# A Hill-Climbing Approach to Construct Near-Optimal Decision Trees

*Xiaorong Sun, Yuping Qiu and Louis Anthony Cox, Jr.*
U S WEST Technologies, 4001 Discovery Drive, Boulder, CO 80303

**Abstract**

We consider the problem of identifying the state of an $n$ component coherent system, where each component can be *working* or *failed*. It is costly to determine the states of the components. The goal is to find a decision tree which specifies the order of the components to be tested with minimum expected cost. The problem is known to be NP-hard. We present an extremely promising heuristic method for creating effective decision trees, and computational results show that the method obtains optimal solutions for 95% of the cases tested.

## 1 Introduction

Following the definition of [1], a coherent system is composed of a set of $n$ components $E = e_1, e_2, \cdots, e_n$ each of which can be in one of two states – working or failed. Component $i$ has a *state value* $x_i$ such that

$$x_i = \begin{cases} 1 & \text{if component } i \text{ is functioning,} \\ 0 & \text{if component } i \text{ is failed.} \end{cases}$$

We call a 0,1-vector $X = (x_1, x_2, \cdots, x_n)$ a *state vector* if $x_i$ is the state value of component $i$ for $i = 1, \cdots, n$. The *structure function* $\phi(X)$ is defined as

$$\phi(X) = \begin{cases} 1 & \text{if the system } i \text{ is functioning,} \\ 0 & \text{if the system } i \text{ is failed.} \end{cases}$$

The $i$th component is *irrelevant* to the structure $\phi$ if $\phi(x_1, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n) = \phi(x_1, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n)$ for any 0,1-vector $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$. Otherwise, the $i$th component is *relevant* to the structure $\phi$. $(E, \phi)$ is called a *coherent system* if

(i) If $X \leq Y$ (i.e., $x_i \leq y_i$ for $i = 1, \cdots, n$), then $\phi(X) \leq \phi(Y)$.

(ii) All components are relevant.

When $\phi(X)$ is defined by a linear inequality $w_1x_1 + w_2x_2 + \cdots + w_nx_n \geq k$ with positive coefficients, i.e.,

$$\phi(X) = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + \cdots + w_nx_n \geq k, \\ 0 & \text{otherwise} \end{cases}$$

then the system is called a *linear threshold system.* As special cases, when $w_1 = \cdots = w_n = 1$, the system is called a *k-out-of-n* system. When $w_1 = \cdots = w_n = 1$ and $k = 1$, it is called a *parallel system.* When $w_1 = \cdots = w_n = 1$ and $k = n$, it is called a *series system.*

For each component $e_i$ of a coherent system $(E, \phi)$, denote by $p_i$ and $q_i = 1 - p_i$ the working and failed probabilities, respectively. In other word, for $i = 1, 2, \cdots, n$, let

$$Pr\{x_i = 1\} = p_i; \quad Pr\{x_i = 0\} = q_i.$$

We assume that the random variables $x_1, \cdots, x_n$ are independent. We also assume that there is a cost $c_i$ to inspect the state of the $i$th component.

An inspection strategy can be described as a *binary decision tree* which is a labeled rooted binary tree. The internal nodes are labeled by the components to be tested, and the leaves of the tree are labeled by the states of the system. Each internal node $v$ with label $i$ has two sons which are labeled by the components to be tested if the $i$th component is found to be working or failing. The two sons are called the *1-son* and *0-son* of $v$ respectively, and $v$ is called the *father* of its sons. Thus the first component to be tested is the one given by the root label. If this component is found to be functioning (failed) and the state of the system is still unknown, then the next component to be

tested is the one labeled by the 1-son (0-son) of the root. As an example, for the following linear threshold system

$$\phi(X) = \begin{cases} 1 & \text{if } 3x_1 + 2x_2 + x_3 + x_4 \geq 5, \\ 0 & \text{otherwise} \end{cases}$$

A decision tree is shown in Figure 1. Circles represent internal nodes with labels inside, 1-son on the right, and 0-son on the left. The squares represent leaves with labels "F" and "W" which mean that the whole system is failed, and working, respectively. The letters on the right side of the nodes are the node names. Node $r$ is the root.

