Decision Graph Search

by

Runping Qi and David Poole*

Technical Report 93-9
April 1993

Department of Computer Science
The University of British Columbia
Vancouver, B. C. V6T 1Z2
Canada

email: qi@cs.ubc.ca, poole@cs.ubc.ca

*Scholar of Canadian Institute for Advanced Research

©1993 Runping Qi and David Poole

Abstract

A decision graph is an AND/OR graph with a certain evaluation function. Decision graphs have been found a very useful representation for a variety of decision making problems. This article present a number of search algorithms computing an optimal solution from a given decision graph. These algorithms include one *depth-first heuristic-search algorithm*, one *best-first heuristic-search algorithm*, one *anytime-algorithm* and two *iterative-deepening depth-first heuristic-search algorithms*. Similar to the *-minimax search algorithms of Ballard, our depth-first heuristic-search algorithm is developed from the alpha–beta algorithm for minimax tree search. In addition, we show how heuristic knowledge can be used to improve search efficiency. Furthermore, we present an anytime algorithm which is conveniently obtained from the depth-first heuristic-search algorithm without incurring much overhead. The best-first heuristic-search algorithm is obtained by modifying the well known AO* algorithm for AND/OR graphs with additive costs. The iterative-deepening algorithms result from combining the iterative-deepening techniques with the depth-first search techniques. Some experimental data on some of these algorithms performance are given.

Key words: decision graphs, heuristic search, anytime algorithms.

1 Introduction

Decision making under uncertainty is an active research topic in AI, decision theory and operations research. Many problems, such as diagnostic reasoning (Poole 1992, Provan and Poole 1991), planning with uncertainty (Haddawy and Hanks 1992, Horvitz *et al.* 1988, Koenig 1992, Wellman 1990), path planning in uncertain environments (Dean *et al.* 1990, Mobasseri 1989, Mobasseri 1990, Qi and Poole 1992a) can be abstracted as decision making under uncertainty. See (Dean and Wellman 1992) for a good introduction to this subject. Problems of decision making under uncertainty can be presented in various forms, such as decision trees (Raiffa 1968), finite stage Markov decision processes (Derman 1970), or influence diagrams (Howard and Matheson 1984, Shachter 1986). The problems presented in different forms may have different semantics attached. This makes it difficult to study the computational issues for decision making with uncertainty in a uniform approach. Although a lot of 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, a lot of work has been done in the operations research community (see (Puterman 1990) for a comprehensive survey) on the computation of Markov decision processes (Derman 1970, Howard 1960), but the algorithms developed there are not directly applicable to influence diagram evaluation. Similarly, influence diagram evaluation has been an active research topic in the AI community, but the algorithms developed for influence diagram evaluation cannot be directly used for solving Markov decision processes.

In order to deal with the computational issues of decision making problems in a uniform way, we proposed in (Qi 1993) *decision graphs* as a common base for the decision making problems, and showed that decision making problems in various forms, such as decision trees, finite stage Markov decision processes, and influence diagrams, can be represented by decision graphs.

A decision graph is an AND/OR graph with a certain *evaluation function*. Decision graphs can also be viewed as a generalization of decision trees (Raiffa 1968) in the sense that they allow for structure sharing. The solution graphs minimizing the evaluation function are the *optimal solution graphs* of the decision graph.

When a decision graph is used to represent a decision problem, the solution graphs of the decision graph are interpreted as the *strategies* of the decision problem, while the evaluation function acts as a *cost function* for the strategies. The optimal solution graphs correspond to the optimal strategies of the decision problem.

Given a decision making problem represented by a decision graph, we need to compute an optimal solution graph of the decision graph. In this article, we describe a number of algorithms for this computational problem. These algorithms include one *depth-first heuristic-search algorithm*, one *best-first heuristic-search algorithm*, one *anytime-algorithm* and two *iterative-deepening depth-first heuristic-search algorithms*. The techniques used in these algorithms are common in the AI literature. Similar to the *-minimax search algorithms of Ballard 1983, the depth-first heuristic-search algorithm is developed from the alpha–beta algorithm for minimax tree search (Knuth and Moore 1975). While Ballard emphasizes the generalization of the alpha-beta pruning technique for handling chance nodes in a minimax tree, in our development we emphasize the pruning technique and the effective use of heuristic knowledge to improve search efficiency. Furthermore, we show that an anytime algorithm can be conveniently obtained by integrating the anytime

concept (Boddy 1991) into the depth-first heuristic-search algorithm. The best-first heuristic-search algorithm is obtained by modifying the well known AO* algorithm (Mahanti and Bagchi 1985, Nilsson 1982, Pearl 1984) for AND/OR graphs with additive costs. The iterative-deepening algorithms result from combining the iterative-deepening techniques (Korf 1985) with the depth-first search techniques.

The remainder of this paper is organized as follows. In the next section, we introduce the decision graph search problem. We present the depth-first heuristic-search algorithm and its anytime version in Section 3. In Sections 4 and 5, we describe the best-first heuristic-search algorithm and the iterative-deepening algorithms. Section 6 provides some experimental data on the performance of the depth-first heuristic-search algorithm and the best-first heuristic-search algorithm. Section 7 concludes the paper.

2 Decision Graphs and Decision Graph Search

From the structural point of view, a *decision graph* is an acyclic AND/OR graph with a certain 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 AND nodes in an AND/OR graph. Each decision graph has exactly one *root*. All children of a node in a decision graph are of the same type. Chance nodes and choice nodes are interleaved in a decision graph. A cost 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 unit. Leaf nodes are terminals, each of which has a value.

A decision graph can be considered as a compact representation of a decision tree[1] in which some subtrees may be identical.

A *solution graph* S, w.r.t. a node n of a decision graph D is a graph with the following characteristics:

1. n is in S;
2. if a non–terminal chance node of D is in S, then all of its children are in S;
3. if a non–terminal choice node of D is in S, then exactly one of its children is in S.

A solution graph w.r.t. the root of a decision graph is simply referred to as a solution graph of the decision graph. Thus a solution graph w.r.t. node n of a decision graph is a solution graph of the (sub–) decision graph rooted at n in the original 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 (states of the world). The root node represents the initial situation. A choice node represents a situation where an agent has to select one of the actions. The arcs emanating from a choice node can be viewed as the actions. The values

[1]More accurately, we should use *Markov decision tree* instead to distinguish the subtle semantic difference between the decision tree we are talking about here and the decision trees in the decision analysis community (Raiffa 1968) where a node in a decision tree depends on all the nodes and the arcs along the path from the root to the node.

associated with the arcs are the costs of the corresponding actions. A chance node represents an uncertain situation. At a choice node, if an agent selects and takes an action, the cost of the action is incurred, and an uncertain situation, represented by a chance node, is reached. From this uncertain situation, some new situation will be reached. The probability that a particular situation will be reached is given by the number associated with the arc from the chance node to the child representing the situation. This process repeats until a terminal is reached. A terminal node represents a situation where an assessment can be made directly.

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 D be a decision graph. Its evaluation function is a real–valued function u defined as follows:

1. If n is a terminal: $u(D, n) = v(n)$.
2. If n is a chance node with l children, $n_1, ..., n_l$ in D:

   $u(D, n) = \sum_{j=1}^{l} p(n, n_j) * u(D, n_j)$.
3. If n is a choice node with l children, $n_1, ..., n_l$ in D:

   $u(D, n) = min_{j=1}^{l}\{c(n, n_j) + u(D, n_j)\}$.

This evaluation function is called a *min–exp evaluation* (in contrast to the minimax evaluation of a minimax tree (von Neumann and Morgenstern 1947)).

The concepts of solution graphs and min–exp evaluation are the natural extension of those defined for decision trees in (Qi and Poole 1992b). The value given by $u(D, n)$ is called the min–exp value of node n in D. The min–exp value of node n in D can be interpreted as the minimal expected cost 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 D with evaluation function u, a solution graph S of D is *optimal* w.r.t. the evaluation function if $u(S, n) = u(D, n)$ for every node n in S.

Fig. 1 shows a simple decision graph. In the figure, boxes, circles, and dotted–line circles denote the choice nodes, chance nodes and terminals respectively.

The problem of decision graph search is defined as follows. *For a given decision graph D and an evaluation function, to find an optimal solution S (w.r.t. the evaluation function).*

This problem can be solved by a brute-force search algorithm or by using the *folding-back and averaging-out* method (Raiffa 1968) that is commonly used in decision analysis for evaluating decision trees. 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 the sections to follow, we present several search algorithms for computing an optimal solution graph from a decision graph. 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.

