, we could realistically evaluate the value of taking uncertainty into consideration at high level planning stages in general, and assess the practical value of U–graphs and U–graph based navigation theory in particular.

Some theoretical questions about U–graph based navigation remain open. For example, can we find an interesting class of navigation problems for which optimal polynomial algorithms exist? Can we find a good bound on the cost ratio of AA1?

Bibliography

[1] R. C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *IEEE International Conference on Robotics and Automation*, pages 264–271, 1987.

[2] R. C. Arkin. Navigational path planning for a vision based mobile robot. *Robotica*, 7:43–48, 1989.

[3] B. W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[4] A. Bar-Noy and B. Schieber. The Canadian traveller problem. In *Proc. 2nd Annul ACM-SIAM Symposium on Discrete Algorithms*, pages 261–270, 1991.

[5] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[6] D. Bertsekas. Dynamic behavior of shortest path algorithms for communication networks. *IEEE transaction on Automatic Control*, AC-27(1), Feb. 1982.

[7] R. Bhatt, D. Gaw, and A. Meystel. A real time guidance system for an autonomous vehicle. In *IEEE Conference on Robotics and Automation*, pages 1785–1791, 1987.

[8] M. Boddy. Anytime problem solving using dynamic programming. In *Proc. of AAAI-91*, pages 360–365, Anaham, CA., USA, 1991.

[9] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource–bounded practical reasoning. *Computational Intelligence*, 4(4):349–255, 1988.

[10] P. P. Chakrabarti, S. Ghose, and S. C. DeSarkar. Admisibility of AO* when heuristics overestimate. *Artificial Intelligence*, 34(1):97–113, 1987.

[11] G. F. Cooper. A method for using belief networks as influence diagrams. In *Proc. of the Fourth Conference on Uncertainty in Artificial Intelligence*, pages 55–63, Univ. of Minnesota, Minneapolis, USA, 1988.

[12] Z. Covaliu and R. M. Oliver. Formulation and solution of decision problems using decision diagrams. Technical report, University of California at Berkeley, April 1992.

[13] J. L. Crowley. Navigation for an intelligent mobile robot. *IEEE Robotics and Automation*, RA-1(1):31–41, 1985.

[14] E. Davis. *Representing and Acquiring Geographic Knowledge.* London: Pitman Publishing, 1986.

[15] T. Dean, K. Basye, R. Chekaluk, S. Hyun, M. Lejter, and M. Randazza. Coping with uncertainty in control system for navigation and exploration. In *Proc. of AAAI–90*, pages 1010–1015, 1990.

[16] T. Dean, R. J. Firby, and D. Miller. Hierarchical planning involving deadlines, travel time and resources. *Computational Intelligence*, 4(4):381–398, 1988.

[17] T. Dean, L. P. Kaebling, J. Kirman, and A. Nicholson. Deliberation scheduling for time-critical sequential decision making. In *Proc. of the Ninth Conference on Uncertainty in Atificial Intelligence*, pages 309–316, Washington, DC, 1993.

[18] T. Dean, L. P. Kaebling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proc. of AAAI–93*, pages 574–579, Washington, DC, 1993.

[19] T. L. Dean and M. P. Wellman. *Planning and Control.* Morgan Kaufmann, 1992.

[20] C. Derman. *Finite State Markovian Decision Process.* Academic Press New York and London, 1970.

[21] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Math.*, 1:269–271, 1959.

[22] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Robotic exploration as graph construction. Technical Report RBCV-TR-88-23, Research in Biological and Computational Vision, University of Toronto, November 1988.

[23] J. A. Feldman and R. F. Sproull. Decision theory and Artificial Intelligence II: The hungry monkey. *Cognitive Science*, 1, 1977.

[24] R. J. Firby. An investigation into reactive planning in complex domains. In *Proc. of AAAI–87*, pages 202–206, 1987.

[25] P. C. Fishburn. *The Foundations of Expected Utility.* Dordrecht, Holland: Reidel, 1982.

[26] R. M. Fung and R. D. Shachter. Contingent influence diagrams, 1990.

[27] M. P. Geogeff and A. L. Lansky. Reactive reasoning and planning: an experiment with a mobile robot. In *Proc. of AAAI-87*, pages 677–682, 1987.