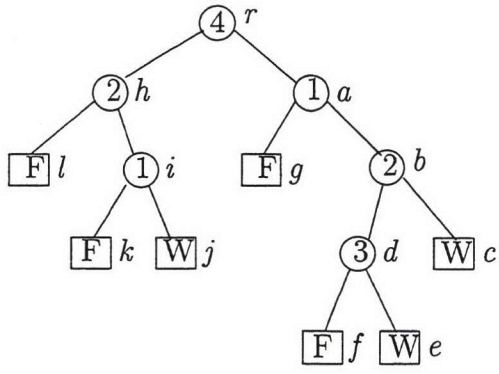


Figure 1: An example of a decision tree

For any decision tree, define the cost of the leaves to be zero. For any internal node $v$ with label $i$, let $v_0$ and $v_1$ be its 0-son and 1-son respectively. Define the cost $C(v)$ of node $v$ is by $c_i + p_i C(v_1) + q_i C(v_0)$. The cost of the tree is defined to be the cost of its root.

With these definitions, our problem now can be stated as: Given an $n$ component coherent system $(E, \phi)$ and $p_1, p_2, \cdots, p_n$, $c_1, c_2, \cdots, c_n$, find a decision tree with minimum expected cost. An optimal decision tree is also called an *optimal inspection strategy.*

The problem arises in many logical inference, reliability testing, pattern recognition, and statistical classification applications. It has been proved to be NP-hard [3]. The only known polynomial-time solvable case is k-out-of-n system[1, 2], which includes series and parallel systems.

It is obvious that an optimal decision strategy does not test any component twice, which implies for any leaf, the path from the root to the leaf contains no pair of nodes which are labeled by the same component. So if we already observed the state values of some components (which form a *partial tree*), then we need only consider the subsystem restricted by these values. Known heuristics all focus on methodologies to grow a partially defined tree [3, 4], using which, a decision tree can be easily built up. But there has been no method to improve the decision tree found, simply because there has been no way to move from one tree to another in the space of all decision trees. This paper develops a new method to improve a given decision tree, which can be either a trivial one, or the one found by heuristic procedures. The key idea is the *delete/add* ( DA ) move defined in Section 2. We shall prove that, in the space of all decision trees for a given coherent system, any tree can be obtained from another by a series of DA moves. Thus, DA moves makes it possible to utilize the idea of hill climbing in the space of all decision trees.

Section 3 shows that our heuristic actually gives the optimum solutions for parallel and series systems. Computational results are reported in Section 4.

## 2 The DA move

For a coherent system $(E, \phi)$, a *path set* is defined as any set of component values for a state vector $X$ that guarantees $\phi(X) = 1$. A *minimal path set* is a path set that ceases to be one if any of the component values in it are removed [3]. Similarly, a *cut set* is defined as any set of component values for a state vector $X$ that guarantees $\phi(X) = 0$, and a *minimal cut set* is a cut set containing no proper subsets that are also cut sets.

Now consider a decision tree $T$, and a path $P$ from the root $r$ to a leaf $v$ labeled "F" ("W"). It can be seen that the set of the component values determined by the path $P$ form a path (cut) set of the system. A path $P$ from the root to a leaf of the tree $T$ is a *minimal path* of the tree if the corresponding path (cut) set is minimal. The following lemma can be easily seen.

**Lemma 2.1** *Suppose $T$ is a decision tree and $P$ is a path from the root to a leaf labeled by "F" ("W"). If $P$ has a component $i$ which has value 1 (0) in path $P$, then $P$ is not minimal.*

Visually, for any decision tree drawn on the plane, if we place 1-sons under the right and 0-sons under the left of their fathers, then any path except the extreme left and right ones is not minimal. For any non-minimal path $P$, there is a node with label component $i$ such that after $i$ is deleted from the path $P$, the values of the other components of $P$ still identify the state of the system. Thus, it is unnecessary to test the component $i$ if the actual system state is determined by the path $P$. In this case, we say that component $i$ is *inessential* to path $P$. For a path $P$ with an inessential component $i$ of a decision tree $T$, we can build a new tree $T_{P,i}$ in which component $i$ is deleted from $P$. We then add the component $i$ to other paths of $T$, and use subtrees of $T$ to build the new tree $T_{P,i}$ in a way described below. This is the main idea of the DA move, i.e., **Delete** a component from a path, and **Add** it to other paths if necessary.