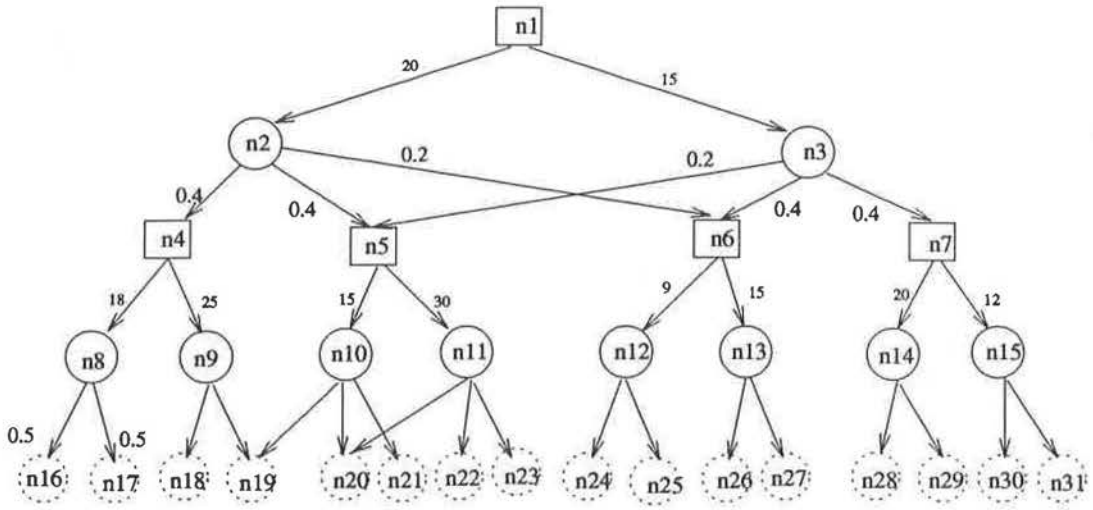

Figure 1: A simple decision graph

3 Depth-First Heuristic-Search Algorithms

In this section, we develop a depth-first heuristic-search algorithm, and its anytime version, for decision graph search. These algorithms use a kind of domain dependent information and a pruning technique similar to the alpha–beta mechanism for minimax tree search (Knuth and Moore 1975).

The depth-first heuristic-search algorithm was originally developed for searching decision trees (Qi and Poole 1992b). 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. The depth-first heuristic-search algorithm is similar to the *-minimax algorithm of Ballard (1983). While Ballard emphasizes the generalization of the alpha–beta technique to deal with chance nodes in a game tree, we emphasize the pruning technique as well as the effective use of domain dependent knowledge to increase search efficiency.

3.1 Pruning mechanism

In order to develop the pruning technique, we contrast a decision tree to 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 edges 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 which estimates the min–exp values of the nodes of the decision tree. A heuristic function h for decision tree D is admissible if, for every node n in D, $h(n) \leq u(D,n)$. For the sake of brevity, we will use $h^*(n)$ to denote $u(D,n)$ when no ambiguity arises.

The pruning mechanism works as follows. For each node n to be searched next, and a given upper bound b, it tries to find out whether $h^*(n)$ is less than b, and find out the value of $h^*(n)$ if it is less than b. Using the terminology in (Knuth and Moore 1975), we call the upper bound the "β-value" of node n.

Let h be an admissible heuristic function for D, and b be the β-value of node n, the node to be searched next. We have the following three cases.

(1) $b \leq h(n)$. In this case, due to the admissibility of h, it can be concluded immediately that $b \leq h^*(n)$. Thus node n (and the subtree rooted at n) need not be searched at all.

(2) $b > h(n)$ and n is a choice node with children $n_1, ..., n_l$. In this case the questions can be answered by searching the subtrees rooted at $n_1, ..., n_l$. Let

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

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 we have obtained when the subtrees rooted at $n_1, ..., n_{i-1}$ have been searched and the subtree rooted at n_i is to be searched next. We call r_{i-1} an *intermediate back-value* of node n. If $r_{i-1} \leq c(n,n_i) + h^*(n_i)$, then $c(n,n_i) + h^*(n_i)$ is either no less than b, the β-value for n, or no less than $c(n,n_j) + h^*(n_j)$ for some j, $1 \leq j < i$. In either case, it is fruitless to search through the subtree rooted at n_i. Therefore, we can set $r_{i-1} - c(n,n_i)$ as the β-value for node n_i. After all of the subtrees under node n have been searched, we know that $h^*(n) \geq b$ if $r_l \geq b$ and that $h^*(n) = r_l$ if $r_l < b$.

(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

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

where $p(n,n_i)$ is the probability of the arc from n to n_i, for $0 \leq i \leq l$. $partial_i$ 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:

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

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

and

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

Since h is admissible, $partial_{i-1} \leq partial_i$ for any i, $1 \leq i \leq l$. Thus, if for some i, $1 \leq i \leq l$, $partial_i \geq b$, then, $h^*(n) \geq b$. In this situation, the search can be discontinued. Since

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

implies $partial_i \geq b$, we can use $(b - partial_{i-1})/p(n, n_i) + h(n_i)$ as the β-value of node n_i. If the subtree rooted at n_l is eventually searched and $partial_l < b$, then $partial_l$ is equal to $h^*(n)$.

3.2 A decision tree search algorithm

A decision tree search algorithm using the pruning mechanism discussed in the previous section, called A, is shown in Fig. 2. 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. `h` corresponds to an admissible heuristic function, and `order-d` and `order-n` correspond to two *tree ordering* functions which can order the children of choice nodes and those of chance nodes respectively. These three functions are the abstraction of the domain dependent information of which A can make use.

The algorithm consists of two mutually recursive functions: `dnode(n, p)` 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` represents the intermediate back–up value 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 next child to be searched is computed. If the β-value for a child is no more than the value given by the heuristic function, then 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 for the next child to be searched 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 over the β-value. Thus no more children need to be searched.

The description of Algorithm A given in Fig. 2 ignores the part of explicitly constructing the optimal solution tree. In case this is needed, the algorithm can be easily modified to do so. Thus for a decision tree D and a node n, Algorithm A can be used either to compute $h^*(n)$ or to compute an optimal subtree rooted at n in D. Since Algorithm A is a depth first search algorithm, the size of the space it needs is linear in the depth of the tree if the solution tree need not be constructed explicitly, and is linear in the size of the largest solution graph the algorithm ever constructs in the search course.

As an illustration of this algorithm, let us consider an example. For the purpose of convenience, we can think that, for a given decision tree, the algorithm orders the tree first (using its tree ordering functions) and then search 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 Algorithm A, is shown in Fig. 3 where we assume that all the terminals have value 10 and all the children of any chance node have the same probability (0.5).