[28] M. P. Georgeff and F. F. Ingrand. Decision–making in an embedded reasoning system. In *Proc. of IJCAI-89*, pages 972–978, Detroit, USA, 1989.

[29] P. Haddawy and S. Hanks. Representations for decision-theoretic planning: Utility functions for deadline goal. In B. Nebel, C. Rich, and W. Swartout, editors, *Proc. of the Fourth International Conference on Knowledge Representation and Reasoning*, pages 71–82, Cambridge, Mass., USA, Oct. 1992. Morgan Kaufmann.

[30] L. R. Harris. The heuristic search under conditions of error. *Artificial Intelligence*, 5(3):217–234, 1974.

[31] E. J. Horvitz, J. S. Breese, and M. Henrion. Decision theory in expert systems and Artificial Intelligence. *International Journal of Approximate Reasoning*, 2:247–302, 1988.

[32] E. J. Hovitz. Reasoning about beliefs and actions under computational resource constraints. In *Uncertainty in Artificial Intelligence, Volume 3.* Armsterdam: North Holland, 1988.

[33] R. A. Howard. *Dynamic Programming and Markov Processes.* Technology Press, Cambridge, Massachusetts, and Wiley New York, 1960.

[34] R. A. Howard. The used car buyer problem. In R. A. Howard and J. E. Matheson, editors, *The Principles and Applications of Decision Analysis, Volume II*, pages 690–718. Strategic Decision Group, Mento Park, CA., 1984.

[35] R. A. Howard and J. E. Matheson. Influence diagrams. In R. A. Howard and J. E. Matheson, editors, *The Principles and Applications of Decision Analysis, Volume II*, pages 719–762. Strategic Decision Group, Mento Park, CA., 1984.

[36] F. V. Jensen, K. G. Olesen, and K. Anderson. An algebra of Bayesian belief universes for knowledge based systems. *Networks*, 20:637–659, 1990.

[37] L. Kleinrock. *Queueing Systems.* John Wiley and Sons, 1976.

[38] D. E. Knuth and R. W. Moore. An analysis of alpha beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[39] S. Koenig. Optimal probabilistic and decision theoretic planning using markov decision theory. Technical Report UCB-CSD-92-685, Computer Science Division (EECS), University of California, Berkeley, 1992.

[40] R. E. Korf. Depth first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[41] B. J. Kuipers. Modeling spatial knowledge. *Cognitive Science*, 2:129–153, 1978.

[42] B. J. Kuipers and Tod S. Levitt. Navigation and mapping in large–scale space. *AI Magzine*, 9(2):25–43, 1988.

[43] V. Kumar, P.S. Gopalakrishnan, and L. N. Kanal (Eds). *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, 1990.

[44] V. Kumar and L. Kanal. The cdp: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 1–27. Springer-Verlag, 1988.

[45] V. Kumar, D. S. Nau, and L. Kanal. A general branch-and-bound formulation for AND/OR graph and game tree search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 91–130. Springer-Verlag, 1988.

[46] C. P. Langlotz and E. H. Shortliffe. Logical and decision-theoretic methods for planning under uncertainty. *AI Magazine*, 10(Spring):39–47, 1989.

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

[48] T. S. Levitt and D. T. Lawton. Qualitative navigation for mobile robots. *Artificial Intelligence*, 44(3):305–360, 1990.

[49] T. A. Linden and J. Glicksman. Contingency planning for an autonomous land vehicle. In *Proc. IJCAI-87*, pages 1047–1054, Milan, Italy, 1987.

[50] A. Mahanti and A. Bagchi. AND/OR graphs heuristic search methods. *J. ACM*, 32(1):28–51, 1985.

[51] M. S. Mannasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for on–line problems. In *the 20th ACM Symp. on Theory of Computing*, pages 322–333, 1988.

[52] A. Martelli and U. Montanari. Additive and/or graphs. In *Proc. of IJCAI-73*, pages 1–7, Stanford, CA., USA, 1973.

[53] J. E. Matheson. Using influence diagrams to value information and control. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, pages 25–63. John Wiley and Sons, 1990.

[54] J. M. McQuillan. The new routing algorithm for the arpanet. *IEEE Transactions on Communications*, COM-28:711–719, May 1980.

[55] A. Meystel. Planning in a hierarchical nested autonomous control system. In *SPIE Mobile Robots*, pages 42–76, 1986.