We now give a formal definition of the DA move. Consider a decision tree $T$ with root $r$, and a path $P = v_1, v_2, \cdots, v_p$ such that $v_1 = r$ and $v_p$ is a leaf ($P$ is shown by the dotted line in Figure 2). Assume $P$ is not minimal with an inessential component $i$ and node $v_k$ has label $i$ (note that there is only one such node in $P$). For any $l$ such that $k \leq l < p$, assume $j$ is the label of node $v_l$ of $P$. Denote by $t_j$ the value of the component $j$ determined by the path $P$, and let $s_j = 1 - t_j$. Denote by $T_l$ the subtree of $v_l$ rooted at the $s_j$-son of $v_l$.

Since $i$ is inessential in path $P$, we may build a new tree in which node $v_k$ will be deleted from $P$, For any $l$ such that $k \leq l < p$, $v_l$ will be also a node in the new tree, and having the same label as in the old tree $T$. Since we want $P$ remain a path of the new tree, the $t_j$-son of $v_l$ is still $v_{l+1}$. The $s_j$-son of $v_l$ in the new tree will be a new node $u_l$ with label $i$ (see Figure 3). Obviously, we may still use $T_l$ as the $t_i$-son of $u_l$ (strictly speaking, as the subtree rooted at the $t_i$-son of $u_l$). The $s_i$-son of $u_l$ will be *derived* from the subtree $T_k$ in the following way.

We can not simply use $T_k$ as the $s_i$ son of $u_l$

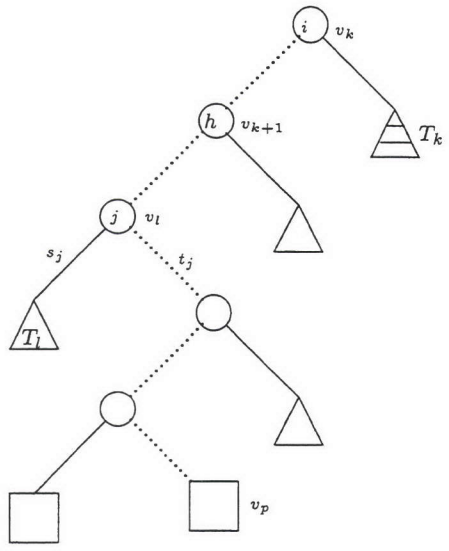


Figure 2: Current tree

since otherwise some paths may contain the some components twice.

Let $h$ be the label of $v_{k+1}$ in $T$. Consider the subtree $N_{k+1}$ rooted at the $s_i$-son of $u_{k+1}$ in the new tree $T_{P,i}$, and the subtree $T_k$ of the old tree $T$. Both subtrees assume exactly the same state values of the components labeled by $v_1, \cdots, v_{k-1}$. The difference between them is that $N_{k+1}$ assumes that the state value of component $h$ is also known. Therefore, we can use the same strategy specified by $T_k$ to test the sub-system restricted by the values of the components labeled by $v_1, \cdots, v_{k+1}$, only remembering that we do not have to test component $h$. Denote by $T_k^{k+1}$ the new subtree corresponding to the strategy specified by $T_k$. The new subtree $T_k^{k+1}$ will be the $s_i$-son of $u_{k+1}$ in the new tree.

In terms of trees, $T_k^{k+1}$ can be obtained from $T_k$ by replacing all subtrees rooted at those nodes which have the same label $h$ as $v_{k+1}$ by the corresponding subtrees rooted at its $s_h$-son. We shall say that $T_k^{k+1}$ is *derived* from $T_k$ by assuming the value of $h$ to be $s_h$.