[2]This convention is used throughout this section.

```
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;
   j = # of children of n;
   let n1, n2, ..., nj = order-d(n);
   for (i = 1 to j) 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;

   j = # of children of N;
   let n1, n2, ..., nj = order-n(n);
   partial = h(n1)* prob(n, 1) + ... + h(nj) * prob(n, j);
   i = 0;
   while (partial < b) and (i < j) 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 2: The pseudo codes of Algorithm A

The optimal solution is indicated by the arrows. Its cost is 35.5. Suppose that Algorithm A uses heuristic function h_0 which 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 cut off[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 cut off. The last pruned node for this problem is node n_{27}. Therefore, five nodes in total are cut off.

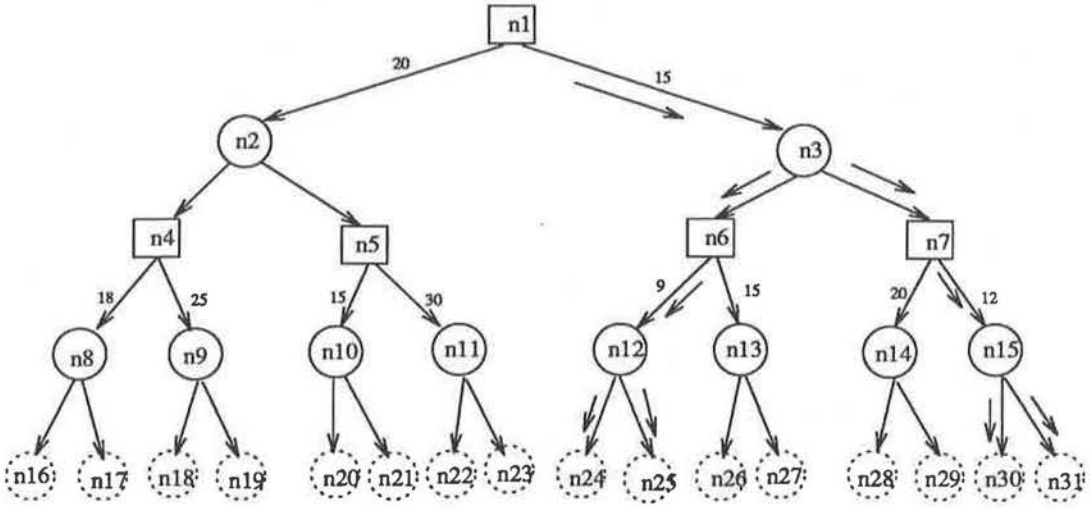

Figure 3: An illustration of the search algorithm

Let dt_1 be a function defined as:

$$dt_1(n,b) = \begin{cases} nnode1(n,b) & \text{if } n \text{ is a chance node;} \\ dnode1(n,b) & \text{otherwise.} \end{cases}$$

The following theorem establishes the correctness of Algorithm A.

Theorem 1 *If the heuristic function used by Algorithm A is admissible, then:*

$$dt_1(n,b) = \begin{cases} h^*(n) & \text{if } h^*(n) < b \\ \texttt{MAXNUM} & \text{otherwise} \end{cases}$$

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

[3]Cutting off a node means the algorithm need not visit the node at all. In the current case, even if node n_{19} is not a terminal, but is an interior node, it will still be cut off.

This theorem can be proved by induction. A proof can be found in the Appendix.

Corollary 1 *If the heuristic function used by Algorithm A is admissible, then:* $dt_1(n,b') \leq dt_1(n,b)$ *for any node* n *in a decision tree and any two numbers* $b' \geq b$. *Furthermore, if* $dt_1(n,b) < b$, *then* $dt_1(n,b) = dt_1(n,b')$.

Corollary 2 *If the heuristic function used by Algorithm A is admissible, then:* $h^*(n) = dt_1(n,\infty)$ *for any node* n *in the decision tree.*

3.3 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 both heuristic functions h_1 and h_2 are admissible, it is clear that the performance of Algorithm A with h_1 will be no worse than that of Algorithm A 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 same decision tree, 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 Algorithm A with h'_0 to the decision tree ordered as shown in Fig. 3, nine nodes (nodes in the subtree rooted at nodes n_9, n_{11} and n_{13}) will be cut off.

3.4 Tree ordering

Note that the correctness of Algorithm A 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 has a great effect on the efficiency of the algorithm. Generally speaking, we want to search first the branch of a choice node that can result in the final (minimal) min–exp value of the choice node in hope that as many other branches as possible can be pruned; and we want to search first the child of a chance node which can contribute most to the min–exp value of the chance node in hope that the partial accumulation can reach b as early as possible.

As an illustration on the effect of tree ordering, let us consider the decision tree shown in Fig. 4. This is the same decision tree as the one in the previous example except that the orderings of the children of some nodes are different. It can be verified that when Algorithm A with heuristic function h_0 is applied to this tree, nine[4] nodes (nodes n_{27}, n_{29}, n_{19}, and the nodes in the subtrees rooted at nodes n_{10} and n_{11}) will be cut off; and when Algorithm A with h'_0 is applied to this tree, twenty one[5] nodes (nodes in the subtrees rooted at nodes n_{13}, n_{14}, and n_2) will be cut off.

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

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

[4] in contrast with five in the previous example.

[5] in contrast with nine in the previous example.

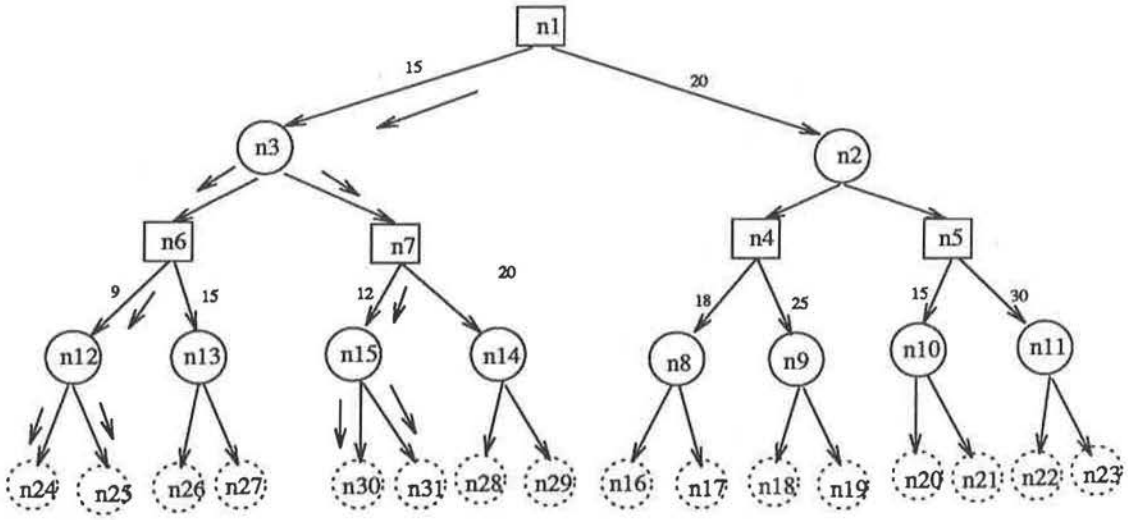

Figure 4: An illustration on the effect of tree ordering

then n_i should be chosen before n_j. With this definition, child n_j of a choice node n will be cut off 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 the brother nodes being pruned.

We can also define $order_n(n)$ 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 chosed before n_j. Similarly, children $n_{j+1}, ...n_l$ of a chance node n will all be cut off if

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

It is easy to verify that with the tree ordering functions defined as above, if $h(n) = h^*(n)$, all the non–optimal children of any choice node will be cut off, then the subtree searched by Algorithm A is exactly the same as an optimal solution tree.

In the other extreme case where the tree ordering functions give an order of the children of a choice node that is just the reverse of the order as defined above, then Algorithm A may visit a lot more nodes.

In Section 6, when we apply our decision graph search algorithms to U–graph based navigation problems (Qi and Poole 1992a), we will encounter some heuristic functions for a realistic problem. Moreover, we will define ordering functions in terms of heuristic functions in the same way as we did above.

For a uniform tree of depth $2d+1$ where each non–terminal choice node has b_1 children and each chance node has b_2 children, then the total number of the terminal nodes in the decision tree is $(b_1 b_2)^d$ while the number of the terminal nodes in an optimal solution tree is only b_2^d. Therefore, the ratio of the performance in the best case to that in the worst case is b_1^d. We call this ratio is the *ideal cutoff ratio*. It is clear that the bigger the b_1, the larger the ideal cutoff ratio, thus the more potential Algorithm A has. An intuitive interpretation on this is that the bigger b_1, the more important the domain knowledge.

3.5 Using inadmissible heuristic functions

In the previous section, it is required that the heuristic function h must be admissible. For A* algorithm, Harris (Harris 1974) 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* (Pearl 1984), we have the following two theorems for Algorithm A which 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 2 *Suppose Algorithm A 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*

$$h^*(n) + \delta \geq b \text{ if } dt_1(n,b) \geq b \quad \text{and}$$
$$h^*(n) + \delta \geq dt_1(n,b) \text{ if } dt_1(n,b) < b.$$

Theorem 3 *Suppose Algorithm A uses heuristic function h. If the costs of all the edges 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:*

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

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

$$h^*(n) * (1+\delta) \geq b \text{ if } dt_1(n,b) \geq b \quad \text{and}$$
$$h^*(n) * (1+\delta) \geq dt_1(n,b) \text{ if } dt_1(n,b) < b.$$

The proofs of these theorems can be found in the Appendix.

3.6 An anytime version of Algorithm A

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 quite a long time before such a solution tree can be computed. Note that Algorithm A will not return any thing before completing the computation of an optimal (suboptimal) solution. However, in some situations, it would be very useful if an algorithm can 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 the algorithms spend. Since an algorithm with this property can give an answer at any time after computing the initial solution, it is called an *anytime algorithm* (Boddy 1991).

We can think of an anytime algorithm as a program which generates a stream of solutions, ordered by their qualities (the expected costs in our case). 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 will systematically enumerate all of the possible solutions of the decision tree, and pass the solution stream to the filter. The filter maintains the minimal cost of the solutions arrived so far and discard the next solution with cost no smaller than the minimal cost. Unfortunately, the performance of this algorithm can be very bad. What we like is an algorithm, called B, which incorporates the pruning mechanism we discussed in Section 3.1 and at the same time behaves like an anytime algorithm. This is the topic of this section.

We observe that Algorithm B should differ from Algorithm A in the the following two aspects:

- When searching a choice node, Algorithm A will exhaust all of the children of the choice node and return the best one. But Algorithm B should return the *first* child which results in an admissible solution. Furthermore, Algorithm B should set a backtrack point at the same time so that it can continue generating better solutions when needed.

- In the course of searching a chance node, if Algorithm A finds that `partial` $\geq$ `b`, it reports a cutoff. But when in a similar situation, Algorithm B should request a backtrack.

Fig. 5 shows the pseudo code of Algorithm B. In the figure, we have a new procedure `a-search` as the interface of the algorithm. The statement `backtracking` means to go on computation 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. 6. Observe that for a given problem, Algorithms A and B will visit the same number of nodes. The only overhead Algorithm B over Algorithm A is to maintain those backtrack points, which needs a storage linear in the maximum of the sizes of solution graphs.

3.7 A remark

It is interesting to note that, although there are many other techniques developed for minimax tree search (e.g. SSS* (Stochman 1979), conspiracy number (McAllester 1988)), it seems that only the alpha–beta technique can be conveniently applied to decision tree search.