[56] A. C. Miller, M. M. Merkhofer, R. A. Howard, J. E. Matheson, and T. T. Rice. Development of automated aids for decision analysis. Technical report, Stanford Research Institute, 1976.

[57] D. P. Miller and M. G. Slack. Global symbolic maps from local navigation. In *Proc. of AAAI-91*, pages 750–755, 1991.

[58] J. S. B. Mitchell, D. W. Payton, and D. M. Keisey. Planning and reasoning for autonomous vehicle control. *International Journal of Intelligent Systems*, 2:129–198, 1987.

[59] B. G. Mobasseri. Path planning under uncertainty: From a decision analytic perspective. In *IEEE International Symposium on Intelligent Control*, pages 556–560, 1989.

[60] B. G. Mobasseri. Decision analytic approach to weighted region problem. In *SPIE Mobile Robots, V*, pages 438–445, 1990.

[61] P. Ndilikilikesha. Potential influence diagrams. Technical report, Business School, University of Kansas, 1991. Working paper No. 235.

[62] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag Berlin Heidelberg New York, 1982.

[63] J. J. Nitao and A. M. Rarodi. An intelligent pilot for an autonomous vehicle system. In *IEEE International Conference on Robotcs and Automation*, pages 176–183, 1985.

[64] S. M. Olmsted. *On representing and Solving Decision Problems*. PhD thesis, Engineering Economics Department, Stanford University, 1983.

[65] C. H. Papadimitriou. Games against nature. *Journal of Computer and System Science*, 31(2), October 1985.

[66] C. H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. In *Proc. the Sixteenth ICALP, Lecture Note in Comp. Sci. No. 372*, pages 610–620. Spring–Verlag, July 1989.

[67] B. G. Patrick. Binary iterative deepening A*: An admissible generalization of IDA* search. In *Proc. of Ninth Canadian Conference on Artificial Intelligence*, pages 54–59, Vancouver, Canada, 1992.

[68] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, 1984.

[69] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, Los Altos, CA, 1988.

[70] L. D. Phillips. Discussion of 'From Influence to Relevance to Knowledge by R. A. Howard'. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, page 22. John Wiley and Sons, 1990.

[71] G. H. Polychronopoulos. *Stochastic and Dynammic Shortest Distance Problems.* PhD thesis, MIT, Operations Research Center, Technical Report 199, May 1992. Technical Report 199.

[72] D. Poole. Probabilistic Horn Abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[73] G. Provan and D Poole. The utility of consistency-based diagnosis. In *Proc. of the Third International Conference on Knowledge Representation and Reasoning*, pages 461–472, 1991.

[74] M. L. Puterman. Markov decision processes. In D. P. Heyman and M. J. Sobel, editors, *Handbooks in Operations Research and Management Science, Volume 2.* Elsevier Science Publishers B. V. (North-Holland), 1990.

[75] R. Qi. A new method for network routing: a preliminary report. In *the Proc. of Pacific Rim Conference on Communications, and Computers and Signal Processing*, 1993.

[76] R. Qi and D. Poole. High level path planning with uncertainty. In B. D. D'Ambrosio, P. Smet, and P. P. Bonissone, editors, *Proc. of the Seventh Conference on Uncertainty in AI*, pages 287–294, UCLA, Los Angeles, USA, 1991. Morgan Kaufmann.

[77] R. Qi and D. Poole. A framework for high level path planning with uncertainty. In *Proc. of the Second Pacific Rim International Conference on Artificial Intelligence*, pages 287–293, Seoul, Korea, 1992.

[78] R. Qi and D. Poole. Two algorithms for decision tree search. In *Proc. of the Second Pacific Rim International Conference on Artificial Intelligence*, pages 121–127, Seoul, Korea, 1992.

[79] H. Raiffa. *Decision Analysis.* Addison–Wesley Publishing Company, 1968.

[80] S. M. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, 1983.

[81] S. Russell. Fine–grained decision–theoretic search control. In *Proc. Sixth Conference on Uncertainty in Artificial Intelligence*, 1990.

[82] S. Russell and E. Wefald. Principles of metareasoning. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proc. of the First International Conference on Knowledge Representation and Reasoning*, pages 400–411, Cambridge, Mass., USA, 1989.

[83] L. J. Savage. *The Foundations of Statistics.* Dover, 1954.