In general, for any $l$ such that $k+1 < l < p$, we can build a subtree $T_k^l$ from $T_k$ as the $s_i$-son of $u_l$ in the new tree. For the sequence $T_k^{k+1}, \cdots, T_k^{p-1}$ of new subtrees built in this way, each one can be obtained from the previous one by assuming

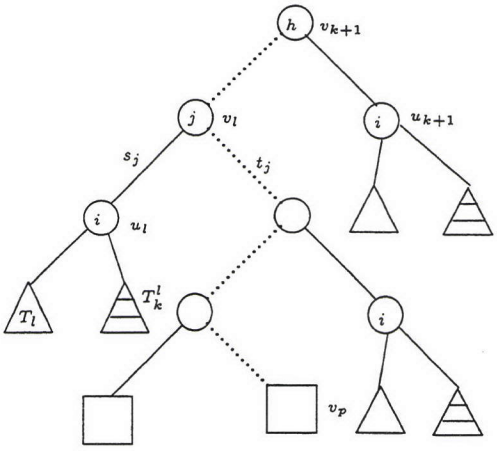


Figure 3: New tree after one DA move

only one more state value of a component. Thus $T_k^{k+1}, \cdots, T_k^{p-1}$ can be derived by letting $T_k^k = T_k$, and for $l = k+1, \cdots, p-1$, let $T_k^l$ be equal to the tree derive from $T_k^{l-1}$ by assuming the value of component $j$ to be $s_j$, where $j$ is the label of node $v_l$.

Using $T_k^{k+1}, \cdots, T_k^{p-1}$, we can define a new tree from a given decision tree $T$. Assume $P = v_1, \cdots, v_p$ is a path of $T$ from the root to a leaf which is not minimal with an inessential node $v_k$ labeled with component $i$. Define a new tree $T_{P,i}$ by the following rules.

(a) If $k \geq 1$, then replace the $t_i$-son of $v_k$ by $v_{k+1}$;

Else replace the root of $T$ by $v_{k+1}$.

(b) For each node $v_l (k < l < p)$ of $P$ with label $j$:

If the set of component values defined by $s_j$ and $t_{j'}$ for every node $v_{l'} (1 \leq l' < l)$ with label $j'$ form a path (cut) set, then add a $s_j$-leaf to $T_{P,i}$ with label "F" ("W");

Else add a new node $u_l$ with label $i$, and let $T_k^l$, $T_l$ be its $s_i$-son and $t_i$-son respectively.

(c) All other nodes will be unchanged.

The following lemma holds:

**Lemma 2.2** *The new tree $T_{P,i}$ is also a decision tree for the same coherent system.*

**Proof**: The definitions of $T_k^{k+1}, \cdots, T_k^{p-1}$, together with rules (a) and (b) guarantee that no path of the tree contains the same component more than once. Rules (b) and (c) guarantee that the leaf labels of $T_{P,i}$ identify the states of the system. □

In the space of all decision trees, the move from $T$ to the new decision tree $T_{P,i}$ constitutes a *DA move.* A given decision tree can be improved by a hill-climbing heuristics using DA moves. One (greedy) approach is to always move to the next tree which has the maximum amount of improvement. Another method is to move to the first direction in which an improvement is found. The two algorithms are presented in Figures 4 and 5.

*ALGORITHM: DA-climb*
*INPUT: A coherent system $(E, \phi)$ and a decision tree $T$.*
*OUTPUT: An improved decision tree $T$.*

*Step 1. Find a path $P$ which is not minimal with an inessential component $i$ such that $C(T_{P,i}) < C(T)$.*
*If there is no such path, then stop and output $T$; Else go to Step 2.*

*Step 2. Let $T = T_{P,i}$, and go to Step 1.*

Figure 4: DA-climb heuristics

*ALGORITHM: Greedy DA-climb*

*Step 1. Find a path $P$ which is not minimal with an inessential component $i$ such that $C(T) - C(T_{P,i})$ is maximized.*
*If the the maximum is 0, then stop and output $T$; Else go to Step 2.*

*Step 2. Let $T = T_{P,i}$, and go to Step 1.*

Figure 5: Greedy DA-climb heuristics

The following theorem theoretically guarantees the quality of the solutions found by the heuristics.

**Theorem 2.1** *Let $T_1$ and $T_2$ be any two decision trees of the same coherent system. Then $T_1$ can be always transformed to $T_2$ by a series of DA moves.*

**Proof**: Let $i_1$ be the label of the root $r_1$ of $T_1$,

and $i_2$ be that of the root $r_2$ of $T_2$, By the definition of coherent system, $T_1$ must have a node $v$ with label $i_2$. If $i_1 \neq i_2$, then the node $v$ of $T_1$ must have a father $u$. By Lemma 2.1, there is a path $P$ in $T_1$ such that $u$ is inessential for $P$. It can be seen that in the tree $T_{P,u}$ obtained by a single DA move, the depth of $v$ which has a label $i_2$ is decrease by 1. After a series of such DA moves, eventually we will have a tree $T'$ whose root has the same label $i_2$ as $T_2$.