```
a_search (n, b)
   if n is a terminal then return MAXNUM else return v(n);
   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;
   j = # of children of n;
   let n1, n2, ..., nj = order_d(n);
   for (i = 1 to j) 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;

   j = # of children of N;
   let n1, n2, ..., nj = order_n(n);
   partial = h(n1)* prob(n, 1) + ... + h(nj) * prob(n, j);
   i = 0;
   while (partial < b) and (i < j) 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 5: The pseudo codes of Algorithm B

```
% 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([], B, R, C) :- fail.
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)|_],  R1,  (LC, R1), C1, C1).
searchchoice_with_s([_|L],  _, R, C1, C):- searchchoice(L, C1, R, C).

%search children list of a chance node

searchchance(L,  B, R, C, C0) :- C0 >= B,   fail.
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 6: A Prolog version of Algorithm B

3.8 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 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[6]. 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 a shared node more than once. In order to overcome this disadvantage, we can use a "cache technique". When the expected cost of a shared node is obtained, it will be stored in a cache for later use. When a node is to be searched, the algorithms first check whether the expected cost of the node is in the cache, and will search the node only if the expected cost of 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. This new lower bound can be used to update $h(n)$, the value of the heuristic function at the node n. This is an example of "learning fron failure".

Incorporating this caching technique into Algorithm A, we can obtain Algorithm A' as shown in Fig. 7. A similar version can be obtained for Algorithm B in the same way.

In the algorithm, when the expected cost of a node is obtained, it is needed to determine whether the expected cost should be cached or not; when a cutoff occurs at a node, it is needed to determine whether the heuristic function should be updated accordingly. If the "caching policy" allows no node to be cached, the algorithm degenerates to Algorithm A. On the other hand, if the "caching policy" allows the expected costs of all shared nodes to be cached, the algorithm can exploit the shared structure of decision graphs to the maximum degree. One way to guarantee this is to cache the expected cost of every searched node. However, this may lead to an exponential space requirement, totally defeating the advantage mentioned above. Clearly, there is a tradeoff between time and space. Generally speaking, we would like to cache those nodes which are likely to be searched again and it would take much time to search them. The searching time of a node can be obtained online. The probability that a node to be searched again may be estimated based on domain dependent knowledge. Russell and Wefald's metareasoning mechanism (Russell and Wefald 1989, Russell 1990) can play a role in deciding which nodes' expected costs should be cached.

4 Applying AO* to Decision Graph Search

AO* can be considered as a counterpart of A* for AND/OR graph search. The algorithm was developed in (Martelli and Montanari 1973). AO* as the name of the algorithm was first given

[6]For Algorithm A, 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 linear in the size of solution graphs otherwise; the space requirement of the anyime algorithm is linear in the size of solution graphs.

```
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;
   j = # of children of n;
   let n1, n2, ..., nj = order-d(n);
   for (i = 1 to j) 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;

   j = # of children of N;
   let n1, n2, ..., nj = order-n(n);
   partial = h(n1)* prob(n, 1) + ... + h(nj) * prob(n, j);
   i = 0;
   while (partial < b) and (i < j) 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 7: The pseudo codes of Algorithm A'

in (Nilsson 1982).

AO* was first developed for searching AND/OR graphs with additive costs (Martelli and Montanari 1973). As shown in (Kumar *et al.* 1988), AO* is applicable to AND/OR graphs with *monotone evaluation functions* (Kumar *et al.* 1988). We say a function $g(x_1, ..., x_l, L)$ is *monotone* if $g(y_1, ..., y_l, L) \leq g(x_1, ..., x_l, L)$ provided $y_i \leq x_i$ for $i = 1, ..., l$. An evaluation function u is *monotone* if there is a monotone function g such that

$$u(n) = g(u(n_1), ..., u(n_l), L)$$

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 be applicable to decision graph search, and present some results on the resultant algorithm.

As usual, 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 *potential solution graph* (psg). A psg is actually 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 can define a minimization–expectation 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.
- $f(n) = min_{i=1}^{l}\{c(n, n_i) + f(n_i)\}$ if n is a choice node with children $n_1, ..., n_l$.
- $f(n) = \sum_{i=1}^{l} p(n, n_i) * f(n_i)$ if n is a chance node with children $n_1, ..., n_l$.

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

Lemma 1 *For any node n in G, $f(n) \leq h^*(n)$.*

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

AO* can be intuitively understood as a repetition of the following two major operations. The first one is the node expansion which finds a non–terminal tip node in the current 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 which, 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 current 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*, expand the node, generating all the children of the node and adding them to G (if some children are already in G, just share them).
4. Set *opsg* to be the optimal potential solution graph in G.
5. Go to step 2.

The above algorithm can be further refined by using the marking technique of (Martelli and Montanari 1973). The following version of AO* is adopted from (Chakrabarti *et al.* 1987), where h denotes a heuristic functions 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 simple version of the algorithm.

Algorithm AO*

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

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

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

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

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

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

   Label SOLVED any 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 such that no descendent of m in G occurs in Z.

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

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

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

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

The only difference between the above algorithm and the AO* algorithm given in (Chakrabarti *et al.* 1987) for AND/OR graphs with additive costs lies in the way of updating function $\hat{f}$ at step 2.3.2 (ii).

Lemma 2 *At any stage during the search process, if a node n is labelled 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 *If there exists some ϵ, $\epsilon \geq 0$, $h(n) \leq h^*(n) + \epsilon$, 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 (Martelli and Montanari 1973). We will prove the lemma by induction on the stage of the algorithm. The lemma is trivially true initially. Let us suppose that it is true at certain stage and let us prove it is true at the next stage, that is after each 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 which 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 $n_1, ..., n_l$ be the children of n. For any child, n_k, $1 \leq k \leq l$, if it has been generated before, we have $\hat{f}(n_k) \leq h^*(n_k) + \epsilon$ by the outer induction assumption, and if n_k is generated for the first time, we also have $\hat{f}(n_k) = h(n_k) \leq h^*(n_k) + \epsilon$ by the hypothesis on the heuristic function h.

If n is a choice node, we have:

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

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

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

Now, let us assume that the inequality 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 (Martelli and Montanari 1973) is actually a special case of Lemma 3 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 4 *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 Lemma 2 and Lemma 3 immediately. □

Corollary 3 *The algorithm with an admissible heuristic function will return an optimal solution, if it terminates.*

Lemma 4 *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.

Theorem 5 *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 Lemma 2 and Lemma 4 immediately. □

As the reader may have already noticed, Theorems 4 and 5 above do not assure that the algorithm must stop even if a *finite* solution graph exists for a given (infinite) decision graph. Thus they are weaker than Theorem 1 in (Martelli and Montanari 1973). 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 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. 8. Node A in the decision tree is the root of the graph. Each choice node has two children, 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 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 the above algorithm 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.

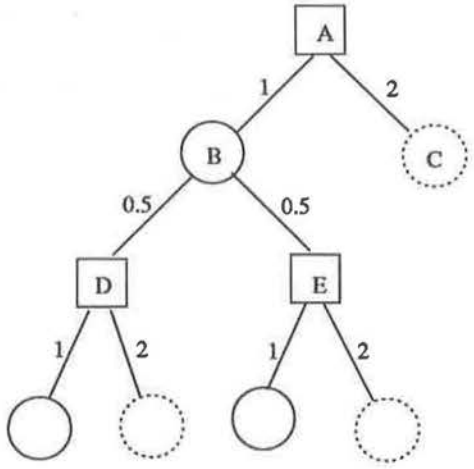

Figure 8: An example for which AO* may not terminate

It can be observed 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 contribute to the costs of the solution graphs consisting of the node a certain amount that can be bounded from below, thus the contribution cannot be arbitrarily small. However, in the case of searching into decision graphs, there is no such guarantee with respect to the cost contribution of a tip node expansion.

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. Although some researchers (Nilsson 1982, Pearl 1984) pointed out the use of heuristic functions for tip node selection, they did so purely out of efficiency consideration. Here, we propose two heuristics for tip node selection out of termination consideration.

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

Heuristic 2: using a best first strategy in tip node selection. Suppose $t_1, ..., t_l$ are the tip nodes of the marked potential solution graph rooted at s. 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.

5 Iterative Deepening Search

The two kinds of algorithms 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 very deep layer in a part of solution graph which is not in an optimal solution graph. Thus it would be nice if we can 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 (Korf 1985) was proposed for such kind of 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.

5.1 Depth bounded iterative deepening

The first iterative-deepening search strategy is a *depth-bounded iterative-deepening* strategy. The strategy repeatedly applies Algorithm A 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* (Korf 1985, Patrick *et al.* 1992), 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 (Korf 1985). 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 in (Slate and Atkin 1977, Winston 1984).

The following result is obvious.

Theorem 6 *The depth bounded iterative deepening algorithm returns an optimal solution graph if the heuristic function it uses is admissible and if it terminates.*

5.2 Cost bounded iterative deepening

The second iterative deepening search strategy is a cost bounded iterative deepening strategy, very much like iterative deepening A* (IDA*) (Korf 1985). 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 which 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 Algorithm A) 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 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 7 *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 Algorithm A is that it is less sensitive to the node ordering in a graph. This can be best illustrated by an example. Suppose we have a decision tree. 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 Algorithm A happens to behave in such a way that it searches 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 founded. This may take a lot of time which turns out to be useless. On the other hand, the cost bounded iterative deepening algorithm will not stick to the subtree below node A for too long (because the β-value can be very small).

The cost–bounded iterative deepening algorithms discussed above is analogous to the binary iterative deepening A* (Patrick 1992) in the sense that both the upper bound and the lower bound of the problem are maintained.