[84] Marcel J. Schoppers. Representation and automatic synthesis of reaction plans. Technical Report UIUCDCS–R–89–1546, Department of Computer Science, University of Illinois at Urbana–Champaign, 1989.

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

[86] R. D. Shachter. An ordered examination of influence diagrams. *Networks*, 20:535–563, 1990.

[87] R. D. Shachter, B. D'Ambrosio, and B. A. Del Favero. Symbolic probabilistic inference in belief networks. In *Proc. of AAAI-90*, pages 126–131, 1990.

[88] R. D. Shachter and M. A. Peot. Decision making using probabilistic inference methods. In *Proc. of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 276–283, San Jose, CA., USA, 1992.

[89] S. Shafer and W. Whittaker. Development of an integrated mobile robot system at Carnegie Mellon University. Technical Report CNU-RI-TR-90-12, The Robotics Institute, Carnegie Mellon University, January 1990.

[90] P. P. Shenoy. Valuation–based systems for Bayesian decision analysis. Technical Report working paper No. 220, School of Business, University of Kansas, April 1990.

[91] P. P. Shenoy. A fusion algorithm for solving Bayesian decision problems. In B. D. D'Ambrosio, P. Smet, and P. P. Bonissone, editors, *Proc. of the Seventh Conference on Uncertainty in Artificial Intelligence*, pages 361–369, UCLA, Los Angeles, USA, 1991. Morgan Kaufmann.

[92] P. P. Shenoy. Valuation network representation and solution of asymmetric decision problems. Technical Report working paper No. 246, School of Business, University of Kansas, April 1993.

[93] D. J. Slate and L. R. Atkin. *CHESS 4.5 — The Northwestern University University Chess Program.* Springer–Verlag, New York, 1977.

[94] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28: 202-208, 1985.

[95] J. E. Smith, S. Holtzman, and J. E. Matheson. Structuring conditional relationships in influence diagrams. *Operations Research*, 41(2):280–297, 1993.

[96] J. Q. Smith. *Decision analysis : a Bayesian approach.* London ; New York : Chapman and Hall, 1988.

[97] W. Stallings. *Data and Computer Communications.* Macmillan Publishing Company New York and Collier Macmillan Publishers London, 1985.

[98] Y. Sun and D. S. Weld. Beyond simple observation: Planning to diagnose. In *Proc. of the 3rd International Workshop on Principles of Diagnosis*, 1992.

[99] Y. Sun and D. S. Weld. A framework for model based repair. In *Proc. of AAAI-93*, pages 182–187, 1993.

[100] A. S. Tanenbaum. *Computer networks.* Englewood Cliffs, N.J. : Prentice-Hall, 1989.

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

[102] J. von Neumann and O. Morgenstern. *Theory and Games and Economic Behavior.* Princeton University Press, 1947.

[103] Z. Wang and J. Crowcroft. Analysis of shortest path routing algorithms in a dynammic network environmrnt. *Computer Communication Review*, 22(2), 1992.

[104] M. P. Wellman. *Formulation of Tradeoffs in Planning Under Uncertainty.* Pitman and Morgan Kaufman, 1990.

[105] P. H. Winston. *Artificial Intelligence.* Addison–Wesley, Reading MA, 1984.

[106] L. Zhang. *A Computational Theory of Decision Networks.* PhD thesis, Department of Computer Science, University of British Columbia, 1993.

[107] L. Zhang and D. Poole. Sidestepping the triangulation problem. In *Proc. of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 360–367, Stanford University, Standford, USA, 1992.

[108] L. Zhang and D. Poole. Stepwise decomposable influence diagrams. In B. Nebel, C. Rich, and W. Swartout, editors, *Proc. of the Fourth International Conference on Knowledge Representation and Reasoning*, pages 141–152, Cambridge, Mass., USA, Oct. 1992. Morgan Kaufmann.

[109] L. Zhang, R. Qi, and D. Poole. A computational theory of decision networks. *accepted by International Journal of Approximate Reasoning, also available as a technical report 93-6, Department of Computer Science, UBC*, 1993.

[110] L. Zhang, R. Qi, and D. Poole. Incremental computation of the value of perfect information in stepwise-decomposable influence diagrams. In *Proc. of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 400–410, Washington, DC, 1993.