Now the two subtrees of the root of $T'$ define the same sub-systems as that of $T_2$, respectively. The theorem then follows by induction on the number of the components in the coherent system. □

## 3 Series and parallel systems

For these special systems, the decision tree corresponds to a permutation of all components of the system. Therefore, we have

**Lemma 3.1** *For series and parallel systems, one DA move is equivalent to switching the order of two adjacent nodes of the permutation tree.*

**Proposition 3.1** *Both heuristics DA-climb and Greedy DA-climb give the optimum decision tree for series and parallel systems.*

**Proof**: This is a direct conclusion of Lemma 3.1 and the known fact that [1] the optimal test order for series system is the permutation $\pi_1, \cdots, \pi_n$ such that

$$\frac{c_{\pi_1}}{q_{\pi_1}} \leq \cdots \leq \frac{c_{\pi_n}}{q_{\pi_n}}.$$

## 4 Computational results

Both hill-climbing heuristics were coded in C, and run on a Sparc 10 workstation. The optimal decision tree is found by a dynamic programming recursion [3].

Heuristic *DA-climb* uses a *depth-first search* [6] method to find if there is a path $P$ which is not minimal with an inessential component $i$ such that $C(T_{P,i}) < C(T)$. If such a path $P$ and $i$ are found, then we immediately replace $T$ by $T_{P,i}$. *Greedy DA-climb* uses the same search method to exhaust all paths of current tree, and finds the maximum defined in Step 1.

Our heuristics have been tested on linear threshold systems, which are an important class of problems [5]. The structure functions are generated using linear inequalities $w_1x_1 + \cdots + w_nx_n \geq k$ whose coefficients are uniformly distributed in the interval $[1, 99]$, and $k$ is set to largest integer not larger than $\beta \sum_{i=1}^{n} w_i$.

From lemma 2.1, a minimal path (cut) set can not contain components with value 0 (1). To check if a subset of components with value 1's form a minimal path set, we need only check if the sum the coefficients $w_i$ of these components is not less than $k$, and if any proper subsets with one component less also has such property. Similarly, to check if a subset of components with value 0's form a minimal cut set, we need only check if the sum the coefficients $w_i$ of the components not in the subset is not less than $k$, and if any subsets with one more component also has such property.

The probabilities $p_1, \cdots, p_n$ are uniformly generated in the interval $(0.01, 0.95)$. The costs $c_1, \cdots, c_n$ are real numbers uniformly distributed in the interval $(1.00, 99.00)$.

The initial decision tree is constructed based on the strategy which examines all components in the order $\pi_1, \cdots, \pi_n$ such that $w_{\pi_1} \geq \cdots \geq w_{\pi_n}$.

For each $n = 5, 6, 7$, both heuristics were run for 100 problems. In Table 1, $\beta = 0.3$. The left and right column under Opt(%) show the number of times heuristic *DA-climb* and *Greedy DA-climb* obtain the optimum solutions. The columns under Ave.Err(%) show the average errors relative to the optimum solutions of the two heuristics. The columns under Larg.Err(%) show the largest error relative to the optimum solutions of the two heuristics.

Table 2 shows the same measures for 100 randomly generated problems generated using $\beta = 0.5$.

Since we can find the exact optimum of $k$-out-of-$n$ system for any $n$ [1], we compared our heuristic

| $n$ | Opt(%) | | Ave.Err(%) | | Larg.Err(%) | |
|---|---|---|---|---|---|---|
| 5 | 98 | 99 | 0.01 | 0.00 | 0.94 | 0.06 |
| 6 | 98 | 97 | 0.01 | 0.01 | 0.60 | 1.07 |
| 7 | 97 | 96 | 0.01 | 0.04 | 0.48 | 2.84 |