It is interesting to note that the cost bounded iterative deepening algorithm and the anytime algorithm B can work as co–routines in the following way. For a given problem, Algorithm B can gradually approach the optimal value of the decision graph from above, 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 end up with an algorithm with anytime property which is simultaneously using cost bounded iterative deepening strategy.

Theorem 8 *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. The result is quite conservative. Nevertheless, it is sufficient for our purpose.

Theorem 9 *All of the algorithms presented in this paper terminate for finite acyclic decision graphs.*

6 Some Experimental Results

In this section, we present some experimental results on the performance of Algorithms A and AO*. The data are obtained when we apply these algorithms to the problem of U–graph based navigation (Qi and Poole 1991, Qi 1993).

6.1 The application domain

A U–graph is an extension of an ordinary distance graph in which the weights of some edges (arcs) are not constants, but are random variables. If the weight of an edge between two vertices is a discrete random variable, it will be referred to as an *uncertain edge*. Otherwise, it will be referred to as an *ordinary edge*. An uncertain edge between two vertices represents a piece of knowledge that there exists a connection between the locations denoted by the vertices but the weight of the connection is not certain. The distribution of the random variable characterizes the uncertainty.

Given a U–graph, a start vertex and a goal vertex, an agent needs to reach the goal vertex from the start vertex by following some strategy. It is assumed that the agent can determine the actual weight of an uncertain edge ony when it is at a vertex adjacent to the edge, and the weight of an uncertain edge will remain the same throughout a navigation course once it is revealed.

The computational problem is to compute a navigation strategy which can lead the agent to the destination with the minimal expected cost.

A navigation task can be modeled by a decision graph. Within such a graph, a node contains the current U–graph and the current vertex. A choice node represents a situation where the agent has to decide where to go from the current vertex, and a chance node represents a situation where the agent is facing some uncertain edges whose status is to be revealed. A leaf node represents the situation where the agent finishes the task. The root represents the initial situation. The mini–exp value of a node is the minimal expected cost for the agent to arrive at the destination from the situation represented by the node. For the details of this modeling, see (Qi 1993, Qi and Poole 1991, Qi and Poole 1992a). The decision graph for the navigation task of going from vertex A to vertex B in the U–graph in Fig. 9-(a) is shown in Fig. 9-(b).

In our experiments, we apply Algorithms A and AO* to a set of U–graph based navigation problems. We consider two classes of U–graphs. The U–graphs in the first class are randomly generated from grids with the following parameters: d_1 , d_2 , p_1 , p_2 . Here, d_1 and d_2 specify the width and height of the grids; p_1 specifies the probability that a connection (either an ordinary edge or an uncertain edge) exists between any pair of neighbour vertices on the grids; p_2 specified the probability of a connection being an uncertain edge. We assume that the distributions of all uncertain edges are binary and independent of one another. For each ordinary edge, a random number is generated as its weight; for each uncertain edge, three random numbers c_1, c_2 and p are generated, with $0 < p < 1$; the weight of the uncertain edge is c_1 with probability p and is c_2 with probability $1 - p$. A U–graph of this kind is an abstraction of the road layout of a city. For each randomly generated U–graph, we assume the navigation task is to go from the upper–left conner to the lower–right conner on the grid.

The U–graphs in the second class are randomly generated from a structure as shown in Fig. 10, which is a model of two parallel–highway systems jointed by a bipartite graph at the middle. The U–graphs are generated with two parameters: m , the number of branches in each parallel

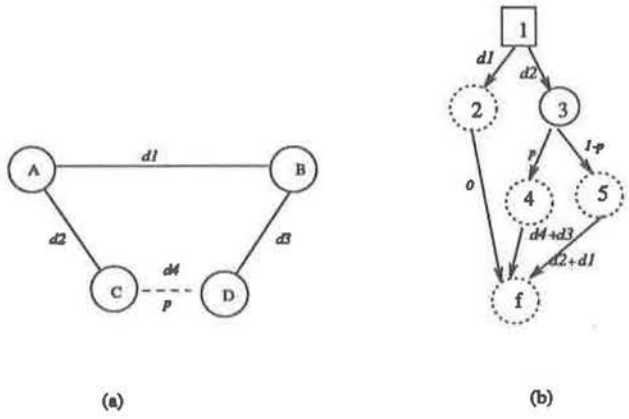

Figure 9: A simple U-graph and a simple decision graph

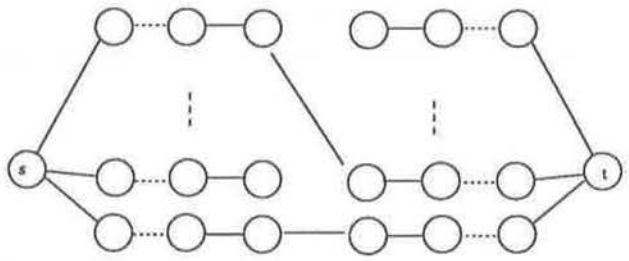

Figure 10: An abstraction of a parallel highway system

highway system, and p, the probability that an ordinary edge exists between a pair of vertices, one at the boundary of each half of the bipartite graph. The weights of the ordinary edges and the weight distributions of the uncertain edges are generated in the same way as that for the U–graphs in the first class. Again, we assume the weight distributions of all uncertain edges are binary and are independent of one another.

6.2 A heuristic function

For a given navigation task, let rg denote the decision graph representing a given navigation task, and u be a min–exp evaluation function defined on rg such that $u(rg, N)$ is the minimal expected cost the agent is going to spend in the situation represented by node N. We define a heuristic function h as follows:

$$h(N) = d_s(G^o, v_c, v_g).$$

In words, $h(N)$ is the shortest distance between the current vertex and the goal vertex in G^o, where G^o is the optimistic induced graph of the current U–graph of node N. The optimistic induced graph of a U–graph G is obtained from G by replacing each uncertain edge in G by an ordinary edge whose weight is the lowest possible value of the distribution of the uncertain edge. It is intuitively clear that $h(N) \leq u(rg, N)$ for any N. Therefore h is admissible.

6.3 Experiment 1

In this experiment, we measured for each problem the number of nodes examined by Algorithm A and that examined by AO*. The experimental data for the U–graphs in the first class and for the U–graphs in the second class are summarized in Tables 1 and 2 respectively. Each row in the tables corresponds to a problem instance. The second and the third columns of each row contain the nodes examined by AO* and by Algorithm A respectively for the problem corresponding to the row. The + (− or =) in the fourth column of each row indicates that the number of the nodes examined by AO* is more than (fewer than or equal to) the number of the nodes examined by Algorithm A for the same problem.

From Table 1 we see that for seventy seven problems out of one hundred, Algorithm A examines fewer nodes than AO* does. From Table 2 we see that for fifty two problems out of one hundred, Algorithm A examines fewer nodes than AO* does. Thus we may conclude that Algorithm A likely examines fewer nodes than AO* does. This may be due to the fact that in our implementation of AO* for U–graph based navigation, we do not check the possibility of structure sharing in the representing graphs (thus each representing graph is essentially treated as an unfolded tree), because otherwise, the overhead (for checking whether a node is already in the graph) will be intolerably high.

Moreover, our experiments show that, for most of the problems, Algorithm A spends less time, even when it examines more nodes than AO* does. This is due to the fact that Algorithm A is a depth first algorithm, involving less overhead. This suggests that Algorithm A is better suited for U–graph based navigation.

Table 1: The performance comparison 1 of Algorithms A and AO* (for the U–graphs in the first class)

	AO*	A	RESULT
1	19	8	+
3	173	1092	−
5	142	137	+
7	341	203	+
9	99	88	+
11	7699	4143	−
13	20	31	−
15	6366	10138	−
17	1165	2168	−
19	53	50	+
21	504	329	+
23	86	59	+
25	1134	808	+
27	118	85	+
29	23	23	=
31	2104	2098	+
33	4847	328	+
35	29	26	+
37	162	231	−
39	316	397	−
41	926	444	+
43	45	112	−
45	197	143	+
47	88	58	+
49	42	39	+
51	52	36	+
53	307	229	+
55	218	117	+
57	61	36	+
59	993	2753	−
61	46	41	+
63	59	52	+
65	128	106	+
67	161	184	−
69	499	321	+
71	325	271	+
73	51	61	−
75	6166	7581	−
77	39	25	+
79	117	115	+
81	115	64	+
83	12	13	−
85	102	43	+
87	18	17	+
89	37	137	−
91	67	65	+
93	1417	1080	+
95	1251	667	+
97	21	19	+
99	3720	2393	+

	AO*	A	RESULT
2	1958	1367	+
4	41	31	+
6	402	118	+
8	194	457	−
10	52	35	+
12	646	507	+
14	7521	5321	+
16	1854	1248	+
18	199	109	+
20	15	13	+
22	14	15	−
24	251	229	+
26	607	280	+
28	21	16	+
30	23	23	=
32	136	469	−
34	341	178	+
36	225	247	−
38	1707	1576	+
40	183	121	+
42	63	60	+
44	6460	6433	+
46	168	149	+
48	356	650	−
50	19	17	+
52	16623	17472	−
54	127	100	+
56	41	24	+
58	145	76	+
60	682	571	+
62	3046	2026	+
64	127	71	+
66	46	28	+
68	13	11	+
70	78	63	+
72	13	11	+
74	28	31	−
76	545	519	+
78	12	11	+
80	23	45	−
82	365	304	+
84	127	61	+
86	85	52	+
88	31	28	+
90	222	172	+
92	132	88	+
94	1367	779	+
96	70	65	+
98	1214	831	+
100	955	772	+