[111] L. Zhang, R. Qi, and D. Poole. Minimizing decision tables in stepwise-decomposable influence diagrams. In *Proc. of the Fourth International Workshop on Artificial Intelligence and Statistics*, Ft. Lauderdale, Florida, USA, Jan. 1993.

Appendix A

Functions for U–graph Generation

In this appendix, the Lisp functions used in our experiments for generating random graphs are included for reference. The functions `random-grid` and `random-parallel-graph` are for generating random grids and random parallel graphs respectively. To generate a random graph, we first generate a list of connections and then generate appropriate weights for those connections by calling function `weight-generation`.

```
(defun random-grid (d1 d2 p1 p2 r1 r2)
    (weight-generation (random-grid-connections d1 d2 p1 p2)  r1 r2))

(defun random-parallel-graph (n p1 r1 r2)
    (weight-generation (parallel-connections n p1)  r1 r2))

(defun random-from-uniform-distribution (a b)
   (+ (min a b) (* (abs (- a b)) (random 1.0))))

(defun random-switch-prob ()
   (/ (round (* 100 (random-from-uniform-distribution 0.01 0.99))) 100.0))

(defun random-weight (r1 r2)
    (let ((a1 (random-from-uniform-distribution 1 r1))
          (a2  (random-from-uniform-distribution 1 r2)))
       (round (random-from-uniform-distribution a1 (* a1 a2)))))

(defun weight-generation(connection-list r1 r2)
  (let* ((switch-list (cadr connection-list))
         (edge-list   (caddr connection-list))
         (edge-inform nil)
```

```
        )
       (dolist (e edge-list)
          (let ((weight  (random-weight r1 r2)))
            (push '(,(car e) ,(cadr e) (,weight) (1.0)) edge-inform)
            ))
       (dolist (e switch-list)
          (let ((weight1 (random-weight r1 r2))
        (weight2 (random-weight r1 r2))
                (p  (random-switch-prob))
               )
            (push '(,(car e) ,(cadr e) (,weight1 ,weight2) (,p ,(- 1 p)))
                 edge-inform)))
       (cons (car connection-list) edge-inform)
))

(defun random-grid-connections (d1 d2 p1 p2)
  (let ((switch-list nil)
       (edge-list   nil)
       (ran-test1 0)
       (ran-test2 0)
       )
      (dotimes (j d1)              ;; generating "vertical connections"
        (dotimes (i (- d2 1))
          (setf ran-test1 (random 1.0))
          (setf ran-test2 (random 1.0))
          (cond ((> ran-test1 p1) nil)
                ((> ran-test2 p2)
                 (push '(,(+ (* i d1) j) ,(+ (* (+ i 1) d1) j)) edge-list))
                (T (push '(,(+ (* i d1) j) ,(+ (* (+ i 1) d1) j)) switch-list))
             )))
      (dotimes (i d2)              ;; generating "horizontal connections"
        (dotimes (j (- d1 1))
          (setf ran-test1 (random 1.0))
          (setf ran-test2 (random 1.0))
          (cond ((> ran-test1 p1) nil)
                ((> ran-test2 p2)
                 (push '(,(+ (* i d1) j) ,(+ (* i d1) j 1)) edge-list))
               (T (push '(,(+ (* i d1) j) ,(+ (* i d1) j 1)) switch-list))
            )))
       (list (* d1 d2) switch-list edge-list)
```

```
    ))

(defun parallel-connections (n p1)
    (let* ((edge-list nil)
           (switch-list nil)
           (m (+ (* 6 n ) 1))
           )
          (dotimes (i n)
            (push (list 0 (+ 1 i)) edge-list)
            (push (list (+ 1 i  n) (+ i 1 (* 2 n))) edge-list)
            (push (list (+ 1 i) (+ i 1 n)) switch-list)

            (push (list (+ i 1 (* 5 n)) m) edge-list)
            (push (list (+ 1 i (* 3 n)) (+ i 1 (* 4 n))) edge-list)
            (push (list (+ 1 i (* 4 n)) (+ i 1 (* 5 n))) switch-list)
            (dotimes (j n)
              (cond ((>= p1 (random 1.0))
                     (push (list (+ i 1 (* 2 n)) (+ 1 j (* 3 n)) edge-list)))
                    (T nil)
                  )))
          (list (+ m 1) switch-list edge-list)
))
```