Table 1: $\beta = 0.3$

| $n$ | Opt(%) | | Ave.Err(%) | | Larg.Err(%) | |
|---|---|---|---|---|---|---|
| 5 | 99 | 99 | 0.03 | 0.03 | 2.69 | 2.69 |
| 6 | 95 | 93 | 0.01 | 0.01 | 0.38 | 0.38 |
| 7 | 95 | 95 | 0.02 | 0.02 | 1.43 | 1.43 |

Table 2: $\beta = 0.5$

solutions with optimum for larger $n$. In Table 3, we can see that for all problems we tested, both heuristics find optimum solutions. The columns under Ave.CPU show the average CPU time in seconds. The columns under Ave.Iter show the number of steps each heuristic took to reach a local optimum. Since *Greedy DA-climb* needs to search the whole tree it is not surprising to see that it takes more CPU time than *DA-climb*, but takes fewer steps to reach a local optimum.

| $n$ | Opt(%) | | Ave.CPU | | Ave.Iter | |
|---|---|---|---|---|---|---|
| 5 | 100 | 100 | 0.01 | 0.02 | 8.6 | 5.0 |
| 6 | 100 | 100 | 0.14 | 0.31 | 22.1 | 11.7 |
| 7 | 100 | 100 | 0.61 | 1.39 | 39.6 | 19.2 |
| 8 | 100 | 100 | 1.91 | 5.40 | 64.8 | 30.4 |
| 9 | 100 | 100 | 9.15 | 31.27 | 102.3 | 46.6 |
| 10 | 100 | 100 | 23.69 | 78.11 | 163.2 | 69.5 |

Table 3: *k-out-of-n* system, $\beta = 0.8$

From these computational experiments it appears that, in a short CPU time, DA-climb gives better solutions. But in the long run, the Greedy method gives better solutions. Figure 6 shows the result for one particular problem with $n = 14$ and $\beta = 0.5$, where the horizontal axis is the CPU time in seconds, and the vertical axis is the value found by the two heuristics. Depending on whether computational time requirement is critical, the user may choose *DA-climb* or *Greedy DA-climb*.

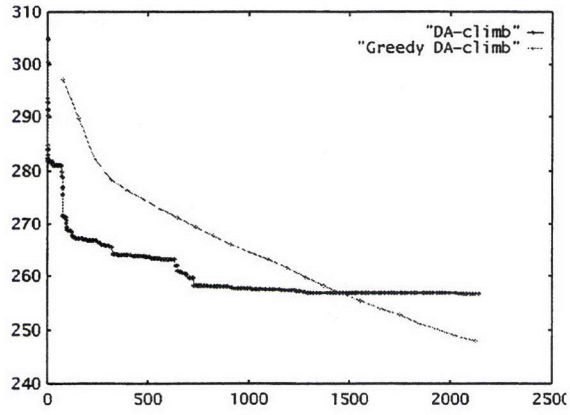


Figure 6: DA-climb vs. Greedy DA-climb

## 5 Conclusion

We have developed a method to search the space of binary decision trees for a coherent system using locally improving (hill-climbing) search. In most cases the heuristics give optimal solutions. While hill-climbing is ubiquitously used on continuous optimization problems, the possibility of applying it to classification trees has not previously been recognized.

Compared to the dynamic programming procedure, which may take a long time without giving any clue of an optimal solution, our hill-climbing heuristics always maintain a feasible solution. Although the heuristic may also take long time because of the size of the classification tree, it can be terminated at any time and output a feasible solution.

## References

[1] Y. Ben-Dov, Optimal testing procedures for special structures of coherent system, *Management Science* 27, No. 12 (December 1981) 1410-1420.

[2] M.F. Chang, W, Shi, W.K. Fuchs, Optimal diagnosis procedures for k-out-of-n system, *IEEE Trans. on Computers* 39(4) (1990) 559-564.

[3] L.A. Cox, Y. Qiu and W. Kuehner, Heuristic least-cost computation of discrete classification functions with uncertain argument values, *Annals of Operations research*, 21(1989) 1-30.

[4] K. Fraughnaugh, J. Ryan, H. Zullo, L.A. Cox, Heuristics for efficient classification, in *this volume.*

[5] C.L. Sheng, *Threshold Logic*, Academic Press, New York, 1969.

[6] R.E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Computing* 1 (1972) 146-160.