Table 2: The performance comparison 2 of Algorithms A and AO* (for the U-graphs in the second class)

	AO*	A	RESULT
1	121	107	+
3	216	152	+
5	63	83	−
7	47	28	+
9	48	82	−
11	138	159	−
13	61	88	−
15	159	131	+
17	481	304	+
19	251	202	+
21	86	60	+
23	217	305	−
25	265	200	+
27	153	176	−
29	112	138	−
31	548	482	+
33	165	124	+
35	502	303	+
37	561	812	−
39	273	400	−
41	2812	1642	+
43	73	120	−
45	137	71	+
47	5566	3539	+
49	560	438	+
51	126	346	−
53	117	158	−
55	3089	2994	+
57	244	449	−
59	5625	10832	−
61	546	1608	−
63	340	780	−
65	1962	1584	+
67	317	418	−
69	2132	2858	−
71	145	877	−
73	7400	9086	−
75	7579	6306	+
77	377	571	−
79	637	1324	−
81	1417	1014	+
83	22	24	−
85	36	28	+
87	140	86	+
89	329	639	−
91	305	206	+
93	479	373	+
95	128	180	−
97	181	122	+
99	766	564	+

	AO*	A	RESULT
2	93	95	−
4	128	88	+
6	57	48	+
8	21	31	−
10	92	90	+
12	203	504	−
14	657	879	−
16	1235	1169	+
18	138	152	−
20	93	129	−
22	956	706	+
24	55	33	+
26	89	110	−
28	1244	881	+
30	133	72	+
32	159	90	+
34	93	90	+
36	821	509	+
38	232	512	−
40	106	79	+
42	263	249	+
44	256	221	+
46	1688	866	+
48	659	465	+
50	414	1323	−
52	906	2461	−
54	413	436	−
56	2653	3348	−
58	13854	8523	+
60	837	1206	−
62	212	103	+
64	440	3004	−
66	1499	4548	−
68	70	344	−
70	1139	1070	+
72	292	412	−
74	101	269	−
76	174	471	−
78	103	40	+
80	3567	3607	−
82	26	23	+
84	45	38	+
86	48	35	+
88	197	365	−
90	758	1425	−
92	158	229	−
94	802	527	+
96	82	57	+
98	7454	4165	+
100	482	297	+

6.4 Experiment 2

Qualitatively, we know that if we cannot assure the admissibility of a heuristic function, then the solution computed by Algorithm A may not be optimal. In this situation, an algorithm with a particular heuristic function can be evaluated by two factors: its performance and its quality. In our case the performance is a measurement on the speed of the of the algorithm while the quality is a measurement of the expected cost it may induce.

Our second experiment is to comapre the effect of heuristic functions on the the performance and the quality of the algorithms. Our experiments cast some light on the tradeoff between time and solution quality.

In our experiments, Algorithm A is tested with five different heuristic functions: h, h_1, h_2, h_3 and h', where h_1, h_2 and h_3 are defined as follows.

$h_1(N) = h(N)/(1+\epsilon)$

$h_2(N) = h(N)/(1+\epsilon)^2$ and

$h_3(N) = h(N)/(1+\epsilon)^3$

with $\epsilon = 0.25$. The heuristic function h' is defined as follows:

$$h'(N) = d_s(G^a, v_c, v_g)$$

where G^a is the average induced graph of the current U–graph of the situation represented by N. The average induced graph of a U–graph G is a graph obtained by replacing each uncertain edge in G with an ordinary edge whose weight is the mean value of the distribution of the uncertain edge.

Heuristic functions h_1, h_2, h_3 and h' are not admissible. Theorems 3 give the quality bounds for Algorithm A with heuristic function h_1, h_2 and h_3. However, we do not have a bound on the admissibility of the heuristic function h'.

For each problem and each heuristic function, we measure the cost of the solution graph and the number of the nodes visited by Algorithm A with the heuristic function for the problem. More specifically, for each problem, we measured the following data:

- c — the cost of the solution graph returned by Algorithm A with heuristic function h. This is the minimal expected cost for the problem.
- m — the number of the nodes examined by Algorithm A with heuristic function h.
- c_1 — the cost of the solution graph returned by Algorithm A with heuristic function h_1.
- m_1 — the number of the nodes examined by Algorithm A with heuristic function h_1.
- c_2 — the cost of the solution graph returned by Algorithm A with heuristic function h_2.
- m_2 — the number of the nodes examined by Algorithm A with heuristic function h_2.
- m_3 — the cost of the solution graph returned by Algorithm A with heuristic function h_3.
- m_3 — the number of the nodes examined by Algorithm A with heuristic function h_3.

Table 3: The average cost ratios and speedup ratios of Algorithm A with different heuristic functions

	acr_{11}	asr_{11}	acr_{12}	asr_{12}	acr_{13}	asr_{13}	acr'	asr'
Class 1	1.011	1.54	1.031	2.07	1.041	2.41	1.006	39.49
Class 2	1.012	2.77	1.064	9.17	1.131	23.04	1.074	23.18

- c' — the cost of the solution graph returned by Algorithm A with heuristic function h'.
- m' — the number of the nodes examined by Algorithm A with heuristic function h'.

From the measured data, we compute the following data:
$cr_j = c_j/c, \quad sr_j = m/m_j, \quad$ for $j = 1, 2, 3$, and
$cr' = c'/c, \quad sr' = m/m'$
Intuitively, for each problem, cr_j is the *cost ratio* of the solution returned by Algorithm A with the heuristic function h_j to the cost of the optimal solution for the problem; sr_j is the *speedup ratio* of Algorithm A with the heuristic function h_j to Algorithm A with the heuristic function h. Similarly, cr' is the *cost ratio* of the solution returned by Algorithm A with the heuristic function h' to the cost of the optimal solution for the problem; sr' is the *speedup ratio* of Algorithm A with the heuristic function h' to Algorithm A with the heuristic function h.

The average values of the computed data overall problem instances are given in Table 3. From this table, we make the following observations.

- The average cost ratios of Algorithm A 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 guaranteed admissible, they usually give very conservative estimations. Therefore, there is a great potential to obtain *more informed* heuristic functions.
- For the U-graphs in both classes, Algorithm A with heuristic function h' outperforms Algorithm A with heuristic functions h_1, h_2 and h_3. This is especially true for the U-graphs in the first class. The average cost ratios of Algorithm A with h' is 1.006, less than the cost ratio of Algorithm A with the heuristic function h_1; and the average speedup ratio is almost 40, far greater than the speedup ratios of Algorithm A with the heuristic function h_3. The good performance of Algorithm A with the heuristic function h' could 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 on the importance of domain dependent knowledge for decision search in particular and for heuristic search in general.

7 Conclusions and Future Work

In this article, we described a number of heuristic search algorithms for decision graph search. Since many decision making problems can be represented by decision graphs, these algorithms have a great application range. In particular, they have been used to solve the problem of U-graph based navigation. We also give some experimental results obtained when applying two of

the algorithms, namely Algorithms A and AO*, to the navigation problem. The experimental results show that for the particular problem, Algorithm A performs better than AO* in average. Furthermore, the experimental results illustrate that the domain–specific heuristic information plays a crucial role for decision graph search.

Our future work includes applying the algorithms presented in this article to other applications such as influence diagram evaluation. It will be very instructive to compare the performance of our algorithms with that of other well known influence diagram evaluation algorithms such as (Shachter 1986, Shachter and Peot 1992, Zhang and Poole 1992, Zhang *et al.* 1993). Also, it will be very interesting to apply our anytime algorithm to real–time decision making problems to see how competitive it is.

Acknowledgement The research reported in this paper is partially supported under NSERC grant OGPOO44121 and Project B5 of the Institute for Robotics and Intelligent Systems. The authors wish to thank Craig Boutilier, Andrew Csinger, Mike Horsch, Keiji Kanazawa, (Nevin) Lianwen Zhang and Ying Zhang for their valuable comments on this paper.

References

[Ballard, 1983] B. W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[Boddy, 1991] M. Boddy. Anytime problem solving using dynamic programming. In *Proc. AAAI–91*, pages 360–365, Anaham, CA., USA, 1991.

[Chakrabarti *et al.*, 1987] P. P. Chakrabarti, S. Ghose, and S. C. DeSarkar. Admisibility of AO* when heuristics overestimate. *Artificial Intelligence*, 34(1):97–113, 1987.

[Dean and Wellman, 1992] T. L. Dean and M. P. Wellman. *Planning and Control.* Morgan Kaufmann, 1992.

[Dean *et al.*, 1990] 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.

[Derman, 1970] C. Derman. *Finite State Markovian Decision Process.* Academic Press New York and London, 1970.

[Haddawy and Hanks, 1992] 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.

[Harris, 1974] L. R. Harris. The heuristic search under conditions of error. *Artificial Intelligence*, 5(3):217–234, 1974.

[Horvitz *et al.*, 1988] 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.

[Howard and Matheson, 1984] 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, VolumII*, pages 720–761. Strategic Decision Group, Mento Park, CA., 1984.

[Howard, 1960] R. A. Howard. *Dynamic Programming and Markov Processes.* Technology Press, Cambridge, Massachusetts, and Wiley New York, 1960.

[Knuth and Moore, 1975] D. E. Knuth and R. W. Moore. An analysis of alpha beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Koenig, 1992] 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.

[Korf, 1985] R. E. Korf. Depth first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kumar *et al.*, 1988] V. Kumar, D. S. Nau, and L. Kanal. A general branch-and-bound formulation for annd/or graph and game tree search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 91–130. Springer-Verlag, 1988.

[Mahanti and Bagchi, 1985] A. Mahanti and A. Bagchi. AND/OR graphs heuristic search methods. *J. ACM*, 32(1):28–51, 1985.

[Martelli and Montanari, 1973] A. Martelli and U. Montanari. Additive and/or graphs. In *Proc. of IJCAI–73*, pages 1–7, Stanford, CA., USA, 1973.

[McAllester, 1988] D. A. McAllester. Conspiracy numbers for minmax search. *Artificial Intelligence*, 35(3):287–310, 1988.

[Mobasseri, 1989] B. G. Mobasseri. Path planning under uncertainty: From a decision analytic perspective. In *IEEE International Symposium on Intelligent Control*, pages 556–560, 1989.

[Mobasseri, 1990] B. G. Mobasseri. Decision analytic approach to weighted region problem. In *SPIE Mobile Robots, V*, pages 438–445, 1990.

[Nilsson, 1982] Nils J. Nilsson. *Principles of Artificial Intelligence.* Springer-Verlag Berlin Heidelberg New York, 1982.

[Patrick *et al.*, 1992] B. G. Patrick, M. Almulla, and M. M. Newborn. An upper bound on the time complexity of iterative–deepening A*. In *Annals of Mathematics and Artificial Intelligence.* 1992.

[Patrick, 1992] 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.

[Pearl, 1984] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, 1984.

[Poole, 1992] D. Poole. Probabilistic Horn Abduction and Bayesian network. Technical Report 92-20, Department of Computer Science, University of British Columbia, B.C. Canada, 1992.

[Provan and Poole, 1991] 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.

[Puterman, 1990] 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.

[Qi and Poole, 1991] 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.

[Qi and Poole, 1992a] 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.

[Qi and Poole, 1992b] 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.

[Qi, 1993] R. Qi. *Decision Graphs: algorithms and applications.* PhD thesis, Department of Computer Science, University of British Columbia, 1993. Forthcoming.

[Raiffa, 1968] H. Raiffa. *Decision Analysis.* Addison–Wesley Publishing Company, 1968.

[Russell and Wefald, 1989] 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.

[Russell, 1990] S. Russell. Fine–grained decision–theoretic search control. In *Proc. sixth Conference on Uncertainty in Artificial Intelligence*, 1990.

[Shachter and Peot, 1992] 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.

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

[Slate and Atkin, 1977] D. J. Slate and L. R. Atkin. *CHESS 4.5 — The Northweatern University University Chess Program.* Springer–Verlag, New York, 1977.

[Stochman, 1979] G. C. Stochman. A minimax algorithm better than alpha–beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[von Neumann and Morgenstern, 1947] J. von Neumann and O. Morgenstern. *Theory and Games and Economic Behavior.* Princeton University Press, 1947.

[Wellman, 1990] M. P. Wellman. *Formulation of Tradeoffs in Planning Under Uncertainty.* Pitman and Morgan Kaufman, 1990.

[Winston, 1984] P. H. Winston. *Artificial Intelligence.* Addison–Wesley, Reading MA, 1984.

[Zhang and Poole, 1992] 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.

[Zhang *et al.*, 1993] L. Zhang, R. Qi, and D. Poole. A comoutational theory of decision networks. *submitted to a journal, also available as a technique report 93-6, Department of Computer Science, UBC*, 1993.

A The Proofs of Theorems

Theorem 1 *If the heuristic function used by A1 is admissible, then*

$$dt_1(n,b) = \begin{cases} h^*(n) & \text{if } h^*(n) < b \\ \texttt{MAXINT} & \text{otherwise.} \end{cases}$$

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:

$$dt_0(n) = h^*(n) = v(n). \tag{1}$$

Case 2 n is a chance node:

$$dt_0(n) = t_0. \tag{2}$$

where $t_0 = t_{0l}$ and $t_{0i}, 0 \leq i \leq l$ is recursively defined as follows:

$$\begin{aligned} t_{00} &= \textstyle\sum_{j=1}^{l} h(n_j) * p_j; \\ t_{0i} &= t_{0i-1} + p_i * (dt_0(n_i) - h(n_j)). \end{aligned} \tag{3}$$

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

Case 3 n is a choice node:

$$dt_0(n) = t_0 \tag{4}$$

where $t_0 = t_{0l}$ and $t_{0i}, 0 \leq i \leq l$ is recursively defined as follows:

$$\begin{aligned} &t_{00} = \infty; \\ &t_{0i} = min\{t_{0i-1}, c_i + dt_0(n_i)\}. \end{aligned} \tag{5}$$

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 algorithms A1 and A2. This convention will be used in the rest of this section.

Second, we observe that, according to the structure of algorithm A1, the definition of dt_1 can be further refined as follows:

Case 1 n is a terminal;

$$dt_1(n,b) = \begin{cases} v(n) & \text{if } v(n) < b \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{6}$$

Case 2 n is a chance node:

$$dt_1(n,b) = \begin{cases} t_1 & \text{if } t_1 < b \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{7}$$

where $t_1 = t_{1l}$ and $t_{1i}, 0 \leq i \leq l$ is recursively defined as follows:

$$\begin{aligned} &t_{10} = \textstyle\sum_{j=1}^{l} h(n_j) * p_j; \\ &b_{1i} = (b - (t_{1i-1} - h(n_i) * p_i))/p_i; \\ &t_{1i} = \begin{cases} t_{1i-1} & \text{if } t_{1i-1} \geq b. \\ t_{1i-1} + p_i * (dt_1(n_i, b_{1i}) - h(n_i)) & \text{otherwise.} \end{cases} \end{aligned} \tag{8}$$

Case 3 n is a choice node:

$$dt_1(n,b) = \begin{cases} t_1 & \text{if } t_1 < b \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{9}$$

where $t_1 = t_{1l}$ and $t_{1i}, 0 \leq i \leq l$ is recursively defined as follows:

$$\begin{aligned} &t_{10} = b; \\ &t_{1i} = \begin{cases} t_{1i-1} & \text{if } t_{1i-1} - c_i \leq h(n_i) \\ min\{t_{1i-1}, c_i + dt_1(n_i, t_{i1-1} - c_i)\} & \text{otherwise} \end{cases} \end{aligned} \tag{10}$$

Thus, to prove the theorem, it suffices to prove

$$dt_1(n,b) = \begin{cases} dt_0(n) & \text{if } dt_0(n) < b \\ \texttt{MAXINT} & \text{otherwise.} \end{cases} \tag{11}$$

for every node n in the decision tree.

Based on the definition of dt_0 and the characterization of dt_1 given above, relation (11) can be proved by induction on the structure of the decision graph. First, we need two intermediate results.

Lemma 5 *Let n be a choice node with children* $n_1, ..., n_l$. *Let* t_{0i} *and* t_{1i} *be defined by equations (5) and (12) respectively. Suppose that for any number* b',

$$dt_1(n_i, b') = \begin{cases} dt_0(n_i) & \text{if } dt_0(n_i) < b' \\ \texttt{MAXINT} & \text{otherwise.} \end{cases}$$

for $1 \leq i \leq l$. *Then, for any* i, $1 \leq i \leq l$
(A) $t_{1i} < b$ *iff* $t_{0i} < b$;
(B) if $t_{0i} < b$, *then* $t_{0i} = t_{1i}$, *otherwise.* $t_{1i} = b$.

Proof We prove this lemma by induction on i. Our observation here is that, under the given assumptions, equation (10) is equivalent to the following simpler one:

$$\begin{aligned} t_{10} &= b; \\ t_{1i} &= min\{t_{1i-1}, c_i + dt_1(n_i, t_{1i-1} - c_i)\} \end{aligned} \tag{12}$$

Basis $i = 1$. $t_{11} = min\{b, c_1 + dt_1(n_1, b - c_1)\}$ 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:

$$dt_1(n_1, b - c_1) = \texttt{MAXINT} > b.$$

Therefore

$$t_{11} = b.$$

If

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

then,

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

By the given assumptions, we have:

$$dt_0(n_1) = dt_1(n_1, b - c_1);$$

$$c_1 + dt_1(n_1, b - c_1) = c_1 + dt_0(n_1) < b.$$

Therefore, $t_{11} = c_1 + dt_0(n_1) = t_{01}$. The induction base holds.
Induction Suppose the lemma is true for $i = k$. For $i = k + 1$, we have:

$$t_{1k+1} = min\{t_{1k}, c_{k+1} + dt_1(n_{k+1}, b - c_{k+1})\};$$

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

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 $t_{1k} = b$ by the induction assumption. Furthermore, since

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

then, by the given assumptions, we have:

$$dt_1(n_{k+1}, t_{1k} - c_{k+1}) = dt_1(n_{k+1}, b - c_{k+1}) = \texttt{MAXINT}.$$

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

(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: $t_{1k} = b$. By the given assumptions, we obtain:

$$dt_1(n_{k+1}, b - c_{k+1}) = dt_0(n_{k+1}).$$

Thus,

$$t_{1k+1} = c_{k+1} + dt_1(n_{k+1}, b - c_{k+1}) = dt_0(n_{k+1}) + c_{k+1}.$$

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

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

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

$$t_{0k} = t_{1k} \text{ (by the induction assumption).}$$

Thus,

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

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

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

$$t_{0k} = t_{1k} \text{ (by the induction assumption);}$$

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

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

Thus, by the given assumptions, we have:

$$dt_1(n_{k+1}, t_{1k} - c_{k+1}) = dt_0(n_{k+1}).$$

Thus,

$$dt_1(n_{k+1}, b - c_{k+1}) + c_{k+1} = dt_0(n_{k+1}) + c_{k+1} < t_{1k}.$$

Therefore, $t_{1k+1} = dt_1(n_{k+1}, b - c_{k+1}) + c_{k+1} = t_{0k+1}$.

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

Lemma 6 *Let n be a chance node with children $n_1, ..., n_l$. Let t_{0i} and t_{1i} be defined by equations (3) and (8) respectively. Suppose that for any number b',*

$$dt_1(n_i, b') = \begin{cases} dt_0(n_i) & \text{if } dt_0(n_i) < b' \\ \texttt{MAXINT} & \text{otherwise.} \end{cases}$$

for $1 \leq i \leq l$. Then, for any i, $1 \leq i \leq l$
(A) $t_{1i} < b$ iff $t_{0i} < b$;
(B) if $t_{0i} < b$, then $t_{0i} = t_{1i}$.

Proof: By induction on i, similar to that for Lemma 6 . □

Proof of Theorem 1. We prove relation (11) by induction on nodes in the decision tree.

1. n is a terminal. Then $v(n) = h^*(n) = dt_0(n)$. Thus, relation (11) holds trivially.

2. n is a choice node. Suppose relation (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 equations (4) and (5))

   $t_1 = t_{1l} \geq b$ (by Lemma 5)

   $dt_1(n) = \texttt{MAXINT}$ (by equation (9)).
   (B). $dt_0(n) \leq b$. We have the following derivation:

   $dt_0(n) = t_{0l} < b$ (by equations (4) and (5)

   $t_{1l} = t_{0l} \geq b$ (by Lemma 5)

   $dt_1(n) = t_1 = t_{0l} = dt_0(n)$ (by equation 9).
   In summary, relation (11) holds for n.

3. n is a chance node. Suppose relation (11) holds for all the children of n. Similarly, it can be proved that the relation holds for n as well by using Lemma 6.

In summary, the theorem holds in general. □

Theorem 2 *Suppose A1 uses heuristic function h. If there exists a number $\delta \geq 0$ such that h satisfies:*

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

then

$$h^*(n) + \delta \geq b \quad \text{if } dt_1(n, b) \geq b;$$

and

$$h^*(n) + \delta \geq dt_1(n, b) \quad \text{if } dt_1(n, b) < b$$

for every node n in the decision tree.

Theorem 3 *Suppose A1 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 $b \geq 0$,

$$h^*(n) * (1+\delta) \geq b \quad \textit{if } dt_1(n,b) \geq b;$$

and

$$h^*(n) * (1+\delta) \geq dt_1(n,b) \quad \textit{if } dt_1(n,b) < b.$$

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 2, it suffices to prove that for every node n in the decision tree

$$dt_0(n) + \delta \geq b \quad \text{if } dt_1(n,b) \geq b;$$

and

$$dt_0(n) + \delta \geq dt_1(n,b) \quad \text{if } dt_1(n,b) < b$$

For Theorem 3, it suffices to prove that for every node n in the decision tree

$$dt_0(n) * (1+\delta) \geq b \quad \text{if } dt_1(n,b) \geq b;$$

and

$$dt_0(n) * (1+\delta) \geq dt_1(n,b) \quad \text{if } dt_1(n,b) < b$$

The proofs of Theorems 2 and 3 are very similar. Here we just present the proof of Theorem 3.

Proof of Theorem 3

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 $dt_1(n,b) \geq b$, then $v(n) = h^*(n) \geq b$, therefore, $dt_0(n) * (1+\delta) \geq b$.

If $dt_1(n,b) < b$, then $dt_1(n,b)v(n) = h^*(n) = dt_0(n)$, thus $dt_0(n) * (1+\delta) \geq dt_1(n,b)$.

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). $dt_1(n,b) < b$. According to equation (7), we have $dt_1(n,b) = t_1 = t_{1l} < b$. Thus, $t_{1i} < b$ for $i = 1,...,l$. Concequently, by equation (8), we have: $dt_1(n_i, b_{1i}) < b_{1i}$. By the induction assumption, we have:

$$dt_1(n_i, b_{1i}) \leq (1+\delta) * dt_0(n_i)$$

for $i = 1,...,l$. According to equations (3) and (2), we obtain:

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

According to equation (8), we obtain:

$$t_{1l} = \sum_{i=1}^{l} p_i * dt_1(n_i, b_{1i}) \leq \sum_{i=1}^{l} p_i * dt_1(n_i) * (1+\delta) = t_{0l} * (1+\delta)$$

Thus, $dt_1(n,b) \leq dt_0(n) * (1+\delta)$.

(B). $dt_1(n,b) \geq b$. According to equations (7) and (8), we know that $t_1 = t_{1l} \geq b$. This implies that either $t_{10} \geq b$ or there exists $k, 1 \leq k \leq l$ such that $t_{1k-1} < b$ and $t_{1k} \geq b$. In the former case, we have:

$$dt_0(n) * (1+\delta) = \sum_{i=1}^{l} p_i * dt_0(n_i) * (1+\delta) \geq \sum_{i=1}^{l} p_i * h(n_i) = t_{10}$$

Thus, $dt_0(n) * (1+\delta) \geq b$.
In the latter case, we can obtain:

$$dt_1(n_j, b_{1j}) < b_{1j} \quad \text{for} \quad 0 \leq j < k$$

$$dt_1(n_k, b_{1k}) \geq b_{1k}$$

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

$$dt_0(n_j) * (1+\delta) \geq dt_1(n_j, b_{1j}) \quad \text{for} \quad 0 \leq j < k$$

and

$$(1+\delta) * dt_0(n_k) \geq b_{1k}$$

Therefore:

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

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

According to equation (8), we have:

$$t_{1k-1} = \sum_{j=1}^{k-1} dt_1(n_j, b_{1j}) * p_j + \sum_{j=k}^{l} h(n_j) * p_j$$

Thus,

$$\sum_{j=1}^{k-1} dt_1(n_j, b_{1j}) * p_j + p_k * (1+\delta) * dt_0(n_k) + \sum_{j=k+1}^{l} h(n_j) * p_j \geq b$$

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

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

$$t_{1i} \leq (1+\delta) * t_{0i} \tag{13}$$

for $i = 0, ..., l$, where t_{0i} and t_{1i} are defined by equations (5) and (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 $t_{1k+1} \leq t_{1k}$, by the inner induction assumption, we conclude $t_{1k+1} \leq (1+\delta) * t_{0k+1}$.

(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 $t_{1k} - c_{k+1} \leq h(n_{k+1})$, then,

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

Since $c_{k+1} \geq 0$ and $\delta \geq 0$, we have $t_{1k+1} \leq (1+\delta) * t_{0k+1}$.
If $t_{1k} - c_{k+1} > h(n_{k+1})$, then,

$$t_{1k+1} = min\{t_{1k}, c_{k+1} + dt_1(n_{k+1}, t_{1k} - c_{k+1})\}$$

When $t_{1k} - c_{k+1} \leq dt_1(n_{k+1}, t_{1k} - c_{k+1})$, by the outer induction assumption, we have:

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

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

Since $c_{k+1} \geq 0$ and $\delta \geq 0$, we have $t_{1k+1} \leq (1+\delta) * t_{0k+1}$.
When $t_{1k} - c_{k+1} > dt_1(n_{k+1}, t_{1k} - c_{k+1})$, by the outer induction assumption, we have:

$$(1+\delta) * dt_0(n_{k+1}) \geq dt_1(n_{k+1}, t_{1k} - c_{k+1})$$

$$t_{1k+1} = c_{k+1} + dt_1(n_{k+1}, t_{1k} - c_{k+1}) \leq c_{k+1} + (1+\delta) * dt_0(n_{k+1})$$

Since $c_{k+1} \geq 0$ and $\delta \geq 0$, we have $t_{1k+1} \leq (1+\delta) * t_{0k+1}$.
By induction, inequality 13 holds for all $i = 0, ..., l$.

In summary, the theorem holds for any node n. □